Na prawach rękopisu

INSTYTUT INFORMATYKI,
AUTOMATYKI I ROBOTYKI
POLITECHNIKI WROCŁAWSKIEJ
Raport serii Sprawozdania nr: / 2013

Programowanie aplikacji współbieżnych i rozproszonych w systemie Linux

Jędrzej UŁASIEWICZ

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

Wrocław 2013

PDF created with pdfFactory trial version www.pdffactory.com

Spis treści

1. Wstęp 3
2. Podstawy posługiwania się systemem Linux 4
2.1 Wstęp 4
2.2 Uzyskiwanie pomocy 4
2.3 Operowanie plikami i katalogami 6
2.4 Operowanie procesami 8
2.5 Zadania 10
3. Tworzenie i uruchamianie programów w języku C 11
3.1 Metoda elementarna – użycie edytor gedit i kompilatora gcc 11
3.2 Uruchamianie programów za pomocą środowiska CodeBlocks 12
3.3 Uruchamianie programów za pomocą środowiska Eclipse CDT 20
3.4 Uruchamianie programów za pomocą narzędzia make 28
4. Tworzenie procesów 30
4.1 Wstęp 30
4.2 Schemat użycia funkcji execl. 30
4.3 Zadania 31
5. Pliki 35
5.1 Podstawowa biblioteka obsługi plików 35
5.2 Niskopoziomowe funkcje dostępu do plików 35
5.3 Standardowa biblioteka wejścia / wyjścia - strumienie 36
5.4 Zadania 36
6. Łącza nienazwane i nazwane 41
6.1 Łącza nienazwane 41
6.2 Łącza nazwane 42
6.3 Funkcja select 43
6.4 Zadania 44
7. Kolejki komunikatów POSIX 48
7.1 Wstęp 48
7.2 Zadania 49
8. Pamięć dzielona i semafory 53
8.1 Pamięć dzielona 53
8.2 Semafory 55
8.3 Zadania 57
9. Interfejs gniazd, komunikacja bezpołączeniowa 61
9.1 Adresy gniazd i komunikacja bezpołączeniowa 61
9.2 Zadania 64
10. Interfejs gniazd, komunikacja połączeniowa 69
10.1 Komunikacja połączeniowa 69
10.2 Zadania 73
11. Sygnały i ich obsługa 76
11.1 Wstęp 76
11.2 Zadania 77
12. Wątki 78
12.1 Tworzenie wątków 78
12.2 Synchronizacja wątków 78
12.3 Zadania 79
13. Zdalne wywoływanie procedur 81
13.1 Podstawy 81
13.2 Zadania 84
14. System pogawędki sieciowej - IRC 85
14.1 Sformułowanie problemu 85
14.2 Wymagania 85
14.3 Format komunikatów 85
Literatura 86

PDF created with pdfFactory trial version www.pdffactory.com

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 CodeBlocks 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

PDF created with pdfFactory trial version www.pdffactory.com

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
`file nazwa_pliku`	Uzyskanie informacji o typie podanego pliku
Katalog `/usr/share/doc`	W katalogu tym zawarta jest dokumentacja dla różnych programów, pakietów i modułów
Internet	Witryna http://kernel.org/doc/manpages 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

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),

PDF created with pdfFactory trial version www.pdffactory.com

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 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.

2.2.3 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 (3posix)- open a file
```

2.2.4 Narzędzie apropos

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

```
$man słowo_kluczowe
```

Uzyskane strony podręcznika można następnie wyświetlić za pomocą narzędzia `man`.

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

PDF created with pdfFactory trial version www.pdffactory.com

2.3 Operowanie plikami i katalogami

2.3.1 Pliki i katalogi

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.
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*)

2.3.2 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`.

2.3.2.1 Uzyskiwanie nazwy katalogu bieżącego

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

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

2.3.2.2 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 2-1.

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

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

Przykład 2-1 Listowanie zawartości katalogu bieżącego.

PDF created with pdfFactory trial version www.pdffactory.com

2.3.2.3 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 2-2.

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

Przykład 2-2 Zmiana katalogu bieżącego

2.3.2.4 Tworzenie nowego katalogu

Nowy katalog tworzy się poleceniem `mkdir`. Polecenie to ma postać: `mkdir nazwa_katalogu`. Tworzenie nowego katalogu ilustruje Przykład 2-3.

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

Przykład 2-3 Tworzenie nowego katalogu

2.3.2.5 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 2-4.

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

Przykład 2-4 Kasowanie katalogu

2.3.3 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 2-5.

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

Przykład 2-5 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
```


PDF created with pdfFactory trial version www.pdffactory.com

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

Zmianę nazwy plików ilustruje Przykład 2-6.

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

Przykład 2-6 Zmiana nazwy pliku `stary.txt` na `nowy.txt`

2.3.3.1 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 2-7.

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

Przykład 2-7 Kasowanie pliku `nowy.txt`

2.3.3.2 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 Operowanie procesami

2.4.1 Wyświetlanie uruchomionych procesów

2.4.1.1 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` | nazwa - sprawdzanie czy wśród procesów istnieje proces nazwa

2.4.1.2 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 2-8 Użycie polecenia top

PDF created with pdfFactory trial version www.pdffactory.com

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. 2-1 Znaczenie parametrów polecenia top

