Na prawach rękopisu

KATEDRA INFORMATYKI
TECHNICZNEJ
WYDZIAŁ ELEKTRONIKI
POLITECHNIKI WROCŁAWSKIEJ

Raport serii PREPRINTY nr: / 2020

Programowanie aplikacji współbieżnych w systemie Linux

Jędrzej UŁASIEWICZ

Słowa kluczowe:
- Aplikacje współbieżne
- Komunikacja międzyprocesowa
- System Linux
- POSIX 1003.1

Wrocław 2020

Spis treści

1. Wstęp ... 5
2. Podstawy posługiwania się systemem Linux ... 7
2.1 Wstęp ... 7
2.2 Uzyskiwanie pomocy ... 7
2.3 Użytkownicy ... 9
2.4 Operowanie plikami i katalogami ... 9
2.5 Program do zarządzania komputerem Midnight Commander ... 13
2.6 Operowanie procesami ... 14
2.7 Zadania ... 15
3. Kompilacja i uruchamianie programów ... 17
3.1 Jak kod źródłowy przekształca się w proces ... 17
3.2 Metoda elementarna – użycie edytor gedit i kompilatora gcc ... 18
3.3 Kompilator gcc ... 21
3.4 Biblioteki ... 23
3.5 Uruchamianie programów za pomocą narzędzia make ... 28
3.6 Uruchamianie programów za pomocą narzędzia gdb ... 32
3.7 Debugowanie biegnacego procesu ... 36
3.8 Zintegrowane środowisko uruchomieniowe Eclipse ... 37
3.9 Pisanie bezpiecznego kodu, makro assert ... 37
3.10 Odwołanie się do argumentów programu ... 38
3.11 Zadania ... 38
4. Tworzenie procesów ... 41
4.1 Funkcja fork ... 41
4.2 Schemat użycia funkcji execl. ... 41
4.3 Ustanawianie ograniczeń na użycie zasobów ... 42
4.4 Zadania ... 45
5. Pliki ... 51
5.1 Podstawowa biblioteka obsługi plików ... 51
5.2 Niskopoziomowe funkcje dostępu do plików ... 51
5.3 Standardowa biblioteka wejścia / wyjścia - strumienie ... 52
5.4 Blokady plików ... 53
5.5 Zadania ... 54
6. Łącza nienazwane i nazwane ... 59
6.1 Łącza nienazwane ... 59
6.2 Łącza nazwane ... 60
6.3 Zadania ... 62
7. Kolejki komunikatów POSIX ... 67
7.1 Wstęp ... 67
7.2 Zadania ... 70
8. Pamięć dzielona i semafory ... 73
8.1 Pamięć dzielona ... 73
8.2 Semafory ... 75
8.3 Zadania ... 77
9. Interfejs gniazd, komunikacja bezpołączeniowa ... 81
9.1 Adresy gniazd i komunikacja bezpołączeniowa ... 81
9.2 Zadania ... 84
10. Interfejs gniazd, komunikacja połączeniowa ... 89
10.1 Komunikacja połączeniowa ... 89
10.2 Zadania ... 93
11. Sygnały i ich obsługa. ... 95
11.1 Wstęp ... 95
11.2 Zadania ... 96
12. Wątki ... 97
12.1 Tworzenie wątków ... 97
12.2 Synchronizacja wątków ... 98
12.3 Zadania ... 98
Literatura ... 103

1. Wstęp

Praca zawiera materiały i zestaw ćwiczeń dotyczących tworzenia złożonych aplikacji w systemach zgodnych ze standardem POSIX 1003.1 co obejmuje w szczególności system LINUX. Ćwiczenia przeznaczone są dla studentów którzy przeszli już podstawowy kurs programowania w języku C i są zaznajomieni z programowaniem sekwencyjnym. Przedstawione tu ćwiczenia są kolejnym etapem zdobywania umiejętności programistycznych. Omawiane są metody tworzenia aplikacji złożonych z wielu wykonywanych współbieżnie procesów sekwencyjnych. Procesy te wykonywać się mogą na jednym komputerze wyposażonym w procesor jedno lub wielo rdzeniowy lub też w komputerach połączonych siecią Internet w więc w systemie rozproszonym. Aby procesy mogły tworzyć aplikację musza się komunikować. Stąd też szczególną uwagę poświęcono różnym metodom komunikacji międzyprocesowej zarówno lokalnym jak i sieciowym. Ćwiczenia obejmują następujące tematy:

1. Podstawy posługiwanie się systemem, uzyskiwanie informacji o stanie systemu, uzyskiwanie informacji o wywołaniach systemowych, funkcjach, aplikacjach (narzędzie `man`, `info`).
2. Tworzenie i uruchamianie programów w języku C z użyciem zintegrowanego środowiska Eclipse i narzędzie `make`.
3. Zarządzanie procesami, tworzenie i kasowanie procesów.
4. Zarządzanie plikami: biblioteka niskiego poziomu, blokady, standardowa biblioteka wejścia / wyjścia, komunikacja poprzez pliki
5. Łącza nienazwane i nazwane, selektywny wybór gotowego wejścia
6. Kolejki komunikatów POSIX
7. Komunikacja przez pamięć dzieloną, synchronizacja procesów przez semafory POSIX
8. Komunikacja sieciowa poprzez gniazda – protokół datagramowy UDP
9. Komunikacja sieciowa poprzez gniazda – protokół połączeniowy TCP
10. Zarządzanie wątkami: tworzenie, synchronizacja, muteksy, zmienne warunkowe
11. Zdalne wywoływanie procedur w standardzie SUN RPC

Poszczególne ćwiczenie składa się z krótkiego omówienia tematu, wyszczególnienia najważniejszych wywołań systemowych, elementarnych przykładów ilustrujących zagadnienie i zadań przeznaczonych do zrealizowania w trakcie laboratorium. Do trudniejszych zadań dostarczone są szkielety zawierające podstawową strukturę aplikacji. Omawiane tu zadania programistyczne zostały rozwiązane w środowisku systemu Linux Debian 6 squeeze

Opracowane tutaj ćwiczenia mogą być wykorzystane w laboratoriach z przedmiotów:

- Programowanie współbieżne
- Sieciowe systemy operacyjne
- Programowanie systemowe i współbieżne

2. Podstawy posługiwania się systemem Linux

2.1 Wstęp

Poniżej podane zostały podstawowe informacje umożliwiające posługiwanie się systemem w zakresie uruchamiana prostych programów napisanych w języku C.

2.2 Uzyskiwanie pomocy

Polecenie	Opis
`man polecenie/funkcja`	Uzyskanie informacji o poleceniu / funkcji – narzędzie `man`
`info polecenie/funkcja`	Uzyskanie informacji o poleceniu / funkcji – narzędzie `info`
`whatis słowo_kluczowe`	Uzyskanie krótkiej informacji o temacie danym w postaci słowa kluczowego
`apropos słowo_kluczowe`	Przeszukanie dokumentacji w poszukiwaniu słowa kluczowego
Katalog `/usr/share/doc`	W katalogu tym zawarta jest dokumentacja dla różnych programów, pakietów i modułów
Internet	Witryna http://kernel.org/doc/man-pages Witryny dystrybucji Linuxa Debiana: http://www.debian.org/ Ubuntu : http://ubuntu.pl/

2.2.1 Narzędzie man

Standardowym systemem przeglądania dokumentacji jest narzędzie `man`. Uruchamiamy je wpisując w terminalu polecenie:

```
$man temat
```

gdzie `temat` jest tekstem określającym na temat który chcemy uzyskać informację. Przykładowo gdy chcemy uzyskać informację na temat funkcji `fork` piszemy:

```
$man fork
```

Dokumentacja pogrupowana jest tradycyjnie w działach które podane są w poniższym zestawieniu:

Dział	Zawartość
1	Polecenia
2	Wywołania systemowe
3	Funkcje biblioteczne
4	Pliki specjalne – katalog /dev
5	Formaty plików
6	Gry
7	Definicje, informacje różne
8	Administrowanie systemem
9	Wywołania jądra

Wiedza o działach bywa przydatna gdyż nieraz jedna nazwa występuje w kilku działach. Wtedy man wywołujemy podając jako drugi parametr numer sekcji.

```
$man numer_sekcji temat
```

Na przykład:

```
$man  3 open
```

Przydatnym poleceniem jest opcja –k

```
$man  -k słowo_kluczowe
```

Pozwala przeszukać manual i znaleźć tematy w których występuje dane słowo kluczowe.Np.:

```
$man  -k open
```

Do poruszania się w manualu stosujemy klawisze funkcyjne:

⇑	Linia do góry
⇓	Linia w dół
PgUp	Strona do góry
PgDn	Strona w dół
/ temat	Przeszukiwanie do przodu
? temat	Przeszukiwanie do tyłu

Strona podręcznika składa się z kilku sekcji: nazwa (NAME) , składnia (SYNOPSIS) , konfiguracja (CONFIGURATION), opis (DESCRIPTION), opcje (OPTIONS), kod zakończenia (EXIT STATUS), wartość zwracana (RETURN VALUE), błędy (ERRORS), środowisko (ENVIROMENT), pliki (FILES), wersje (VERSIONS), zgodne z (CONFORMING TO), uwagi, (NOTES), błędy (BUGS), przykład (EXAMPLE), autorzy (AUTHORS), zobacz także (SEE ALSO).
Dokumentacja man dostępna jest w postaci HTML pod adresem : http://www.kernel.org/doc/man-pages

Narzędzia do przeglądania man'a:

- `tkman` – przeglądanie w narzędziu Tkl
- `hman` – przegladanie w trybie HTML

2.2.2 Narzędzie apropos

Narzędzie apropos wykonuje przeszukanie stron podręcznika man w poszukiwaniu podanego jako parametr słowa kluczowego.

```
$apropos słowo_kluczowe
```

Jest ono równoważne:

```
man -k słowo_kluczowe
```

2.2.3 Narzędzie whatis

Narzędzie `whatis` wykonuje przeszukanie stron podręcznika man w poszukiwaniu podanego jako parametr słowa kluczowego. Następnie wyświetlana jest krótka informacja o danym poleceniu / funkcji.

```
$whatis słowo_kluczowe
```

Przykład:

```
$whatis open
open (1)- start a program on a new virtual terminal
open (2)- open and possibly create a file or device
open (3)- posix, open a file
```

Uzyskane strony podręcznika można następnie wyświetlić za pomocą narzędzia `man`. Jest ono równoważne:

```
man -f słowo_kluczowe
```

2.2.4 Narzędzie info

Dodatkowym systemem przeglądania dokumentacji jest narzędzie `info`. Uruchamiamy je wpisując w terminalu polecenie:

```
$info temat
```

Narzędzie `info` jest łatwiejsze w użyciu i zwykle zawiera bardziej aktualną informację.

2.2.5 Klucz --help

Większość poleceń GNU może być uruchomiona z opcją `-- help`. Użycie tej opcji pozwala na wyświetlenie informacji o danym poleceniu.

Dokumentacja systemu Linux dostępna jest w Internecie. Można ją oglądać za pomocą wchodzącej w skład systemu przeglądarki Firefox.

Ważniejsze źródła podane są poniżej:

- Dokumentacja man w postaci HTML: http://www.kernel.org/doc/man-pages
- Materiały Linux Documentation Project: http://tldp.org
- Machtelt Garrels, Introduction to Linux - http://tldp.org/LDP/intro-linux/intro-linux.pdf
- Dokumentacja na temat dystrybucji UBUNTU: - http://help.ubuntu.com
- Brian Ward, Jak działa Linux, Podręcznik administratora

2.3 Użytkownicy

Jakiekolwiek operacje w systemie mogą wykonywać tylko zarejestrowani w nim użtkownicy. Nowych użytkowników dodaje administrator za pomocą polecenia adduser.

```
#adduser nowy_uzytkownik
```

Informacje o tym jaki jest numer użytkownika i jaki jest numer grupy uzyskiwana jest z plików **/etc/passwd** i **/etc/group.** Plik **passwd** składa się z linii. Każda z linii odpowiada jednemu użytkownikowi i składa się z 7 pól oddzielonych dwukropkiem ":".

Nazwa	Nazwa użytkownika
Hasło	Znak x gdy wymagane podanie hasła
UID	Liczbowy identyfikator użytkownika. 0 dla administratora, 1-999 zarezerwowane,
GID	Numer grupy do której należy użytkownik
Informacja	Dodatkowe informacje o użytkowniku
Katalog	Ścieżka określająca katalog domowy – katalog w który będzie bieżący po zalogowaniu
Shell	Ścieżka do interpretera poleceń, zwykle /bin/bash

Tab. 2-1 Zawartość linii pliku **passwd**

Przykład linii pliku passwd:

```
juka:x:1000:1000:Jedrzej Ulasiewicz :/home/juka:/bin/bash
```

Każdy proces utworzony przez użytkownika znakowany jest jego UID co wykorzystywane jest przy kontroli dostępu do zasobów.

2.4 Operowanie plikami i katalogami

W systemie Linux prawie wszystkie zasoby są plikami. Dane i urządzenia są reprezentowane przez abstrakcję plików. Mechanizm plików pozwala na jednolity dostęp do zasobów tak lokalnych jak i zdalnych za pomocą poleceń i programów usługowych wydawanych z okienka terminala. Plik jest obiektem abstrakcyjnym z którego można czytać i do którego można pisać. Plik posiada szereg atrybutów które zostały pokazane w poniższej tabeli.

Typ pliku	Plik regularny, katalog, gniazdko, kolejka FIFO, urządzenie blokowe, urządzenie znakowe, link
Prawa dostępu	Prawo odczytu, zapisu, wykonania określone dla właściciela pliku, grupy do której on należy i innych użytkowników systemu
Wielkość	Wielkość pliku w bajtach
Właściciel pliku	UID właściciela pliku
Grupa do której należy właściciel	GID grupy do której należy właściciel pliku
Czas ostatniej modyfikacji	Czas kiedy nastąpił zapis do pliku
Czas ostatniego dostępu	Czas kiedy nastąpił odczyt lub zapis do pliku
Czas ostatniej modyfikacji statusu	Kiedy zmieniano atrybuty takie jak prawa dostępu, właściciel, itp
Liczba dowiązań	Pod iloma nazwami (ang. *links*) występuję dany plik
Identyfikator urządzenia	Identyfikator urządzenia na którym plik jest pamiętany

Tab. 2-2 Atrybuty pliku

Oprócz zwykłych plików i katalogów w systemie plików widoczne są pliki specjalne. Zaliczamy do nich łącza symboliczne, kolejki FIFO, bloki pamięci, urządzenia blokowe i znakowe. System umożliwia dostęp do plików w trybie odczytu, zapisu lub wykonania. Symboliczne oznaczenia praw dostępu do pliku dane są poniżej:

`r` - Prawo odczytu (*ang. read*)
`w` - Prawo zapisu (*ang. write*)
`x` - Prawo wykonania (*ang. execute*)

Prawa te mogą być zdefiniowane dla właściciela pliku, grupy do której on należy i wszystkich innych użytkowników.

`u` - Właściciela pliku (*ang. user*)
`g` - Grupy (*ang. group*)
`o` - Innych użytkowników (*ang. other*)

Do wyświetlania atrybutów pliku służy polecenie `stat`.

```
$stat hello1_m.s
Plik: hello1_m.s
  rozmiar: 697       bloków: 8          bloki I/O: 4096   zwykły plik
Urządzenie: 801h/2049d inody: 2359930     dowiązań: 1
Dostęp: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Dostęp:      2018-05-29 20:33:09.678262341 +0200
Modyfikacja: 2018-01-09 16:09:58.000000000 +0100
Zmiana:      2018-03-23 22:54:14.820896226 +0100
Utworzenie:  -
```

Przykład 2-1 Polecenie `stat`

2.4.1 Polecenia dotyczące katalogów

Pliki zorganizowane są w katalogi. Katalog ma postać drzewa z wierzchołkiem oznaczonym znakiem /. Położenie określonego pliku w drzewie katalogów określa się za pomocą ścieżki. Rozróżnia się ścieżki absolutne i relatywne. Ścieżka absolutna podaje drogę jaką trzeba przejść od wierzchołka drzewa do danego pliku. Przykład ścieżki absolutnej to `/home/juka/prog/hello.c`. Ścieżka absolutna zaczyna się od znaku /. Ścieżka relatywna zaczyna się od innego znaku niż /. Określa ona położenie pliku względem katalogu bieżącego. Po zarejestrowaniu się użytkownika w systemie katalogiem bieżącym jest jego katalog domowy. Może on być zmieniony na inny za pomocą polecenia `cwd`.

2.4.1.1 Uzyskiwanie nazwy katalogu bieżącego

Nazwę katalogu bieżącego uzyskuje się pisząc polecenie `pwd`. Na przykład:

```
$pwd
/home/juka
```

2.4.1.2 Listowanie zawartości katalogu

Zawartość katalogu uzyskuje się wydając polecenie `ls`. Składnia polecenia jest następująca:

```
ls [-l] [nazwa]
```

Gdzie:
`l` - Listowanie w „długim” formacie, wyświetlane są atrybuty pliku
`nazwa` - Nazwa katalogu lub pliku

Gdy nazwa określa pewien katalog to wyświetlona będzie jego zawartość. Gdy nazwa katalogu zostanie pominięta wyświetlana jest zawartość katalogu bieżącego. Listowane są prawa dostępu, liczba dowiązań, właściciel pliku, grupa, wielkość, data utworzenia oraz nazwa. Wyświetlanie katalogu bieżącego ilustruje Przykład 2-2.

```
$ls -l
-rwxrwxr-x 1 root root  7322 Nov 14  2003 fork3
-rw-rw-rw- 1 root root   886 Mar 18  1994 fork3.c
```

właściciel, typ, inni, grupa, właściciel, grupa, liczba dowiązań, wielkość, data utworzenia, nazwa

Przykład 2-2 Listowanie zawartości katalogu bieżącego.

Typy plików:

oznaczenie	Opis
-	Regularny
d	Katalog (directory)
b	Plik specjalny – urządzenie blokowe
c	Plik specjalny – urządzenie znakowe
p	Plik specjalny – łącze lub plik FIFO
l	Plik specjalny – link symboliczny
s	Plik specjalny - gniazdko

Tabela 2-1 Typy plików w systemie Linux

2.4.1.3 Listowanie drzewa katalogów

```
tree -L poziom katalog
```

Przykład

```
$tree -L 2  /etc
```

2.4.1.4 Zmiana katalogu bieżącego

Katalog bieżący zmienia się na inny za pomocą polecenia `cd`. Składnia polecenia jest następująca: `cd nowy_katalog` .Gdy jako parametr podamy dwie kropki `..` to przejdziemy do katalogu położonego o jeden poziom wyżej. Zmianę katalogu bieżącego ilustruje Przykład 2-3.

```
$pwd
/home/juka
$cd   prog
$pwd /home/juka/prog
```

Przykład 2-3 Zmiana katalogu bieżącego

2.4.1.5 Tworzenie nowego katalogu

Nowy katalog tworzy się poleceniem `mkdir`. Polecenie to ma postać: `mkdir  nazwa_katalogu`. Tworzenie nowego katalogu ilustruje Przykład 2-4.

```
$ls
prog
$mkdir src
$ls
prog src
```

Przykład 2-4 Tworzenie nowego katalogu

2.4.1.6 Kasowanie katalogu

Katalog kasuje się poleceniem `rmdir`. Składnia polecenia `rmdir` jest następująca: `rmdir  nazwa_katalogu`. Aby możliwe było usuniecie katalogu musi on być pusty. Kasowanie katalogu ilustruje Przykład 2-5.

```
$ls
prog src
$rmdir src
$ls
prog
```

Przykład 2-5 Kasowanie katalogu

2.4.2 Polecenia dotyczące plików

Kopiowanie pliku

Pliki kopiuje się za pomocą polecenia `cp`. Składnia polecenia `cp` jest następująca:

```
cp [-ifR] plik_źródłowy plik_docelowy
cp [-ifR] plik_źródłowy katalog_docelowy
```

Gdzie:

i - Żądanie potwierdzenia gdy plik docelowy może być nadpisany.
f - Bezwarunkowe skopiowanie pliku.
R - Gdy plik źródłowy jest katalogiem to będzie skopiowany z podkatalogami.

Kopiowanie plików ilustruje Przykład 2-6.

```
$ls
nowy.txt prog
$ls prog
$
$cp nowy.txt prog
$ls prog
nowy.txt
```

Przykład 2-6 Kopiowanie pliku `nowy.txt` z katalogu bieżącego do katalogu `prog`

Zmiana nazwy pliku

Nazwę pliku zmienia się za pomocą polecenia `mv`. Składnia polecenia `mv` dana jest poniżej:

```
mv [-if] stara_nazwa nowa_nazwa
mv [-if] nazwa_pliku katalog_docelowy
```

Gdzie:
i - Żądanie potwierdzenia gdy plik docelowy może być nadpisany.
f - Bezwarunkowe skopiowanie pliku.

Zmianę nazwy plików ilustruje Przykład 2-7.

```
$ls
stary.txt
$mv stary.txt nowy.txt
$ls
nowy.txt
```

Przykład 2-7 Zmiana nazwy pliku `stary.txt` na `nowy.txt`

2.4.2.1 Kasowanie pliku

Pliki kasuje się za pomocą polecenia `rm`. Składnia polecenia rm jest następująca:

```
rm [-Rfi] nazwa
```

Gdzie:
i - Żądanie potwierdzenia przed usunięciem pliku.
f - Bezwarunkowe kasowanie pliku.
R - Gdy nazwa jest katalogiem to kasowanie zawartości wraz z podkatalogami.

Kasowanie nazwy pliku ilustruje Przykład 2-8.

```
$ls
prog nowy.txt
$rm nowy.txt
$ls
prog
```

Przykład 2-8 Kasowanie pliku `nowy.txt`

2.4.2.2 Listowanie zawartości pliku

Zawartość pliku tekstowego listuje się za pomocą poleceń:

- `more nazwa_pliku`,
- `less nazwa_pliku`, `cat nazwa_pliku`.
- `cat nazwa_pliku`

Można do tego celu użyć też innych narzędzi jak edytor `vi`, edytor `gedit` lub wbudowany edytor programu `Midnight Commander`.

2.4.2.3 Szukanie pliku

- `locate wzorzec`
- `find ścieżka wzorzec`

Przykład:
`$find /etc passwd`

2.5 Program do zarządzania komputerem Midnight Commander

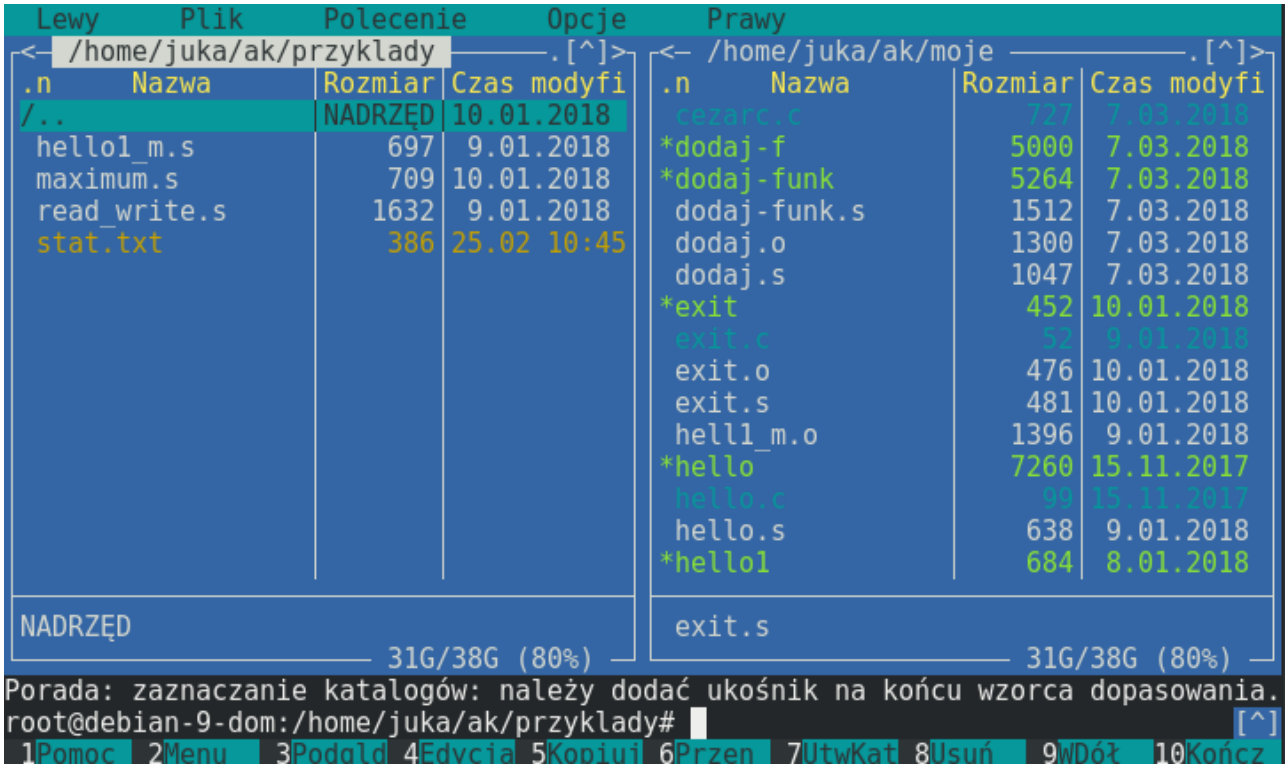
Wiele funkcji dotyczących zarządzania komputerem może być wykonanych za pomocą programu Midnight Commander. Program ten umożliwia realizację nastepujących funkcji:

- Zarządzanie plikami
- Zarządzanie katalogami
- Kopiowanie plików pomiędzy komputerami za pomocą pcrotokołów FTP, SCP, SFTP
- Edycję plików
- Wyświetlanie zawartości plików
- Wyszukiwanie plików
- Dekompresję archiwów

Program można zainstalować za pomocą polecenia:

```
#apt-get install mc
```

a uruchamia się go za pomocą polecenia `mc`. Widok okna programu `mc` pokazany jest ponizej.



Ekran 2-1 Program Midnight Commander do zarządzania komputerem

Dostęp do podstawowych funkcji programu uzyskuje się za pomocą klawiszy funkcyjnych (pokazanych na dolnej listwie) a dostęp do pozostałych za pomocą menu aktywowanego klawiszami Alt+9. Program Midnight Commander dostępny jest w większości dystrybucji Linuksa a także w systemie Windows jako Total Commander lub Windows Commander.

2.6 Operowanie procesami

2.6.1 Abstrakcja procesu

Drugą fundamentalną abstrakcją współczesnych systemów operacyjnych jest abstrakcja procesu. Proces można rozumieć jako wykonujący się program na witrualnym procesorze. Aby proces mógł się wykonać potrzebny jest jego kod wykonywalny i zasoby takie jak procesor, pamięć, pliki i ewentualnie urządzenia wejścia/wyjścia.

2.6.2 Wyświetlanie uruchomionych procesów

2.6.2.1 Polecenie ps

Polecenie ps pozwala uzyskać informacje o uruchomionych procesach. Posiada ono wiele przełączników.

`ps` - wyświetlane są procesy o tym samym EUID co proces konsoli.
`ps - ef` - wyświetlanie wszystkich procesów w długim formacie.
`ps - ef | grep nazwa` - sprawdzanie czy wśród procesów istnieje proces nazwa

2.6.2.2 Polecenie top

Pozwala uzyskać informacje o procesach sortując je według czasu zużycia procesora. Lista odświeżana jest c0 5 sekund. Poniżej podano przykład wywołania polecenia top.

```
$top
 PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
1831 juka      20   0 83340  20m  16m S   37  0.5   1:07.64 gnome-system-
 951 root      20   0 76812  21m  10m S   10  0.5   0:41.70 Xorg
   1 root      20   0  2892 1684 1224 S    0  0.0   0:00.58 init
```

Przykład 2-9 Użycie polecenia top

Symbol	Opis
PID	Identyfikator procesu
USER	Nazwa efektywnego użytkownika
PR	Priorytet procesu
NI	Wartość parametru nice
VIRT	Całkowita wielkość pamięci wirtualnej użytej przez proces
RES	Wielkość pamięci rezydentnej (nie podlegającej wymianie) w kb
SHR	Wielkość obszaru pamięci dzielonej użytej przez proces
S	Stan procesu: R – running, D – uninteruptible running, S – sleeping, T – traced or stoped, Z - zombie
%CPU	Użycie czasu procesora w %
%MEM	Użycie pamięci fizycznej w %
TIME+	Skumulowany czas procesora zużyty od startu procesu
COMMAND	Nazwa procesu

Tab. 2-3 Znaczenie parametrów polecenia top

2.6.2.3 Polecenie `uptime`

Polecenie wyświetla czas bieżący, czas pracy systemu, liczbę użytkowników i obiążenie systemu w ostatnich 1, 5 i 15 minutach.

```
$uptime
18:26:47 up  1:14,  4 users,  load average: 0.32, 0.25, 0.19
```

Przykład 2-10 Użycie polecenia top

2.6.2.4 Polecenie `pstree`

Polecenie wyświetla drzewo procesów. Można obserwować zależność procesów typu macierzysty – potomny. Pokazuje to poniższy ekran.

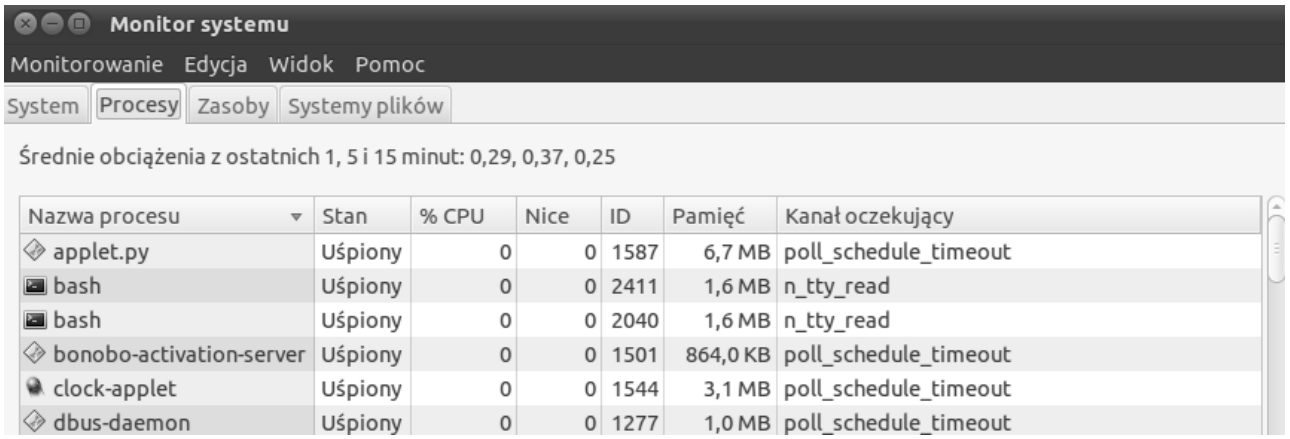
```
root@debian-9-dom:/home/juka/ak/przyklady# pstree
systemd─┬─ModemManager─┬─{gdbus}
        │               └─{gmain}
        ├─NetworkManager─┬─dhclient
        │                ├─{gdbus}
        │                └─{gmain}
        ├─accounts-daemon─┬─{gdbus}
        │                 └─{gmain}
```

Ekran 2-2 Drzewo procesów – polecenie `pstree`

2.6.2.5 Monitor systemu

W systemie Ubuntu dostępny jest monitor systemu który wyświetla informacje dotyczące aktywnych procesów. Uruchomienie następuje poprzez:

- Wybór opcji System / Administracja / Monitor systemu
- Wpisanie w terminalu polecenia: `gnome-system-monitor`



Przykład 2-11 Użycie polecenia monitora systemu

2.6.3 Kasowanie procesów

Procesy kasuje się poleceniem kill. Składnia polecenia jest nastepująca:

```
kill [-signal | -s signal] pid
```

Gdzie:
`signal` – numer lub nazwa sygnału
`pid` - pid procesu który należy skasować

Nazwa sygnału	Numer	Opis
SIGTERM	15	Zakończenie procesu w uporządkowany sposób
SIGINT	2	Przerwanie procesu, sygnał ten może być zignorowany
SIGKILL	9	Przerwanie procesu, sygnał ten nie może być zignorowany
SIGHUP	1	Używane w odniesieniu do demonów, powoduje powtórne wczytanie pliku konfiguracyjnego.

Tab. 2-4 Częściej używane sygnały

2.7 Zadania

2.7.1 Uzyskiwanie informacji o stanie systemu

Zobacz jakie procesy i wątki wykonywane są aktualnie w systemie.

2.7.2 Uzyskiwanie informacji o obciążeniu systemu

Używając polecenia `top` zbadaj który z procesów najbardziej obciąża procesor.