2.4.1.3 Polecenie uptime

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

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

Przykład 2-9 Użycie polecenia top

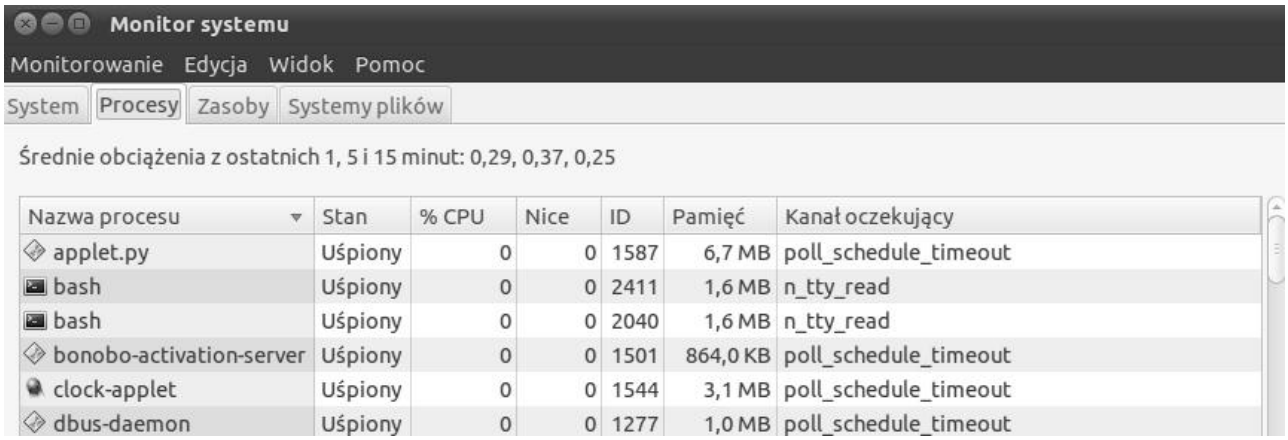
2.4.1.4 Polecenie pstree

Polecenie wyświetla drzewo procesów. Można obserwować zależność procesów typu macierzysty – potomny.

2.4.1.5 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 2-10 Użycie polecenia monitora systemu

2.4.2 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

PDF created with pdfFactory trial version www.pdffactory.com

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. 2-2 Częściej używane sygnały

2.5 Zadania

2.5.1 Uzyskiwanie informacji o stanie systemu

Zobacz jakie procesy i wątki wykonywane są aktualnie w systemie.

2.5.2 Uzyskiwanie informacji o obciążeniu systemu

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

2.5.3 Archiwizacja i kopiowania plików

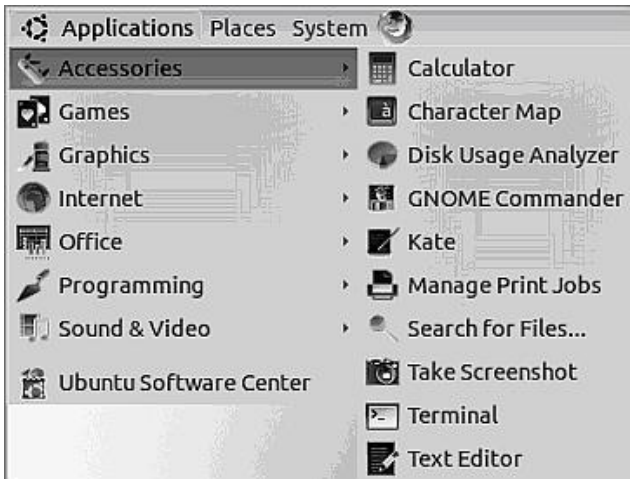
W systemie pomocy znajdź opis archiwizatora `tar`. Używając programu `tar` spakuj wszystkie pliki zawarte w katalogu bieżącym do pojedynczego archiwum o nazwie `programy.tar` (polecenie: `tar –cvf programy.tar *`). Następnie zamontuj dyskietkę typu MSDOS i skopiuj archiwum na dyskietkę. Dalej utwórz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

PDF created with pdfFactory trial version www.pdffactory.com

3. Tworzenie i uruchamianie programów w języku C

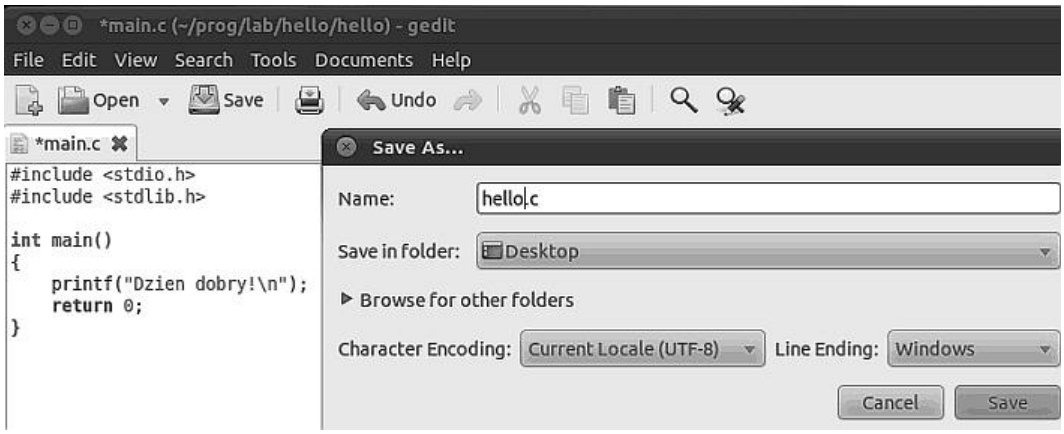
3.1 Metoda elementarna – użycie edytor gedit i kompilatora gcc

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 (patrz Przykład 3-3) kompilujemy program wpisując polecenie:

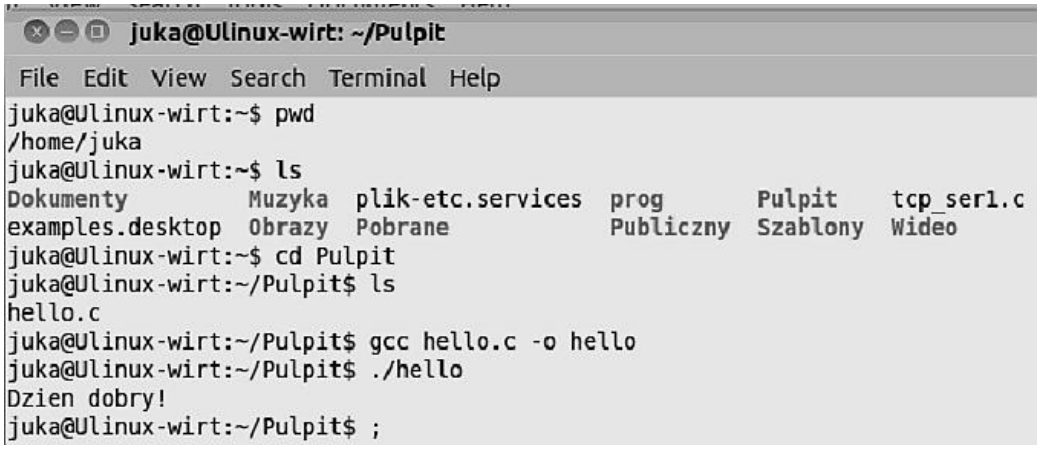
```
$gcc hello.c –o hello
```


PDF created with pdfFactory trial version www.pdffactory.com

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
```

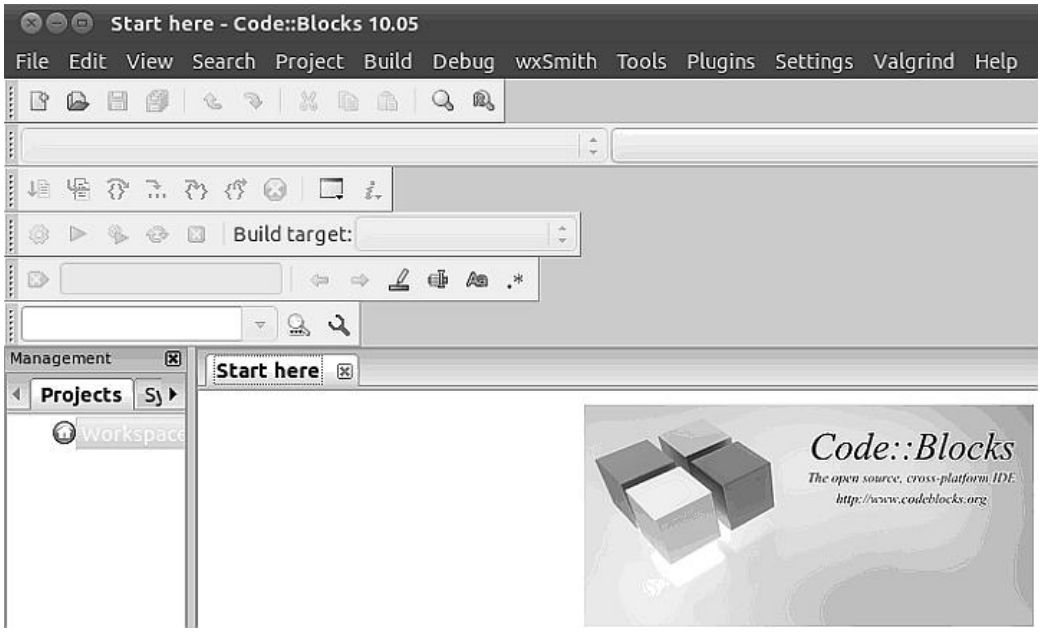
Otrzymamy wynik jak pokazuje Przykład 3-3.



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

3.2 Uruchamianie programów za pomocą środowiska CodeBlocks

CodeBlocks jest zintegrowanym środowiskiem do tworzenia i uruchamiania programów w języku C i C++. Zawiera w sobie edytor, narzędzie wywoływania kompilatora i interfejs okienkowy do programu uruchomieniowego (ang. *debuger*). Opis środowiska dostępny jest na stronie http://www.codebloks.org. Środowisko instaluje się za pomocą dowolnego instalatora np. zarządcę pakietów Synaptec. Po zainstalowaniu w menu programów pojawia się pozycja Code::Blocks IDE i klikając na nią uruchamiamy środowisko które zgłasza się jak poniżej.



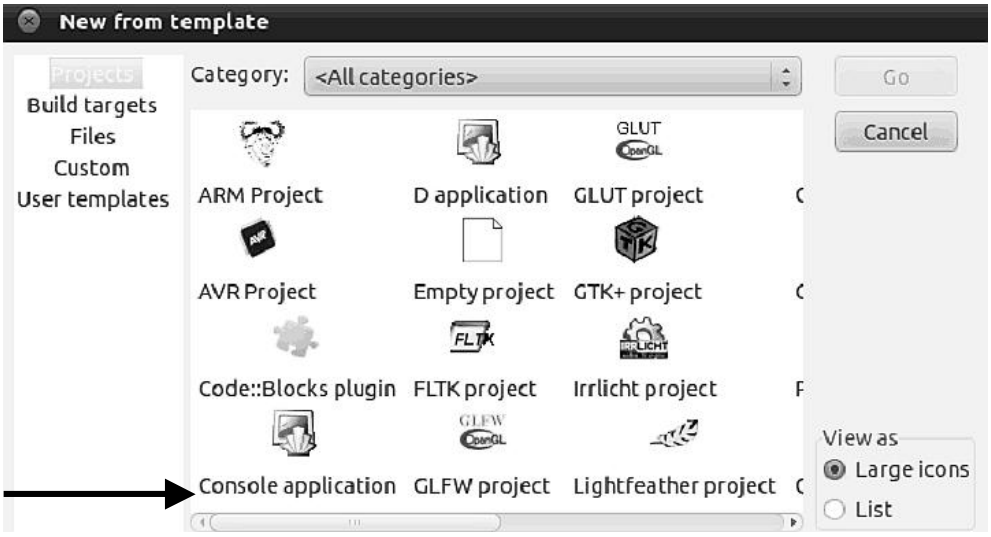
Przykład 3-4 Zgłoszenie środowiska CodeBlocks

Aby utworzyć i uruchomić program konsolowy należy:

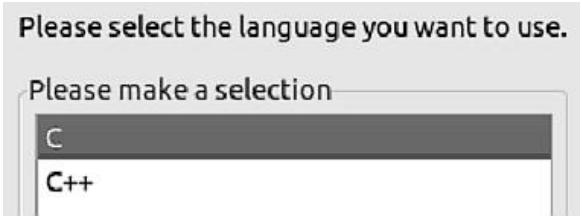
1. Kliknać w pozycję menu **File** (na górnej belce)
2. Wybrać pozycję **New / Project**. Pojawi się okno wyboru typu aplikacji jak pokazuje Przykład 3-5
3. Z okna wybrać opcję **Console aplication** (aplikacja konsolowa).
4. Pojawi się okno wyboru języka z którego należy wybrać język C jak pokazuje Przykład 3-6.

PDF created with pdfFactory trial version www.pdffactory.com

5. Dalej pojawi się okno wyboru nazwy programu. W oknie **Project tittle** wpisać należy nazwę projektu (np. pierwszy) jak pokazuje Przykład 3-7. Dalej należy wcisnąć położony w dole okna przycisk **Next**.
6. Następnie pojawi się ono wyboru typu projektu jak pokazuje Przykład 3-8. W tym momencie można zdecydować czy wybieramy opcję programu przeznaczonego do uruchamiania (domyślna opcja **Create "Debug" configuration**) lub też jako wersja końcowa (Opcja **Create "Release" configuration**). Wybieramy opcję domyslną "**Debug**". Przy okazji zaobserwować można w jakich folderach znajduję się tworzone programy. **Dalej** naciskamy przycisk **Finish**.