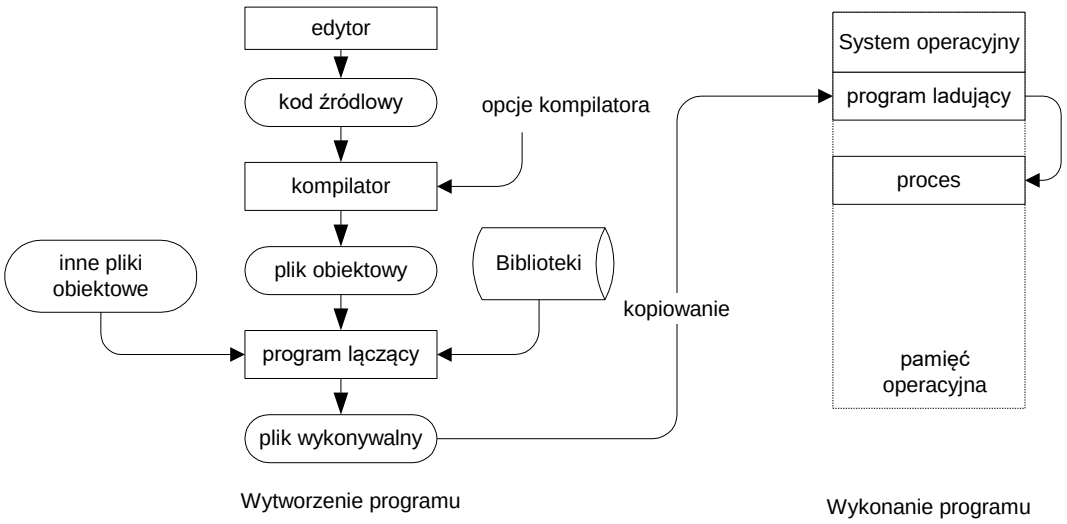
2.7.3 Archiwizacja i kopiowania plików

W systemie pomocy znajdź opis archiwizatora `tar`. Używając programu `tar` spakuj wszystkie pliki zawarte w katalogu bieżącym do pojedynczego archiwum o nazwie `programy.tar` (polecenie: `tar -cvf programy.tar *`). Następnie zamontuj dyskietkę typu MSDOS i skopiuj archiwum na dyskietkę. Dalej utwórz katalog `nowy` i skopiuj do niego archiwum z dyskietki i rozpakuj do postaci pojedynczych plików (polecenie: `tar -xvf programy.tar`).

3. Kompilacja i uruchamianie programów

3.1 Jak kod źródłowy przekształca się w proces

Kod aplikacji tworzony jest zazwyczaj w języku wysokiego poziomu, my posługiwać się będziemy językiem C. W języku wysokiego poziomu tworzy się tak zwany kod źródłowy który po zapisaniu będzie plikiem z programem źródłowym. Plik źródłowy przetwarzany jest następnie przez kompilator do tak zwanego programu wykonywalnego. Jeżeli procesor (i ewentualnie system operacyjny) na którym kompilowany jest program jest inny niż ten na którym jest wykonywany to kompilator nazywa się kompilatorem skrośnym (ang. *cross compiler*). Następnie program wykonywalny przekształcany jest w wykonujący się proces, co wykonywane jest przez program ładujący systemu operacyjnego (ang. *loader*). Proces tworzenia i wykonywania programu pokazany jest na poniższym Rys. 3-1. Program wykonywalny może się wykonać na tej samej maszynie na której został utworzony, lub tak jak się to dzieje w systemach wbudowanych, jest on przesyłany do systemu docelowego i tam dopiero wykonany.



Rys. 3-1 Przebieg procesu wytworzenia i wykonania programu

3.1.1.1 Przetworzenie kodu źródłowego w wykonywany proces odbywa się w kilku etapach. Najważniejsze z nich to kompilacja, łączenie i ładowanie programu.

3.1.2 Kompilacja

Podstawowym narzędziem przekształcającym kod źródłowy zrozumiały dla procesora w kod wykonywalny jest kompilator. Celem kompilacji jest transformacja kodu źródłowego będącego zapisem algorytmu w języku wysokiego poziomu (który nie może być wykonany przez procesor) na kod maszynowy danego procesora. Kompilacja przebiega w kilku etapach i prowadzi ona do wytworzenia tak zwanego pliku obiektowego. Plik obiektowy zawiera kod maszynowy właściwy dla procesora na którym kod będzie wykonywany i informacje dodatkowe. Typowy plik obiektowy składa się z takich części jak: nagłówek, kod maszynowy, dane, tablica symboli, informacje o relokacji, informacje dla programu uruchomieniowego (ang. *debugger*). Na etapie kompilacji nie sposób określić pod jaki adres w pamięci należy załadować utworzony program, gdyż kompilator nie posiada informacji o stanie pamięci procesora w chwili wykonania programu. Stąd pliki obiektowe i wykonywalne zawierają tak zwaną tablicę relokacji (ang. *relocation table*) . Składa się ona z pozycji, z których każda zawiera wskaźnik do adresu w kodzie obiektowym, który musi być zmodyfikowany w procesie ładowania programu do pamięci operacyjnej. W systemie Linux plik obiektowy jak i wykonywalny tworzony jest w tak zwanym formacie ELF (ang. *Executable and Linkable Format*). Informacje o plikach w formacie ELF uzyskać można za pomocą narzędzi Linuksowych takich jak `readelf i objdump`, . Istnieje wiele kompilatorów języka C ale my używać będziemy standardowego narzędzia kompilacyjnego systemu Linux czyli kompilatora `gcc` .

3.1.3 Łączenie

Plik obiektowy zawiera tłumaczenie kodu źródłowego na instrukcje kodu maszynowego i dane na których te instrukcje operują ale nie jest jeszcze kompletnym programem gdyż nie zawiera bibliotek i być może innych segmentów programu. Kompletny program wykonywalny powstanie na etapie łączenia. Operację łączenia wykonuje program nazywany konsolidatorem lub linkerem (ang. *linker*). Konsolidator dołącza programu głównego inne pliki obiektowe i biblioteki w wyniku czego powstaje program wykonywalny. Jest on także w formacie ELF. W systemie Linux rolę linkera pełni program `ld`.

3.1.4 Ładowanie programu

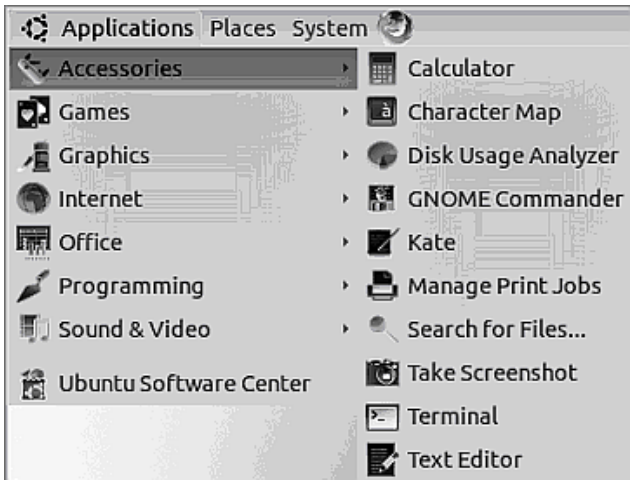
Plik wykonywalny (nazywany też plikiem binarnym) kopiowany jest następnie w miejsce przeznaczenia. Może to być inny folder w tym samym komputerze, inny komputer lub też system wbudowany. Kolejną czynnością która musi być wykonana jest utworzenie procesu na podstawie pliku wykonywalnego. Czynność tę wykonuje program ładujący (ang. *loader*). Funkcje programu ładującego to:

- Weryfikacja pozwoleń, wymagań na zasoby
- Utworzenie deskryptora procesu
- Skopiowanie segmentów programu do pamięci operacyjnej
- Skopiowanie argumentów linii poleceń na stos
- Inicjalizacja rejestrów procesora
- Umieszczenie deskryprora nowego procesu w kolejce procesów gotowych

Po wykonaniu powyższych czynności program zostaje przekształcony w proces i przystępuje do wykonywania swojej funkcji. W dalszej części tego rozdziału przedstawimy sposoby kompilacji programów i opiszemy stosowane do tego celu narzędzia.

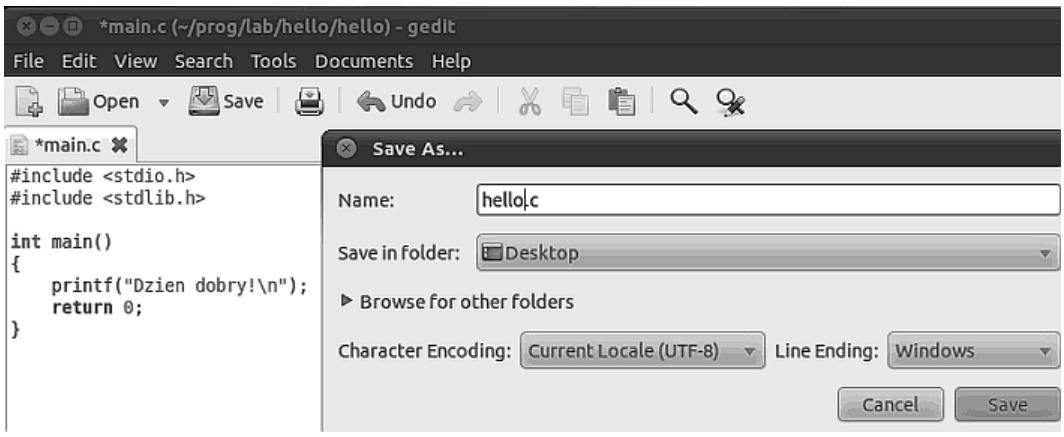
3.2 Metoda elementarna – użycie edytor gedit i kompilatora gcc

Najprostszą metodą tworzenia i uruchamiania programów w systemie Linux jest użycie systemowego edytora `gedit` i kompilatora `gcc` uruchamianego w trybie wsadowym. Aby uruchomić edytor `gedit` należy wybrać opcję Text Editor z głównego menu Applications / Accessories jak pokazuje poniższy przykład.



Przykład 3-1 Uruchomienie edytora `gedit`

Następnie gdy edytor się zgłosi wybieramy opcję File / New i otwiera się okno edycyjne. W oknie edycyjnym wpisujemy tekst programu. Może to być najprostszy program wyprowadzający na konsolę powitanie tak jak w przykładzie poniżej.



Przykład 3-2 Edycja programu hello.c

Po wprowadzeniu tekstu wybieramy opcję edytora File / Save As. Pojawi się okienko o nazwie "Save As ..." w którym w okienku Name wpisujemy nazwę pliku (hello.c) i ewentualnie wybieramy folder roboczy wybierając go w okienku Save i folder tak jak pokazuje to przykład Przykład 3-2. Gdy plik z programem jest już zapamiętany wtedy uruchamiamy terminal wybierając z głównego menu opcję Accessories / Terminal (patrz Przykład 3-1). Gdy terminal się zgłosi zmieniamy folder bieżący na ten folder w którym zapisaliśmy nasz program. W naszym przykładzie będzie to folder Pulpit wpisujemy więc polecenie:

```
$cd Pulpit
```

Następnie sprawdzamy czy w folderze znajduje się nasz plik źródłowy wpisując polecenie:

```
$ls
```

Gdy zostanie wyświetlona zawartość folderu Pulpit i zawierał on będzie nasz plik `hello.c` z programem źródłowym kompilujemy program wpisując polecenie:

```
$gcc hello.c -o hello
```

`gcc` jest tu nazwą kompilatora, `hello.c` nazwą pliku źródłowego z programem, opcja `-o hello` specyfikuje nazwę pliku wykonywalnego. Gdy kompilacja wykona się można sprawdzić obecność pliku wykonywalnego który powinien być utworzony przez kompilator wykonując ponownie polecenie `ls`. Gdy zobaczymy że w folderze Pulpit pojawił się plik `hello` możemy go uruchomić wpisując polecenie:

```
$./hello
```

Otrzymamy wynik jak pokazuje przykład poniżej.

```
juka@debian6:~$ gcc hello.c -o hello
juka@debian6:~$ ./hello
Dzien dobry !
```

Przykład 3-1 Kompilacja i uruchomienie programu `hello.c`

W systemie Linux istnieje pożyteczne polecenie o nazwie `file` które pozwala na zbadanie rodzaju pliku który z jakichś względów nas interesuje. Po wpisaniu polecenia `file` podajemy nazwę testowanego pliku. Zastosowanie programu `file` do otrzymanego wcześniej programu `hello` daje wynik jak poniżej.

```
$file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked (uses shared libs), for GNU/Linux 2.6.18, not stripped
```

Przykład 3-2 Ilustracja działania programu `file` dla pliku wykonywalnego

Widzimy że program `hello` jest plikiem wykonywalnym w formacie ELF dla procesora Intel 30386. Program źródłowy można skompilować do postaci pliku obiektowego który następnie będzie konsolidowany. Robi się to korzystając z opcji `-c` kompilatora `gcc`. Aby skompilować program `hello.c` do pliku obiektowego stosujemy polecenie:

```
gcc hello.c -c -o hello.o
```

Plik `hello.o` jest plikiem obiektowym. Gdy zastosujemy do tego pliku polecenie `file` otrzymamy:

```
$file hello
hello.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

Przykład 3-3 Ilustracja działania programu `file` dla pliku obiektowego

O pliku obiektowym można uzyskać różne informacje posługując się programem `objdump` . Gdy wykonamy go z opcją `-x` otrzymamy wiele informacji o zawartości pliku obiektowego `hello.o` co pokazuje poniższy przykład.

```
$objdump -x hello.o
hello.o:     file format elf32-i386
architecture: i386, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text           0000001c  00000000  00000000  00000034  2**2
  1 .data           00000000  00000000  00000000  00000050  2**2
  2 .bss            00000000  00000000  00000000  00000050  2**2
  3 .rodata         0000000d  00000000  00000000  00000050  2**0
  4 .comment        0000001d  00000000  00000000  0000005d  2**0
  5 .note.GNU-stack 00000000  00000000  00000000  0000007a  2**0
…
SYMBOL TABLE:
…
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
…
```

Przykład 3-4 Ilustracja działania programu objdump dla pliku obiektowego

Z powyższego przykładu widać jakie informacje zawiera plik obiektowy. Są to nagłówki, segmenty, tablica symboli i dane do relokacji. Ważniejsze segmenty to:

- `.text` – segment kodu, zawiera instrukcje
- `.data` – segment danych zainicjowanych, dane którym nadano wartości początkowe
- `.bss` – segment danych nie zainicjowanych, dane którym nie nadano wartości początkowych
- `.rodata` – segment danych zawierających stałe (tylko do odczytu)
- `.comment` – komentarze
- `.note.GNU-stack` – informacja że potrzebny będzie stos

Za pomocą polecenia `objdump` z opcją `-d` można uzyskać kod assemblera zawarty w segmencie kodu co pokazuje poniższy przykład.

```
objdump -d hello.o
00000000 <main>:
   0: 55                      push   %ebp
   1: 89 e5                   mov    %esp,%ebp
   3: 83 e4 f0                and    $0xfffffff0,%esp
   6: 83 ec 10                sub    $0x10,%esp
   9: c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  10: e8 fc ff ff ff          call   11 <main+0x11>
  15: b8 00 00 00 00          mov    $0x0,%eax
  1a: c9                      leave
  1b: c3                      ret
```

Przykład 3-5 Ilustracja działania programu `objdump` dla pliku obiektowego – disassemlacja segmentu kodu

W kolejnym etapie możemy dokonać konsolidacji pliku `hello.o` pisząc polecenie jak niżej i otrzymamy nowy plik wykonywalny `hello`.

```
gcc hello.o -o hello
```

W tym przykładzie Budowanie programu wykonywalnego przebiegało w dwóch etapach. W pierwszym etapie utworzyliśmy plik obiektowy a drugim dokonaliśmy jego konsolidacji otrzymując plik wykonywalny. Wiele informacji o pliku wykonywalnym można uzyskać za pomocą programów `file`,`size`. Polecenie `size` pozwala na uzyskanie informacji o rozmiarach programu co pokazuje poniższy przykład.

```
$size hello
text     data         bss         dec         hex     filename
916       264           8        1188         4a4     hello
```

Przykład 3-6 Ilustracja działania programu `size` dla pliku programu `hello`

Chodzi przy tym nie o rozmiar pliku wykonywalnego zapisanego na dysku ale o rozmiar programu umieszczonego w pamięci operacyjnej. Część `text` podaje wielkość segmentu kodu, `data` podaje wielkość segmentu danych zainicjowanych, `bss` wielkość segmentu danych nie zainicjowanych a `dec` całkowitą wielkość pamięci zajmowaną przez program `hello`. Zestawienie narzędzi przydatnych w analizie programów pokazuje poniższa tabela.

Program	Opis	Przykład
`objdump`	Podaje zawartość plików obiektowych	`objdump -x hello.o`
`readelf`	Odczyt pliku w formacie ELF	`readelf -a hello`
`size`	Podaje ile pamięci zajmuje proces	`size hello`
`file`	Podaje typ pliku	`file hello`

Tabela 3-1 Zestawienie narzędzi do analizy plików programowych

3.3 Kompilator gcc

3.3.1 Wiele plików źródłowych

W licznych przypadkach program źródłowy składa się z wielu plików. Podział dużego pliku z kodem źródłowym na mniejsze jest wygodny gdyż umożliwia modułowe podejście do programowania. Pliki składowe mogą być kompilowane oddzielnie a potem łączone. Załóżmy że kod źródłowy składa się z plików: `pierwszy.c` i `wspolny.c`.

```
#include "wspolny.h"
int main(void)
{
  pisz("program 1");
  return 0;
}
```

Kod 3-1 Plik programu `pierwszy.c`

```
#include <stdio.h>
void pisz(char * tekst) {
  printf("%s\n",tekst);
}
```

Kod 3-2 Plik biblioteki `wspolny.c`

```
void pisz(char * tekst);
```

Kod 3-3 Plik nagłówkowy `wspolny.h`

Kompilujemy je oddzielnie poprzez polecenia:

```
$gcc -c wspolny.c
$gcc -c pierwszy.c
```

W wyniku kompilacji utworzone zostaną dwa pliki obiektowe: `pierwszy.o` i `wspolny.o`. Plik obiektowy jest plikiem wykonywalnym ale jeszcze niekompletnym. Do zbudowania pliku wykonywalnego potrzebny jest jeszcze proces konsolidacji która to wykonywana jest przez program nazywany linkerem (W Linuksie nazywa się `ld`). Nie jest zwykle wywoływany wprost lecz wywołuje go program `gcc`. Nawiązując do powyższego programu plik wykonywalny o nazwie `main` tworzy się jak poniżej.

```
$gcc -o pierwszy pierwszy.o wspolny.o
```

3.3.2 Pliki nagłówkowe

Pliki nagłówkowe są także plikami źródłowymi zawierającymi deklaracje typów i funkcji. Potrzebne są po to aby kompilator mógł sprawdzić prawidłowość użycia funkcji i zmiennych które zaimplementowane są w innych plikach.

Wiele problemów z kompilacją ma swoje źródło w tym że kompilator nie wie gdzie położone są pliki nagłówkowe. Przykładem pliku nagłówkowego jest plik `stdio.h` który jest włączony do kodu źródłowego w linii:

```
#include <stdio.h>
```

Linuks przechowuje pliki nagłówkowe w katalogu `/usr/include`. Gdyby plik nagłówkowy o nazwie `pierwszy.h` był w innym katalogu powiedzmy `/home/juka/include` należałoby poinformować o tym kompilator jak poniżej.

```
$gcc -c -I/home/juka/include pierwszy.c
```

Jeżeli plik nagłówkowy jest w tym samym katalogu co plik źródłowy to umieszczamy jego nazwę w podwójnym cudzysłowie.

```
#include "pierwszy.h"
```

3.3.3 Preprocesor języka C

Zanim właściwy kompilator przystąpi do pracy, plik źródłowy przetwarzany jest wstępnie przez program zwany preprocesorem. Informacje dla preprocesora, nazywane dyrektywami, poprzedzane są znakiem krzyżyka #. Preprocesor można uruchomić bezpośrednio, nazywa się `cpp`. Na przykład:

```
$cpp hello.c
```

Można też wywołać kompilator `gcc` z opcją –E, na przykład:

```
$gcc -E hello.c
```

Preprocesor wyróżnia trzy rodzaje dyrektyw:

- Pliki nagłówkowe
- Makrodefinicje
- Dyrektywy warunkowe

Dyrektywa `#include plik` nakazuje preprocesorowi włączyć do kodu cały `plik` wymieniony po `#include`.

<u>Makrodefinicja</u> nakazuje zastąpienie jednego łańcucha znaków innym łańcuchem. Na przykład

```
#define SUMA(a,b)  (a + b)
```

W tym przypadku napotkane w programie napisy `SUMA(2,3)` zostaną zastąpione przez napis `(2 + 3)`. Makrodefinicje mogą być podane także w linii poleceń kompilatora poprzez użycie opcji `-D`, np.:

```
$gcc main.c -DDEBUG=1
```

Działanie tej opcji będzie takie same jak pojawienie się w kodzie linii:

```
#define DEBUG 1
```

Dyrektywy warunkowe pozwalają na włączenie/wyłączenie pewnych fragmentów kodu za pomocą dyrektyw: `#ifdef, #if, #endif.`

`#ifdef MAKRO`
`  tekst`
`#endif`
	Gdy MAKRO zostało wcześniej zdefiniowane, to tekst zostanie włączony

`#if warunek`
`  tekst`
`#endif`
	Gdy warunek ma wartość większą od zera, to tekst zostanie włączony

Wykorzystanie kompilacji warunkowej i makrodefinicji podawanej jako opcji kompilatora pokazane zostanie poniżej. Fragment kodu pomiędzy liniami `#ifdef DEBUG` a `#endif` zostanie włączony tylko wtedy, gdy zdefiniowane będzie makro `DEBUG`. Można to osiągnąć albo poprzez odkomentowanie trzeciej linii poniższego przykładu lub poprzez użycie opcji –D kompilatora.

```
#include <stdio.h>
#include <stdlib.h>
#define DEBUG 1

int main(int argc, char *argv[])
{
    #ifdef DEBUG
     printf("argc: %d argv[0]: %s\n",argc,argv[0]);
    #endif
    printf("Dzien dobry\n");
    return 0;
}
```

Przykład 3-7 Kompilacja warunkowa program `hello2.c`

```
$gcc hello2.c -o hello2 -DDEBUG=1
```

Wykonanie programu skompilowanego jak powyżej da rezultat:

```
$./hello2
argc: 1 argv[0]: ./hello2
Dzien dobry
```

Gdy w linii kompilacji pominiemy opcję –D wynik działania programu nie będzie zawierał informacji o parametrach funkcji `main`.

```
$gcc hello2.c -o hello2
$./hello2
Dzien dobry
```

Jak widzimy kompilacja warunkowa i możliwość przekazywania makrodefinicji podczas kompilacji jest wygodnym mechanizmem różnicowania trybu uruchamiania i docelowego.

3.4 Biblioteki

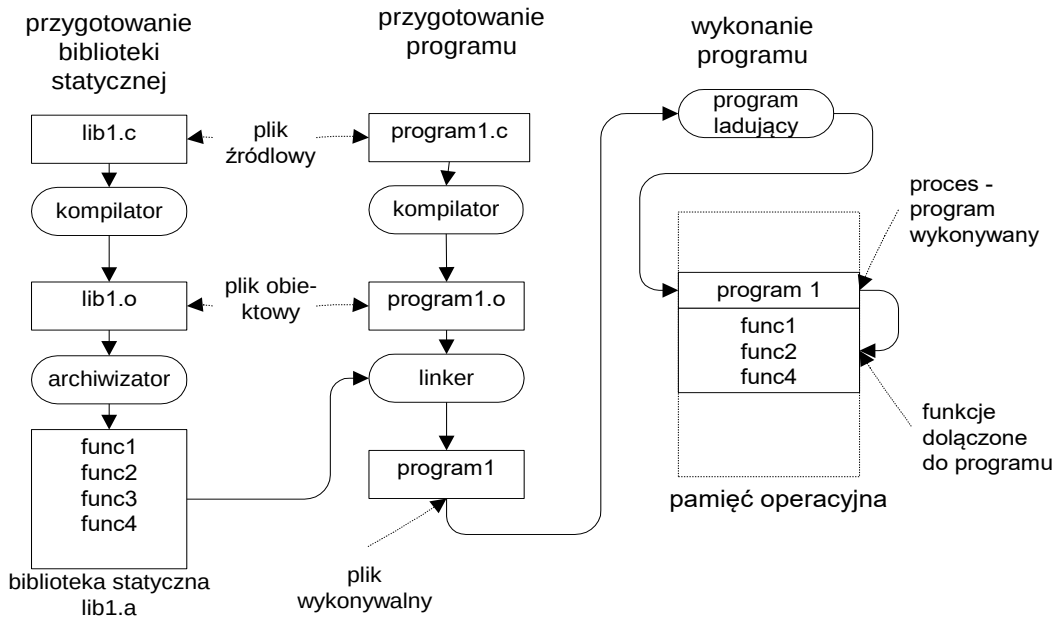
Programy wykonują wiele typowych czynności których samodzielne programowanie byłoby niecelowe. Czynności te, typowo realizowane są przez napisane wcześniej funkcje. Funkcje te zgrupowane są w bibliotekach, które w dużej części są nieodłącznym elementem systemu. Wyróżniamy dwa rodzaje bibliotek:

- Biblioteki statyczne
- Biblioteki współdzielone

Zostaną one omówione dalej.

3.4.1 Biblioteki statyczne

Biblioteka statyczna składa się z funkcji i danych na których funkcje te operują. Bibliotekę tworzy się kompilując zestawy funkcji, zawarte w jednym lub wielu plikach źródłowych, do postaci plików obiektowych. Pliki te następnie łączone są w archiwum za pomocą polecenia `ar`. Gdy program łączący buduje plik wykonywalny i gdy jest wskazanie by korzystał z bibliotek statycznych, to przeszukuje on bibliotekę i dołącza do pliku wykonywalnego moduły zawierające potrzebne funkcje. Schemat tworzenia biblioteki statycznej pokazuje Rys. 3-2.



Rys. 3-2 Tworzenie i wykorzystanie biblioteki statycznej

Powyższe rozważania zilustrować można przykładem w którym program `pierwszy.c` i `drugi.c` korzystają z funkcji `pisz(char * text)` która umieszczona jest w pliku `wspolny.c`.

```
#include "wspolny.h"
int main(void)
{
  pisz("program 1");
  return 0;
}
```

Kod 3-4 Plik programu `pierwszy.c`

```
#include "wspolny.h"
int main(void)
{
  pisz("program 2");
  return 0;
}
```

Kod 3-5 Plik programu `drugi.c`

```
#include <stdio.h>
void pisz(char * tekst) {
  printf("%s\n",tekst);
}
```

Kod 3-6 Plik biblioteki `wspolny.c`

```
void pisz(char * tekst);
```

Kod 3-7 Plik nagłówkowy `wspolny.h`

Standardowy sposób utworzenia pliku wykonywalnego `pierwszy` podany jest poniżej.

```
gcc -c pierwszy.c -o pierwszy.o
gcc -c wspolny.c -o wspolny
gcc -o pierwszy pierwszy.o wspolny.o
```

Pokażemy jak utworzyć bibliotekę statyczną zawierającą funkcję `pisz` która to funkcja wykorzystywana jest w programie `pierwszy.c i drugi.c`. Utwórzmy katalog o nazwie `biblioteka` i skopiujmy tam pliki `pierwszy.c`, `drugi.c wspolny.c, wspolny.h`. Bibliotekę tworzymy w dwóch krokach:

- Krok1 - utworzenie pliku obiektowego `wspolny.o` zawierającego funkcję `pisz` za pomocą kompilatora `gcc`. Użyta będzie opcja `-c` nakazująca tylko kompilację bez tworzenia pliku wynikowego
- Krok 2 - utworzenie archiwum za pomocą programu archiwizatora `ar`

Aby utworzyć bibliotekę o nazwie `libwspolny.a` należy wykonać następujące polecenia:

```
$gcc -c wspolny.c -o wspolny.o
$ar rcsv libwspolny.a wspolny.o
```

Polecenie `ls` pokazuje że powstał plik biblioteki statycznej o nazwie `libwspolny.a`. Ważne jest aby nazwa biblioteki zaczynała się od liter `lib`. Można sprawdzić zawartość tej biblioteki za pomocą polecenia: `nm nazwa_biblioteki`.

```
$nm libwspolny.a
libwspolny.o:
00000000 T pisz
         U puts
```

Widzimy że biblioteka zawiera funkcję `pisz`. Następnie aby uzyskać plik wykonywalny o nazwie drugi wykonujemy łączenie programu informując że dołączamy bibliotekę statyczną `libwspolny.a`

```
$gcc pierwszy.c -o pierwszy -L. -lwspolny
```

W powyższym poleceniu opcja –L. informuje kompilator że należy szukać biblioteki w katalogu bieżącym. Należy zwrócić uwagę że po literze `-l` nie wpisujemy całej nazwy biblioteki ale część bez przedrostka `lib`. Brakujący przedrostek `lib` kompilator doda sam. Teraz wystarczy się przekonać że program działa prawidłowo pisząc polecenie:

```
$./drugi
Program drugi
```

Częstym źródłem problemów jest to że program łączący nie może znaleźć właściwych bibliotek. Standardowo biblioteki umieszczone są w katalogu `/lib` i `/usr/lib`. Gdy biblioteki umieszczone są w innym miejscu należy poinformować o tym program łączący. Gdyby biblioteka `libwspolny.a` umieszczona była w katalogu `/home/juka/lib` należało by użyć następującej linii kompilacyjnej:

```
$gcc pierwszy.c -o pierwszy -L/home/juka/lib -lwspolny
```

3.4.2 Biblioteki współdzielone

3.4.2.1 Zasada działania

Zastosowanie bibliotek statycznych ma tak skutek że:

- Kod występujących w nich funkcji występuje wielokrotnie w różnych programach co powoduje niepotrzebne straty przestrzeni systemu plików
- W przypadku znalezienia w bibliotece błędu, należy przekompilować wszystkie programy

Powstał więc pomysł by biblioteki umieścić w ogólnie znanym miejscu, a wtedy wiele programów mogłoby z nich korzystać. Jest to koncepcja biblioteki współdzielonej.

Aby posługiwać się bibliotekami współdzielonymi należy posiadać informację:

- Jak sprawdzić jakich bibliotek potrzebuje dany program
- Jak program szuka bibliotek współdzielonych
- Jak łączyć program z bibliotekami współdzielonymi
- Jak tworzyć biblioteki współdzielone

3.4.2.2 Wykorzystanie bibliotek współdzielonych

Aby program mógł odnaleźć bibliotekę współdzieloną musi być ona umieszczona w dobrze zdefiniowanym miejscu. Zgodnie z zaleceniem FHS (ang. *Filesystem Hierarchy Standard*) biblioteki współdzielone powinny być umieszczone w katalogu `/usr/lib` i `/usr/local/lib`. Biblioteki współdzielone mają rozszerzenie `so`, przykładową biblioteką jest plik `/lib/ldlinux.so.2`. Za pomocą programu `ldd` można uzyskać informacje jakich bibliotek współdzielonych używa dany program.