7. Po chwili pojawi się ekran z nowym projektem. Gdy klikniemy w napis sources / main znajdujący się w oknie **Projects** pokaże się okno edycji pliku źródłowego jak pokazuje przykład Przykład 3-9.
8. Dalej można dokonać edycji pliku `main.c` zastępując na przykład napis „`Hello Wold`" napisem „`Witam w pierwszym programie`".
9. Następnie kompilujemy i uruchamiamy program wybierając opcję **`Build and run`** z menu **Build** lub wciskając klawisz funkcyjny F7.
10. W wyniku działanie programu pojawi się okno wynikowe jak pokazuje Przykład 3-10.
11. Aby poprawić i skompilować ponownie program należy koniecznie zamknąć okno wynikowe gdyż inaczej opcja kompilacji zostanie zablokowana.

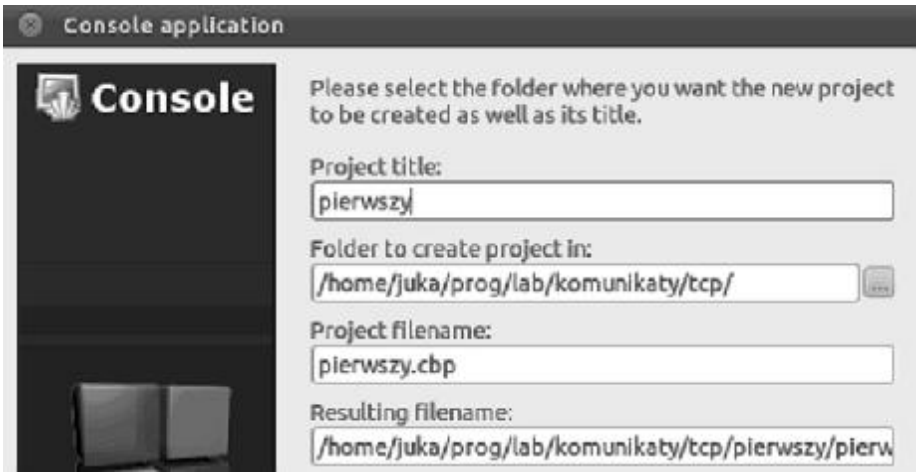


Przykład 3-5 Menu wyboru typu aplikacji



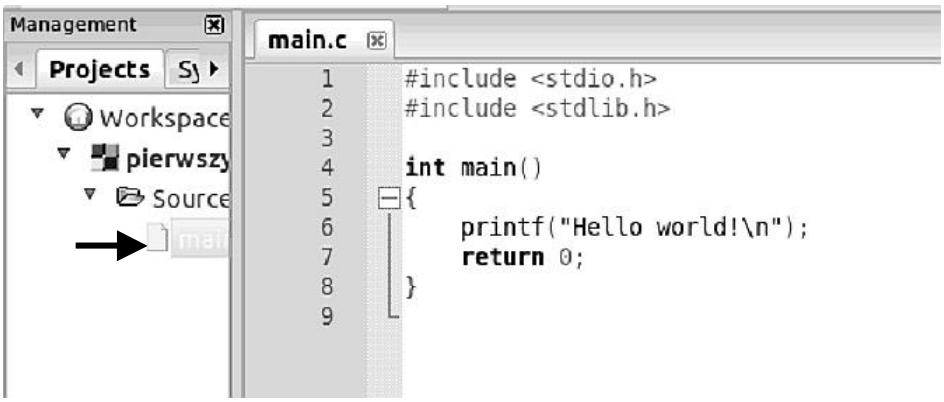
Przykład 3-6 Menu wyboru języka programowania

PDF created with pdfFactory trial version www.pdffactory.com



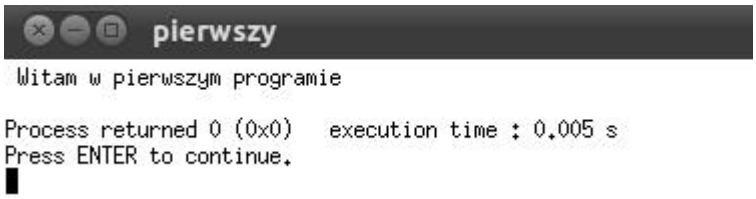
Przykład 3-7 Okno nazwy projektu

Przykład 3-8 Okno wyboru typu projektu



Przykład 3-9 Okno edycji pliku źródłowego

PDF created with pdfFactory trial version www.pdffactory.com



Przykład 3-10 Okno wyników programu

3.2.1 Debugowanie programów w środowisku CodeBlocks

Każdy programista, czy to początkujący czy zaawansowany doświadczył sytuacji gdy jego program nie zachowuje się zgodnie z oczekiwaniami czego powodem są błędy w programie. Systemy tworzenia oprogramowania oferują wiele narzędzi wspomagających uruchamianie programów. Jednym z najważniejszych są programy uruchomieniowe (ang. *debuggers*) . Pozwalają one na śledzenie wykonywania programu na poziomie kodu źródłowego. Główne mechanizmy które umożliwiają śledzenie to:

- Praca krokowa
- Ustawianie punktów wstrzymania programu (ang. *breakpoints*)
- Podgląd zmiennych w programie

Podstawowym narzędziem uruchomieniowym w systemie Linux jest GNU debugger gdb opisany w [7], [12]. Jest to jednak skomplikowane narzędzie pracujące w trybie konsolowym. Omawiane środowisko CodeBlocks zawiera Interfejs okienkowy do `gdb` co znacznie ułatwia z niego korzystanie. Elementarny sposób debugowania programów w tym środowisku zostanie pokazany dalej na przykładzie.