```
$ldd hello
      linux-gate.so.1 =>  (0xb7721000)
      libc.so.6 => /lib/i686/cmov/libc.so.6 (0xb75c5000)
      /lib/ld-linux.so.2 (0xb7722000)
```

Przykład 3-8 Uzyskanie informacji o bibliotekach dzielonych za pomocą polecenia `ldd`

W celu zachowania elastyczności, programy zwykle nie zawierają informacji o bezwzględnym położeniu bibliotek współdzielonych a tylko ich nazwy. Odnajdowaniem bibliotek współdzielonych zajmuje się konsolidator dynamiczny (ang. *runtime dynamic linker*). W systemie Linux jest on widoczny w powyższym przykładzie jako `ld-linux.so.2.`

Stosunkowo częstą przyczyną błędów jest niemożność odnalezienia właściwej biblioteki współdzielonej. Konsolidator dynamiczny poszukuje bibliotek w następujący sposób:

1. Sprawdza czy istnieje zmienna środowiska LD_LIBRARY_PATH. Gdy tak poszukuje biblioteki w wskazanej przez tę zmienna ścieżce.
2. Sprawdza czy lokalizacja biblioteki umieszczona jest w systemowym schowku (ang. *cache*) `/etc/ld.so.cache`
3. Odczytuje plik `/etc/ld.so.conf` gdzie są nazwy plików z bibliotekami dzielonymi. (dla przykładu zawiera on treść: `include /etc/ld.so.conf.d/*.conf`). W przykładzie są to pliki: `i486-linux-gnu.conf, libc.conf, vmware-tools-libraries.conf.`
4. Pliki zawierają znane systemowi położenia bibliotek współdzielonych, np. plik `libc.conf` zawiera wpis `/usr/local/lib`

Gdyby zostały wprowadzone zmiany w powyższych plikach konfiguracyjnych, należy zaktualizować schowek przez polecenie:

```
#ldconfig -v
```

Gdy tworzymy program który ma używać niestandardowej biblioteki współdzielonej należy poinformować o tym program łączący. Powiedzmy że w programie `pierwszy` mamy użyć biblioteki: `/home/juka/lib/libwspolny.so`. Jak utworzyć tę bibliotekę pokazane zostanie dalej. Kompilacja powinna przebiegać następująco:

```
gcc -o pierwszy pierwszy.c -Wl,-rpath=/home/juka/lib -L/home/juka/lib -lwspolny
```

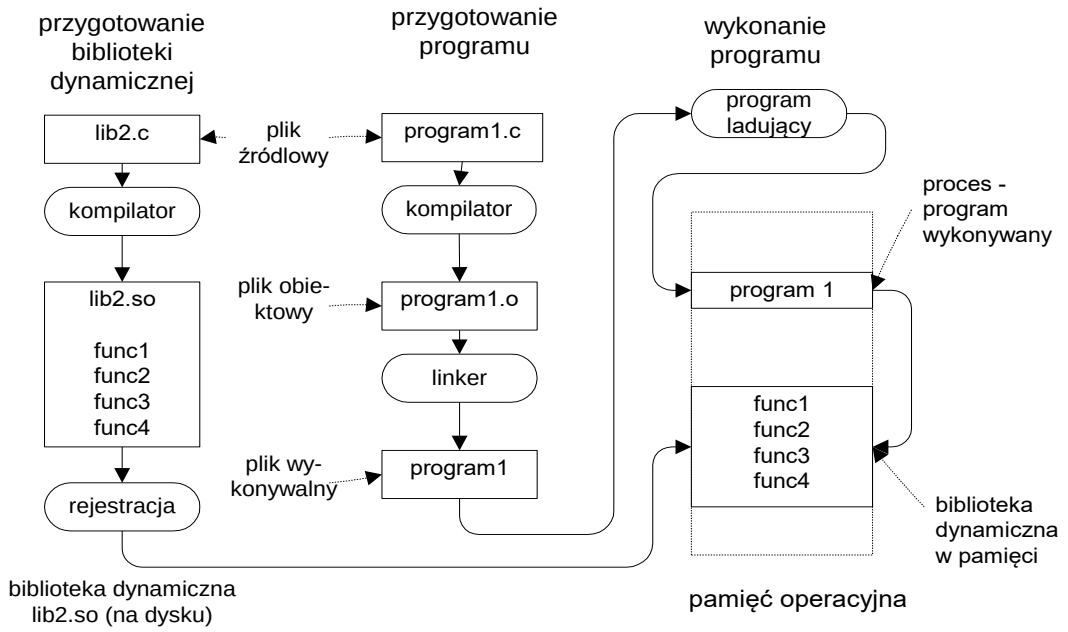
Można teraz za pomocą polecenia `ldd` sprawdzić z jakich bibliotek korzysta program `pierwszy`.

```
$ldd pierwszy
linux-gate.so.1 =>  (0xb76ea000)
libwspolny.so => /home/juka/lib/libwspolny.so (0xb76e6000)
libc.so.6 => /lib/i686/cmov/libc.so.6 (0xb758c000)
/lib/ld-linux.so.2 (0xb76eb000)
```

Przykład 3-9 Testowanie bibliotek używanych przez program `pierwszy` za pomocą narzędzia `ldd`

3.4.2.3 Tworzenie biblioteki współdzielonej

Aby utworzyć bibliotekę współdzieloną trzeba napisać kod źródłowy funkcji wchodzących w skład biblioteki i skompilować do postaci biblioteki współdzielonej używając do tego odpowiednich opcji kompilatora (`-shared -fpic)`. Następnie należy zainstalować bibliotekę czyli poinformować system o nazwie biblioteki i o lokalizacji pliku ją zawierającego. W procesie konsolidacji programu wykonywalnego biblioteki nie są dołączane do pliku wynikowego, ale są tylko rejestrowane to znaczy program łączący wpisuje do pliku wykonywalnego informację o używanej bibliotece, tak aby było możliwe, załadowanie biblioteki w czasie wykonania programu. Proces przygotowania biblioteki współdzielonej, przygotowania programu ja zawierającego i wykonania takiego programu pokazuje poniższy rysunek.



Rys. 3-3 Tworzenie i wykorzystanie biblioteki współdzielonej

Przytoczone rozważania zilustrować można przykładem takim jak poprzednio. Pokażemy zatem jak utworzyć bibliotekę współdzieloną. Plik `wspolny.c` zawiera kod funkcji `pisz(char * tekst)` wykorzystywanej w programach `pierwszy` i `drugi`. Aby zbudować bibliotekę współdzieloną zawierającą tę funkcję wykonajmy następujące polecenie:

```
$gcc -shared -fpic -o libwspolny.so wspolny.c
```

Powyższe polecenie zleca utworzenie biblioteki współdzielonej o nazwie `libwspolny.so` z pliku `wspolny.c`. Dalej kopiujemy tę bibliotekę do katalogu `/home/juka/lib`. Następnie tworzymy plik wykonywalny dla programu `pierwszy` pisząc polecenie:

```
$gcc -o pierwszy pierwszy.c -Wl,-rpath=/home/juka/lib     -L/home/juka/lib -
lwspolny
```

Polecenie nakazuje utworzenie pliku wykonywalnego `pierwszy` z pliku `pierwszy.c` i użycie biblioteki współdzielonej `libwspolny.so` umieszczonej w katalogu `/home/juka/lib`. Za pomocą polecenia `ldd` możemy sprawdzić jakich bibliotek dynamicznych używa program pierwszy.

```
$ldd pierwszy
linux-gate.so.1 =>  (0xb76ea000)
libwspolny.so => /home/juka/lib/libwspolny.so (0xb76e6000)
libc.so.6 => /lib/i686/cmov/libc.so.6 (0xb758c000)
/lib/ld-linux.so.2 (0xb76eb000)
```

Przykład 3-10 Testowanie bibliotek używanych przez program `pierwszy` za pomocą narzędzia `ldd`

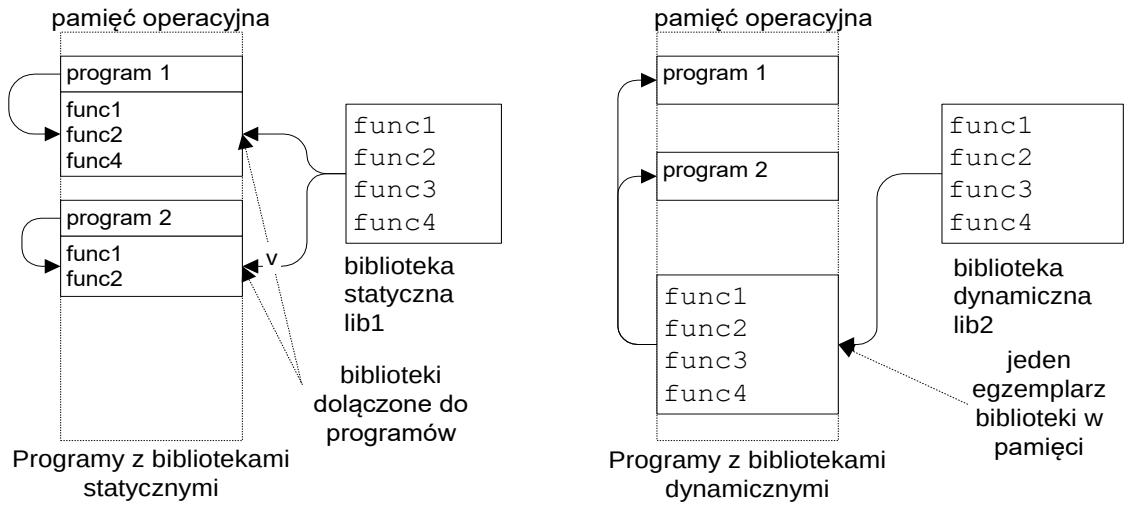
Narzędzia służące do instalowania, testowania zawartości bibliotek i testowania jakich bibliotek dynamicznych wymaga program wykonywalny podaje Tabela 3-2.

Program	Opis	Przykład
`ldconfig`	Instalacja biblioteki współdzielonej	`ldconfig -p` (testowanie jakie biblioteki współdzielone są zainstalowane)
`ldd`	Podaj z jakich bibliotek dynamicznych korzysta program	`ldd hello`
`nm`	Podaj zawartość pliku biblioteki	`nm library.a`
`readelf`	Odczyt pliku w formacie ELF	`readelf -a /lib/libutil.so1`

Tabela 3-2 Zestawienie narzędzi do analizy bibliotek

3.4.3 Biblioteki statyczne i współdzielone – porównanie

Gdy program wykonywalny korzysta z bibliotek statycznych, to zawiera on w sobie wszystkie potrzebne funkcje biblioteczne. Podejście to ma tak zalety jak i wady. Wadą jest to że jeżeli w komputerze wykonuje się kilkadziesiąt procesów korzystających z tych samych bibliotek to występujące w nich moduły wielokrotnie się powielają co prowadzi do niepotrzebnej straty pamięci.

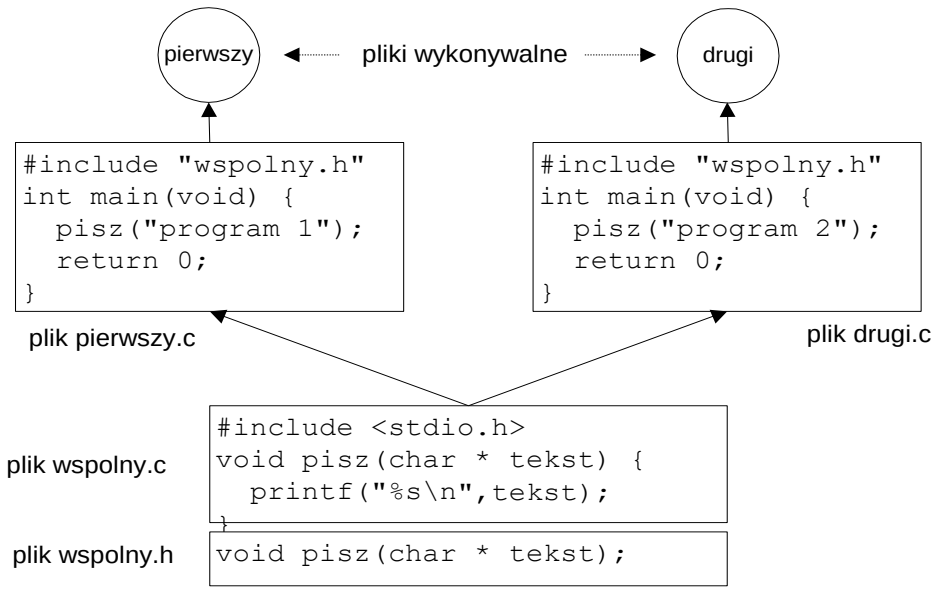


Rys. 3-4 Ilustracja działania biblioteki statycznej i współdzielonej

Sytuacja taka zilustrowana jest na powyższym rysunku. Biblioteka statyczna lib1 zawiera funkcje func1, func2, func3 i func4. Program 1 po konsolidacji zawiera funkcje func1, func2 i func4, a program 2 zawiera funkcje func1 i func2. Widać że funkcje func1 i func2 dublują się co powoduje utratę pamięci. Odmienna sytuacja jest w przypadku zastosowania biblioteki współdzielonej. Ważną cechą biblioteki współdzielonej jest fakt że wystarczy gdy w pamięci operacyjnej będzie tylko jedna jej kopia. Sytuacja gdy program 1 i program 2 używają biblioteki współdzielonej przedstawiona jest na powyższym rysunku . Tak więc zastosowanie bibliotek dynamicznych powoduje zwykle oszczędność pamięci. Zaletą biblioteki współdzielonej jest także większa łatwość aktualizacji. Załóżmy że w bibliotece wykryjemy błąd. Gdy jest to biblioteka statyczna, to w celu naprawienia błędu, należy dokonać ponownego łączenia wszystkich plików wykonywalnych. W przypadku użycia biblioteki współdzielonej poprawiamy tylko samą bibliotekę. Biblioteki współdzielone mają jednak także poważne wady. Jedną z nich jest trudność określenia czy dany program wykona się na innych komputerach niż ten na którym został opracowany i przetestowany. Wynika to z faktu że nie wiemy czy na innym komputerze zainstalowana jest potrzebna biblioteka dynamiczna i w jakiej jest ona wersji.

3.5 Uruchamianie programów za pomocą narzędzia make

Opisane wyżej metody uruchamiania programów są odpowiednie dla prostych aplikacji składających się z jednego programu utworzonego z jednego bądź niewielkiej liczby plików źródłowych. Jednak w praktyce najczęściej mamy do czynienia z bardziej zaawansowanymi aplikacjami. Aplikacje te składają się z wielu programów a te z kolei składają się z wielu plików. W trakcie ich uruchamiania modyfikujemy niektóre z nich. Następnie musimy uruchomić aplikację aby sprawdzić efekty wprowadzonych zmian. Powstaje pytanie które pliki skompilować i połączyć aby wprowadzone w plikach źródłowych zmiany były uwzględnione a aplikacja aktualna. Z pomocą przychodzi nam narzędzie `make` powszechnie stosowane w tworzeniu złożonych aplikacji. W praktyce programistycznej typowa jest sytuacja gdy aplikacja składa się z pewnej liczby programów wykonywalnych zawierających jednak pewne wspólne elementy (stałe, zmienne, funkcje). Pokazuje to poniższy przykład. Przykładowa aplikacja składa się z dwóch programów: `pierwszy.c` i `drugi.c`. Każdy z programów wypisuje na konsoli swoją nazwę i w tym celu korzysta z funkcji `void pisz(char * tekst)` zdefiniowanej w pliku `wspolny.c` a jej prototyp zawarty jest w pliku `wspolny.h`. Sytuację pokazuje poniższy rysunek.



Rys. 3-1 Aplikacja składająca się z dwóch programów

Aby skompilować aplikację należy dwukrotnie wpisać polecenia kompilacji np. tak jak poniżej:

```
$gcc pierwszy.c wspolny.c –o pierwszy
$gcc drugi.c wspolny.c –o drugi
```

Analogiczny efekt osiągnąć można tworząc plik definicji `makefile` dla narzędzia `make` a następnie pisząc z konsoli polecenie `make`. Narzędzie `make` opisane jest obszernie w literaturze i dokumentacji systemu Linux. Plik `makefile` dla powyższego przykładu pokazany jest poniżej. Należy zauważyć że plik `makefile` i pliki źródłowe powinny być umieszczone w jednym folderze. Po wpisaniu polecenia `make` system szuka w folderze bieżącym pliku o nazwie `Makefile` a następnie `makefile` po czym go przetwarza.

```
# Plik makefile dla aplikacji skladajacej się z dwoch programow
all: pierwszy drugi
pierwszy: pierwszy.c wspolny.c wspolny.h
      gcc -o pierwszy pierwszy.c wspolny.c
drugi: drugi.c wspolny.c wspolny.h
      gcc -o drugi drugi.c wspolny.c
```

Przykład 3-3 Plik `makefile` dla aplikacji składającej się z dwóch plików

Wyniki działania polecenia `make` pokazuje polecenie `ls`

```
$ ls
drugi.c makefile pierwszy.c wspolny.c wspolny.h
$make
gcc –o pierwszy pierwszy.c wspolny.c
gcc –o drugi drugi.c wspolny.c
$ls
drugi drugi.c makefile pierwszy pierwszy.c wspolny.c wspolny.h
```

`Przykład 3-4` Działanie polecenia make

Z przykładu widać że kompilator `gcc` został uruchomiony samoczynnie z właściwymi parametrami a w wyniku jego działania zostały utworzone pliki wykonywalne `pierwszy` i `drugi`. Plik definicji `makefile` składa się z **zależności** i **reguł**. Zależność podaje jaki cel ma być osiągnięty (zwykle jest to nazwa pliku który ma być utworzony) i od jakich innych plików zależy. Na podstawie zależności program `make` określa jakie pliki są potrzebne do kompilacji, sprawdza czy ich kompilacja jest aktualna - jeśli tak, to pozostawia bez zmian, jeśli nie, sam kompiluje to co jest potrzebne zgodnie z poleceniem. W prostym wariancie reguła składa się z nazwy celu (może to być nazwa pliku wynikowego lub akcji którą należy przeprowadzić) a po dwukropku listy plików od których dany cel zależy. Jeżeli program `make` stwierdzi że plik wynikowy jest starszy od któregoś z plików od którego zależy dokonywana jest jego kompilacja zgodnie z regułą zawartą w kolejnej linii.

```
cel:  plik1 plik2 ... plikn
      polecenia
```

Należy zwrócić uwagę że linia polecenia zaczyna się niewidocznym znakiem tabulacji. Zastąpienie znaku tabulacji spacjami spowoduje błędne działanie programu. Nawiązując do omawianego przykładu występuje tam definiująca zależność linia:

```
pierwszy: pierwszy.c wspolny.c wspolny.h
```

Informuje ona system że plik `pierwszy` zależy od plików `pierwszy.c wspolny.c wspolny.h` toteż jakakolwiek zmiana w tych plikach spowoduje konieczność powtórnego tworzenia pliku `pierwszy`. Natomiast reguły mówią jak taki plik utworzyć. W tym przykładzie aby utworzyć plik wykonywalny `pierwszy` należy uruchomić kompilator z parametrami jak poniżej.

```
gcc -o pierwszy pierwszy.c wspolny.c
```

Obecnie możemy wykonać eksperyment polegający na modyfikacji pliku `wspolny.c` Modyfikacji można wykonać edytorem (należy zapisać zmiany) lub zasymulować zmiany za pomocą polecenia: `touch nazwa_pliku`. Polecenie `touch` zmienia atrybut czas dostępu do pliku, ustawiając go na czas wykonania polecenia `touch`. Wykonajmy polecenie: `touch wspolny.c`. Wtedy czas modyfikacji pliku `wspolny.c` będzie większy niż czas utworzenia plików `pierwszy` i `drugi` a więc program `make` ma podstawy do stwierdzenia że pliki wykonywalne `pierwszy` i `drugi` są już nieaktualne gdyż zmieniono coś w pliku `wspolny.c`. Gdy napiszemy na konsoli polecenie `make`, program wykona rekompilację tworząc ponownie pliki `pierwszy` i `drugi`.

```
$touch wspolny.c
$make
gcc –o pierwszy pierwszy.c wspolny.c
gcc –o drugi drugi.c wspolny.c
```

Przykład 3-5 Działanie polecenia make – rekompilacja programów pierwszy i drugi

Natomiast gdy zmienimy czas dostępu tylko do pliku pierwszy.c i napiszemy polecenie `make` to zostanie skompilowany tylko program pierwszy co pokazuje poniższy przykład.

```
$touch pierwszy.c
$make
gcc –o pierwszy pierwszy.c wspolny.c
```

Przykład 3-6 Działanie polecenia make – rekompilacja programu pierwszy

W plikach `makefile` umieszczać można linie komentarza poprzez umieszczenie na pierwszej pozycji takiej linii znaku #. W linii określającej cel zależność może być pominięta. Tak jest w poniższym przykładzie gdzie cel archiw nie zawiera żadnej zależności. Natomiast akcja definiuje wykonanie archiwizacji plików źródłowych. Gdy program `make` zostanie wywołany z parametrem będącym nazwą pewnego celu można spowodować wykonanie reguły odpowiadające temu celowi. Do poprzedniego pliku `makefile` dodać można regułę o nazwie `archiw` wykonania archiwizacji plików źródłowych co pokazuje Przykład 3-7. Wpisanie polecenia: `make archiw` spowoduje utworzenie archiwum plików źródłowych i zapisanie ich w pliku `prace.tgz`..

```
all: pierwszy drugi
pierwszy: pierwszy.c wspolny.c wspolny.h
      gcc -o pierwszy pierwszy.c wspolny.c
drugi: drugi.c wspolny.c wspolny.h
      gcc -o drugi drugi.c wspolny.c
clean:
      rm *.o pierwszy drugi
archiw:
      tar -cvf prace.tar *.c *.h makefile
      gzip prace.tar
      mv prace.tar.gz prace.tgz
```

Przykład 3-7 Plik `make` z opcją archiwizacji plików źródłowych

3.5.1 Argumenty polecenia make

Program `make` może być wywołany z parametrami lub bez. Wywołany bez argumentów powoduje realizację pierwszego celu. Przykłady wywołania programu z argumentami podane są poniżej.

`make cel` - powoduje realizację podanego w argumencie celu

`make -f plik` - powoduje przetwarzanie pliku

`make -n` - wypisuje działania ale ich nie wykonuje

3.5.2 Wbudowane makra i zmienne

System make posiada wbudowane makra. Przykłady niektórych pokazane są poniżej.

CFLAGS - opcje kompilatora języka C
CC - nazwa kompilatora języka C, domyślnie `cc`

Więcej informacji na temat wbudowanych makr znaleźć można w dokumentacji systemu. Poniżej podano przykład użycia wbudowanych makr $(CC) i $(CFLAGS).

```
all: pierwszy drugi
pierwszy: pierwszy.c wspolny.c wspolny.h
      $(CC) -o pierwszy $(CFLAGS) pierwszy.c wspolny.c
drugi: drugi.c wspolny.c wspolny.h
      $(CC) -o drugi $(CFLAGS) drugi.c wspolny.c
```

Przykład 3-11 Przykład użycia wbudowanych makr

3.5.3 Makrodefinicje użytkownika

W plikach makefile można stosować makrodefinicje którym przypisuje się pewne wartości. Odwołanie do zmiennej `nazwa` ma postać: `$(nazwa)`. Postępowanie takie jest stosowane gdy w pliku `makefile` powtarzają się pewne napisy, na przykład nazwy plików. Wygodnie wtedy oznaczyć je jako makrodefinicje i przypisać im na początku wartość. Postępowanie takie ilustruje poniższy przykład. Zdefiniowano tam zmienne `ZRODLA1, ZRODLA2` którym przypisano początkowe wartości. Przykładowo użycie wewnątrz pliku zmiennej np. `$(ZRODLA1)` powoduje że na miejsce tej zmiennej podstawiony zostanie napis `"pierwszy.c wspolny.c wspolny.h"` co pokazano poniżej.

```
all: pierwszy drugi
ZRODLA1= pierwszy.c wspolny.c wspolny.h
ZRODLA2= drugi.c  wspolny.c wspolny.h
pierwszy: $(ZRODLA1)
      $(CC) -o pierwszy $(CFLAGS) $(ZRODLA1)
drugi:  $(ZRODLA1)
      $(CC) -o drugi $(CFLAGS) $(ZRODLA2)
```

Przykład 3-8 Plik `make`– ilustracja użycia makrodefinicji

3.5.4 Typowe cele kompilacji

Znaczna liczba plików `makefile` zawiera typowe cele kompilacji, które realizują części procesu przygotowania programu. Typowe cele kompilacji dane są poniżej:

- `all` – jest to pierwszy cel kompilacji zdefiniowany w pliku makefile.
- `install` – akcja wykonywana w ramach tego celu ma spowodować skopiowanie plików wykonywalnych w ich właściwe miejsce przeznaczenia.
- `clean` – usunięcie plików pośrednich i wykonywalnych
- `test` – sprawdzenie czy utworzone programy działają poprawnie

3.6 Uruchamianie programów za pomocą narzędzia gdb

Rzadko zdarza się by napisany przez nas program od razu działał poprawnie. Na ogół zawiera wiele błędów które trzeba pracowicie poprawiać. Typowy cykl uruchamiania programów polega na ich edycji, kompilacji i wykonaniu. Przy kompilacji mogą się ujawnić błędy kompilacji które wymagają poprawy a gdy program skompiluje się poprawnie przystępujemy do jego wykonania. Często zdarza się że uruchamiany program nie zachowuje się w przewidywany przez nas sposób. Wówczas należy uzyskać dodatkowe informacje na temat:

- Ścieżki wykonania programu
- Wartości zmiennych a ogólniej zawartości pamięci związanej z programem

Informacje takie uzyskać można na dwa sposoby:

- Umieścić w kodzie programu dodatkowe instrukcje wyprowadzania informacji o przebiegu wykonania i wartości zmiennych.
- Użyć programu uruchomieniowego (ang. *Debugger*)

Gdy używamy pierwszej metody, dodatkowe informacje o przebiegu wykonania programu są zwykle wypisywane na konsoli za pomocą instrukcji `printf` lub też zapisywane do pliku. Po uruchomieniu programu, instrukcje wypisywania dodatkowych informacji są z programu usuwane. Użycie pierwszej metody jest w wielu przypadkach wystarczające. Jednak w niektórych, bardziej skomplikowanych przypadkach, wygodniej jest użyć programu uruchomieniowego. Program taki daje następujące możliwości:

- Uruchomienie programu i ustawienie dowolnych warunków jego wykonania (np. argumentów, zmiennych otoczenia, itd)
- Doprowadzenie do zatrzymania programu w określonych warunkach.
- Sprawdzenie stanu zatrzymanego programu (np. wartości zmiennych, zawartość rejestrów, pamięci, stosu)
- Zmiana stanu programu (np. wartości zmiennych) i ponowne wznowienie programu.

W świecie systemów klasy POSIX , szeroko używanym programem uruchomieniowym jest `gdb` (ang. *gnu debugger*) który jest częścią projektu GNU Richarda Stallmana. Może on być użyty do uruchamiania programów napisanych w językach C, C++, assembler, Ada , Fortran, Modula-2 i częściowo OpenCL. Program działa w trybie tekstowym, jednak większość środowisk graficznych IDE takich jak Eclipse czy CodeBlocks potrafi się komunikować z `gdb` co umożliwia pracę w trybie okienkowym. Istnieją też środowiska graficzne specjalnie zaprojektowane do współpracy z `gdb` jak chociażby DDD (ang. *Data Display Debugger*). Program gdb posiada wiele możliwości i obszerną dokumentację podaną w a tutaj podane zostaną tylko najważniejsze polecenia.

Kompilacja programu

Aby możliwe było uruchamianie programu z użyciem `gdb` testowany program należy skompilować z kluczem: `-g`. Użycie tego klucza powoduje że do pliku obiektowego z programem dołączona zostanie informacja o typach zmiennych i funkcji oraz zależność pomiędzy numerami linii programu a fragmentami kodu binarnego.
Rozważmy przykładowy program `test.c` podany w Przykład 3-12. Aby skorzystać z debuggera program `test.c` należy skompilować następująco: `gcc test.c -o test -g`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
      int i,j;
      puts("Witamy w Lab PRW");
      system("hostname");
      for(i=0;i<10;i++) {
         j=i+10;
         printf("Krok  %d\n",i);
         sleep(1);
      }
      printf("Koniec\n");
      return EXIT_SUCCESS;
}
```

Przykład 3-12 Program test.c

3.6.1.1 Uruchomienie i zakończenie

Program gdb uruchamia się w następujący sposób:

```
gdb [opcje] [prog [obraz-pam lub pid]]
gdb [opcje] --args prog [argumenty_prog ...]
```

gdzie:

opcje	Opcje programu które można uzyskać pisząc: gdb --h
prog	Nazwa pliku wykonywalnego
obraz-pam	Obraz pamięci utworzony przy awaryjnym zakończeniu programu
pid	Pid procesu do którego chcemy się dołączyć
args	Argumenty programu

Najprostszy sposób uruchomienia debuggera gdb w celu uruchamiania programu zawartego w pliku prog to napisanie na konsoli polecenia: gdb prog

Można też dołączyć się do już działającego programu. W tym celu po nazwie programu należy podać jego pid co pokazuje Tabela 3-3. Program gdb może też służyć do analizy przyczyny awaryjnego zakończenia programu. Gdy proces jest kończony, na skutek otrzymania jednego z pewnych sygnałów, system operacyjny tworzy plik zawierający obraz pamięci procesu. Obraz ten może być analizowany przez gdb w celu znalezienia przyczyny awaryjnego zakończenia procesu. Aby dokonać analizy procesu prog który został awaryjnie zakończony, a jego oraz pamięci został zapisany w pliku core, gdb uruchamia się jak następuje: gdb prog core. Jeszcze jeden wariant uruchomienia programu gdb pozwala na podanie argumentów uruchamianego programu. Gdy program prog należy uruchomić z argumentami a1 a2 ... an to wtedy gdb należy uruchomić z opcją --arg jak następuje: gdb --arg prog a1 a2 ... an. Typowe sposoby uruchomienia programu gdb podaje Tabela 3-3.

gdb prog	Zwykłe uruchomienie gdb dla programu zawartego w pliku prog
gdb prog core	Uruchomienie gdb dla programu z pliku prog. Obraz pamięci znajduje się w pliku core.
gdb prog pid	Dołączenie się do działającego programu, oprócz nazwy podajemy też jego pid
gdb --args prog argumenty	Uruchomienie gdb dla programu z argumentami
gdb -help	Uzyskiwanie pomocy

Tabela 3-3 Różne sposoby uruchomienia programu gdb

Program gdb kończy się wpisując polecenie quit, skrót q lub też kombinację klawiszy Ctrl+d.
Możemy uruchomić program gdb w celu testowania podanego w programu test. W tym celu piszemy polecenie:

```
$gdb test
```

Po wpisaniu tego polecenia zgłasza się program gdb i oczekuje na wprowadzenie poleceń.

Uzyskiwanie pomocy

Program `gdb` posiada znaczną liczbę poleceń. Wpisując polecenie `help` uzyskujemy zestawienie kategorii poleceń, wpisując polecenie `help all` uzyskujemy zestawienie wszystkich poleceń.

Listowanie programu źródłowego

Uzyskanie fragmentu kodu źródłowego następuje przez użycie polecenia list. Polecenie to występować może w różnych wariantach co pokazuje `Tabela 3-4`.

`list`	Listowanie fragmentu kodu źródłowego począwszy od bieżącej pozycji
`list nr`	Listowanie fragmentu kodu źródłowego w pobliżu linii o numerze `nr`
`list pocz, kon`	Listowanie fragmentu kodu źródłowego od linii `pocz` do linii `kon`
`list +`	Listowanie nastepnego fragmentu kodu źródłowego (od pozycji bieżącej)
`list -`	Listowanie poprzedniego fragmentu kodu źródłowego (od pozycji bieżącej)

Tabela 3-4 Polecenia listowania fragmentu kodu źródłowego

Gdy mamy już uruchomiony gdb i testujemy program test możemy wylistować fragment kodu źródłowego pisząc polecenie list jak pokazuje `Ekran 3-1`.

```
(gdb) list
1
2       #include <stdio.h>
3       #include <stdlib.h>
4       #include <unistd.h>
5
6       int main(void) {
7               int i,j;
8               puts("Witamy w Lab PRW");
9               for(i=0;i<20;i++) {
10                 j=i+10;
(gdb)
```

Ekran 3-1 Listowanie kodu źródłowego

Zatrzymywanie procesu

Testowanie programów polega zwykle na próbie ich wykonania i zatrzymania, po czym następuje zbadanie stanu programu. Momenty zatrzymania określane przez tak zwane punkty zatrzymania (ang. *breakpoint*). Są trzy metody zdefiniowania punktu zatrzymania:

- Wskazanie określonej linii w kodzie programu lub też nazwy fukcji. Jest to zwykły punkt zatrzymania.
- Zatrzymanie programu gdy zmieni się wartość zdefiniowanego przez nas wyrażenia (ang. *watchpoint*).
- Zatrzymanie programu gdy zajdzie określone zdarzenie (ang. catchpoint) jak wystąpienie wyjątku czy załadowanie określonej biblioteki.

Tworzonym punktom zatrzymania nadawane są kolejne numery począwszy od 1. Po utworzeniu mogą one być aktywowane (ang *enable*), dezaktywowane (ang. *disable*) i kasowane (ang. *delete*). Najprostszym sposobem ustawienia punktu zatrzymania jest użycie polecenia: break nr_lini. Następuje wtedy ustawienie punktu zatrzymania w danej linii programu. Polecenie: info break powoduje wypisanie ustawionych punktów wstrzymania. Możemy teraz, w naszym przykładowym programie, ustawić punkt zatrzymania na linii 10 co robimy poleceniem: break 10 jak pokazuje Ekran 3-2. Polecenie info break podaje ustawione punkty zatrzymania.

```
(gdb) break 10
Breakpoint 1 at 0x8048453: file test.c, line 10.
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x08048453 in main at test.c:10
(gdb)
```

Ekran 3-2 Ustawienie punktu zatrzymania

Punkty wstrzymania można kasować za pomocą polecenia: `clear nr_linii`.