Aby utworzyć nowy projekt zawierający przykład postępujemy tak jak w poprzednim rozdziale. Niech nasz projekt nazywa się `iter`. Kod źródłowy niech będzie jak podaje Przykład 3-11. Program ten ma wypisać na konsoli 10 kolejnych liczb. Ważne aby nie zapomnieć zaznaczyć opcji **Debug** w oknie wyboru typu projektu (Przykład 3-8) . Następnie kompilujemy program wybierając z menu głównego opcję **Build** tak jak pokazuje Przykład 3-12. Gdy kompilacja będzie już poprawna ustawimy punkt wstrzymania programu (ang. *breakpoint*) w 6 linii kodu tak jak pokazuje Przykład 3-11. Punkt wstrzymania ustawiamy poprzez:

- Ustawienie kursora w linii 6 okna kodu źródłowego
- Naciśnięcie klawisza funkcyjnego F5 lub wybór opcji: **Debug / Toggle brakpoint** z menu głównego jak pokazuje Przykład 3-13.

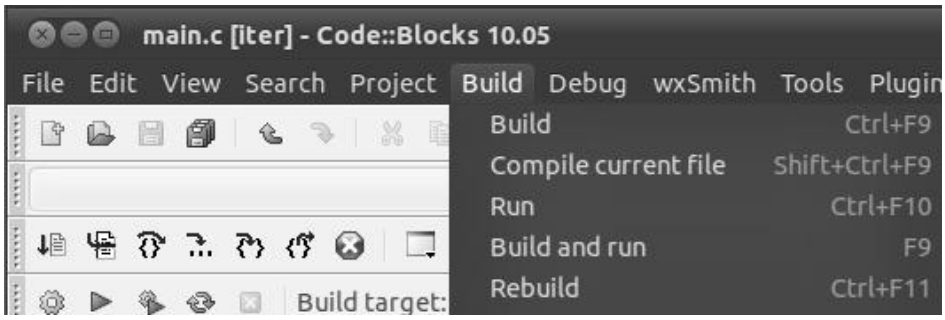
Nowy punkt wstrzymania uwidoczni się poprzez znacznik kółka w linii 6 kodu źródłowego jak widać to w Przykład 3-11. Następnie uruchamiamy program wybierając opcję **Debug / Start** z menu głównego lub naciskając klawisz funkcyjny F8. Program zatrzymuje się na linii 6 kodu. Bieżąca linia kodu wskazywana jest poprzez trójkąt pojawiający się po numerze linii tak jak pokazuje Przykład 3-14. Następnie klikając w ikonę **Program C ...** w dolnej części panelu możemy wyświetlić okno konsoli programu i zaobserwować działanie instrukcji printf tak jak pokazuje Przykład 3-14. Aby wykonać kolejne instrukcje naciskamy kilkakrotnie klawisz funkcyjny F7 (lub wybieramy opcję **Debug / Next line**) i dochodzimy do instrukcji 13 co pokazuje Przykład 3-15. Będąc w tym stanie można sprawdzić aktualne wartości zmiennych programowych. Dokonujemy tego klikając w opcję **Debug / Debugging windows / Watches**.

PDF created with pdfFactory trial version www.pdffactory.com

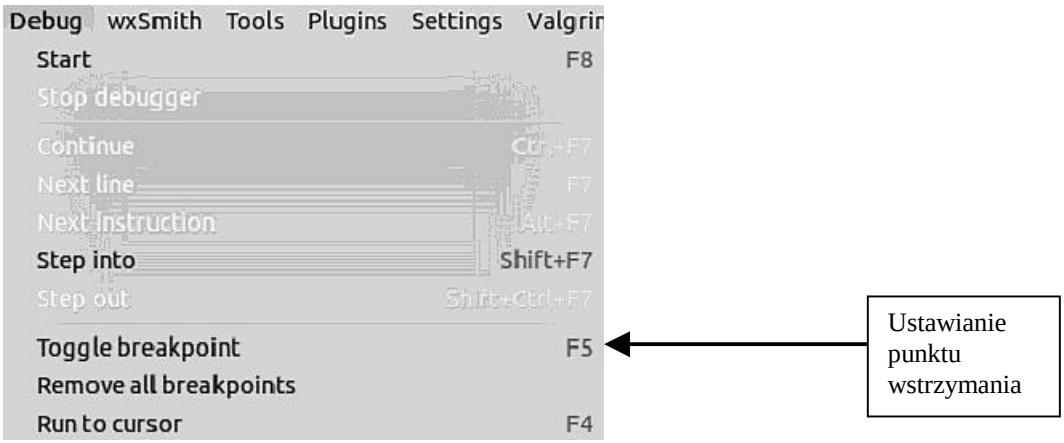
main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int k=0;
4  int wypisz(int i) {
5    printf("Krok %d\n",i);
6    return(i+1);
7  }
8
9  int main() {
10     int i,j;
11     printf("Start!\n");
12     for(i=1;i<10;i++) {
13         j=i*i+2;
14         k=wypisz(i);
15     }
16     return(0);
17 }
```

Przykład 3-11 Kod programu podlegającego debugowaniu

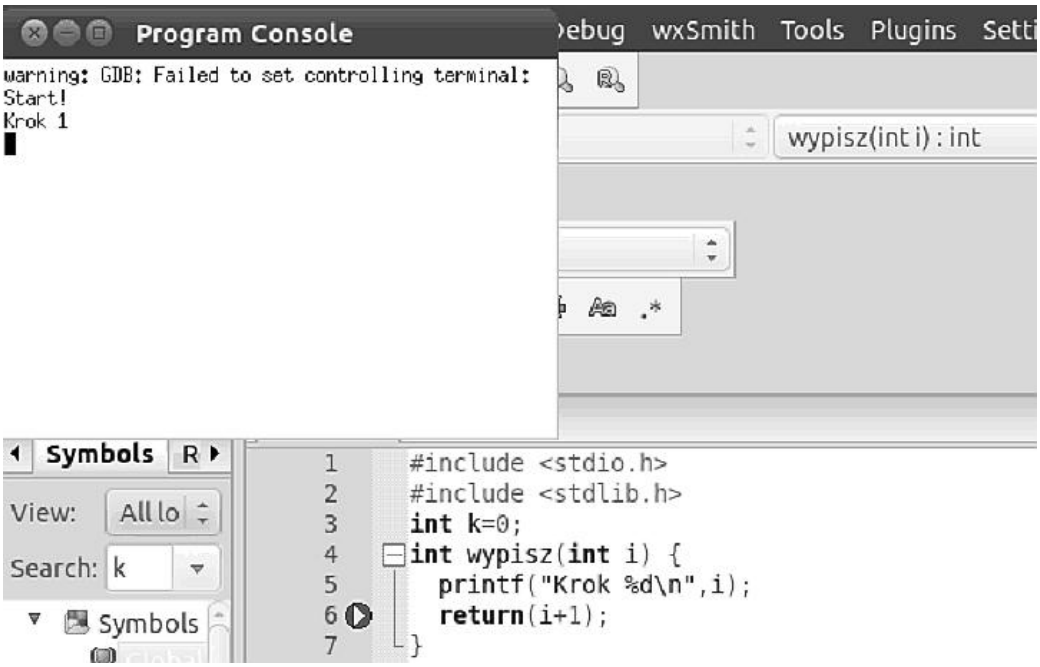


Przykład 3-12 Kompilacja programu



Przykład 3-13 Ustawianie punktu wstrzymania programu

PDF created with pdfFactory trial version www.pdffactory.com



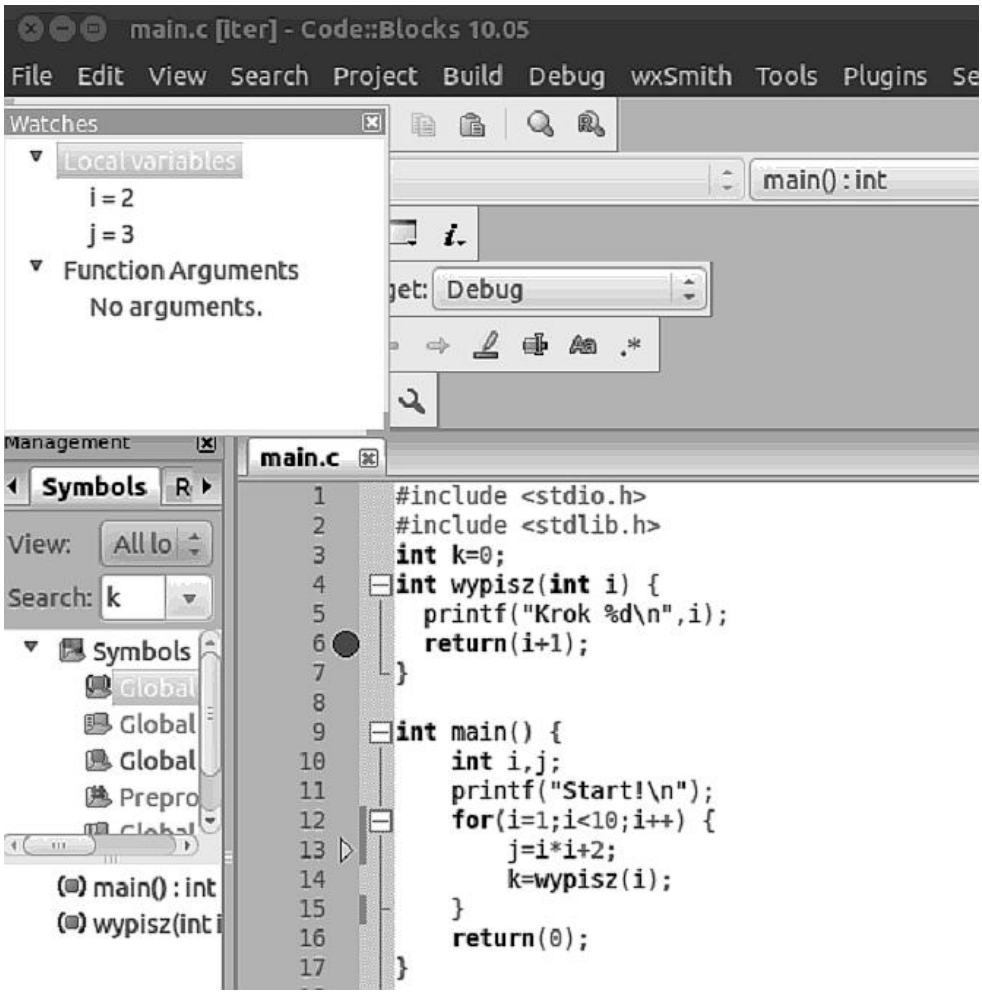
Przykład 3-14 Stan programu po dojściu do punktu wstrzymania