Polecenie `break` występuje w wielu wariantach co opisane jest w dokumentacji. Między innymi można ustawić zatrzymanie procesu gdy spełniony jest pewien warunek. Polecenie warunkowe ma wtedy postać:

`break nr_linii if warunek` którego skutkiem będzie zatrzymanie procesu w danej linii gdy warunek będzie spełniony. Przykładowo możemy ustawić punkt zatrzymania na linii 10 gdy zmienna i w programie z osiągnie wartość większą niż 5. Polecenie będzie miało postać: `break 10 if i > 5`. Proces ustawiania warunkowego punktu zatrzymania pokazuje Ekran 3-3.

```
(gdb) break 10 if i > 5
Breakpoint 2 at 0x8048453: file test.c, line 10.
(gdb) i b
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x08048453 in main at test.c:10
        stop only if i > 5
(gdb)
```

Ekran 3-3 Ustawienie warunkowego punktu zatrzymania

Uruchamianie procesu

Jeżeli w programie ustawiono punkty zatrzymania to można go uruchomić. Wykonuje się to poprzez polecenie `run`.

```
(gdb) run
Starting program: /home/juka/prog/beagle/test
Witamy w Lab PRW
Krok  0
Krok  1
Krok  2
Krok  3
Krok  4
Krok  5

Breakpoint 1, main () at test.c:10
10                  j=i+10;
```

Ekran 3-4 Wykonanie programu do punktu zatrzymania

Listowanie programu	`list [numer linii]`	`skrót`
Uruchomienie programu	`run [argumenty]`	

Ustawienie punktu zatrzymania	`break nr_linii`
	`break nazwa_funkcji`
	`break nr_linii if warunek`
		`b`

Ustawienie pułapki, program zatrzyma się gdy zmieni się wartość obserwowanej zmiennej	`watch nazwa_zmiennej`	`w`
Listowanie punktów zatrzymania	`info break`	
Kasowanie punktu zatrzymania	`clear nr_linii`	
Wyprowadzenie wartości zmiennych	`print nazwa_zmiennej`	`p`
Kontynuacja do następnego punktu wstrzymania	`Continue`	`c`
Wykonanie następnej instrukcji, nie wchodzimy do funkcji	`Next`	`n`
Wykonanie następnej instrukcji, wejście do funkcji	`Step`	`s`
Uzyskanie pomocy	`Help`	`h`
Ustawienie wartości zmiennej `var` na `wart`	`set var nazwa_zmienn=wart`	
Zakończenie pracy programu	`quit`	`q`

Tabela 3-5 Najczęściej używane polecenia programu uruchomieniowego gdb

Częściej spotykane polecenia debuggera gdb podano poniżej.

- b main - Put a breakpoint at the beginning of the program
- b - Put a breakpoint at the current line
- b N - Put a breakpoint at line N
- b +N - Put a breakpoint N lines down from the current line
- b fn - Put a breakpoint at the beginning of function "fn"
- d N - delete breakpoint number N
- info break - list breakpoints
- r - Run the program until a breakpoint or error

- c - continue running the program until the next breakpoint or error
- f - Run until the current function is finished
- s - run the next line of the program
- s N - run the next N lines of the program
- n - like s, but don't step into functions
- u N - run until you get N lines in front of the current line
- p var - print the current value of the variable "var"
- bt - print a stack trace
- u - go up a level in the stack
- d - go down a level in the stack
- q - Quit `gdb`

3.7 Debugowanie biegnacego procesu

Debugger `gdb` można również wykorzystać do testowania uruchomionego już procesu. W celu wykonania eksperymentu napiszmy program jak poniżej:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
    int i,j,k;
    puts("Witamy w Lab PRW");
    system("hostname");
    for(k=1;k<100;k++) {
        printf("Nowa epoka %d\n",k);
        for(i=0;i<10;i++) {
          j=i+10;
          printf("Epoka %d krok  %d\n",k,i);
          sleep(1);
        }
    }
    printf("Koniec\n");
    return EXIT_SUCCESS;
}
```

Przykład 3-9 Program test2.c

Program kompilujemy z opcją –g a nastepnie uruchamiamy.

```
$gcc test2.c -o test2 -g
./test2
```

Otwieramy drugi terminal, ustalamy `pid` procesu i uruchamiamy `gdb` jak poniżej.

```
$ps -ef | grep test2
$ ps -ef | grep test2
juka     24719 24714  0 12:02 pts/4    00:00:00 ./test2
juka     24724 24615  0 12:02 pts/5    00:00:00 grep test2
```

Następnie uruchamiamy program gdb podając nazwę biegnącego procesu i jego pid.

```
$gdb test2 24719
```

Gdy debugger się zgłosi ustawiamy breakpoint na linii 10 i obserwujemy zatrzymanie się procesu test2. Dalej debugowanie programu przebiega w zwyczajny sposób.

3.8 Zintegrowane środowisko uruchomieniowe Eclipse

Ręczne wypisywanie skomplikowanych poleceń kompilacyjnych jest uciążliwe i podatne na błędy. Dlatego pojawiło się wiele zintegrowanych środowisk kompilacyjnych (ang. IDE – *Integrated Development Enviroment*) pracujących w trybie graficznym. Znanym środowiskiem tego typu jest Visual Studio firmy Microsoft. W dziedzinie systemów wbudowanych najczęściej stosowane jest środowisko Eclipse. Projekt ten zainicjowany przez firmę IBM został przekazany na zasadach wolnego programowania fundacji Eclipse i jest przez nią dalej rozwijany. Eclipse jest napisane w Javie i pierwotnie było przeznaczone do tworzenia oprogramowania w tym właśnie języku. Środowisko to jest w znacznym stopniu konfigurowalne i umożliwia dość łatwą instalację rozszerzeń w postaci tak zwanych wtyczek (ang. *Plug In*). Istnieją wtyczki dla języka C,C++, Pythona i wielu innych. Środowisko Eclipse jest chętnie stosowane w dziedzinie systemów wbudowanych. Na jego podstawie skonstruowano zaawansowane środowiska uruchomieniowe ułatwiające programowanie systemów wbudowanych. Wymienić tu można platformę Momentics firmy QNX Sotfware Systems czy CodeComposer firmy Texas Instruments. Środowisko Eclipse skonfigurowane do potrzeb systemów wbudowanych posiada typowo następujące funkcje:

- Utworzenie projektu i edycja plików źródłowych
- Kompilacja projektu na wybrany system docelowy
- Przesyłanie programów na system docelowy
- Uruchamianie (ang. Debug) programów w systemie docelowym , Eclipse jest zwykle interfejsem do debugera `gdb`.

Po odpowiednim skonfigurowaniu Eclipse może pełnić różne dodatkowe funkcje. Przykładem może być zintegrowane środowisko uruchomieniowe Momentics. Środowisko może być wykonywane na komputerze bazowym pracującym w systemie Windows, Linux lub QNX Neutrino. Programy mogą być tworzone dla procesorów ARM, MIPS, PowerPC, SH-4, XScale, x86. Momentics zawiera takie narzędzia jak edytory, kompilatory, biblioteki, profilery kodu, analizatory systemów docelowych i inne. Dostępna jest także obszerna dokumentacja.

3.9 Pisanie bezpiecznego kodu, makro assert

W pisaniu bezpiecznego kodu przydatne jest makro `assert`. Zdefiniowane jest ono w pliku nagłówkowym `assert.h`.

```
#include <assert.h>

void assert(int expression)
```

Gdy wartość wyrażenia expression jest równa zero program będzie przerwany i wyprowadzone będą informacje:

- Nazwa pliku z programem
- Numer linii w której wystąpił błąd
- Nazwa funkcji w której wystąpił błąd

Makra assert należy użyć do sprawdzenia czy spodziewane warunki są spełnione, np.:

```
assert(a > b+1);
assert(c == f);
assert(e);
```

Poniżej podany został przykład użycia funkcji `assert`.

```
#include <stdio.h>
#include <assert.h>
// #define NDEBUG

int main(int argc, char * argv[]) {
    printf("Start, argc= %d\n",argc);
    assert(argc > 1);
    printf("argv[1]= %s\n", argv[1]);
    printf("Koniec\n");
}
```

Przykład 3-10 Użycie makra assert

Gdy uruchomimy program bez argumentów (czyli wartość wyrażenia `argc >1` będzie fałszywa (zero)) makro zatrzyma program i wyprowadzi komunikat:

```
$./assert
Start, argc= 1
Assert: assert.c:6: main: Assertion 'argc > 1' failed.
Przerwane
$
```

Natomiast prawidłowe uruchomienie programu, z jednym parametrem spowoduje że wartość wyrażenia `argc >1` będzie prawdziwa i makro nie zadziała.

```
$./assert 222
Start, argc= 2
argv[1]= 222
Koniec
$
```

Działanie makra może być wyłączone przez dodanie definicji

`#define NDEBUG` lub opcji kompilacyjnej `-DNDEBUG (gcc assert.c -o assert -DNDEBUG)`

3.10 Odwołanie się do argumentów programu

Często zachodzi potrzeba przekazania do wnętrza programu określonych parametrów które mogą być przekazane jako parametry przy wywołanie programu. Dla przykładu:

```
prog 123 a2 a3
```

Poniższy przykład pokazuje jak odwołać się do argumentów.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
      int i,k;
      printf("argc= %d\n",argc);
      for(i=0;i<argc;i++) {
        printf("argv[%d] = %s\n",i,argv[i]);
      }
      k = atoi(argv[1])
      return EXIT_SUCCESS;
}
```

Przykład 3-11 Program wypisujący swoje argumenty

Program daje wyniki jak poniżej.

```
argc= 4
argv[0] = ./prog
argv[1] = 123
argv[2] = a2
argv[3] = a3
```

Należy zauważyć że przekazane do programu parametry są łańcuchami tekstowymi (ang. *string*). Jeżeli chcielibyśmy zamienić parametr 123 na liczbę należy użyć funkcji `int atoi(char *lstring)` przekształcającą napis `lstring` na liczbę `int` tak jak w powyższym przykładzie.

3.11 Zadania

3.11.1 Makrodefinicje i kompilacja warunkowa

Napisz program używający kompilacji warunkowej który w zależności od podanego w czasie kompilacji makra DEBUG wyświetla szczegółową informację o swym działaniu lub ją pomija.

3.11.2 Biblioteki statyczne

Zmodyfikuj podany w 3.4 przykład poprzez dodanie jeszcze jednej funkcji:

```
int dlugosc(char *napis)
```

która oblicza i zwraca długość napisu. Utwórz bibliotekę statyczną składającą się z funkcji pisz i dlugość. Zademontruj działanie programu korzystającego z tej biblioteki.

3.11.3 Biblioteki dynamiczne

Zmodyfikuj podany w 3.4 przykład poprzez dodanie jeszcze jednej funkcji:

```
int dlugosc(char *napis)
```

która oblicza i zwraca długość napisu. Utwórz bibliotekę dynamiczną składającą się z funkcji pisz i dlugość. Zademontruj działanie programu korzystającego z tej biblioteki.

4. Tworzenie procesów

4.1 Funkcja fork

Do tworzenia nowych procesów wykorzystuje się funkcję `fork`. Proces bieżący przekształca się w inny proces za pomocą funkcji `exec`. Funkcja `exit` służy do zakończenia procesu bieżącego, natomiast funkcji `wait` używa się do oczekiwania na zakończenie procesu potomnego i do uzyskania jego statusu.

- Funkcja `int fork()` – powoduje utworzenie nowego procesu będącego kopią procesu macierzystego. Segment kodu jest taki sam w obu zadaniach. Proces potomny posiada własny segment danych i stosu. Funkcja zwraca 0 w kodzie procesu potomnego a PID nowego procesu w procesie macierzystym (lub -1 gdy nowy proces nie może być utworzony).
- Fukcja `execl(fname,arg1,arg2,...,NULL)` przekształca bieżący proces w proces o kodzie zawartym w pliku wykonywalnym `fname`, przekazując mu parametry `arg1,arg2`, itd.
- Funkcja `pid = wait(&status)` powoduje zablokowanie procesu bieżącego do czasu zakończenia się jednego zadania potomnego. Gdy zad. potomne wykona funkcję `exit(status)`; funkcja zwróci PID procesu potomnego i nada wartość zmiennej status. Gdy nie ma procesów potomnych funkcja wait zwróci -1.
- Funkcja `exit(int stat)` powoduje zakończenie procesu bieżącego i przekazanie kodu powrotu stat do procesu macierzystego.

Podstawowy schemat tworzenia nowego procesu podany jest poniżej.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void main(void){
  int pid,status,i;
  if((pid = fork()) == 0) { /* Proces potomny ---*/
      for(i=0;i<10;i++) {
         printf(" Potomny krok: %d\n",i);
         sleep(1);
      }
      exit(0);
  }
 /* Proces macierzysty */
 printf("Macierzysty = %d \n",getpid());
 pid = wait(&status);
 printf("Proces %d zakonczony status: %d\n",pid,WEXITSTATUS(status));
}
```

Przykład 4-1 Podstawowy wzorzec tworzenia procesu potomnego

4.2 Schemat użycia funkcji execl.

Funkcja `execl` używana jest do przekształcania bieżącego procesu w inny proces.

Funkcja 4-1 `exec` – transformacja procesu	
`pid_t execl(char * path, arg0,arg1,..., argN,NULL)`	
`path`	Ścieżka z nazwą pliku wykonywalnego.
`arg0`	Argument 0 przekazywany do funkcji `main` tworzonego procesu. Powinna być to nazwa pliku wykonywalnego ale bez ścieżki.
…	…
`argN`	Argument N przekazywany do funkcji `main` tworzonego procesu .

Przykład programu tworzącego proces potomny za pomocą funkcji `execl` podano poniżej.

```
// Ilustracja działania funkcji execl – uruchomienie programu potomny
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void main(void){
  int pid,status;
  if((pid = fork()) == 0) { /* Proces potomny pot ---*/
     execl("./pot","pot",NULL);
  }
 /* Proces macierzysty */
 printf("Macierzysty = %d \n",getpid());
 pid = wait(&status);
 printf("Proces %d zakończony status: %d\n",pid,WEXITSTATUS(status));
}
```

Przykład 4-2 Przykład utworzenia procesu potomnego za pomocą funkcji `execl`

```
#include <stdlib.h>
main() {
  int id, i ;
  for(i=1;i <= 10;i++) {
       printf("Potomny krok: %d \n",i);
       sleep(1);
  }
  exit(i);
}
```

Przykład 4-3 Kod procesu potomnego `pot.c`

4.3 Ustanawianie ograniczeń na użycie zasobów

W każdym systemie komputerowym zasoby potrzebne do tworzenia i wykonywania procesów są ograniczone. W przypadku gdy w systemie działa wiele procesów ważną rzeczą jest zabezpieczenie systemu przed wyczerpaniem zasobów spowodowanym przez nadmierne zużycie zasobów przez procesy wchodzące w skład aplikacji.

☞ W bezpiecznym systemie operacyjnym powinien istnieć mechanizm limitujący pobieranie zasobów przez procesy.

System Linux posiada mechanizmy pozwalające na ustanowienie limitu na takie zasoby jak:

- czas procesora,
- pamięć operacyjna,
- pamięć wirtualna
- wielkość pamięci pobranej ze sterty,
- wielkość segmentu stosu,
- maksymalna liczba deskryptorów plików,
- maksymalna wielkość pliku utworzonego przez proces
- maksymalna liczba procesów potomnych tworzonych przez proces.
- Maksymalna liczba blokad plików i obszarów pamięci operacyjnej

Dla każdego z tych zasobów istnieje:

- ograniczenie miękkie (*ang. soft limit*)
- ograniczenie twarde (*ang. hard limit*).

Ograniczenie miękkie może być zmieniane przez proces bieżący ale nie może przekroczyć twardego.
Ograniczenie twarde może być zmieniane przez proces o statusie administratora.

4.3.1 Testowanie i ustawianie limitu zasobów z poziomu shell

Do testowania i ustawiania poziomu zużycia zasobu przez uzytkownika służy polecenie `ulimit`

`ulimit [-acdfHlmnpsStuv] [limit]`

Opcje:

-S	Zmień lub pokaż miękkie ograniczenie
-H	Zmień lub pokaż twarde ograniczenie
-a	Pokaż wszystkie ograniczenia
-c	Maksymalna wielkość pamięci operacyjnej
-d	Maksymalna wielkość segmentu danych
-f	Maksymalna wielkość tworzonego pliku
-l	Maksymalna wielkość pamięci operacyjnej która może być zablokowana
-n	Maksymalna liczba deskryptorów plików
-p	Maksymalna wielkość bufora na łącza nienazwane
-s	Maksymalna wielkość stosu
-t	Maksymalna wielkość jednostek czasu procesora
-u	Maksymalna liczba tworzonych przez użytkownika procesów
-v	Maksymalna wielkość pamięci wirtualnej dla procesu

```
$ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 16382
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) unlimited
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
$ulimit -Sn 10
$ulimit -n
10
```

Przykład 4-1 Testowanie i ustawianie limitów zasobów

Ustawiane limity zmieniane są w jednostkach 1024 bajtowych z wyjątkiem:

- –t – sekundy,
- -p – bloki 512 bajtów
- -n – sztuki
- -u – sztuki,

4.3.2 Testowanie i ustawianie limitu zasobów z poziomu programu

Do testowania limitów zasobów służy funkcja `getrlimit`.

`getrlimit` – pobranie aktualnego limitu zasobów

```
int getrlimit(int resource, struct rlimit *rlp)
```

Gdzie:
`resource` Określenie zasobu.
`rlp` Wskaźnik na strukturę zawierającą bieżące i maksymalne ograniczenie.

Funkcja zwraca `0` gdy sukces a `-1` gdy błąd. Jako pierwszy parametr funkcji podać należy numer testowanego zasobu które podaje tabela. Funkcja powoduje skopiowanie do struktury `rlp` aktualnych ograniczeń. Struktura ta zawiera co najmniej dwa elementy:

`rlim_cur` - zawiera ograniczenie miękkie
`rlim_max` zawierający ograniczenie twarde.

Do ustawiania limitów zasobów służy funkcja `setrlimit`.

`setrlimit` – ustanowienie nowego limitu zasobów

```
int setrlimit(int resource, struct rlimit *rlp)
```

Funkcja zwraca `0` gdy sukces a `-1` gdy błąd. Gdy proces próbuje pobrać zasoby ponad przydzielony limit system operacyjny może:
1. Zakończyć proces.
2. Wysłać do niego sygnał .
3. Zakończyć błędem funkcję pobierającą dany zasób.

Oznaczenie	Opis	Akcja przy przekroczeniu
RLIMIT_AS	Pamięć wirtualna	Wysłanie sygnału SIGSEGV do procesu przekraczającego zasób
RLIMIT_CORE	Pamięć operacyjna	Zakończenie procesu z zapisaniem na dysku obrazu pamięci operacyjnej.
RLIMIT_CPU	Czas procesora	Wysłanie sygnału SIGXCPU do procesu przekraczającego zasób.
RLIMIT_DATA	Wielkość pamięci pobranej ze sterty.	Funkcja pobierająca pamięć kończy się błędem.
RLIMIT_FSIZE	Maksymalna wielkość pliku utworzonego przez proces. Gdy 0 to zakaz tworzenia plików.	Wysłanie sygnału SIGXFSZ do procesu przekraczającego zasób. Gdy sygnał jest ignorowany to plik nie zostanie powiększony ponad limit.
RLIMIT_NOFILE	Maksymalna liczba deskryptorów plików tworzonych przez proces.	Funkcja tworząca ponad limitowe pliki skończy się błędem.
RLIMIT_STACK	Maksymalny rozmiar stosu	Wysłanie sygnału SIGSEGV do procesu przekraczającego stos.
RLIMIT_NPROC	Maksymalna liczba procesów potomnych tworzonych przez proces.	Procesy przekraczające limit nie będą utworzone.
RLIMIT_MSGQUEUE	Pamięć zajmowana przez kolejki komunika tów POSIX	

Tab. 4-1 Zestawienie niektórych zasobów systemowych podlegających ograniczeniu

Ustanowienie ograniczenie `RLIMIT_CPU` na czas zużycia procesora w systemach działających nieprzerwanie nie ma dużego zastosowania

```
#include <stdlib.h>
#include <sys/resource.h>
int main(int argc, char *argv[]) {
  int res, i, num = 0;
  struct rlimit rl;
  printf("       CUR          MAX \n");
  getrlimit(RLIMIT_CPU,&rl);
  printf("CPU    %d   %d \n",rl.rlim_cur, rl.rlim_max);
  getrlimit(RLIMIT_CORE,&rl);
  printf("CORE   %d   %d \n",rl.rlim_cur, rl.rlim_max);
  rl.rlim_cur = 2;
  setrlimit(RLIMIT_CPU,&rl);
  while (1);
  return 0;
}
```

Program 4-1 Program `rlimit.c` testujący i nakładający ograniczenia na pobierane przez proces zasoby

Gdy przydzielony czas procesora ulegnie wyczerpaniu proces zakończy się z komunikatem:

```
$CPU time limit exceeded (core dumped)
```

4.4 Zadania

4.4.1 Atrybuty procesów

Napisz proces o nazwie `prinfo` który wyświetla następujące atrybuty procesów:

- identyfikator procesu PID,
- identyfikator procesu macierzystego PPID,
- rzeczywisty identyfikator użytkownika UID,
- rzeczywisty identyfikator grupy GID,
- efektywny identyfikator użytkownika EUID,
- efektywny identyfikator grupy EGID,
- priorytet procesu
- otoczenie procesu

A) Uruchom program `prinfo` i wyprowadź na konsolę atrybuty procesu.

B) Wprowadź do otoczenia procesu nowy parametr MOJPAR i nadaj mu wartość wczytywaną z klawiatury i przetestuj czy zmiana została wprowadzona poprawnie.

C) Ustaw w napisanym programie `prinfo` bit `setuid`, zaloguj się jako inny użytkownik i zaobserwuj rzeczywisty i efektywny identyfikator użytkownika.

4.4.2 Tworzenie procesów za pomocą funkcji fork - struktura 1 poziomowa.

Proces macierzysty o nazwie `procm1` powinien utworzyć zadaną liczbę procesów potomnych PP-1, PP-2,..., PP-N za pomocą funkcji `fork` a następnie czekać na ich zakończenie. Zarówno proces macierzysty jak i procesy potomne powinny w pętli wyświetlać na konsoli swój numer identyfikacyjny jak i numer kroku w odstępach 1 sekundowych. Numer identyfikacyjny procesu potomnego NR wynika z kolejności jego utworzenia. Np. proces o numerze 3 wykonujący N kroków powinien wyświetlać napisy:

```
Proces 3 krok 1
Proces 3 krok 2
...............
Proces 3 krok N
```

Aby wykonać zadanie do procesu `procm1` należy przekazać informacje:

- Ile ma być procesów potomnych

- Ile kroków ma wykonać każdy z procesów potomnych.

Informacje te przekazuje się jako parametry programu `procm1` z linii poleceń.

`procm1 K0 K1 K2 .... KN` gdzie:

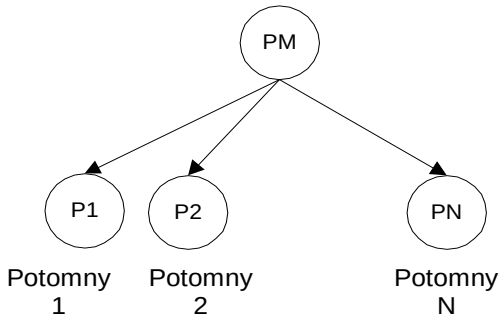
`K0` – liczba kroków procesu macierzystego
`K1` – liczba kroków procesu potomnego `P1`
`...`
`KN` - liczba kroków procesu potomnego PN

Np. wywołanie `procm1 10 11 12` oznacza że należy utworzyć 2 procesy potomne i maja one wykonać 11 i 12 kroków. Proces macierzysty ma wykonać 10 kroków. Na zakończenie procesu potomnego powinien on wykonać funkcję `exit(NR)` przekazując jako kod powrotu swój numer identyfikacyjny procesowi macierzystemu. Proces macierzysty powinien czekać na zakończenie się procesów potomnych (funkcja `pid = wait(&status)`) i dla każdego zakończonego procesu wyświetlić: `pid` i kod powrotu. W tej wersji programu procesy potomne nie posiadają swoich procesów potomnych. Proces macierzysty powinien:

1. Utworzyć `n = argc - 2` procesów potomnych. Należy w pętli użyć funkcji `fork()`.
2. Wypisywać w pętli (K0 kroków) co 1 sek. komunikaty:

```
Macierzysty krok1
Macierzysty krok2
...
Macierzysty krokK0
```

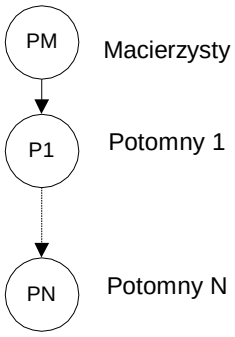
3. Zaczekać na zakończenie wszystkich procesów potomnych, wypisać ich pid i kod powrotu:



Rys. 4-1 Proces macierzysty i procesy potomne – struktura dwupoziomowa

4.4.3 Tworzenie procesów za pomocą funkcji fork - struktura N poziomowa.

Zadanie to jest analogiczne do poprzedniego ale struktura tworzonych procesów ma być N poziomowa. Znaczy to że zarówno proces macierzysty jak i każdy proces potomny (z wyjątkiem ostatniego procesu N) tworzy dokładnie jeden proces potomny. Drzewo procesów będzie wyglądało jak poniżej.



Rys. 4-2 Proces macierzysty i procesy potomne – struktura pionowa

Zadanie należy rozwiązać stosując funkcję rekurencyjną `tworz(int poziom, char *argv[])`. W funkcji tej argument poziom oznacza zmniejszany przy każdym kolejnym wykonaniu poziom wywołania funkcji a argument `argv` zadaną liczbę kroków.

4.4.4 Tworzenie procesów za pomocą funkcji fork i exec.

Zadanie to jest podobne do zadania 1. Różnica jest taka ze procesy potomne powinny być przekształcone w inne procesy o nazwie `proc_pot` za pomocą funkcji `execl`.

Tak wiec:

- Proces macierzysty uruchamia się poleceniem `procm3 K0 K1 K2 .... KN`
- Proces macierzysty `procm3` powinien utworzyć zadaną liczbę procesów potomnych `PP-1, PP-2,...,PP-N` gdzie `N = argc-2` za pomocą funkcji `fork` a następnie wyprowadzić na konsole komunikaty:

```
Proces macierzysty krok 1
Proces macierzysty krok 2
...............
Proces macierzysty krok K0
```

Następnie proces macierzysty ma czekać na zakończenie się procesów potomnych. Dla każdego zakończonego procesu potomnego należy wyświetlić jego `pid` i kod powrotu.

- Procesy potomne przekształcają się w procesy `proc_pot` z których każdy ma wyświetlać w pętli na konsoli swój numer identyfikacyjny i numer kroku w odstępach 1 sekundowych. Numer identyfikacyjny i liczba kroków do wykonania ma być przekazana z procesu macierzystego jako parametr. Na zakończenie procesu potomnego powinien on wykonać funkcje `exit(NR)` przekazując jako kod powrotu swój numer identyfikacyjny procesowi macierzystemu.

- Procesy `proc_pot` należy utworzyć edytorem w postaci oddzielnych plików, skompilować, uruchomić i przetestować. Uruchomienie procesu `proc_pot` jako `proc_pot 4 6` powinno spowodować wyprowadzenie komunikatów:

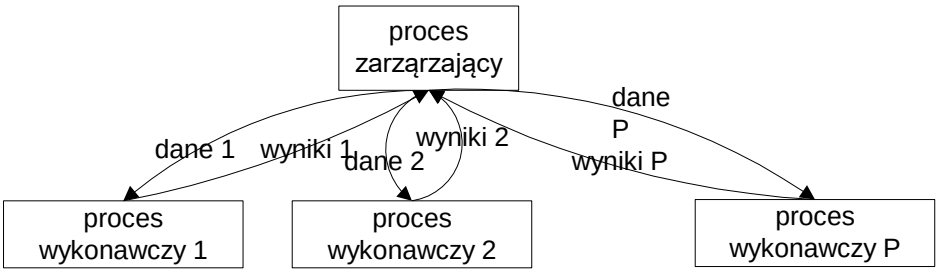
```
Proces 4 krok 1
Proces 4 krok 2
...............
Proces 4 krok 6
Proces 4 zakończony
```

4.4.5 Tworzenie procesów potomnych za pomocą funkcji system.

Wykonaj zadanie analogiczne jak w poprzednim punkcie z tą różnicą że nowe procesy mają być tworzone za pomocą funkcji system.

4.4.6 Znajdowanie liczb pierwszych

Celem tego ćwiczenia jest sprawdzenie jakie korzyści odnieść można z współbieżności. Wiele praktycznych problemów wymaga bardzo znacznej mocy obliczeniowej której pojedynczy procesor nie jest w stanie dostarczyć. Należy więc problem podzielić na mniejsze jednostki które mogły by być przetwarzane przez oddzielne procesory lub rdzenie. Jest to zadanie dekompozycji problemu, a jednostkami mogą być procesu, które są szeregowane na oddzielne rdzenie, procesory czy nawet komputery (elementy klastra). Istnieje wiele modeli takiego przetwarzania, jeden z bardziej znanych to model zarządca – wykonawca. Proces zarządzający dzieli zadanie na mniejsze jednostki (dekompozycja), tworzy procesy wykonawcze i przekazuje im zadania do wykonania. Następnie czeka na ich zakończenie i zbiera otrzymane wyniki (agregacja).



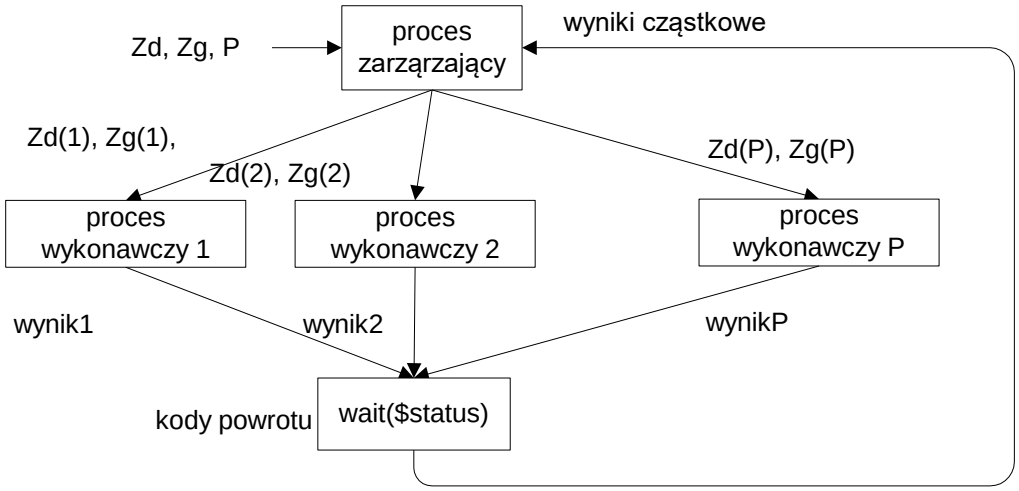
Rys. 4-1 Model zarządca - wykonawca

Właśnie to jest tematem tego ćwiczenia. Zadaniem do wykonania jest sprawdzenie ile liczb pierwszych znajduje się w zadanym przedziale. Nie jest to zbytnio twórcze zadanie, ale ma te zaletę że daje się łatwo zdekomponować na podproblemy.
Napisz program `pierwsze` który ma znajdować liczby pierwsze w zadanym przedziale `[Zd,...,Zg]`. Liczba jest pierwsza gdy dzieli się przez 1 i przez siebie samą. Prymitywny algorytm sprawdzania, czy dana liczba n jest liczbą pierwszą dany jest poniżej:

```
int pierwsza(int n)
// Funkcja zwraca 1 gdy n jest liczbą pierwsza 0 gdy nie
{ int i,j=0;
  for(i=2;i*i<=n;i++) {
    if(n%i == 0) return(0) ;
 }
 return(1);
}
```

Przykład 4-4 Funkcja sprawdzająca czy liczba n jest pierwsza

Obliczenia można przyspieszyć dzieląc zakres [Zd,...,Zg] na P podprzedziałów [Zd(1),...,Zg(1)], [Zd(2),...,Zg(2)],..., [Zd(P),...,Zg(P)] gdzie P jest liczbą dostępnych procesorów. W każdym z podprzedziałów [Zd(i),...,Zg(i)] możemy znajdować liczby pierwsze niezależnie, co robi proces wykonawczy o nazwie `licz`. Tak więc o ile dysponujemy procesorem wielordzeniowym obliczenia wykonane mogą być równolegle. Wyniki pośrednie `ile_pierw_c` (liczba liczb pierwszych w przedziale) uzyskane przez poszczególne procesy wykonawcze mają być przekazane poprzez kod powrotu w funkcji `exit(ile_pierw_c)`. Po zakończeniu procesów wykonawczych proces macierzysty odczytuje poszczególne wyniki cząstkowe wykonując funkcje `wait(&status)` i sumuje wyniki cząstkowe podając na końcu czas obliczeń i liczbę znalezionych liczb pierwszych.



Rys. 4-1 Znajdowanie liczb pierwszych – wiele procesów obliczeniowych

Zadanie powinno być rozwiązane w następujący sposób:

1. Program zarządzający dzieli przedział [Zd,...,Zg] na P podprzedziałów. Następnie tworzy procesy potomne używając funkcji `fork i execl("./licz","licz",pocz,kon,numP,0 ).` Funkcje te uruchamiają procesy wykonawcze o nazwie `licz`. Każdemu z tych procesów Pi mają być jako argumenty przekazane: granice przedziału `pocz=Zd(i), kon=Zg(i),` nazwa pliku z wynikami pośrednimi i numer procesu `numP`. Tak więc, proces wykonawczy powinien mieć postać:

   `licz granica_dolna granica_górna numer_procesu`

2. Proces zarządzający czeka na zakończenie procesów wykonawczych wykonując funkcje `wait(&status)` i odczytuje ze zmiennej status dane cząstkowe o znalezionych liczbach liczb pierwszych. Następnie oblicza ich sumę która ma być wyprowadzona na konsolę i czas obliczeń. Z uwagi na ograniczenie kodu powrotu z programu do zakresu 0-255 dla większych zakresów obliczeń nie otrzymamy prawidłowych wyników.

Proces wykonawczy znajduje liczby pierwsze w przedziale [Zd(i),...,Zg(i)]. Znalezioną liczbę liczb pierwszych `ile_pierw_c` przekazane poprzez kod powrotu w funkcji `exit(ile_pierw_c)` do procesu zarządzającego.

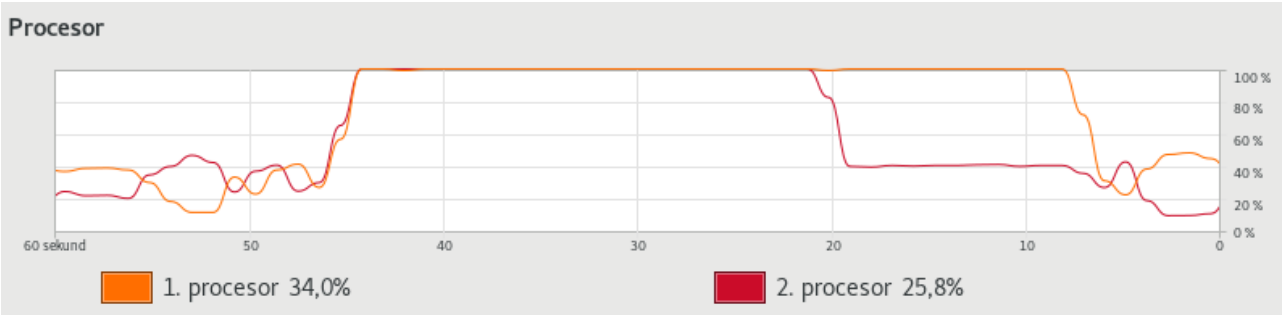
Program główny powinien mieć następujące argumenty:

- Zakres dolny przedziału - `zd`
- Zakres górny przedziału - `zg`
- Liczbę procesów wykonawczych - `P`

Ma być wywołany jak poniżej:

```
$./pierwsze zd zg P
```

Program ma podawać czas obliczeń, do jego pomiaru można użyć funkcji `time(NULL)`. Proszę narysować wykres pokazujący zależność czasu obliczeń od liczby procesów.



Ekran 4-1 Wykres pokazujący obciążenie rdzeni procesora uzyskany za pomocą Monitora systemu (ang. System Monitor)

4.4.7 Ustawianie i testowanie limitów użycia zasobów z poziomu shell

W systemie Linux istnieje możliwość ograniczenia wielkości zasobu przyznawanego procesom. Ograniczeniu podlegać może czas zużycia procesora, wielkość pamięci zajmowanej przez proces, maksymalnej liczby deskryptorów plików i inne parametry. Z poziomu shella można uzyskiwać informacje o ograniczeniach zasobu za pomocą polecenia ulimit. Zapoznaj się z tym poleceniem w podręczniku man (man 1 bash). Wyświetl na terminalu aktualne wielkości ograniczeń na zasoby:

Napisz program `test_limit` testujący przekroczenia limitów zasobów i wytwarzający sytuację ich przekroczenia. Sytuacja ta ma być wykryta i zasygnalizowana. Do ograniczenia zasobu należy użyć polecenia `ulimit` z odpowiednim parametrem co pokaże polecenie `ulimit --help`. Np. polecenie `ulimit -u 10` ogranicza liczbę procesów potomnych do 10. Należy przeprowadzić testy dla następujących zasobów:

`RLIMIT_CPU`	Czas zużycia procesora
`RLIMIT_STACK`	Wielkość pamięci zajmowanej przez stos
`RLIMIT_DATA`	Wielkość pamięci zajmowanej przez dane zainicjowane, niezainicjowane i stertę
`RLIMIT_FSIZE`	Maksymalna wielkość pliku utworzonego przez proces
`RLIMIT_NOFILE`	Maksymalna liczba plików tworzonych przez proces
`RLIMIT_NPROC`	Maksymalna liczba procesów tworzonych przez proces

Tab. 4-2 Niektóre limity zasobów systemu operacyjnego

5. Pliki

5.1 Podstawowa biblioteka obsługi plików

W systemie Linux prawie wszystkie zasoby są plikami. Dane i urządzenia są reprezentowane przez abstrakcję plików. Mechanizm plików pozwala na jednolity dostęp do zasobów tak lokalnych jak i zdalnych za pomocą poleceń i programów usługowych wydawanych z okienka terminala. Plik jest obiektem abstrakcyjnym z którego można czytać i do którego można pisać. Oprócz zwykłych plików i katalogów w systemie plików widoczne są pliki specjalne. Zaliczamy do nich łącza symboliczne, kolejki FIFO, bloki pamięci, urządzenia blokowe i znakowe.

5.2 Niskopoziomowe funkcje dostępu do plików

Niskopoziomowe funkcje dostępu do plików zapewniają dostęp do plików regularnych, katalogów, łącz nazwanych, łącz nie nazwanych, gniazdek, urządzeń (porty szeregowe, równoległe). Ważniejsze niskopoziomowe funkcje dostępu do plików podaje poniższa tabela. Są one szczegółowo opisane w manualu.

Nr	Funkcja	Opis
1	`open`	Otwarcie lub utworzenie pliku
2	`creat`	Tworzy pusty plik
3	`read`	Odczyt z pliku
4	`write`	Zapis do pliku
5	`lseek`	Pozycjonowanie bieżącej pozycji pliku
6	`fcntl`	Ustawianie i testowanie różnorodnych atrybutów pliku
7	`fstat`	Testowanie statusu pliku
8	`close`	Zamknięcie pliku
9	`unlink, remove`	Usuwa plik
10	`lockf`	Blokada pliku

Tab. 5-1 Ważniejsze niskopoziomowe funkcje dostępu do plików

Podstawowy sposób dostępu do plików polega na tym że najpierw plik powinien być otwarty co wykonywane jest za pomocą funkcji `open`.

```
int open(char *path,int oflag,[mode_t mode])
```

`path` Nazwa pliku lub urządzenia
`oflag` Tryb dostępu do pliku – składa się z bitów – opis w pliku nagłówkowym <fcntl.h>
`mode` Atrybuty tworzonego pliku (prawa dostepu)

Funkcja powoduje otwarcie pliku lub urządzenia o nazwie wyspecyfikowanej w parametrze `path`. Otwarcie następuje zgodnie z trybem `oflag`. Funkcja zwraca deskryptor pliku (uchwyt) będący niewielką liczbą `int`. Uchwyt pliku służy do identyfikacji pliku w innych funkcjach systemowych np. funkcji `read(...)` i `write(...)`. Gdy plik nie jest już używany powinien być zamknięty za pomocą funkcji `close(...)`. Prosty program czytający plik tekstowy podany został poniżej.

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
```

```
int main(int argc, char *argv[]) {
  int fd,rd;
  char buf[80];
  if(argc < 2) return 0;
  fd = open(argv[1],O_RDONLY);
  if(fd<0) {
     perror("open");
     exit(0)'
  }
  do {
     rd = read(fd,buf,80);
     printf("%s",buf);
  } while(rd > 0);
  close(fd);
  return 1;
}
```

Przykład 5-1 Przykład wykorzystania niskopoziomowych funkcji we/wy do odczyt pliku tekstowego

5.3 Standardowa biblioteka wejścia / wyjścia - strumienie

Standardowa biblioteka wejścia / wyjścia rozszerza możliwości funkcji niskopoziomowych. Zapewnia ona wiele rozbudowanych funkcji ułatwiających formatowanie wyjścia i skanowania wejścia, obsługuje buforowanie. Należące do niej funkcje zadeklarowane są w pliku nagłówkowym stdio.h. Odpowiednikiem uchwytu jest strumień (ang. *stream*) widziany w programie jako FILE*. Ważniejsze funkcje należące do standardowej biblioteki wejścia wyjścia podaje poniższa tabela. Są one szczegółowo opisane w manualu.

Funkcja	Opis
`fopen, fclose`	Otwarcie lub utworzenie pliku, zamknięcie pliku
`fread`	Odczyt z pliku
`fwrite`	Zapis do pliku
`fseek`	Pozycjonowanie bieżącej pozycji pliku
`fgetc,getc, getchar`	Odczyt znaku
`fputc, putc, putchar`	Zapis znaku
`frintf, fprintf, sprintf`	Formatowane wyjście
`scanf, fscanf, sscanf`	Skanowanie wejścia
`fflush`	Zapis danych na nośnik

Tab. 5-2 Ważniejsze funkcje wysokiego poziomu dostępu do plików

Podstawowy sposób dostępu do plików polega na tym że najpierw plik powinien być otwarty co wykonywane jest za pomocą funkcji `fopen`.

```
FILE* fopen(char *path,char *tryb)
```

`path` Nazwa pliku lub urządzenia
`tryb` Tryb dostępu do pliku

Funkcja powoduje otwarcie pliku lub urządzenia o nazwie wyspecyfikowanej w parametrze `path`. Otwarcie następuje zgodnie z trybem tryb. Funkcja zwraca identyfikator strumienia który służy do identyfikacji pliku w innych funkcjach biblioteki. Prosty program czytający plik tekstowy podany został poniżej.

```
#include <stdio.h>
#define SIZE 80

int main() {
   int ile;
   FILE *f;
   char buf[SIZE];
   f = fopen("fread.c","r");
   if(f == NULL) { perror("fopen"); exit(0);}
   do {
    ile = fread(&buf,sizeof(buf),1,f);
    printf("%s\n",buf);
   } while(ile == 1);
   fclose(f);
   return 0;
}
```

Przykład 5-2 Przykład wykorzystania standardowej biblioteki we/wy do odczytu pliku

5.4 Blokady plików

Pliki mogą być odczytywane i zapisywane przez wiele współbieżnych procesów. Współbieżny dostęp do pliku musi być kontrolowany aby nie naruszyć jego spójności. W szczególności jak jeden proces odczytuje dane z pliku inny nie może do niego zapisywać. Na czas dostępu pliki powinny być blokowane. Jedną z możliwości jest zastosowanie funkcji lockf. Funkcja lockf jest obudową obszerniejszej funkcji **`fcntl`** i ułatwia z niej korzystanie.

Funkcja 5-1 **`lockf`** – zablokowanie dostepu do pliku	
`int lockf(int fd, int funkcja, int zakres)`	
`fd`	Uchwyt pliku
`funkcja`	Specyfikacja operacji: F_LOCK, F_ULOCK, F_TEST, F_TLOCK
`ile`	Zakres blokowanie

Funkcja zwraca:

`-1`	Gdy błąd
`>0`	Sukces

F_LOCK	Zablokuj dostęp do pliku na długości **`zakres`** od pozycji bieżącej
F_ULOCK	Zwolnij dostęp do pliku.
F_TEST	Testuj czy fragment pliku jest zablokowany przez inny proces
F_TLOCK	Testuj czy fragment pliku jest zablokowany przez inny proces. Gdy nie to zajmij plik

Przykład działania funkcji lockf podany jest poniżej.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[]){
     int res, fd;

    if ((fd = open("blokada2.c", O_RDWR)) == -1) {
        perror("open");
        exit(0);
    }
    printf("Proba zajecia blokady\n");
    res = lockf(fd,F_LOCK,0);
    if (res == -1) {
        perror("lockf - 1");
```

```
        exit(1);
    }
    printf("Blokada zajeta\n");
    printf("Nacisnij <RETURN> aby zwolnic blokade\n");
    getchar();
    res = lockf(fd,F_ULOCK,0);
    if (res == -1) {
        perror("lockf - 2");
        exit(1);
    }
    printf("Blokada zwolniona\n");
    close(fd);
    return 0;
}
```

Przykład 5-1 Ilustracja blokowania pliku, program blokada2.c

5.5 Zadania

5.5.1 Program kopiowania plików - funkcje niskiego poziomu

Napisz program `copy` który kopiuje pliki używając funkcji niskiego poziomu. Program ma być uruchamiany poleceniem `copy file1 file2` i kopiować podany jako parametr pierwszy plik `file1` na podany jako parametr drugi plik `file2`. Użyj w programie funkcji dostępu do plików niskiego poziomu: `open()`, `read()`, `write()`, `close()`. Znajdź opis tych funkcji w systemie pomocy. Program powinien działać według następującego schematu:

1. Utwórz bufor `buf` o długości 512 bajtów (tyle wynosi długość sektora na dysku).
2. Otwórz plik `file1`.
3. Utwórz plik `file2`.
4. Czytaj 512 bajtów z pliku `file1` do bufora `buf`.
5. Zapisz liczbę rzeczywiście odczytanych bajtów z bufora `buf` do pliku `file2`.
6. Gdy z `file1` odczytałeś 512 bajtów to przejdź do kroku 5.
7. Gdy odczytałeś mniej niż 512 bajtów to zamknij pliki i zakończ program.

5.5.2 Program kopiowania plików – użycie strumieni

Napisz program `fcopy` który kopiuje pliki używając funkcji standardowej biblioteki wejścia wyjścia. Program ma być uruchamiany poleceniem `fcopy file1 file2` i kopiować podany jako parametr pierwszy plik `file1` na podany jako parametr drugi plik `file2`. Użyj w programie funkcji dostępu do plików: `fopen()`, `fread()`, `fwrite()`, `fclose()`. Znajdź opis tych funkcji w systemie pomocy. Program powinien działać według następującego schematu:

1. Utwórz bufor `buf` o długości 512 bajtów (tyle wynosi długość sektora na dysku).
2. Otwórz plik `file1`.
3. Utwórz plik `file2`.
4. Czytaj 512 bajtów z pliku `file1` do bufora `buf`.
5. Zapisz liczbę rzeczywiście odczytanych bajtów z bufora `buf` do pliku `file2`.
6. Sprawdź funkcją feof czy wystąpił koniec pliku `file1` . Gdy nie przejdź do kroku 4. Gdy plik się skończył to zamknij pliki i zakończ program.

5.5.3 Listowanie atrybutów pliku

Napisz program `fstat` wyprowadzający na konsolę atrybuty pliku będącego parametrem programu. Wywołanie: `fstat nazwa_pliku`. Przykładowo:

```
$./fstat fstat
Plik: fstat
wielkosc :  7318 b
liczba linkow:    1
pozwolenia:       -rwxr-xr-x
link symboliczny: nie
```

W programie należy wykorzystać funkcję `int fstat(int file, struct stat fileStat)` oraz podane w tabeli maski bitowe. Pomogą one zidentyfikować prawa dostępu zwrócone przez element `fileStat.st_mode`.

Dalsze wyjaśnienia dotyczące znaczenia atrybutów pliku, makra i maski bitowe znaleźć można w manualu w opisie wywołania `fstat`.

Wartośc ósemkowa	Nazwa symboliczna	Pozwolenie na
0400	S_IRUSR	Odczyt przez właściciela
0200	S_IWUSR	Zapis przez właściciela
0100	S_IXUSR	Wykonanie przez właściciela
0040	S_IRGRP	Odczyt przez grupę
0020	S_IWGRP	Zapis przez grupę
0010	S_IXGRP	Wykonanie przez grupę
0004	S_IROTH	Odczyt przez innych użytkowników
0002	S_IWOTH	Zapis przez innych użytkowników
0001	S_IXOTH	Wykonanie przez innych użytkowników

Tab. 5-3 Specyfikacja niektórych bitów określających prawa dostępu do pliku

```
file = open(argv[1],O_RDONLY);
res = fstat(file,&fileStat);
…
printf( (S_ISDIR(fileStat.st_mode)) ? "d" : "-");
printf( (fileStat.st_mode & S_IRUSR) ? "r" : "-");
…
```

Przykład 5-3 Fragment programu podającego atrybuty pliku

5.5.4 Listowanie zawartości katalogu

Napisz program wyprowadzający na konsolę pliki zawarte w katalogu będącym parametrem programu. Wywołanie programu ma postać: `dir katalog`. W przypadku braku parametru katalog podawana ma być zawartość katalogo bieżącego. Dla plików mają być podane nazwa, wielkość, typ, prawa dostępu. Użyj funkcji `opendir(...)` i `readdir(...)` opisanych w manualu.

5.5.5 Równoległe znajdowanie liczb pierwszych – komunikacja przez wspólny plik

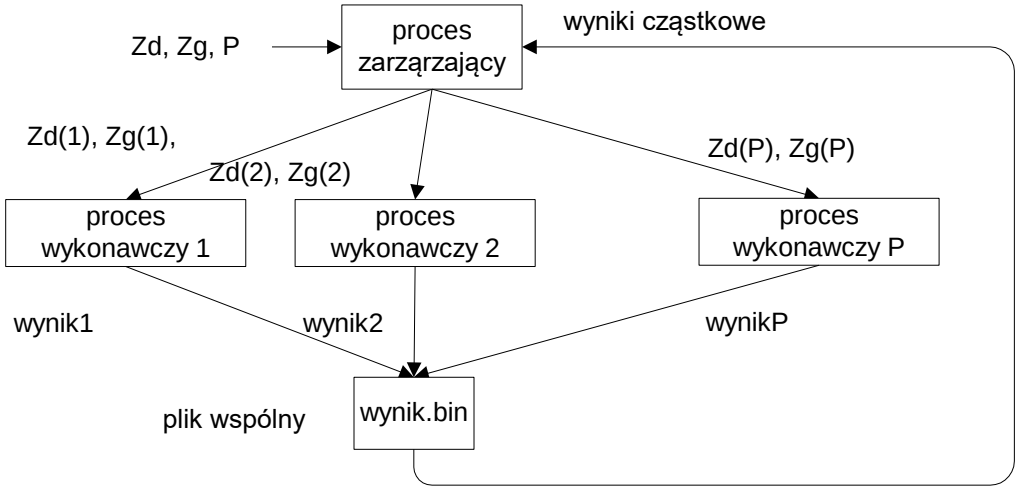
Napisz program który ma znajdować liczby pierwsze w zadanym przedziale `[Zd,...,Zg]`. Liczba jest pierwsza gdy dzieli się przez 1 i przez siebie samą. Prymitywny algorytm sprawdzania, czy dana liczba n jest liczbą pierwszą dany jest poniżej:

```
int pierwsza(int n)
// Funkcja zwraca 1 gdy n jest liczbą pierwsza 0 gdy nie
{ int i,j=0;
  for(i=2;i*i<=n;i++) {
    if(n%i == 0) return(0) ;
 }
 return(1);
}
```

Obliczenia można przyspieszyć dzieląc zakres `[Zd,...,Zg]` na `P` podprzedziałów `[Zd(1),...,Zg(1)]`, `[Zd(2),...,Zg(2)]`,..., `[Zd(P),...,Zg(P)]` gdzie `P` jest liczbą dostępnych procesorów. W każdym z podprzedziałów `[Zd(i),...,Zg(i)]` możemy znajdować liczby pierwsze niezależnie, co robi proces wykonawczy o nazwie `licz`. Tak więc o ile dysponujemy procesorem wielordzeniowym obliczenia wykonane mogą być równolegle. Wyniki pośrednie (liczba liczb pierwszych w przedziale) uzyskane przez poszczególne procesy wykonawcze mają być przekazane poprzez wspólny plik `wynik.bin`. Znalezioną liczbę liczb pierwszych każdy z procesów wykonawczych zapisuje w danej niżej strukturze.

```
struct {
   int pocz;// początek przedzialu
   int kon; // koniec przedzialu
   int ile; // Ile liczb w przedziale
} wynik;
```

Następnie struktura zapisywana jest do pliku `wynik.bin` za pomocą funkcji `write(fd,&wynik,sizeof(wynik))`. Po zakończeniu procesów wykonawczych proces macierzysty odczytuje z pliku poszczególne struktury za pomocą funkcji `read(fd,&wynik,sizeof(wynik))` i sumuje wyniki cząstkowe podając na końcu czas obliczeń i liczbę znalezionych liczb pierwszych.



Rys. 5-1 Znajdowanie liczb pierwszych – wiele procesów obliczeniowych

Zadanie powinno być rozwiązane w następujący sposób:

1. Program zarządzający tworzy pusty plik `wynik.bin`.
2. Program zarządzający dzieli przedział `[Zd,...,Zg]` na `P` podprzedziałów. Następnie tworzy procesy potomne używając funkcji `fork`. Każdy z tych procesów Pi dziedziczy z procesu macierzystego: granice przedziału `pocz=Zd(i), kon=Zg(i)`. Nie jest konieczne użycie funkcji execl.
3. Proces zarządzający czeka na zakończenie wykonawczych i odczytuje z pliku `wynik.bin` dane o znalezionych liczbach liczb pierwszych, oblicza sumę i czas obliczeń które mają być wyprowadzona na konsolę

Proces wykonawczy o numerze `i` znajduje liczby pierwsze w przedziale `[Zd(i),Zg(i)]`. Znalezioną liczbę liczb pierwszych zapisuje w pliku `wynik.bin`.

Program główny powinien mieć następujące argumenty:

- Zakres dolny przedziału Zd
- Zakres górny przedziału Zg
- Liczbę procesów wykonawczych P

Program ma podawać czas obliczeń - do jego pomiaru można użyć funkcji `time(NULL)`. Proszę narysować wykres pokazujący zależność czasu obliczeń od liczby procesów.

5.5.6 Blokowanie plików

Aby zapoznać się z blokowaniem pliku za pomocą blokady doradczej wykonaj kompilację podanego wyżej przykładowego programu blokada2.c. Następnie uruchom dwie kopie tego programu z różnych konsol. Zaobserwuj wyniki i przebieg blokowania.

5.5.7 Równoległe znajdowanie liczb pierwszych – komunikacja przez wspólny plik i użycie blokady pliku.

W zadaniu 5.5.5 procesy wykonawcze mogą próbować współbieżnego zapisu do tego samego pliku wynik.bin co może naruszyć jego integralność. Należy się przed takim przypadkiem zabezpieczyć używając funkcji:

```
int lockf(int fd, int funkcja, int zakres)
```

`fd` – Uchwyt pliku
`funkcja` – Specyfikacja operacji: `F_LOCK, F_ULOCK, F_TEST,F_TLOCK`
`ile` – Zakres blokowanie

6. Łącza nienazwane i nazwane

6.1 Łącza nienazwane

Najprostsza chyba metoda komunikacji miedzyprocesowej są łącza (*ang. pipe lub unnamed FIFO*). Łącze tworzy jednokierunkowy kanał komunikacyjny pomiędzy dwoma procesamii. Jeden z procesów może zapisywać bajty do łącza za pomocą funkcji `write`, podczas gdy drugi z procesów może je odczytywać korzystając z funkcji `read`. Komunikacja za pomocą łączy nienazwanych możliwa jest tylko dla procesów
pozostających w relacji macierzysty potomny. Łącze tworzy się za pomocą funkcji `pipe`.

```
#include <unistd.h>
int pipe(int fildes[2])
```

Wykonanie tej funkcji tworzy dwa deskryptory plików:
`fildes[0]` - deskryptor strumienia do czytania
`fildes[1]` - deskryptor strumienia do pisania

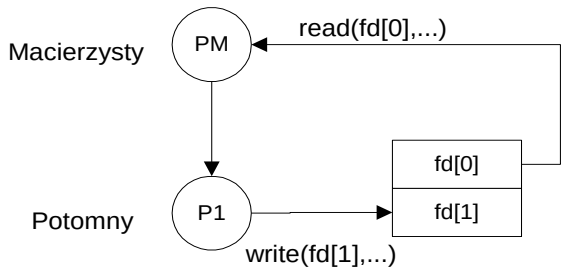
Funkcja zwraca:
0 gdy sukces
-1 gdy operacja się nie udała.

Nieużywany w procesie deskryptor musi być zamknięty (funkcją close). Deskryptory łącz utworzonych
przy pomocy funkcji pipe są dziedziczone przez proces potomny utworzony przez funkcje fork(). Próba czytania z pustego łącza powoduje zablokowanie procesu czytającego (domyślnie zmienna O_NONBLOCK nie jest ustawiona). Poniżej podano przykład procesów komunikujących się poprzez łącze.

```
// Program przykladowy zapis - odczyt z lacza
// Kompilacja: gcc lacze.c -o lacze
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

 int main() {
  int fd[2], child;
  char buf[] = "Programisci wszystkich krajow laczcie sie ! ";
  char buf2[64];
  /* Utworzenie lacza */
  pipe(fd);
  if ((child = fork()) == 0) {
       /* Proces  potomny - przesyła wiadomosc  macierzystego */
       close(fd[0]);
       write(fd[1], buf, sizeof(buf));
       close(fd[1]);
       exit(0);
  }
  /*  Proces macierzysty - odczytuje wiadomosc od potomka */
  close(fd[1]);
  read(fd[0], buf2, sizeof(buf));
  printf("%s\n", buf2);
  close(fd[0]);
  return 0;
}
```

Przykład 6-1 Program lacze.c - proces potomny przesyła wiadomość do macierzystego poprzez łącze



Rys. 6-1 Procesy komunikują się poprzez łącze nienazwane

6.2 Łącza nazwane

Gdy procesy nie pozostają w relacji macierzysty - potomny komunikacja przy pomocy łącz nienazwanych nie może być zastosowana. Należy zastosować wtedy łącza nazwane zwane inaczej plikami FIFO. Pliki FIFO są plikami specjalnymi. Posiadają takie atrybuty zwykłych plików jak nazwa, właściciel, grupa i prawa dostępu. Pliki FIFO różnią się tym od zwykłych plików że element odczytany jest z pliku usuwany. Pliki FIFO tworzy się przy pomocy funkcji `mkfifo`.

```
int mkfifo(char *name,mode_t mode, int flags)
```

`name` - nazwa pliku FIFO
`mode` - tryb otworzenia (np. S_IRUSR | O_CREAT)
`flags` - gdy plik jest tworzony to można mu nadać prawa dostępu (argument opcjonalny)

Funkcja zwraca: 0 – gdy sukces, -1 gdy błąd

Utworzony plik FIFO należy otworzyć za pomocą funkcji `open`. Zapis i odczyt następuje za pomocą funkcji `write` i `read`. Gdy używane są pliki FIFO wywołanie funkcji `open` może być blokujące.

Na plikach FIFO można także wykonywać operacje z poziomu powłoki. Plik FIFO tworzy się poleceniem:

```
mkfifo [-m mode] nazwa_pliku
mode – prawa dostępu
```

Dla przykładu utwórzmy plik FIFO o nazwie nowy i wyświetlmy zawartość katalogu bieżącego.

```
$mkfifo nowy
$ls -l nowy
prw-rw-rw  1 juka juka       0   Mar 30 19:25  nowy
```

Przykład 6-2 Tworzenie pliku FIFO przy pomocy polecenia `mkfifo`

Litera p na pierwszej pozycji wskazuje że nowy jest plikiem specjalnym typu FIFO. Można pisać i czytać z pliku FIFO posługując się standardowymi narzędziami systemu. Dla przykładu z pierwszej konsoli wydajmy poleceni jak niżej.

```
$ ls –l >  nowy
$
```

Przykład 6-3 Listowanie zawartości katalogu do pliku FIFO

Z drugiej konsoli wylistujmy zawartość tego pliku.

```
$cat < nowy
drwxrwxr-x  2 juka      juka           4096 Mar 30 19:39 .
drwxrwxr-x 20 juka      juka           8192 Nov 17 08:53 ..
prw-rw-rw-  1 juka      juka              0 Mar 30 19:37 nowy
-rw-rw-rw-  1 juka      juka            529 Mar 30 10:33 fifo_a.c
-rw-rw-rw-  1 juka      juka            510 Mar 30 10:33 fifo_b.c
```

Przykład 6-4 Listowanie zawartości pliku nowy typu FIFO

Podczas uruchamiania powyższych procesów zaobserwować synchronizację procesów. Polecenie `ls` będzie wstrzymane, do czasu uruchomienia polecenia `cat` na drugiej konsoli. Przykład procesów komunikujacych się przez pliki FIFO podano poniżej. Proces `mpipw` pisze znaki do pliku FIFO a proces `mpipr` czyta.

```
// Program przykladowy piszacy do pliku FIFO
// Czyta program mpipr
// Kompilacja gcc mpipw.c -o mpipw
 #include <stdio.h>
 #include <unistd.h>
 #include <stdlib.h>
 #include <sys/types.h>
 #include <sys/stat.h>
 #include <errno.h>
 #include <fcntl.h>

 int main() {
  int fdes,res;
  static char c;
  /* open a named pipe */
  printf("Zapis \n");
  if(mkfifo("FIFO",0666) < 0) {
       perror("mkfifo");
       // return 0;
  }
  fdes = open("FIFO",O_RDWR);
  if(fdes < 0) {
     perror("Open"); return 1;
  }
  printf("Plik otwarty %d\n",fdes);
  c = '0';
  do {
       sleep(1);
       res = write(fdes, &c, 1);
       if(res < 0)  perror("writing message");
       printf("W -->%c \n", c);
       c++;
  } while((res > 0) && (c < '9'));
  close(fdes);
  return 0;
}
```

Przykład 6-5 Program npipw.c - zapis do pliku FIFO

```
// Program przykladowy czytajacy z lacza MyPip, pisze program mpipw
// Kompilacja gcc mpipr.c -o mpipr
 #include <stdio.h>
 #include <unistd.h>
 #include <stdlib.h>
 #include <sys/types.h>
 #include <sys/stat.h>
 #include <errno.h>
 #include <fcntl.h>
```

```
 int main() {
  int fdes,res;
  static char c;
  printf("Program czytajacy \n");
  /* Utworz lacze */
  if(mkfifo("FIFO",0666) < 0) {
       perror("mkfifo");
  }
  // Otwarcie pliku
  fdes = open("FIFO",O_RDWR);
  if(fdes < 0) { perror("Open"); exit(1); }
  printf("Plik otwarty, fdes = %d \n",fdes);
  do {
     // Odczyt
     res = read(fdes, &c, 1);
     if(res < 0)  perror("read");
     printf("R -->%c \n", c);
     sleep(1);
  } while(res > 0);
  printf("Koniec\n");
  close(fdes);
  return 0;
}
```

Przykład 6-6 Program `npipr.c` – odczyt z pliku FIFO

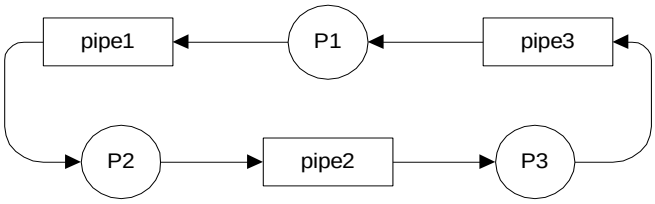
6.3 Zadania

6.3.1 Prosta komunikacja przez łącza nienazwane

Proszę napisać aplikację składającą się z procesów P1 i P2. Proces P2 jest procesem potomnym procesu P1. Proces P1 przekazuje co 1 sekundę do P2 kolejne liczby 1,2,...,10 które mają być wyświetlane przez P2.

6.3.2 Łącza nienazwane - Procesy modyfikują przekazywane dane

Proszę napisać aplikację składającą się z procesów P1,P2,P3. Proces P1 generuje kolejne liczby 1,2,..,10 i przekazuje je do P2 który dodaje do liczb 1 i przekazuje je do P3. P3 dodaje do otrzymanych liczb 1 zwraca je do P1. Proces P1 tworzy procesy P2 i P3 jako procesy potomne.



Rys. 6-2 Procesy komunikują się poprzez łącza nienazwane

6.3.3 Wykorzystanie funkcji popen

Wykorzystując funkcje `popen` napisz aplikację pobierającą listę procesów poleceniem `ps -ef` i wyprowadzającą ja na konsolę sortując po czasie zużycia procesora.

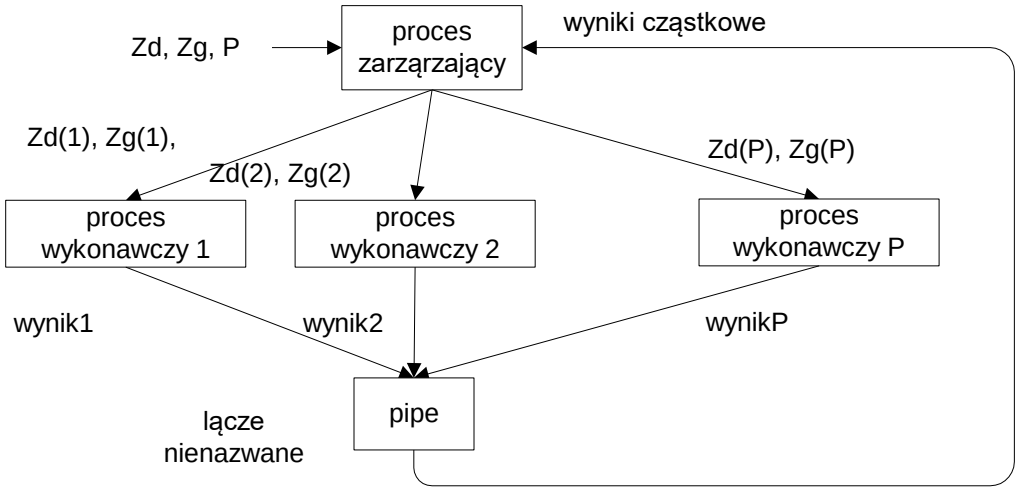
6.3.4 Znajdowanie liczb pierwszych, komunikacja poprzez łącze nienazwane

Napisz program który ma znajdować liczby pierwsze w zadanym przedziale [Zd,...,Zg]. Obliczenia można przyspieszyć dzieląc zakres [Zd,...,Zg] na P podprzedziałów [Zd(1),...,Zg(1)], [Zd(2),...,Zg(2)],..., [Zd(P),...,Zg(P)] gdzie P jest liczbą dostępnych procesorów. W każdym z podprzedziałów [Zd(i),...,Zg(i)] możemy znajdować liczby pierwsze niezależnie, co robi proces potomny tworzony funkcją `fork`. Zakres początkowy i końcowy obliczeń dla każdego z procesów obliczeniowych ma być przekazany na zasadzie dziedziczenia segmentu danych przez proces potomny. Gdy w procesie macierzystym zmienna x=5 to w procesie potomnym wartość tej zmiennej także będzie 5, jednak od tego momentu zmienne te będą już niezależne. Gdy dysponujemy procesorem wielordzeniowym obliczenia wykonane mogą być równolegle. Do komunikacji pomiędzy procesami wykonawczymi a macierzystym wykorzystamy łącze nienazwane.

Proces wykonawczy i znajduje liczby pierwsze w przedziale [Zd(i),...,Zg(i)]. Znalezioną liczbę liczb pierwszych zapisuje w danej niżej strukturze.

```
struct {
   int pocz;// początek przedzialu
   int kon; // koniec przedzialu
   int ile; // Ile liczb w przedziale
} wynik;
```

Następnie struktura zapisywana jest do łącza. Po zakończeniu procesów wykonawczych proces macierzysty odczytuje poszczególne struktury i sumuje wyniki cząstkowe podając na końcu czas obliczeń i liczbę znalezionych liczb pierwszych.



Rys. 6-1 Znajdowanie liczb pierwszych – wiele procesów obliczeniowych

Zadanie powinno być rozwiązane w następujący sposób:

1. Program zarządzający tworzy łącze nienazwane `fd[2]` za pomocą funkcji `pipe(fd)`.
2. Program zarządzający dzieli przedział [Zd,...,Zg] na P podprzedziałów. Następnie tworzy procesy potomne używając funkcji `fork`. Funkcja ta uruchamia proces wykonawczy. Każdemu z tych procesów Pi mają być przekazane: granice przedziału `pocz=Zd(i)`, `kon=Zg(i)` i numer procesu. Proces wykonawczy dziedziczy parametry `pocz` i `kon` i liczy ile w tym zakresie znalazł liczb pierwszych. Następnie zapisuje zakres i wynik do struktury wynik i strukturę zapisuje do łącza funkcją `write(fd[1],&wynik,sizeof(wynik))`.
3. Proces zarządzający czeka na zakończenie wykonawczych i odczytuje z łącza dane o znalezionych liczbach liczb pierwszych poprzez odczyt struktur z łącza za pomoca funkcji `read(fd[o],&wynik,sizeof(wynik))`. Dalej oblicza sumę która ma być wyprowadzona na konsolę i czas obliczeń.

Program ma podawać czas obliczeń - do jego pomiaru można użyć funkcji `time(NULL)`. Proszę narysować wykres pokazujący zależność czasu obliczeń od liczby procesów.

6.3.5 Łącza nazwane – problem producenta i konsumenta

Rozwiąż problem producenta / konsumenta za pomocą kolejek FIFO. Struktura komunikatu jest następująca:

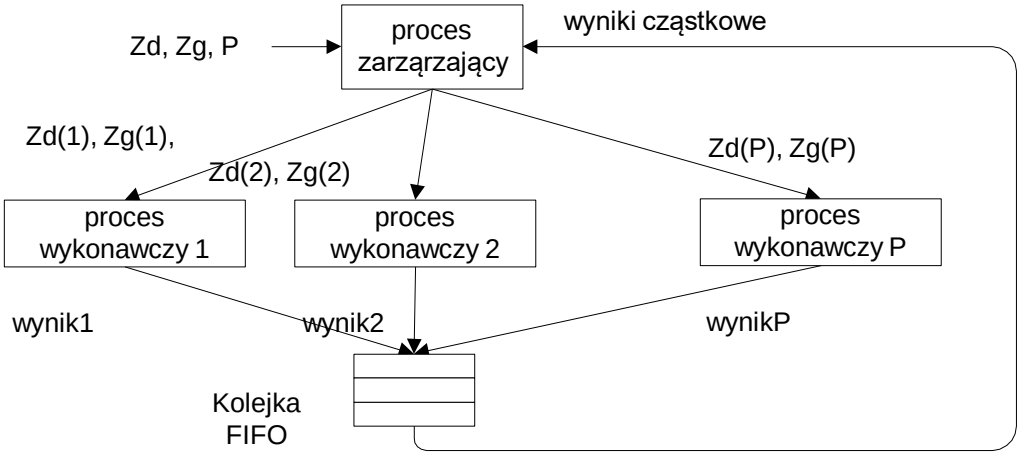
```
typedef struct {
  int from;
  char text[SIZE];
} mmsg_t;
```

Program producenta powinien z linii poleceń przyjmować nazwę kolejki i numer identyfikacyjny (umieszczany w polu from). Program konsumenta powinien z linii poleceń przyjmować nazwę kolejki. Uruchom program dla co najmniej 2 producentów i 2 konsumentów.

Uwagi:
- Proszę zaobserwować co się dzieje przy różnej kolejności uruchomiania programów.
- W jaki sposób można przesyłać komunikaty o zmiennej długości. Proszę spróbować rozwiązać to zagadnienie.

6.3.6 Znajdowanie liczb pierwszych

Napisz program który ma znajdować liczby pierwsze w zadanym przedziale [Zd,...,Zg]. Obliczenia można przyspieszyć dzieląc zakres [Zd,...,Zg] na P podprzedziałów [Zd(1),...,Zg(1)], [Zd(2),...,Zg(2)],..., [Zd(P),...,Zg(P)] gdzie P jest liczbą dostępnych procesorów. W każdym z podprzedziałów [Zd(i),...,Zg(i)] możemy znajdować liczby pierwsze niezależnie co robi proces wykonawczy o nazwie `licz`. Tak więc o ile dysponujemy procesorem wielordzeniowym obliczenia wykonane mogą być równolegle. Wyniki pośrednie (liczba liczb pierwszych w przedziale) uzyskane przez poszczególne procesy wykonawcze mają być przekazane do kolejki komunikatów.



Rys. 6-2 Znajdowanie liczb pierwszych – komunikacja poprzez kolejkę FIFO

Zadanie powinno być rozwiązane w następujący sposób:

1. Program zarządzający tworzy kolejkę FIFO
2. Program zarządzający dzieli przedział [Zd,...,Zg] na P podprzedziałów. Następnie tworzy procesy potomne używając funkcji `execl("./licz",... )`. Funkcja ta uruchamia proces wykonawczy o nazwie `licz`. Każdemu z tych procesów mają być jako argumenty przekazane granice przedziału Zd(i),Zg(i).. Tak więc, proces wykonawczy powinien mieć postać: `licz granica_dolna granica_górna`
3. Proces zarządzający czeka na zakończenie wykonawczych i odczytuje z pliku FIFO dane o
4. znalezionych liczbach liczb pierwszych, odczytuje te dane, oblicza sumę która ma być wyprowadzona na konsolę.

Proces wykonawczy i znajduje liczby pierwsze w przedziale [Zd(i),...,Zg(i)]. Znalezioną liczbę liczb pierwszych zapisuje w danej niżej strukturze.

```
struct {
   int pocz;// początek przedzialu
   int kon; // koniec przedzialu
   int ile; // Ile liczb w przedziale
} wynik;
```

Następnie struktura zapisywana jest do kolejki komunikatów. Program główny powinien mieć następujące argumenty:
- Zakres dolny przedziału
- Zakres górny przedziału
- Liczbę procesów wykonawczych

Program ma podawać czas obliczeń - do jego pomiaru można użyć funkcji `time(NULL)`.

6.3.7 Znajdowanie liczb pierwszych – wersja równoważąca obciążenia

Poprzednia wersja programu znajdowania liczb pierwszych nie pozwalała na poprawne rozwiązanie problemu równoważenia obciążenia. Znaczy to tyle że procesy przeszukujące przedziały zawierające mniejsze liczby kończyły działanie wcześniej pozostawiając procesor niewykorzystany. Należy więc rozwiązać problem dążąc do równomiernego wykorzystania procesorów. Można to osiągnąć to dzieląc wyjściowy przedział na mniejsze

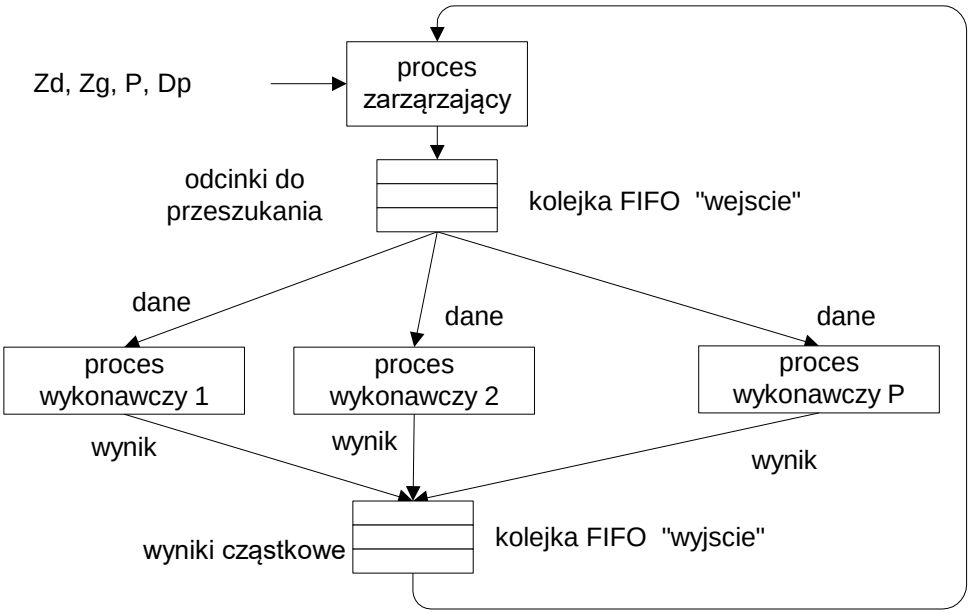
podprzedziały które będą następnie przekazane procesom wykonawczym poprzez kolejkę FIFO „wejscie”. Program uruchamiamy jak poniżej:

```
$pierwsze granica_dolna granica_górna liczba_procesow dlugosc_przedzialu
```

Do kolejki „wejscie” proces zarządzający wpisuje struktury:

```
struct {
   int pocz;  // Początek przedziału
   int kon;   // Koniec przedziału
   int numer; // Kolejny numer odcinka
} odcinek;
```

Proces wykonawczy oblicza ile liczb pierwszych mieści się w danym odcinku (pomiędzy `pocz` i `kon`) a następnie zapisuje wynik cząstkowy (w postaci struktury) do kolejki komunikatów o nazwie „wyjście”. Dalej program zarządzający odczytuje z kolejki wyniki cząstkowe i dokonuje ich sumowania otrzymując wynik końcowy.



Rys. 6-3 Znajdowanie liczb pierwszych – wersja równoważąca obciażenia

W algorytmie tym występują dwa problemy;

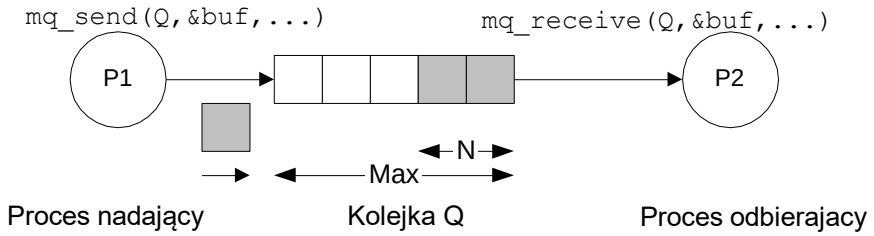
1. Jak powiadomić proces wykonawczy o tym że nie ma już danych i należy się zakończyć.
2. Jak zapobiec zablokowaniu aplikacji z powodu przepełnienia się kolejki wejściowej lub wyjściowej.

Problem 1 można rozwiązać wprowadzając specjalny odcinek np. dla którego elementy pocz i kon = 0. Problem drugi można rozwiązać tworząc oddzielny proces pobierający wyniki z kolejki wyjściowej. Należy wykonać eksperyment zmierzający do okreslenia optymalnej długości przedziału i liczby procesów.

7. Kolejki komunikatów POSIX

7.1 Wstęp

Kolejki komunikatów POSIX są wygodnym mechanizmem komunikacji międzyprocesowej działającym w obrębie jednego komputera. W przeciwieństwie do kolejek FIFO kolejki komunikatów zachowują strukturę komunikatu, można testować ich stan oraz mogą sygnalizować zmianę statusu z pustej na niepustą.



Rys. 7-1 Procesy P1 i P2 komunikują się za pomocą kolejki Q

Kolejkę komunikatów tworzy się za pomocą polecenia `mq_open()`.

Funkcja 7-1 `mq_open` – tworzenie kolejki komunikatów	
`mqd_t mq_open(char *name,int oflag, int mode, mq_attr *attr)`	
`name`	Nazwa kolejki komunikatów
`oflag`	Tryb tworzenia kolejki
`mode`	Prawa dostępu (`r` - odczyt, `w` - zapis) dla właściciela pliku, grupy i innych, analogiczne jak w przypadku plików regularnych.
`attr`	Atrybuty kolejki komunikatów lub NULL gdy domyślne

Aby użyć kolejki komunikatów należy zadeklarować zmienną typu `mqd_t` (kolejka komunikatów) i zmienną typu `mq_attr` (atrybuty kolejki komunikatów pokazane w Tabela 7-2). Następnie należy otworzyć kolejkę komunikatów używając funkcji `mq_open`. Z kolejkami komunikatów związane są następujące funkcje:

`mq_open()`	Tworzenie lub otwieranie kolejki komunikatów
`mq_send()`	Zapis do kolejki komunikatów
`mq_receive()`	Odczyt z kolejki komunikatów
`mq_getattr()`	Pobranie atrybutów i statusu kolejki.
`mq_setattr()`	Ustawienie atrybutów kolejki
`mq_notify()`	Ustalenie trybu zawiadamiania o zdarzeniach w kolejce.
`mq_close()`	Zamykanie kolejki komunikatów.
`mq_unlink()`	Kasowanie kolejki komunikatów

Tabela 7-1 Funkcje obsługi kolejek komunikatów

`long mq_maxmsg`	Maksymalna liczba komunikatów w kolejce.
`long mq_msgsize`	Maksymalna wielkość pojedynczego komunikatu.
`long mq_curmsg`	Aktualna liczba komunikatów w kolejce.
`long mq_flags`	Flagi
`long mq_sendwait`	Liczba procesów zablokowanych na operacji zapisu.
`long mq_recvwait`	Liczba procesów zablokowanych na operacji odczytu.

Tabela 7-2 Atrybuty `mq_attr` kolejki komunikatów

Aby użyć kolejek komunikatów należy do programu dołączyć plik nagłówkowy `<mqueue.h>`.
Podczas kompilacji należy dołączyć bibliotekę rt a więc użyć opcji –lrt. Na przykład:
`gcc program.c -o program -lrt.`

Uwaga!

- W standardowej dystrybucji Ubuntu nazwa kolejki komunikatów powinna się zaczynać od znaku "/"
- Długość kolejki komunikatów nie może przekraczać 8 (`mq_maxmsg <= 8`)

Poniżej pokazany został przykład użycia kolejki komunikatów. Program `odbior` tworzy kolejkę komunikatów o nazwie `Kolejka` i czeka na komunikaty. Po uruchomieniu programu można sprawdzić czy kolejka została utworzona pisząc na konsoli polecenie:

```
$ odbior &
$ ls -l
nrw-rw---- 1 juka   juka 0 Apr 27 15:45 Kolejka
```

Kody źródłowy programów odbierania i wysyłania pokazano poniżej.

```
//-----------------------------------------------------------------------------
// Proces wysylajacy komunikaty do kolejki POSIX
// Odbiera program mq_rcv
// Kompilacja:   gcc mq_snd.c -o mq_snd -lrt
// Uruchomienie: ./mq_snd numer
#include <stdio.h>
#include <mqueue.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define  SIZE     80
#define  MQ_NAME "/Kolejka"
 struct {
      int  typ;        // Typ komunikatu
      char text[SIZE];  // Tekst komunikatu
  } msg;

int  main(int argc, char *argv[]) {
   int i, res, num=0;
   unsigned int prior;
   mqd_t  mq;
   struct mq_attr attr;
   char kname[40];
   prior = 10;
   if(argc < 2) { printf("Uzycie: mq_send numer\n"); exit(0); }
   num = atoi(argv[1]);
   if(argc >= 3) strcpy(kname, argv[2]); else strcpy(kname,MQ_NAME);
   printf("Kolejka: %s\n",kname);
   // Utworzenie kolejki komunikatow ----------------
   attr.mq_msgsize = sizeof(msg);
   attr.mq_maxmsg = 4;
   attr.mq_flags = 0;
   mq=mq_open(kname , O_RDWR | O_CREAT , 0660, &attr );
   if( mq == -1 ) {  perror("Kolejka "); exit(0);  }
   printf("Kolejka: %s otwarta, mq: %d\n", kname,mq);
   for(i=0; i < 10 ;i++) {
     sprintf(msg.text,"Proces %d komunikat %d",num,i);
     msg.typ = num;
     res = mq_send(mq,(char *)&msg,sizeof(msg),prior);
     if (res == -1 ) { perror("Blad zapisu do mq"); continue; }
     printf("Wyslano: %s\n",msg.text);
     sleep(1);
  }
  mq_close(mq);
  return 0;
}
```

Przykład 7-1 Proces `mq_snd.c` wysyłający komunikaty do kolejki

```
// Proces odbierajacy komunikaty z kolejki
// Kompilacja gcc mq_rcv.c -o mq_rcv -lrt
#include <stdio.h>
#include <mqueue.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#define  SIZE     80
#define  MQ_NAME "/Kolejka"
 struct {
        int  typ;        // Typ komunikatu
        char text[SIZE];  // Tekst komunikatu
 } msg;

int main(int argc, char *argv[]) {
 int i, res, num=0;
 unsigned int prior;
 mqd_t  mq;
 struct mq_attr attr;
 prior = 10;
 if(argc < 2) {
    printf("Uzycie: mq_rcv numer\n");
    exit(0);
 }
 num = atoi(argv[1]);
 // Utworzenie kolejki komunikatow ----------------
 attr.mq_msgsize = sizeof(msg);
 attr.mq_maxmsg = 4;
 attr.mq_flags = 0;
 mq=mq_open(MQ_NAME , O_RDWR | O_CREAT , 0660, &attr );
 if( mq == -1 ) {   perror("Kolejka "); exit(0); }
 printf("Kolejka utworzona: %d \n",mq);
 for(i=0; i < 10 ;i++) {
    res = mq_receive(mq,(char *)&msg,sizeof(msg),&prior);
    if (res == -1 )  perror("Blad odczytu z mq");
    else  printf("Odebrano: %s\n",msg.text);
    sleep(1);
  }
  mq_close(mq);
  return 0;
}
```

Przykład 7-2 Proces mq_rcv.c odbierający komunikaty z kolejki

7.2 Zadania

7.2.1 Rozwiązanie problemu producenta i konsumenta za pomocą kolejek komunikatów

Należy rozwiązać problem producenta i konsumenta używając mechanizmu kolejek komunikatów. Należy napisac będą procesy:

`init - proces inicjujący kolejkę komunikatów`

`prod nr_prod kroki` - producent komunikatów, może być wiele kopii tego procesu

`kons kroki` - konsument komunikatów, może być wiele kopii tego procesu

1. Postać przesyłanego komunikatu powinna być dana strukturą jak poniżej:

```
typedef struct  {
        int  type;            /* typ procesu:  1 PROD, 2 KONS  */
        int  pnr ;            /* numer procesu     */
        char text[SIZE];      /* tekst  komunikatu */
    } ms_type;
```

Definicja struktury powinna być zawarta w pliku nagłówkowym `common.h`

2. Procesy producenta `prod( )` powinien być napisany według poniższego wzoru:
   - Pobrać z linii poleceń swój numer nr i liczbę kroków
   - Utworzyć lub otworzyć kolejkę komunikatów.
   - Wykonywać w pętli następującą sekwencję instrukcji:

```
    for(i=0;i<10;i++) {
        msg.pnr = nr;
        msg.type = PROD;
        sprintf(msg.text,"Producent %d krok %d",nr,i);
        // Przeslanie komunikatu do kolejki
        res = mq_send(mq,&msg,sizeof(msg),priority);
        ......
        sleep(1);
    }
```

3. Proces konsumenta kons należy napisać podobnie jak proces producenta. Zamiast funkcji `mq_send` należy użyć funkcji `mq_receive`.

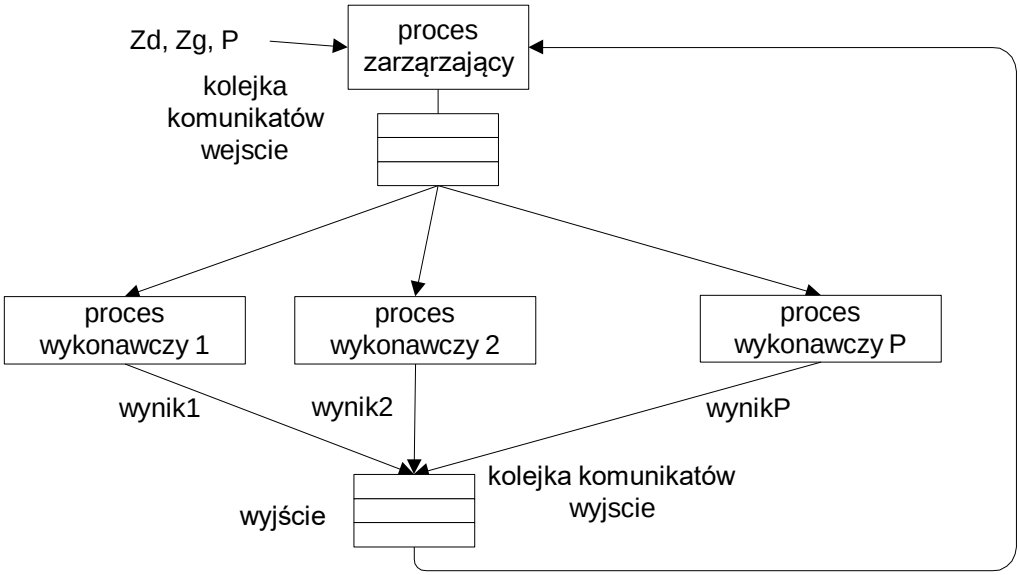
4. O długości kolejki decyduje parametr `attr.mq_maxmsg` który należy ustawić na zadana wartość,co ma robić proces `init`. W procesach wykorzystać funkcje `mq_getattr()` za pomocą której uzyskać można informacje o liczbie komunikatów w kolejce.

7.2.2 Znajdowanie liczb pierwszych w przedziale

Napisz program który ma znajdować liczby pierwsze w zadanym przedziale [Zd,...,Zg]. Obliczenia można przyspieszyć dzieląc zakres [Zd,...,Zg] na P podprzedziałów [Zd(1),...,Zg(1)], [Zd(2),...,Zg(2)],..., [Zd(P),...,Zg(P)] gdzie P jest liczbą dostępnych procesorów. W każdym z podprzedziałów [Zd(i),...,Zg(i)] możemy znajdować liczby pierwsze niezależnie co robi proces wykonawczy o nazwie `licz`. Tak więc o ile dysponujemy procesorem wielordzeniowym obliczenia wykonane mogą być równolegle. Dane wejściowe o podprzedziałach do procesów wykonawczych maja być przekazanie przez kolejkę komunikatów o nazwie „wejście" w której należy umieścić rekordy o strukturze danej poniżej (pole `liczb` niewykorzystane).

```
typedef  struct {
    int nr;         // numer przedzialu i
    int pocz;       // poczatek zakresu obliczen Zd(i)
    int kon;        // koniec zakresu obliczen Zg(i)
    int liczb;      // ile liczb pierwszych w przedziale
} msg_t;
```

Wyniki końcowe zapisane w strukturach `msg_t` mają być przekazane do kolejki komunikatów o nazwie „wyjście".



Rys. 7-1 Znajdowanie liczb pierwszych z użyciem kolejek FIFO

Zadanie powinno być rozwiązane w następujący sposób:

Proces zarządzający:

- Tworzy kolejki komunikatów POSIX „wejscie” i „wyjscie”
- Dzieli przedział [Zd,...,Zg] na P podprzedziałów. Następnie wpisuje do kolejki „wejscie” struktury typu `msg_t` zawierające kolejne podprzedziały 1,2,..,P.
- Tworzy procesy potomne używając funkcji `execl(”./licz”,... ).` Funkcja ta uruchamia procesy wykonawczy o nazwie `licz`.
- Czeka na zakończenie wykonawczych.
- Odczytuje z kolejki „wyniki” dane o znalezionych liczbach liczb pierwszych, oblicza sumę wszystkich liczb pierwszych (ma być wyprowadzona na konsolę) oraz czas obliczeń.

Proces wykonawczy

- Odczytuje z kolejki komunikatów „wejscie” zakres obliczeniowy pocz i kon.
- Znajduje liczby pierwsze w przedziale [pocz,kon] i ich sumaryczną ilość.
- Znalezioną liczbę liczb pierwszych oraz zakres obliczeń zapisuje w strukturze `msg_t`.
- Struktura zapisywana jest do kolejki komunikatów „wyjscie”.

Program główny powinien mieć następujące argumenty:

- Zakres dolny przedziału
- Zakres górny przedziału
- Liczbę procesów wykonawczych

Program ma podawać czas obliczeń a do jego pomiaru można użyć funkcji `time(NULL).`

7.2.3 Znajdowanie liczb pierwszych – wersja równoważąca obciążenia

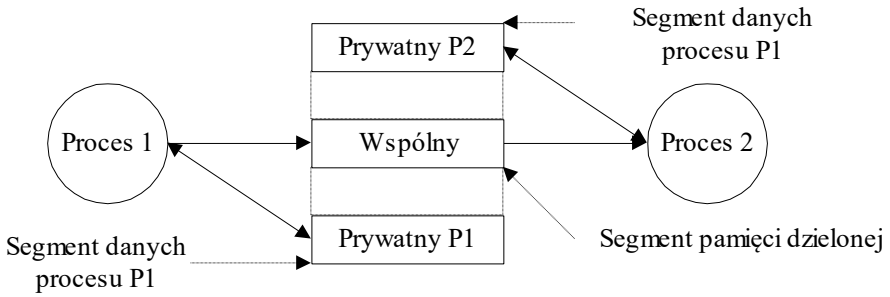
W poprzednim zadaniu obciążenie poszczególnych procesów wykonawczych jest nierównomierne wskutek czego pewne procesy kończą się szybciej a zwolniony procesor pozostaje niewykorzystany. Opracuj wersję programu równoważącą obciążenia procesorów. Można to osiągnąć dzieląc zakres obliczeń na niewielkie przedziały zapisywane następnie do kolejki wejściowej. Określ przyspieszenie programu w porównaniu z wersją poprzednią.

8. Pamięć dzielona i semafory

8.1 Pamięć dzielona

Jedną z możliwość komunikowania się procesów jest komunikacja przez pamięć dzieloną. Ta metoda komunikacji może być użyta gdy procesy wykonywane są na maszynie jednoprocesorowej lub wieloprocesorowej ze wspólną pamięcią. Nie ma natomiast zastosowania przy innych architekturach. Aby procesy mogły mieć wspólny dostęp do tych samych danych należy:

1. Utworzyć oddzielny segment pamięci.
2. Udostępnić dostęp do segmentu zainteresowanym procesom.



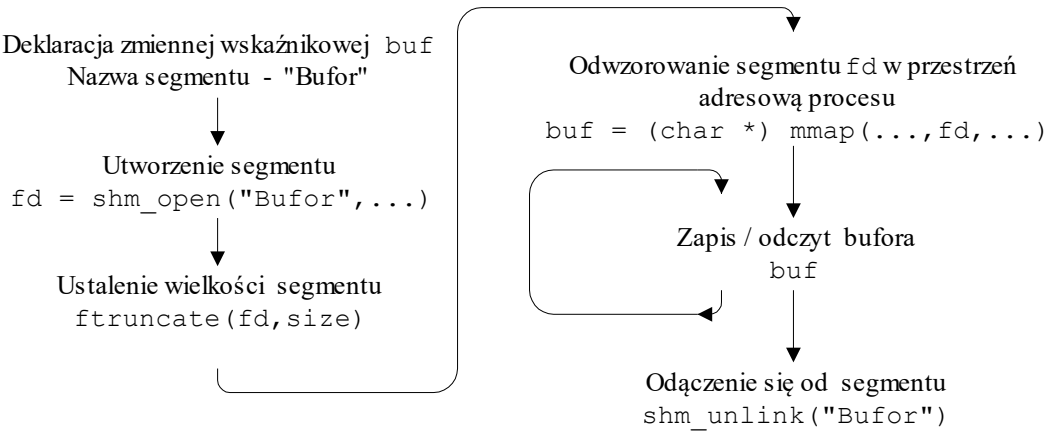
Rys. 8-1 Procesy P1 i P2 komunikuję się poprzez wspólny obszar pamięci

Standard Posix 1003.4 definiuje funkcje pozwalające na tworzenie i udostępnianie segmentów pamięci. Są to funkcje `shm_open()`, `ftruncate()`, `mmap()`, `munmap()` ,`mprotect()`, `shm_unlink`. Najważniejsze z funkcji podane są poniżej.

Opis	Funkcja
Tworzenie segmentu pamięci dzielonej	`shm_open()`
Ustalanie rozmiaru segmentu pamięci	`ftruncate()`
Odwzorowanie segmentu pamięci dzielonej w obszar procesu	`mmap()`
Odłączenie się od segmentu pamięci	`shm_unlink()`

Tabela 8-1 Funkcje operowania na pamięci dzielonej

Schemat utworzenia i udostępnienia segmentu pamięci dzielonej podano na poniższym rysunku.



Rys. 8-2 Schemat użycia segmentu pamięci dzielonej

Podany dalej Przykład 8-1 ilustruje sposób użycia segmentu pamięci dzielonej do wymiany danych pomiędzy procesami.

```
// Kompilacja gcc pam-dziel.c -o pam-dziel -lrt
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

#define SIZE 60   // Rozmiar bufora

 typedef struct {
    int typ;
    char text[SIZE];
 } buf_t;


main(int argc,char *argv[]) {
 int i,stat,pid,k,res;
 buf_t  *buf;
 char name[16];
 int  fd; // Deskryptor segmentu

 strcpy(name,"Bufor");
 shm_unlink(name);
 // Utworzenie segmentu pamieci  ---------------------
 if((fd=shm_open(name,O_RDWR|O_CREAT,0664))==-1) {
    perror("shm_open");
    exit(-1);
 }
 printf("fh = %d\n",fd);
 // Okreslenie rozmiaru obszaru pamieci ---------------
 res = ftruncate(fd,sizeof(buf_t));
 if(res < 0) { perror("ftrunc"); return 0; }

 // Odwzorowanie segmentu fd w obszar pamieci procesow
 buf = (buf_t *) mmap(0,sizeof(buf_t), PROT_READ|PROT_WRITE,MAP_SHARED, fd, 0);
 if(buf == NULL) { perror("mmap"); exit(-1); }
 printf("start\n");
 // Proces potomny P2 - pisze do pamieci wspolnej -----
 if(fork() == 0) {
     buf-> typ = 1;
     for(k=0;k<10;k++) { // Zapis do bufora
        printf("Zapis - Komunikat %d\n",k);
        sprintf(buf->text,"Komunikat %d",k);
        sleep(1);
     }
     exit(0);
 }
 // Proces macierzysty P1 czyta z pamięci wspólnej -
 for(i=0;i<10;i++) {
     printf("Odczyt %s\n",buf->text);
     sleep(1);
 }
 // Czekam na potomny --
 pid = wait(&stat);
 return 0;
}
```

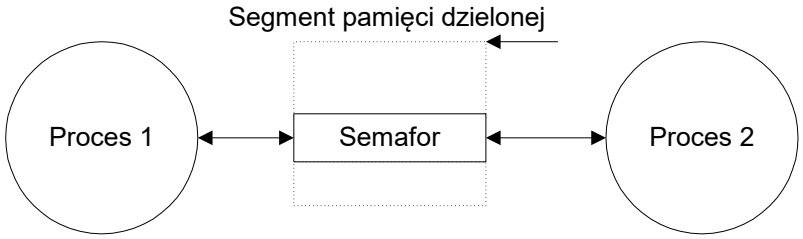
Przykład 8-1 Procesy P1 i P2 komunikują się przez wspólny obszar pamięci – program `pam-dziel.c`

8.2 Semafory

Standard POSIX definiuje dwa typy semaforów:

- Semafory nienazwane
- Semafory nazwane

Dostęp do semafora nienazwanego następuje po adresie semafora. Może on być użyty do synchronizacji procesów o ile jest umieszczony w pamięci dzielonej. Stąd nazwa semafor nienazwany. Inny typ semafora to semafor nazwany. Dostęp do niego następuje po nazwie.



Rys. 8-3 Semafor nienazwany umieszczony w pamięci dzielonej

Przed użyciem semafora nienazwanego musi on być zadeklarowany jako obiekt typu `sem_t` a pamięć używana przez ten semafor musi zostać mu jawnie przydzielona. O ile semafor nienazwany ma być użyty w różnych procesach powinien być umieszczony w wcześniej zaalokowanej pamięci dzielonej. Funkcje operujące na semaforach podaje Tabela 8-2.

Działanie	Funkcja
Utworzenie semafora nazwanego	`sem_open()`
Inicjacja semafora nienazwanego	`sem_init()`
Czekanie na semaforze	`sem_wait()`
Sygnalizacja na semaforze	`sem_post()`
Sygnalizacja warunkowa	`sem_trywait()`
Zamknięcie semafora nazwanego i nienazwanego	`sem_close()`
Zamknięcie semafora nazwanego	`sem_unlink()`
Skasowanie semafora nienazwanego	`sem_destroy()`

Tabela 8-2 Operacje na semaforach w standardzie POSIX 1003.1

Przed użyciem semafora nienazwanego trzeba:

1. Utworzyć segment pamięci za pomocą funkcji `shm_open()`.
2. Określić wymiar segmentu używając funkcji `ftruncate()`.
3. Odwzorować obszar pamięci wspólnej w przestrzeni danych procesu – `mmap()`.
4. Zainicjować semafor za pomocą funkcji `sem_init()`.

Aby użyć semafora nazwanego należy go otworzyć lub utworzyć (o ile nie istnieje) do czego wykorzystuje się funkcję `sem_open()`. Pobieranie i zwrot jednostek abstrakcyjnego zasobu następuje przez wykonanie funkcji semaforowych `sem_wait()` i `sem_post()`.