Debugger oferuje kilka możliwości dalszego wykonywania programu. Są one następujące:

Akcja	Opcje z menu	Klawisze	Ikona
Przejście do kolejnego punktu wstrzymania	Debug / Continue	Ctrl+F7	
Przejście do następnej linii kodu źródłowego	Debug / Next line	F7	
Przejście do następnej linii kodu maszynowego	Debug / Next instruction	Alt+F7	
Przejście do pozycji wskazywanej przez kursor	Debug / Run	F4	
Wykonanie następnej instrukcji z wejściem do funkcji	Debug / Step into	Ctrl+F7	
Wyjście z funkcji	Debug / Step out	Shift+Ctrl+F7	
Zakończenie debugowania	Debug / Stop debugger		

Tab. 3-1 Sposoby poruszania się po kodzie programu

PDF created with pdfFactory trial version www.pdffactory.com



Przykład 3-15 Inspekcja zmiennych programu po zatrzymaniu na instrukcji 13

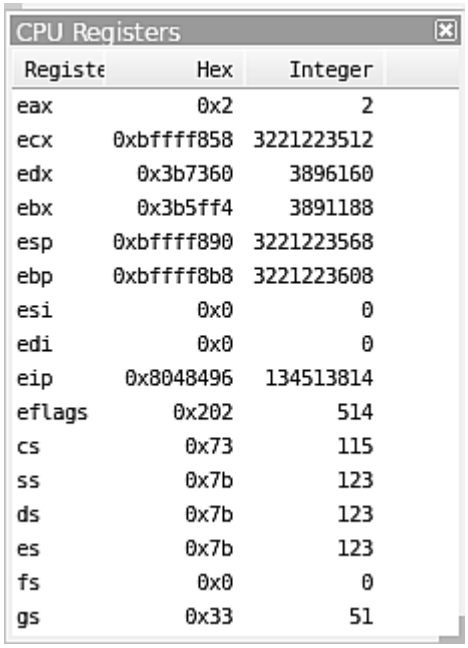
Środowisko CodeBlocks umożliwia dodatkowo wiele innych funkcji. W szczególności umożliwia obserwację:

- Punktów wstrzymania
- Stosu wywołań funkcji
- Rejestrów procesora
- Instrukcji kodu maszynowego
- Pamięci danych programu
- Wykonywalnych wątków
- Pułapek
- Aktualnego stosu wywołań funkcji
- Załadowanych bibliotek
- Użytych plików
- Stanu koprocesora zmiennoprzecinkowego
- Schematu obsługi sygnałów

Dostęp do tych funkcji możliwy jest poprzez wybór opcji **Debug / Debugging windows** co pokazuje Przykład 3-16 oraz opcji **Debug / Information**. Wybierając opcje **Debug / Debugging windows / CPU registers** możemy obejrzeć rejestry procesora co pokazuje Przykład 3-17.

PDF created with pdfFactory trial version www.pdffactory.com

Przykład 3-16 Dostęp do funkcji inspekcyjnych debuggera



Przykład 3-17 Obserwacja rejestrów procesora

Debugowanie przykładowego programu kończymy ustawiając punkt wstrzymania na ostatniej instrukcji programu w linii 16, usunięcie punktu wstrzymania z linii 6 i wybór opcji **Debug / Continue**. Wyniki programu pokazuje Przykład 3-18.

```
Start!
Krok 1
Krok 2
Krok 3
Krok 4
Krok 5
Krok 6
Krok 7
Krok 8
Krok 9
```