```
// Program producenta-konsumenta (C) J. Ulasiewicz 2019
// Demonstruje dzialanie pamieci dzielonej i semaforow
// Kompilacja gcc prod_kons.c -o prod_kons -lrt -lpthread
#include <sys/mman.h>
#include <fcntl.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>


#define BSIZE     4   // Rozmiar bufora
#define LSIZE    80   //  Dlugosc linii

typedef struct {
    char buf[BSIZE][LSIZE];
    int head;
    int tail;
    int cnt;
    sem_t mutex;
    sem_t empty;
    sem_t full;
} bufor_t;

int main(int argc,char *argv[]) {
 int i,stat,k, pid, size, fd,res;
 bufor_t  *wbuf ;
 char c;
 // Utworzenie segmentu ---------------------------
 shm_unlink("bufor");
 fd=shm_open("bufor", O_RDWR|O_CREAT , 0774);
 if(fd == -1){
   perror("open"); _exit(-1);
 }

 printf("fd: %d\n",fd);
 size = ftruncate(fd,sizeof(bufor_t));
 if(size < 0) {perror("trunc"); _exit(-1); }
 // Odwzorowanie segmentu fd w obszar pamieci procesow
 wbuf = (bufor_t *)mmap(0, sizeof(bufor_t)
         ,PROT_READ|PROT_WRITE,MAP_SHARED, fd, 0);
 if(wbuf == NULL) {perror("map");  _exit(-1); }

 // Inicjacja obszaru --------------------------------
 wbuf-> cnt = 0;
 wbuf->head = 0;
 wbuf->tail = 0;
 if(sem_init(&(wbuf->mutex),1,1)){
   perror("mutex"); _exit(0);
 }
 if(sem_init(&(wbuf->empty),1,BSIZE)) {
   perror("empty"); _exit(0);
 }
 if(sem_init(&(wbuf->full),1,0)) {
   perror("full"); _exit(0);
 }
 // Tworzenie procesow -------------
 if(fork() == 0) { // Producent
    for(i=0;i<10;i++) {
       // printf("Producent: %i\n",i);
```

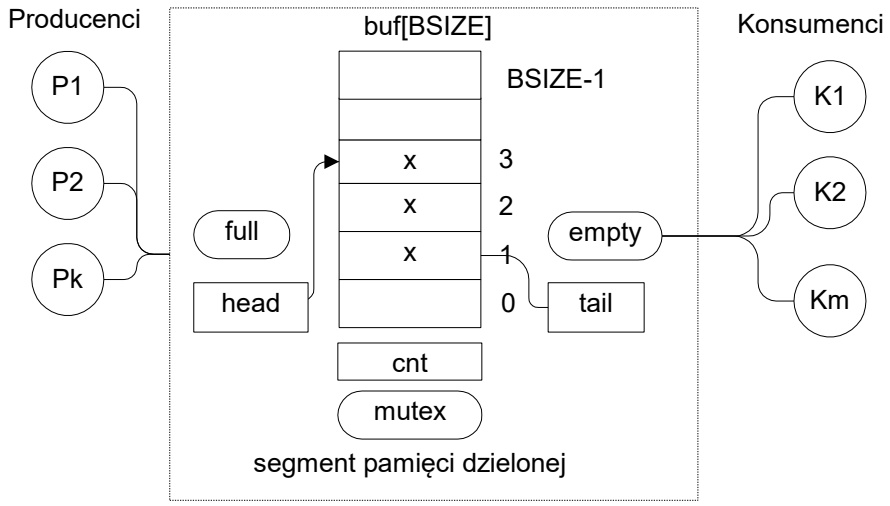
```
        sem_wait(&(wbuf->empty));
        sem_wait(&(wbuf->mutex));
        sprintf(wbuf->buf[wbuf->head],"Komunikat %d",i);
        printf("Producent - cnt:%d head: %d tail: %d\n",
                wbuf-> cnt,wbuf->head,wbuf->tail);
        wbuf-> cnt ++;
        wbuf->head = (wbuf->head +1) % BSIZE;
        sem_post(&(wbuf->mutex));
        sem_post(&(wbuf->full));
        sleep(1);
      }
      _exit(i);
  }
  // Konsument ------------------
  for(i=0;i<10;i++) {
      sem_wait(&(wbuf->full));
      sem_wait(&(wbuf->mutex));
      printf("Konsument - cnt: %d odebrano %s\n",wbuf->cnt
              ,wbuf->buf[wbuf->tail]);
      wbuf-> cnt --;
      wbuf->tail = (wbuf->tail +1) % BSIZE;
      sem_post(&(wbuf->mutex));
      sem_post(&(wbuf->empty));
      sleep(1);
  }
  pid = wait(&stat);
  sem_close(&(wbuf->mutex));
  sem_close(&(wbuf->empty));
  sem_close(&(wbuf->full));
  return 0;
}
```

Przykład 8-2 Program prod_kons.c - rozwiązanie problemu producenta i konsumenta za pomocą semaforów nienazwanych

8.3 Zadania

8.3.1 Problem producenta i konsumenta

Wzorując się na podanym wczesniej przykładzie rozwiąż problem producenta konsumenta implementując bufor cykliczny położony we wspólnym segmencie pamięci. W rozwiązaniu należy wprowadzić mechanizm wstrzymywania producenta (gdy bufor był pełny) i konsumenta (gdy bufor był pusty za pomocą semaforów nienazwanych, co prowadzi do prawidłowego rozwiązania problemu. Semafory powinny być położone w tym segmencie pamięci w którym umieszczony jest bufor.



Rys. 8-4 Bufor cykliczny w pamięci dzielonej

```
typedef struct {
    int head; // Tutaj producent wstawia nowy element
    int tail; // Stąd pobiera element konsument
    int cnt;  // Liczba elementów w buforze
    sem_t mutex; // Semafor chroniący sekcję krytyczną
    sem_t empty; // Semafor wstrzymujący producenta
    sem_t full;  // Semafor wstrzymujący konsumenta
    char buf[BSIZE][LSIZE];
} bufor_t
```

Należy napisać trzy programy:

- `Init` – program tworzący segment pamięci dzielonej, inicjujący liczniki i semafory.
- `producent nr_prod liczba_krokow` – program producenta o numerze nr_prod wykonującu zadaną liczbę kroków.
- `konsument nr_kons liczba_krokow` – program konsumenta o numerze nr_kons wykonującu zadaną liczbę kroków.

Procesy `init`, `producent` i `konsument` mają być niezależne – uruchamiane oddzielnie z konsoli. Program producenta ma wpisywać do bufora komunikaty postaci:

```
Producent nr_prod krok 1,
Producent nr_prod krok 2,
...
Producent nr_prod krok liczba_krokow,
```

Komunikaty te mają być odbierane z bufora przez proces konsumenta i wyświetlane na konsoli.

8.3.2 Problem czytelników i pisarzy

Problem czytelników i pisarzy polega na zorganizowaniu pracy czytelni. W czytelni przebywać może wielu czytelników ale tylko jeden pisarz. Poprawne rozwiązanie jest następujące.

- Wpuszczać na przemian czytelników i pisarzy
- Gdy wchodzi jeden z czytelników, to wpuszcza on wszystkich czekających czytelników

Rozwiązanie poprawne nie dopuszcza do zagłodzenia czy to czytelników czy pisarzy. Można przyjąć dla uproszczenia że w czytelni jest ograniczona liczba miejsc (PLACES). Proszę rozwiązać problem czytelników i pisarzy używając semaforów nazwanych. Można użyć danej niżej struktury i semaforów.

```
typedef struct {
    char text[SIZE];
    int odczyt; // Liczba odczytow
    int zapis;  // Liczba zapisow
} bufor_t

wolne; // Semafor nazwany odpowiadający liczbie wolnych miejsc w czytelni
wr;    // Semafor nazwany zapewniający obecność tylko jednego pisarze
```

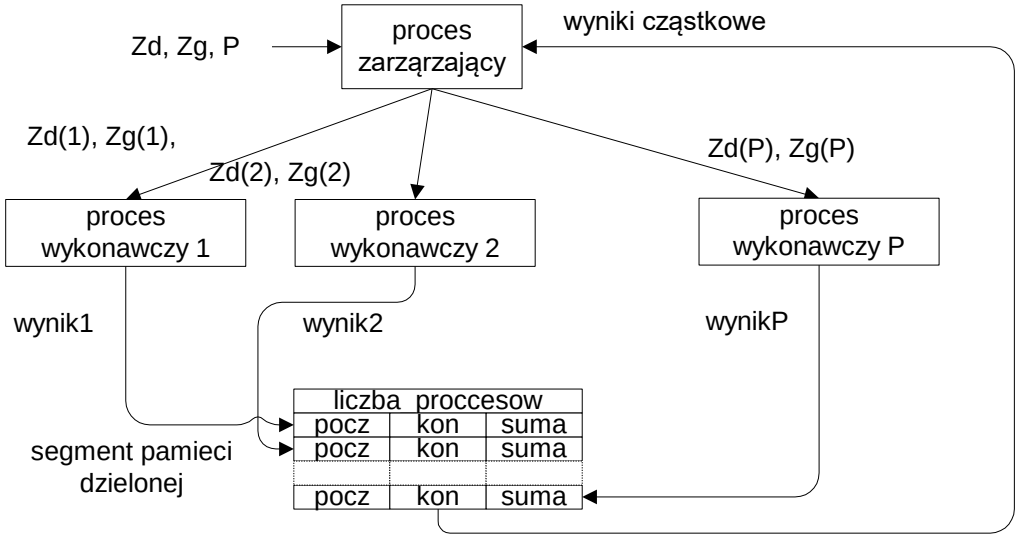
8.3.3 Szukanie liczb pierwszych w przedziale – komunikacja przez pamięć dzieloną

Napisz program który ma znajdować liczby pierwsze w zadanym przedziale [Zd,...,Zg]. Obliczenia można przyspieszyć dzieląc zakres [Zd,...,Zg] na P podprzedziałów [Zd(1),...,Zg(1)], [Zd(2),...,Zg(2)],..., [Zd(P),...,Zg(P)] gdzie P jest liczbą dostępnych procesorów. W każdym z podprzedziałów [Zd(i),...,Zg(i)] możemy znajdować liczby pierwsze niezależnie, co robi proces wykonawczy o nazwie `licz`. Tak więc o ile dysponujemy procesorem wielordzeniowym obliczenia wykonane mogą być równolegle. Wyniki pośrednie (liczba liczb pierwszych w przedziale) uzyskane przez poszczególne procesy wykonawcze mają być przekazane poprzez obszar pamięci dzielonej o nazwie "`bufor`". Każdy z procesów wykonawczych ma dodać do obszaru jeden rekord typu `dane_t` zawierający: początek zakresu obliczeń, koniec zakresu obliczeń i liczbę znalezionych w tym przedziale liczb pierwszych. Strukturę obszaru pamięci dzielonej podaje struktura `buf_t`.

```
typedef struct {
    int pocz;
    int kon ;
    int suma;
 } dane_t;
```

```
typedef struct {
  int wymiar;
  dane_t dane[SIZE];
 } buf_t;
```

Po zakończeniu procesów wykonawczych proces macierzysty odczytuje z poszczególne rekordy z obszaru pamięci dzielonej i sumuje wyniki cząstkowe podając na końcu czas obliczeń i liczbę znalezionych liczb pierwszych.



Rys. 8-5 Znajdowanie liczb pierwszych – komunikacja przez pamięć dzieloną

Zadanie powinno być rozwiązane w następujący sposób:

1. W programie zarządzającym deklarujemy w nim zmienną wskaźnikową `buf_t * buf` a nastepnie tworzy on segment pamięci dzielonej "`bufor`"o strukturze `buf_t`. Następnie ustalamy wielkość segmentu (funkcja `ftruncate`) i odwzorowujemy go w lokalną przestrzeń adresową (`buf = (bufor_t *) mmap(....))`.
2. Program zarządzający dzieli przedział `[Zd,...,Zg]` na P podprzedziałów. Następnie tworzy procesy potomne używając funkcji `fork`. Są to procesy wykonawcze. Każdy z tych procesów dziedziczy granice przedziału `pocz=Zd(i), kon=Zg(i),` numer procesu i, uchwyt segmentu pamięci dzielonej i wskaźnik `buf`. Nie

używamy tu funkcji `execl(...)`. Proces wykonawczy znajduje liczbę liczb pierwszych i wpisuje do odpowiedniego rekordu pamięci dzielonej zakres obliczeń i znalezioną liczbę liczb pierwszych .

```
buf->dane[i].pocz = pp;  buf->dane[i].kon = kp; buf->dane[i].suma = suma;
```

3. Proces zarządzający czeka na zakończenie wykonawczych i odczytuje z segmentu pamięci dzielonej dane o znalezionych liczbach liczb pierwszych a nastepnie oblicza sumę która ma być wyprowadzona na konsolę i czas obliczeń.

Program główny powinien mieć następujące argumenty:

- Zakres dolny przedziału
- Zakres górny przedziału
- Liczbę N procesów wykonawczych

W zadaniu tym zakładamy że liczba procesów N jest mniejsza od wymiaru SIZE tablicy z danymi. Program ma podawać czas obliczeń - do jego pomiaru można użyć funkcji `time(NULL)`. Proszę narysować wykres pokazujący zależność czasu obliczeń od liczby procesów.

8.3.4 Szukanie liczb pierwszych w przedziale – komunikacja przez pamięć dzieloną, ograniczony wymiar bufora.

Napisz program `lpierwsze` , będący modyfikacją programu poprzedniego, który ma znajdować liczby pierwsze w zadanym przedziale. Program ma być wywoływany z parametrami:

```
lpierwsze poczatek koniec N
```

Podobnie jak w poprzednim programie dane wejściowe do poszczególnych wątków maja być przekazywane przez tablicę `dane[SIZE]`. Załóż że wymiar tej tablicy jest mniejszy od liczby procesów czyli N < SIZE. Opracuj mechanizm zabezpieczenia przed nadpisaniem tablicy i użyj do tego celu semaforów.

9. Interfejs gniazd, komunikacja bezpołączeniowa

9.1 Adresy gniazd i komunikacja bezpołączeniowa

Jeżeli mające się komunikować procesy znajdują się na różnych komputerach do komunikacji może być użyty protokół TCP/IP wraz z interfejsem gniazdek BSD. Możliwe jest użycie jednego z dwóch styli komunikacji:

- Komunikacji bezpołączeniowej (datagramy) UDP
- Komunikacji połączeniowej TCP

Dokumentacja dotycząca gniazd zawarta jest w [16]. Różnice pomiędzy stylami komunikacji podaje [12]. W komunikacji UDP każdy komunikat adresowany jest oddzielnie a ponadto zachowywane są granice przesyłanych komunikatów. W komunikacji bezpołączeniowej stosowane są następujące funkcje:

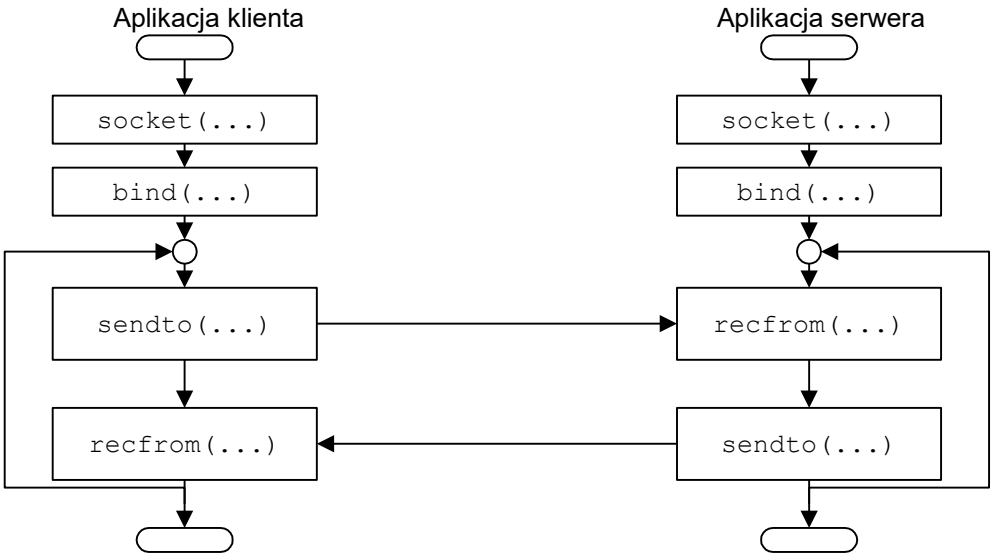
`socket`	Utworzenie gniazdka
`bind`	Powiązanie z gniazdkiem adresu IP
`sendto`	Wysłanie datagramu do odbiorcy
`recfrom`	Odbiór datagramu
`htons, htonl`	Konwersja formatu lokalnego liczby do formatu sieciowego (s – liczba krótka, s – liczba długa)
`ntohs, ntohl`	Konwersja formatu sieciowego liczby do formatu lokalnego (s – liczba krótka, s – liczba długa)
`close`	Zamknięcie gniazdka
`gethostbyname`	Uzyskanie adresu IP komputera na podstawie jego nazwy
`inet_aton`	Zamiana kropkowego formatu zapisu adresu IP na format binarny

Tabela 9-1 Ważniejsze funkcje używane w interfejsie gniazdek – komunikacja bezpołączeniowa

Sprawdź w dokumentacji ich parametry i znaczenia. Kolejność działań podejmowanych przez klienta i serwera podana jest poniżej a ich współpracę pokazuje Rys. 9-1.

<u>Klient:</u>
Tworzy gniazdko - `socket`
Nadaje gniazdku adres - `bind`
Nadaje lub odbiera dane - `sendto, recfrom,`

<u>Serwer:</u>
Tworzy gniazdko - `socket`
Nadaje gniazdku adres - `bind`
Nadaje lub odbiera dane - `sendto, recfrom`



Rys. 9-1 Przebieg komunikacji bezpołączeniowej

Dane poniżej przykłady mogą być użyte jako wzorce do budowania programów korzystających z komunikacji bezpołączeniowej.

```
// Proces odbierajacy komunikaty - wysyla udp_cli
// Wspolpracuje z udp_cli
// Kompilacja gcc udp_serw.c -o udp_serw -lrt
 #include <arpa/inet.h>
 #include <netinet/in.h>
 #include <stdio.h>
 #include <sys/types.h>
 #include <sys/socket.h>
 #include <unistd.h>
 #include <string.h>
 #define BUFLEN 80
 #define KROKI 10
 #define PORT 9950

 typedef struct {
  int typ;
  char buf[BUFLEN];
} msgt;

 void blad(char *s) {
     perror(s);
     _exit(1);
 }

int main(void) {
   struct sockaddr_in adr_moj, adr_cli;
   int s, i, slen=sizeof(adr_cli),snd, rec, blen=sizeof(msgt);
   char buf[BUFLEN];
   msgt msg;
   gethostname(buf,sizeof(buf));
   printf("Host: %s\n",buf);
   s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
   if(s < 0) blad("socket");
   printf("Gniazdko %d utworzone\n",s);
   // Ustalenie adresu IP nadawcy
   memset((char *) &adr_moj, 0, sizeof(adr_moj));
   adr_moj.sin_family = AF_INET;
   adr_moj.sin_port = htons(PORT);
   adr_moj.sin_addr.s_addr = htonl(INADDR_ANY);
   if (bind(s,(struct sockaddr *) &adr_moj, sizeof(adr_moj))==-1)
       blad("bind");
   // Odbior komunikatow ------------
   for (i=0; i<KROKI; i++) {
      rec = recvfrom(s, &msg, blen, 0,(struct sockaddr *) &adr_cli, &slen);
      if(rec < 0) blad("recvfrom()");
      printf("Odebrano komunikat z %s:%d res %d\n  Typ: %d %s\n",
            inet_ntoa(adr_cli.sin_addr), ntohs(adr_cli.sin_port),
            rec,msg.typ,msg.buf);
      // Odpowiedz -----
      sprintf(msg.buf,"Odpowiedz %d",i);
      snd = sendto(s, &msg, blen, 0,(struct sockaddr *) &adr_cli, slen);
      if(snd < 0) blad("sendto()");
      printf("wyslano odpowiedz -res %d\n",snd);
   }
   close(s);
   return 0;
}
```

Przykład 9-1 Proces odbierający komunikaty – serwer, udp_serw.c

```
// Proces wysyla a potem odbiera komunikaty udp
// Wspolpracuje z udp_serw
// Kompilacja gcc udp_cli.c -o udp_cli -lrt
 #include <netinet/in.h>
 #include <stdio.h>
 #include <sys/types.h>
 #include <sys/socket.h>
 #include <unistd.h>
 #include <string.h>
 #include <arpa/inet.h>
 #define BUFLEN 80
 #define KROKI 10
 #define PORT 9950
 #define SRV_IP "127.0.0.1"

typedef struct {
  int typ;
  char buf[BUFLEN];
} msgt;

  void blad(char *s) {
     perror(s);
     _exit(1);
   }

int main(int argc, char * argv[]) {
   struct sockaddr_in adr_moj, adr_serw, adr_x;
   int s, i, slen=sizeof(adr_serw), snd, blen=sizeof(msgt),rec;
   char buf[BUFLEN];
   msgt msg;
   s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
   if(s < 0) blad("socket");
   printf("Gniazdko %d utworzone\n",s);
   memset((char *) &adr_serw, 0, sizeof(adr_serw));
   adr_serw.sin_family = AF_INET;
   adr_serw.sin_port = htons(PORT);
   if (inet_aton(argv[1], &adr_serw.sin_addr)==0) {
        fprintf(stderr, "inet_aton() failed\n");
        _exit(1);
   }
```

```
   for (i=0; i<KROKI; i++) {
      msg.typ = 1;
      sprintf(msg.buf, "Wysylam komunikat %d", i);
      snd = sendto(s, &msg, blen, 0,(struct sockaddr *) &adr_serw,
      (socklen_t) slen);
      if(snd < 0) blad("sendto()");
      printf("Wyslano komunikat res: %d\n", snd);
      printf("Czekam na odpowiedz\n");
      rec = recvfrom(s, &msg, blen, 0,(struct sockaddr *) &adr_x,
      (socklen_t *) &slen);
      if(rec < 0) blad("recvfrom()");
      printf("Otrzymana odpowiedz %s\n",msg.buf);
      sleep(1);
   }
   close(s);
   return 0;
}
```

Przykład 9-2 Proces wysyłający komunikaty – klient, udp_klient.c.

Przykłady należy skompilować a następnie uruchomić w oddzielnych oknach tego samego komputera lub też na różnych komputerach. Program klienta uruchomić podając adres IP komputera na którym wykonywany jest program serwera.

```
$./udp_cli adres_ip_serwera
```

Gdy przykłady uruchamiamy lokalnie jako adres serwera podajemy 127.0.0.1. Gdy mamy dwa komputery o adresach IP: komputer klienta IP=192.168.0.158, komputer serwera IP=192.168.0.160 to najpierw na komputerze serwera uruchamiamy program serwera pisząc

```
$./udp_serw
```

Następnie na komputerze klienta uruchamiamy program

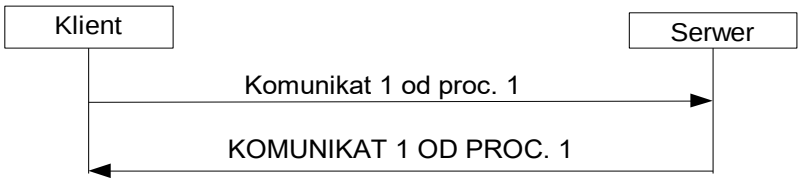
```
$./udp_cli 192.168.0.160
```

9.2 Zadania

9.2.1 Przesyłanie komunikatów pomiędzy niezależnymi procesami – zamiana małych liter na duże

Serwer odbiera komunikaty wysyłane przez klientów i odsyła napisy otrzymane w polu `text` ale zamienia małe litery na duże. Procesy klienta i serwera uruchamiane są niezależnie z linii poleceń.

```
typedef struct  {
   int  typ;        // typ komunikatu
   int  from;       // nr procesu który wysłał komunikat
   int  ile;        // ile było malych liter
   char text[SIZE]; // tekst komunikatu
 } mss_t;
```



Rys. 9-1 Współpraca klienta i serwera

<u>Proces serwera</u>
Serwer wykonuje następujące kroki:

- Utworzenie gniazdka
- Odbiór zleceń klientów.
- Odpowiedź na zlecenia klientów polegająca na zamianie małych liter na duże. W polu `ile` należy umieścić liczbę zamienionych liter.

- Co 10 sekund serwer ma wyświetlać informację o liczbie otrzymanych dotychczas komunikatów.

Proces klienta
Proces klienta uruchamiany jest z parametrem: adres IP węzła na którym uruchomiony jest klient (np. `klient 192.168.0.158`). Klient wykonuje następujące kroki:
- Utworzenie gniazdka
- Wysyłanie komunikatów do serwera. Pole `type` ma zawierać 1, pole `from` numer procesu, pole `text` łańcuch wprowadzany z konsoli
- Odbiór i wyświetlanie odpowiedzi serwera.

9.2.2 Klient i serwer usługi FTP

Napisz proces klienta i proces serwera realizujących:
- Przesyłanie plików od serwera do klienta
- Przesyłanie plików od klienta do serwera
- Listowanie zdalnego katalogu

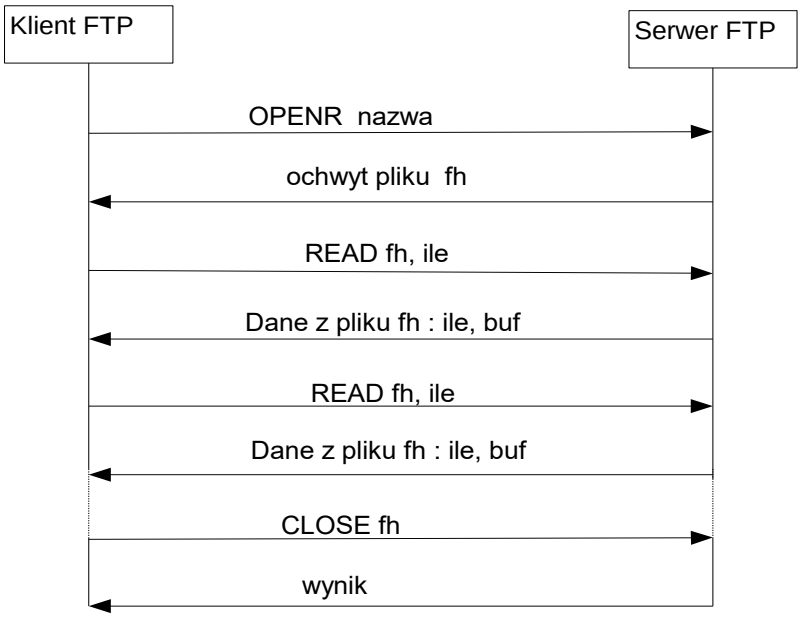
Od klienta do serwera przesyłane następujące rodzaje komunikatów:

```
#define OPENR  1  // Otwarcie pliku do odczytu
#define OPENW  2  // Otwarcie pliku do zapisu
#define READ   3  // Odczyt fragmentu pliku
#define CLOSE  4  // Zamkniecie pliku
#define WRITE  5  // Zapis fragmentu pliku
#define OPENDIR 6 // Otworz zdalny katalog
#define READDIR 7 // Czytaj zdalny katalog
#define STOP  10  // Zatrzymanie serwera
```

Format komunikatu przesyłanego pomiędzy klientem a serwerem jest następujący:

```
#define SIZE = 512 bajtów.
typedef struct  {
    int  typ;       // typ zlecenia
    int  ile;       // liczba bajtow
    int  fh;        // uchwyt pliku
    char buf[SIZE]; // bufor
} mms_t;
```

Serwer odbiera komunikaty wysyłane przez klienta i realizuje je. W poleceniu OPENR klient żąda podania pliku którego nazwa umieszczona jest w polu `buf`. Serwer otwiera ten plik umieszczając jego uchwyt w polu `fh`. Następnie klient żąda podania porcji pliku `fh` w buforze `buf` w ilości `ile` = SIZE. Plik sprowadzany jest fragmentami o długości SIZE. W polu `ile` ma być umieszczona liczba przesyłanych bajtów. Klient może wykryć koniec pliku gdy liczba rzeczywiście przesłanych bajtów ile jest mniejsza od żądanej. Po zakończeniu przesyłania pliku klient wysyła polecenie CLOSE `fh`.



Rys. 9-2 Współpraca klienta i serwera FTP

Procesy klienta i serwera uruchamiane są niezależnie z linii poleceń. Jako argument programów podajemy nazwę pod którą rejestruje się serwer. Przedstawiony wyżej serwer jest iteracyjnym serwerem bezstanowym. Jego szkic podano poniżej.

```
#define PORT 9950

main(int argc, char *argv[]) {
  mms_t msg;
  struct sockaddr_in adr_moj, adr_cli;
  int s,rec,snd,blen=sizeof(msg),slen;

  // Utworzenie i rejestracja nazwy -----------------
  s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
  if(s < 0) perror("socket");
  // Ustalenie adresu gniazdka serwera
  memset((char *) &adr_moj, 0, sizeof(adr_moj));
  adr_moj.sin_family = AF_INET;
  adr_moj.sin_port = htons(PORT);
  adr_moj.sin_addr.s_addr = htonl(INADDR_ANY);
  if (bind(s, &adr_moj, sizeof(adr_moj))==-1) perror("bind");

  do {
      // Odbior polecenia --------------
      rec = recvfrom(s, &msg, blen, 0, &adr_cli, &slen);
      if(rec < 0) blad("recvfrom()");
      printf("Odebrano komunikat %d\n",msg.typ);
      // Dekodowanie polecenia i realizacja
      switch (msg.typ) {
             case OPENR: msg.fh = open(msg.buf,O_RDONLY);
                   break;
             case READ:
                  msg.ile  = read(msg.fh,msg.buf,SIZE);
                 break;
             case CLOSE: ….
             case OPENW: ….
             case WRITE: ….
             case OPENDIR: ….
             case READDIR: ….
```

```
            case CLOSE: ….
       }
      // Wyslanie odpowiedzi ---------
      snd = sendto(s, &msg, blen, 0, &adr_cli, slen);
      if(snd < 0) perror("sendto");
  } while(msg.typ != STOP);
 }
```

Przykład 9-3 Szkic procesu serwera FTP

Klienta uruchamiamy podając jako argument adres IP serwera . Klient powinien wyświetlić proste menu zawierające pozycje:

- Pobranie pliku (z serwera do klienta)
- Przesłanie pliku (od klienta do serwera)
- Listowanie bieżącego katalogu zdalnego
- Zmiana katalogu zdalnego

Poniżej podano szkic kodu klienta. Na początku zrealizuj dwie pierwsze funkcje.

```
main(int argc, char *argv[]) {
  mms_t msg;
  ...
  s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
  ...
  do {  //Otwarcie pliku
        printf("Podaj nazwe pliku: ");
        gets(msg.buf);
        msg.typ = OPEN;
        snd = sendto(s, &msg, blen, 0, &adr_serw, slen);
        if(snd < 0) perror("sendto()");
        rec = recvfrom(s, &msg, blen, 0, &adr_moj, &slen);
        if(rec < 0) perror("recvfrom()");
  } while(msg.fh < 0);

do { // Odczyt --------------------
      msg.typ = READ;
      msg.ile = SIZE;
      snd = sendto(s, &msg, blen, 0, &adr_serw, slen);
      if(snd < 0) perror("sendto()");
      rec = recvfrom(s, &msg, blen, 0, &adr_moj, &slen);
      if(rec < 0) perror("recvfrom()");
      printf("Odebrano: %d bajtow\n",msg.ile);
      if(msg.ile > 0)   printf("%s\n",msg.buf);
        ...
  } while(msg.ile == SIZE);
```

Przykład 9-4 Szkic procesu klienta FTP

Rozszerzenia:

- Dodaj funkcję zapisu na dysku pliku który przesyłany jest od klienta do serwera
- Dodaj funkcję listowania zawartości katalogu którego nazwa podawana jest przez klienta
- Dodaj funkcję zmiany katalogu bieżącego
- Zrealizuj serwer jako serwer współbieżny

9.2.3 Szukanie liczb pierwszych w przedziale – komunikacja przez gniazdka

Napisz program który ma znajdować liczby pierwsze w zadanym przedziale [Zd,...,Zg]. Obliczenia można przyspieszyć dzieląc zakres [Zd,...,Zg] na P podprzedziałów [Zd(1),...,Zg(1)], [Zd(2),...,Zg(2)],..., [Zd(P),...,Zg(P)] gdzie P jest liczbą dostępnych procesorów. W każdym z podprzedziałów [Zd(i),...,Zg(i)] możemy znajdować liczby pierwsze niezależnie, co robi proces wykonawczy o nazwie `licz`. Tak więc o ile dysponujemy procesorem wielordzeniowym obliczenia

wykonane mogą być równolegle. Dane wejściowe dla procesów wykonawczych i wyniki pośrednie (liczba liczb pierwszych w przedziale) uzyskane przez poszczególne procesy wykonawcze mają być przekazane poprzez komunikaty UDP. Strukturę komunikatu podaje struktura `buf_t`.

```
typedef struct {
    int pocz;
    int kon ;
    int suma;
 } dane_t;
```

10. Interfejs gniazd, komunikacja połączeniowa

10.1 Komunikacja połączeniowa

W komunikacji połączeniowej najpierw należy utworzyć połączenie pomiędzy komunikującymi się procesami. Po nawiązaniu połączenia pomiędzy procesami można przesyłać bajty. W komunikacji połączeniowej stosowane są następujące funkcje:

`socket`	Utworzenie gniazdka
`bind`	Powiązanie z gniazdkiem adresu IP
`connect`	Próba nawiązania połączenia
`listen`	Ustalenie długości kolejki procesów oczekujących na połączenie
`accept`	Oczekiwanie na połączenie
`write, send`	Wysyłanie bajtów
`read, recv`	Odbiór bajtów
`close`	Zamknięcie gniazdka
`inet_aton`	Zamiana kropkowego formatu zapisu adresu IP na format binarny
`gethostbyname`	Uzyskanie adresu IP komputera na podstawie jego nazwy

Tabela 10-1 Ważniejsze funkcje używane w interfejsie gniazdek – komunikacja połączeniowa

Sprawdź w podręczniku ich parametry i znaczenia. Kolejność działań podejmowanych przez klienta i serwera podana jest poniżej a ich współpracę pokazuje Rys. 10-1.

Klient:

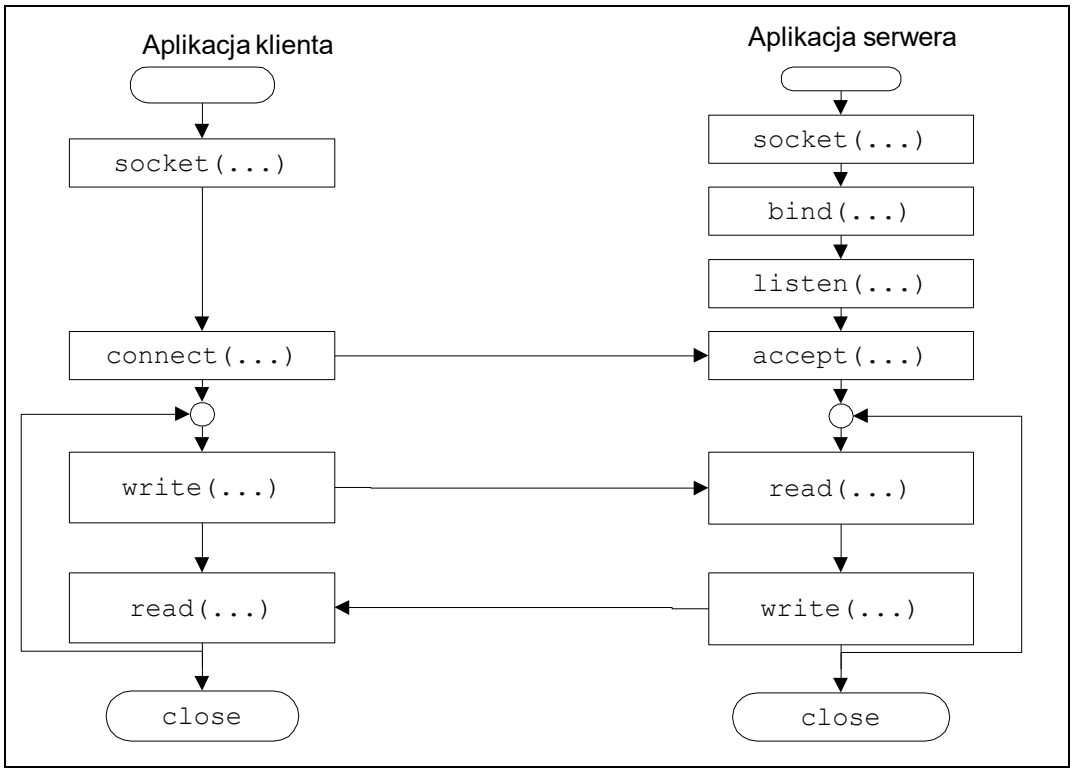
1. Tworzy gniazdko `socket`
2. Nadaje gniazdku adres `bind` (konieczne przy odbiorze)
3. Łączy się z serwerem `connect`
4. Nadaje lub odbiera dane `write, read, recv, send`

Serwer:

1. Tworzy gniazdko `socket`
2. Nadaje gniazdku adres `bind` (konieczne przy odbiorze)
3. Wchodzi w tryb akceptacji połączeń `listen`
4. Oczekuje na połączenia `accept`

Gdy połączenie zostanie nawiązane serwer wykonuje następujące czynności:

1. Tworzy dla tego połączenia nowe gniazdko
2. Nadaje lub odbiera dane - `write`, `read`, `recv`, `send`
3. Zamyka gniazdko



Rys. 10-1 Przebieg komunikacji z kontrolą połączenia

Przykład komunikacji połączeniowej dany jest poniżej. Program `tcp_serw.c` tworzy gniazdko, nadaje mu adres i przechodzi w tryb oczekiwania na połączenie. Gdy połączenie nadejdzie, odbiera bufor komunikatu i odpowiada na niego. Program serwera uruchamiamy poleceniem:

```
$./tcp_serw
```

Program klienta `tcp_cli` uruchamiamy poleceniem:

```
$./tcp_cli adres_serwera
```

Program klienta może być uruchomiony na tym samym komputerze co program serwera bądź na innym. Gdy uruchamiamy program klienta lokalnie nazwa serwera będzie `localhost` a gdy sieciowo nazwa serwera będzie jego adresem IP w postaci kropkowej bądź nazwą znakową (156.17.40.20 lub leo5). Program klienta `tcp_cli.c` tworzy gniazdko, nadaje mu adres i próbuje nawiązać połączenie z serwerem. Adres serwera pobierany jest z linii argumentów.

```
// Gniazdka - przyklad trybu polaczeniowego
// Program wspolpracuje z tcp_cli
// Uruchomienie: tcp_serwer
// Kompilacja gcc tcp_serw.c -o tcp_serw -lrt
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define MY_PORT 2000
#define TSIZE 32

typedef struct { // Komunikat
            int  typ;
            char tekst[TSIZE];
} komunikat_t;
```

```
void blad(char *s) {
     perror(s);
     _exit(1);
}

main() {
  int sock, msgsock, length;
  struct sockaddr_in server;
  int rval, res,i , cnt;
  komunikat_t msg;

  // Tworzenie gniazdka
  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock < 0) blad("Blad gniazdka");

  // Adres gniazdka
  server.sin_family = AF_INET;
  server.sin_addr.s_addr = INADDR_ANY;
  // server.sin_port = ntohs(MY_PORT);
  server.sin_port = htons(MY_PORT);
  if (bind(sock,(struct sockaddr *) &server, sizeof(server)))  blad("bind");
  // Uzyskanie danych poloczenia
  length = sizeof(server);
  if (getsockname(sock,(struct sockaddr *) &server, &length))
  blad("getsocketname");
  printf("Numer portu %d\n", ntohs(server.sin_port));

  // Start przyjmowania polaczen
  listen(sock, 5);
  do {
       printf("Czekam na polaczenie \n");
       msgsock = accept(sock, 0, 0);
       cnt = 0;
       if (sock == -1) perror("accept");
       else {
         printf("Polaczenie nawiazane - %d \n",sock);
         do { /* przesylanie bajtow ------------*/
           res = recv(msgsock,&msg,sizeof(msg),MSG_WAITALL);
           if(res < 0) blad("recv");
           if(res == 0) { printf("Rozlaczenie\n"); break; }
           printf("Otrzymano komunikat: %s\n",msg.tekst);
           cnt++;
           msg.typ = 1;
           sprintf(msg.tekst,"Komunikat %d",cnt);
           res = send(msgsock,&msg,sizeof(msg),0);
           if(res < 0) blad("send");
           printf("Odpowiedz wyslana - %d bajtow\n",res);
           sleep(1);
         } while (1);
         close(msgsock);
       }
  } while (1);
  printf("Koniec\n");
} /* Main */
```

Przykład 10-1 Serwer `tcp_serw.c` działający w trybie z kontrolą połączenia

```
// Program odbiera dane od programu tcp-serw
// uruchomionego na wezle addr. Uzywany port 2000
// Uruchomienie: tcp-client addr
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdlib.h>
#include <string.h>
#define MY_PORT 2000
#define TSIZE 32

typedef struct { // Komunikat
                int  typ;
                char tekst[TSIZE];
        } komunikat_t;

main(int argc, char *argv[]){
   int sock, cnt,res;
   struct sockaddr_in server;
   struct hostent *hp, *gethostbyname();
   komunikat_t msg;

   // Tworzenie gniazdka
   sock = socket(AF_INET, SOCK_STREAM, 0);
   if (sock < 0) {
       perror("Blad gniazdka");
       exit(1);
   }

   // Uzyskanie adresu maszyny z linii polecen
   server.sin_family = AF_INET;
   hp = gethostbyname(argv[1]);
   if (hp == 0) {
        printf("%s nieznany\n",argv[1]);
        exit(2);
   }
   memcpy(&server.sin_addr, hp->h_addr,
          hp->h_length);
   server.sin_port = htons(MY_PORT);

   // Proba polaczenia
   if (connect(sock,(struct sockaddr *) &server, sizeof(server)) < 0) {
        perror("Polaczenie"); exit(1);
   }
   printf("Polaczenie nawiazane\n");
```

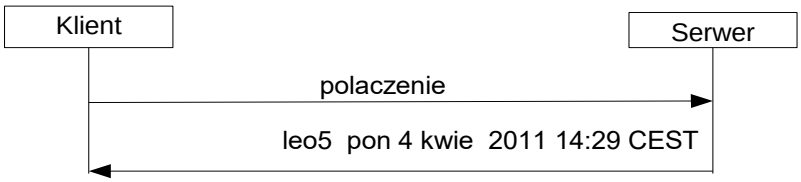
```
   // Petla odczytu ------------------------
   cnt = 0;
   memset(&msg,0,sizeof(msg));
   msg.typ = 1;
   do {
    sprintf(msg.tekst,"Komunikat %d",cnt);
    res = send(sock,&msg,sizeof(msg),0);
    printf("Zapis %d bajtow\n",res);
    cnt++;
    res = recv(sock,&msg,sizeof(msg),MSG_WAITALL);
    if(res < 0) { perror("Blad odczytu"); break; }
    if(res == 0) {
      printf("Polaczenie zamkniete");   break;
    }
    printf("Msg = %d Tekst = %s\n",cnt,msg.tekst);
   } while( cnt < 10  );
}
```

Przykład 10-2 Klient `tcp_cli.c` w trybie z kontrolą połączenia

10.2 Zadania

10.2.1 Uzyskiwanie czasu i nazwy zdalnego komputera

Napisz program klienta i serwera do uzyskiwania czasu i nazwy komputera wykorzystując tryb połączeniowy.



Rys. 10-1 Współpraca klienta i serwera

Proces serwera
1. Tworzy gniazdko AF_INET stylu SOCK_STREAM
2. Wiąże gniazdko do portu
3. Czeka na połączenie od klienta
4. Wpisuje do łąńcucha line swoja nazwę, date i czas bieżacy a następnie znak nowej linii.
5. Wysyła łańcuch line do klienta
6. Czeka 2 sekundy
7. Przechodzi do punktu 4

Proces klienta
Proces klienta uruchamiany jest z parametrem: adres IP węzła na którym uruchomiony jest serwer (np. `klient 192.168.0.158`). Klient wykonuje następujące kroki:

1. Utworzenie gniazdka AF_INET stylu SOCK_STREAM
2. Połączenie się z serwerem
3. Odbiór znaków aż do znaku nowej linii i ich wyświetlanie.
4. Sprawdzenie czy operator chce się rozłączyć
5. Gdy tak to się rozłączyć, gdy nie przejść do 3.

Procesy klienta i serwera uruchamiane są niezależnie z linii poleceń.

Do testowania serwera użyj narzędzia telnet. Uruchomienie:

```
$telnet adres_IP_serwera port
```

Przetestuj polecenie `netstat` aby zaobserwować nasłuchujący proces serwerowy i połączenie pomiędzy klientem i serwerem. Aby zaobserwować połaczenie musi ono być przez jakiś czas utrzymane.

10.2.2 Uzyskiwanie czasu i nazwy zdalnego komputera – wersja współbieżna

Napisz program klienta i serwera realizujących funkcję jak w poprzednim przykładzie. Serwer powinien być współbieżny to znaczy dla każdego połączenia należy utworzyć oddzielny proces.

10.2.3 Klient i serwer usługi FTP

Napisz proces klienta i proces serwera realizujących przesyłanie plików. Wykorzystaj połączeniowy wariant komunikacji pomiędzy procesami.

10.2.4 Komunikator internetowy

Napisz aplikację komunikatora znakowego działającego w trybie klient – serwer. Program powinien umożliwiać dwustronną komunikację terminalową.

Serwer:

```
Utworzenie gniazdka strumieniowego TCP w domenie internetu - f. socket
Nadaje gniazdku adres - ustalenie numeru portu f. bind.
Przejście do odbioru połączeń f. listen
do {
      akceptuje połączenia
      do {
       // Oczekuje na gotowość gniazdka sieciowego lub klawiatury
       select(....)
       gdy gotowa klawiatura {
              Odbiór znaków
              Wysłanie znaków do korespondenta
       }
       gdy gotowe gniazdko sieciowe {
              Odbierz znaki
              Wyswietl znaki na konsoli
       }
    } while(polaczenie);
 }
```

Przykład 10-3 Schemat działania serwera

Klient

```
Utworzenie gniazdka strumieniowego TCP w domenie internetu - f. socket
Ustalenie adresu serwera
Nawiązanie połączenia - f. connect
do {
  // Oczekuje na gotowość gniazdka sieciowego lub klawiatury
  select(....)
       gdy gotowa klawiatura {
              Odbiór znaków
              Wysłanie znaków do korespondenta
       }
       gdy gotowe gniazdko sieciowe {
              Odbierz znaki
              Wyswietl znaki na konsoli
       }
 } while(polaczenie);
```

Przykład 10-4 Schemat działania klienta

Jeżeli do odbioru znaków z klawiatury wykorzystamy funkcję `gets(...)` to do czasu naciśnięcia `Enter` proces nie będzie wyświetlał informacji przychodzącej. Jak rozwiązać ten problem? Przetestuj aplikację najpierw lokalnie a potem w sieci.

11. Sygnały i ich obsługa.

11.1 Wstęp

Sygnały są reprezentacja asynchronicznych i zwykle awaryjnych zdarzeń zachodzących w systemie w systemie. Listę obsługiwanych sygnałów można uzyskać pisząc na konsoli:

```
$kill -l
```

System obsługuje następujące sygnały (pominieto sygnały czasu rzeczywistego):

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL	5) SIGTRAP
6) SIGABRT	7) SIGBUS	8) SIGFPE	9) SIGKILL	10) SIGUSR1
11) SIGSEGV	12) SIGUSR2	13) SIGPIPE	14) SIGALRM	15) SIGTERM
16) SIGSTKFLT	17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO	30) SIGPWR
31) SIGSYS				

Sygnały mogą być generowane:
- przez system operacyjny, gdy wystąpi zdarzenie awaryjne,
- w programie za pomocą funkcji kill(), alarm() i raise(),
- z konsoli za pomocą polecenia kill.

Sygnał może być obsłużony przez program aplikacyjny. Funkcja systemowa `signal` pozwala na zainstalowanie procedury obsługi sygnału.

```
signal(int sig, void(*funct) (int))
```

Gdzie:

`sig` Numer sygnału

`funct` Nazwa funkcji obsługującej sygnał

Procedura `void funct(int)` wykonana będzie gdy pojawi się sygnał sig. W systemie pierwotnie zdefiniowane są dwie funkcje obsługi sygnałów:

SIG_DFL - akcja domyślna, powoduje zwykle zakończenie procesu,

SIG_IGN - zignorowanie sygnału (nie zawsze jest to możliwe).

Prosty program przechwytujący sygnał SIGINT generowany przy naciśnięciu klawiszy (Ctrl+Break) podano poniżej.

```
#include <signal.h>
#include <stdlib.h>
#include <setjmp.h>
int sigcnt = 0;
int sig = 0;

void sighandler(int signum) {
/* Funkcja obslugi sygnalu */
  sigcnt++;
  sig = signum;
}

void main(void) {
   int i =0;
   printf("Program wystartował \n");
   signal(SIGINT, sighandler);
   do {
     printf(" %d %d %d  \n",i,sigcnt,sig);
     sleep(1); i++;
   } while(1);
}
```

Przykład 11-1 Obsługa sygnału SGINT

11.2 Zadania

11.2.1 Obsługa sygnału SIGINT

Uruchom podany w Przykładzie 1 program. Sprawdź co stanie się przy próbie jego przerwania poprzez jednoczesne naciśnięcie klawiszy (Ctrl+Break).

11.2.2 Wprowadzanie hasła

Napisz program który wykonuje w pętli następujące czynności:
1. Ustawia czasomierz (funkcja alarm) na generacje sygnału za 5 sekund.
2. Wypisuje komunikat „Podaj hasło:" i próbuje wczytać łańcuch z klawiatury.
3. Gdy uda się wprowadzić hasło przed upływem 5 sekund, alarm jest kasowany i następuje wyjście z pętli.
4. Gdy nie uda się wprowadzić hasła w ciągu 5 sekund należy wyprowadzić napis: „Ponów próbę" i przejść do kroku 2.

11.2.3 Przesyłanie sygnałów pomiędzy procesami

Napisz dwa procesy – macierzysty i potomny. Proces macierzysty czeka w pętli na sygnał. Proces potomny generuje cykliczne sygnały (za pomocą funkcji kill).

11.2.4 Restarty procesu

Napisz program wykonujący restart po każdorazowej próbie jego przerwania wykonanej poprzez naciśnięcie klawiszy (Ctrl+Break). Naciśnięcie tej kombinacji klawiszy powoduje wygenerowanie sygnału SIGINT. W programie skorzystaj z funkcji `setjmp` i `longjmp`.

11.2.5 Implementacja funkcji alarm

Dokonaj próby samodzielnej implementacji funkcji `myalarm(int t)`. Wykonanie tej funkcji spowoduje wygenerowanie sygnału SIGUSR1 po upływie t sekund. Wykonanie tej funkcji z parametrem 0 ma spowodować odwołanie alarmu.

11.2.6 Implementacja przeterminowanie wysyłania komunikatu

W zadaniu dotyczącym komunikacji bezpołączeniowej proces klienta wysyłał komunikaty do procesu serwera. Dokonaj modyfikacji procesu klienta aby narzucić przeterminowanie T na wysłanie komunikatu. Rozwiąż zadanie dla przypadków:

a) Z użyciem funkcji `alarm` (T >= 1 sek).
b) Z użyciem timera (T – dowolny) .

12. Wątki

12.1 Tworzenie wątków

Aby wykorzystać możliwości wątków należy dysponować funkcjami umożliwiającymi administrowania wątkami. Zestaw operujących na wątkach funkcji zdefiniowany jest w pochodzącej z normy POSIX 1003 bibliotece `pthreads` (*ang. posix threads*) dostępnej w systemie QNX6 Neutrino. Prototypy operujących na wątkach funkcji zawarte są w pliku nagłówkowym `<pthread.h>`. Biblioteka `pthreads` zawiera następujące grupy funkcji:

1. Tworzenie wątków.
2. Operowanie na atrybutach wątków (ustawianie i testowanie).
3. Kończenie wątków.
4. Zapewnianie wzajemnego wykluczania.
5. Synchronizacja wątków.

Pierwsze trzy grupy funkcji zawierają mechanizmy do tworzenia wątków, ustalania ich własności, identyfikacji, kończenia oraz oczekiwania na zakończenie. Ważniejsze funkcje z tej grupy podaje Tabela 12-1.

Tworzenie wątku	`pthread_create()`
Uzyskanie identyfikatora wątku bieżącego	`Pthread_self()`
Kończenie wątku bieżącego	`pthread_exit()`
Oczekiwanie na zakończenie innego wątku	`pthread_join()`
Kończenie innego wątku	`pthread_cancel()`
Wywołanie procedury szeregującej	`pthread_yield()`

Tabela 12-1 Ważniejsze funkcje systemowe dotyczące tworzenia i kończenia wątków

Prosty program tworzący watki podaje Przykład 12-1.

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#define NUM_THREADS 2
#define KROKOW      4
pthread_t tid[NUM_THREADS]; // Tablica identyfikatorow watkow
int     wynik[NUM_THREADS]; // Tablica wynikow

void * kod(void *arg) {
   int numer = (int) arg;
   int i;
   for(i=0;i<KROKOW;i++) {
      printf("Watek: %d krok: %d \n",numer,i);
      sleep(1);
   }
   wynik[numer] = numer;
   pthread_exit((void*) &wynik[numer]);
}

int main(int argc, char *argv[]) {
   int i, status;
   void ** statp;
   for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i], NULL, kod,(void *)(i+1));
   for (i = 0; i < NUM_THREADS; i++) {
             pthread_join(tid[i], (void *) &statp);
             printf("Watek %d zakonczony\n",(int)*statp );
   }
   return  0;
}
```

Przykład 12-1 Program `watki_examp1.c` tworzący wątki

12.2 Synchronizacja wątków

Współbieżny dostęp do danych może naruszyć ich integralność. Aby zapewnić integralność należy zapewnić wzajemne wykluczanie w dostęp do wspólnych danych. Do zapewnienia wyłączności dostępu do danych stosuje się mechanizm muteksu (*ang. mutex*). Najważniejsze operacje na muteksach podaje Tabela 12-2.

Inicjacja muteksu	`pthread_mutex_init()`
Zajęcie muteksu	`pthread_mutex_lock()`
Zajęcie muteksu z przeterminowaniem	`pthread_mutex_timedlock()`
Próba zajęcia muteksu	`pthread_mutex_trylock()`
Zwolnienie muteksu	`pthread_mutex_unlock()`
Skasowanie muteksu	`pthread_mutex_destroy()`
Ustalanie protokołu zajmowania muteksu	`pthread_mutexattr_setprotocol()`
Ustalanie pułapu priorytetu	`pthread_mutexattr_ setprioceiling()`

Tabela 12-2 Ważniejsze funkcje operowania na muteksach

Do synchronizacji wątków stosuje się zmienne warunkowe. Najważniejsze operacje wykonywane na zmiennych warunkowych podaje Tabela 12-3.

Inicjacja zmiennej warunkowej	`pthread_cond_init()`
Zawieszenie wątku w kolejce	`pthread_cond_wait()`
Zawieszenie wątku w kolejce zmiennej warunkowej i czekanie z limitem czasowym	`pthread_cond_timedwait()`
Wznowienie wątku zawieszonego w kolejce	`pthread_cond_signal()`
Wznowienie wszystkich wątków zawieszonych w kolejce zmiennej warunkowej	`pthread_cond_broadcast()`
Skasowanie zmiennej warunkowej	`pthread_cond_destroy()`

Tabela 12-3 Najważniejsze operacje na zmiennych warunkowych

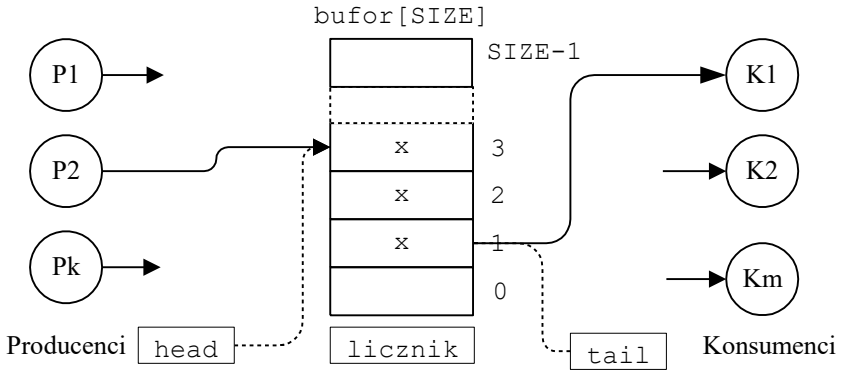
12.3 Zadania

12.3.1 Problem producenta i konsumenta

Rozwiąż pokazany na Rys. 12-1 problem producenta i konsumenta posługując się mechanizmem wątków. Bufor ma być tablicą `char buf[SIZE][LSIZE]` zawierająca napisy oraz wskaźniki `head` i `tail` oraz zmienną `licznik`. Wątek producenta ma wpisywać do bufora łańcuchy: „`Producent: i krok: k`” które mają być następnie pobierane przez konsumenta. Wpis następuje na pozycji `head`. Konsument pobiera zawartość bufora z pozycji `tail` i wyświetla ją na konsoli. Do synchronizacji użyj muteksu `mutex` oraz zmiennych warunkowych `empty` i `full`. Zmienne związane z obsługą bufora cyklicznego dane są poniżej.

```
#define SIZE 4          // Liczba pozycji (napisow) w buforze
#define LSIZE[80]       // Dlugosc napisu
char buf[SIZE][LSIZE];  // Bufor na napisy
int head;  // Tutaj wpisujemy do bufora nowy element
int tail;  // Stad pobieramy element z bufora
int cnt;   // Liczba elementow w buforze
pthread_cond_t empty;  // Tu czekamy gdy brak miejsca w buforze
pthread_cond_t full;   // Tu czekamy gdy brak rekordow w buforze
pthread_mutex_t mutex; // Wzajemne wykluczanie
```

Program należy uruchamiać podając liczbę producentów i konsumentów: `prodkons liczba_prod liczba_kons`.



Rys. 12-1 Problem producenta i konsumenta

12.3.2 Szukanie liczb pierwszych w aplikacji wielowątkowej

Napisz aplikację wielowątkową który ma znajdować liczby pierwsze w zadanym przedziale [Zd,...,Zg]. Jeżeli dysponujemy maszyną z procesorem wielordzeniowym obliczenia można istotnie przyspieszyć dzieląc obliczenia pomiędzy wątki. Podziel zakres [Zd,...,Zg] na P podprzedziałów [Zd(1),...,Zg(1)], [Zd(2),...,Zg(2)],..., [Zd(P),...,Zg(P)] gdzie P jest liczbą wątków. Wątek i ma znajdować liczby pierwsze w podprzedziale [Zd(i),...,Zg(i)]. Gdy dysponujemy maszyną wieloprocesorową to obliczenia wykonane mogą być równolegle. Program `pierwsze` powinien być uruchamiany z parametrami:

```
pierwsze Zd Zg liczba_watkow
```

Dane do wątków powinny być przekazywane jako elementy struktury:

```
typedef struct {
   int pocz;    // poczatek zakresu
   int kon;     // koniec zakresu
   int numer;   // numer watku
   int wynik;   // Ilosc liczb pierwszych dla przedzialu (wynik)
} par_t
```

Aby uniknąć problemów z przekazywaniem danych do poszczególnych wątków, umieść dane w tablicy:

```
#define MAXW 256  // Maksymalna liczba watkow
part_t dane[MAXW]; // Dane o zakresach obliczen i wyniki dla watkow
```

Dla wątku i zakres obliczeń i wyniki znajdują się na pozycji `dane[i]`. Wyniki działania wątków (liczba liczb pierwszych w zakresie) powinna być zwracana jako status zakończenia wątku. Przyjmij założenie że liczba watków jest mniejsza od MAXW. Porównaj szybkość działania tej metody szukania liczb pierwszych z metodą opartą o procesy.

12.3.3 Szukanie liczb pierwszych w aplikacji wielowątkowej – tablica z danymi mniejsza niż liczba wątków

Zwiększenie liczby wątków najpierw prowadzi do zmniejszenia czasu obliczeń a po przekroczeniu pewnej ich liczby czas obliczeń się zwiąksza. Dokonaj modyfikacji poprzedniego programu tak aby wielkość MAXW tablicy z danymi wejściowymi dla wątków była mniejsza niż liczba wątków. Wystąpi tu problem synchronizacji wątku głównego z wątkami roboczymi. Zaproponuj:

- Sposób uruchamiania watków
- Sposób przekazywania danych do wątków
- Sposób odbierania wyników od wątków i obliczania podsumowania.

Następnie napisz program implementujący tę koncepcję.

12.3.4 Szukanie liczb pierwszych w aplikacji wielowątkowej – równomierne obciążenie wątków

W poprzednim przykładzie wątki liczące wyższe zakresy liczb są bardziej obciążone i kończą się później. Dokonaj takiej modyfikacji programu aby problem ten był rozwiązany przy równomiernym obciążeniu procesorów. Można postępować w taki sposób że wątki robocze zgłaszają swoją gotowość wątkowi sterującemu a ten przekazuje im do obliczeń kolejne podprzedziały. Wykonaj eksperymenty obliczeniowe mierząc czas obliczeń dla tego samego zakresu obliczeń i różnej długości podprzedziału. Określ optymalną wielkość podprzedziału.

Katedra Informatyki Technicznej, Wydział Elektroniki Politechniki Wrocławskiej
ul. Janiszewskiego 11/17
50-372 Wrocław

dr inż. Jędrzej Ułasiewicz

Niniejszy raport otrzymują		Egz.
1	OINT – Biblioteka międzyinstytutowa I-6	1
2	Zakład Architektury Komputerów	1
3	Autor	1
RAZEM		3

Raport wpłynął do Redakcji w lipcu 2019 r.

Literatura

[1] Brian Ward, Jak działa Linux, Helion 2015.
[2] Robert Love, Linux Programowanie systemowe, Helion 2013.
[3] Ben-Ari M.; Podstawy programowania współbieżnego i rozproszonego, WNT Warszawa 1996.
[4] Fusco John, Linux niezbędnik programisty, Helion Gliwice 2009.
[5] Kernigan B, Ritchie D. Język ANSI C, WNT Warszawa 2002.
[6] K. Haviland, D. Gray, B. Salama; UNIX Programowanie systemowe, RM Warszawa 1999.
[7] Gabassi Michel, Dupoy Bertrand, Przetwarzanie rozproszone w systemie UNIX, Lupus, Warszawa 1995.
[8] The Gnu make manual, http://www.gnu.org/software/make/manual/make.html
[9] Matthew N. Stones R. Linux Programowanie,Wyd. RM Warszawa 1999.
[10] Manual systemu Linux: http://www.kernel.org/doc/man-pages
[11] Mitchell Mark, Oldham Jeffrey, Samuel Alex, LINUX programowanie dla zaawansowanych, Wydawnictwo RM Warszawa 2002.
[12] Stevens Richard W. ,Programowanie zastosowań sieciowych w systemie UNIX, WNT Warszawa 1996.
[13] J. Ułasiewicz, Systemy czasu rzeczywistego QNX6 Neutrino, wyd. BTC Warszawa 2007
[14] The GNU project debugger. http://www.gnu.org/software/gdb/documentation/
[15] David A. Wheeler Program Library HOWTO http://www.tldp.org/HOWTO/Program-Library-HOWTO/
[16] The GNU C Library – gniazdka, https://www.gnu.org/software/libc/manual/html_mono/libc.html#toc-Sockets-1