Przykład 3-18 Wyniki działania uruchamianego programu

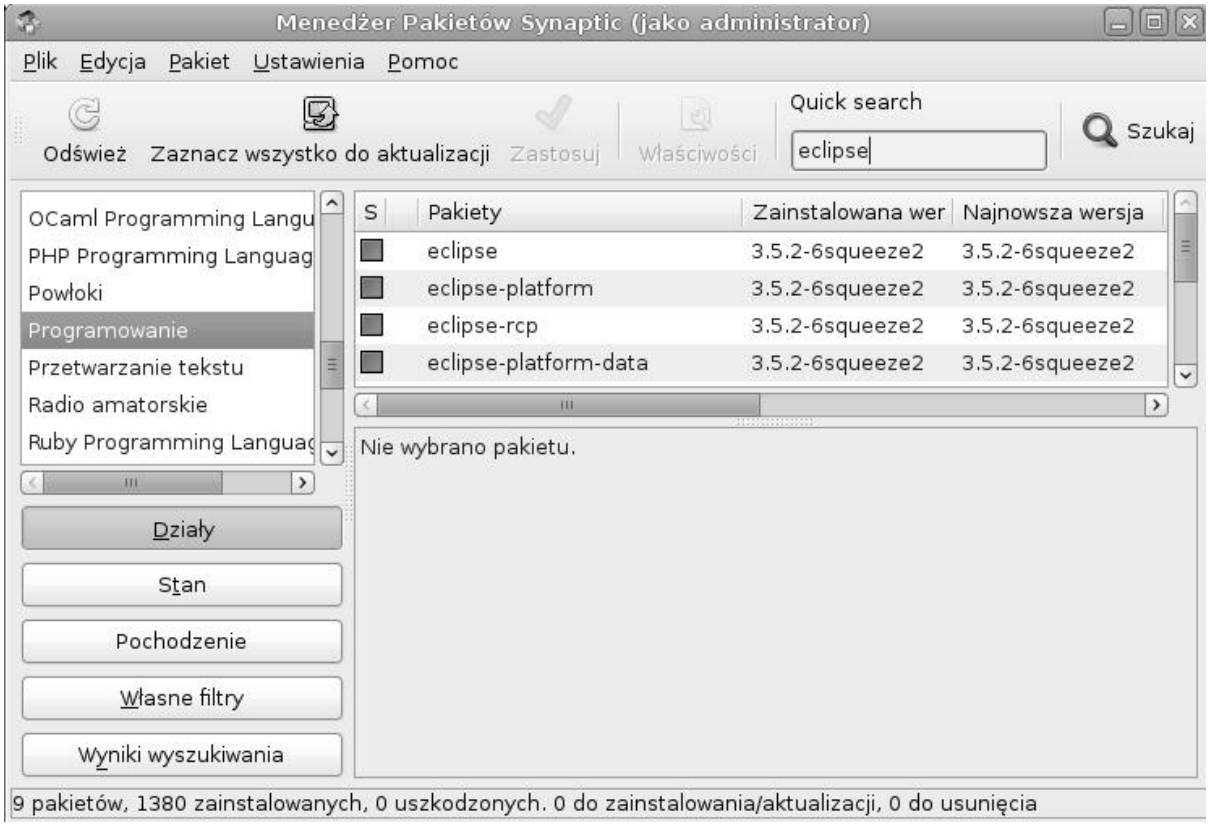
PDF created with pdfFactory trial version www.pdffactory.com

3.3 Uruchamianie programów za pomocą środowiska Eclipse CDT

Eclipse jest zintegrowanym środowiskiem do tworzenia i uruchamiania programów w różnych językach, w tym w języku C i C++. Zawiera w sobie edytor, narzędzie wywoływania kompilatora i interfejs okienkowy do programu uruchomieniowego (ang. *debuger*). Opis środowiska dostępny jest na stronie http://www.eclipse.org/cdt/documentation.php .

3.3.1 Instalacja środowiska

Środowisko instaluje się za pomocą dowolnego instalatora np. zarządcę pakietów Synaptic. Po uruchomieniu instalatora Synaptec w lewym oknie zaznacza się opcję Wszystkie lub Programowanie, w oknie Quick search należy wpisać eclipse. Po pojawieniu się pakietów jak w poniższym rysunku należy zaznaczyć pakiet eclipse do instalacji i kliknąć w ikonę Zastosuj.



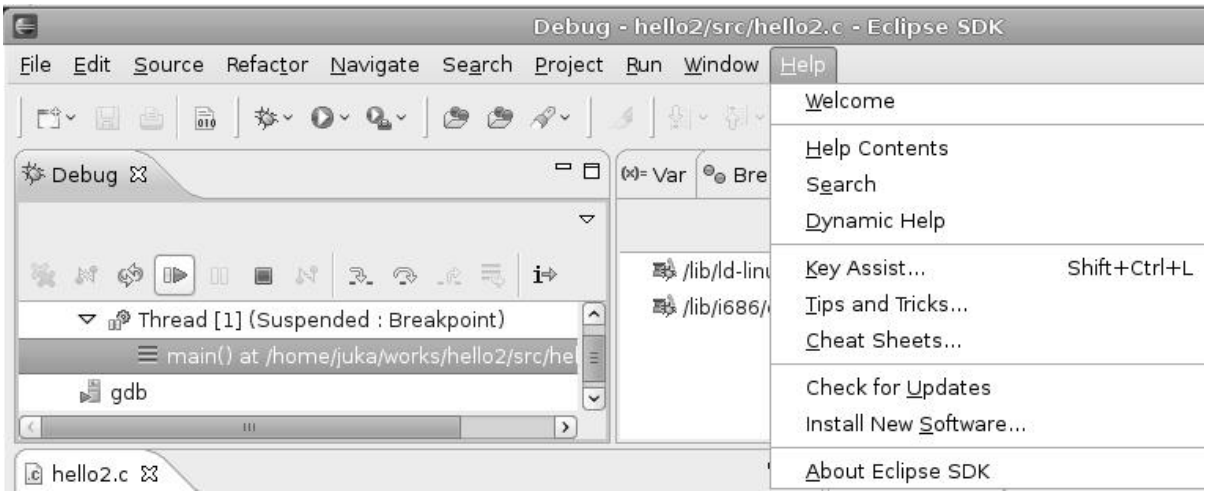
Rys. 3-1 Instalacja aplikacji Eclipse za pomocą zarządcy pakietów Synaptic

Po zainstalowaniu aplikacji należy ją uruchomić. Można to zrobić klikając w opcję Programy / Programowanie / Eclipse. Ekran startowy pokazany jest na poniższym rysunku.

PDF created with pdfFactory trial version www.pdffactory.com

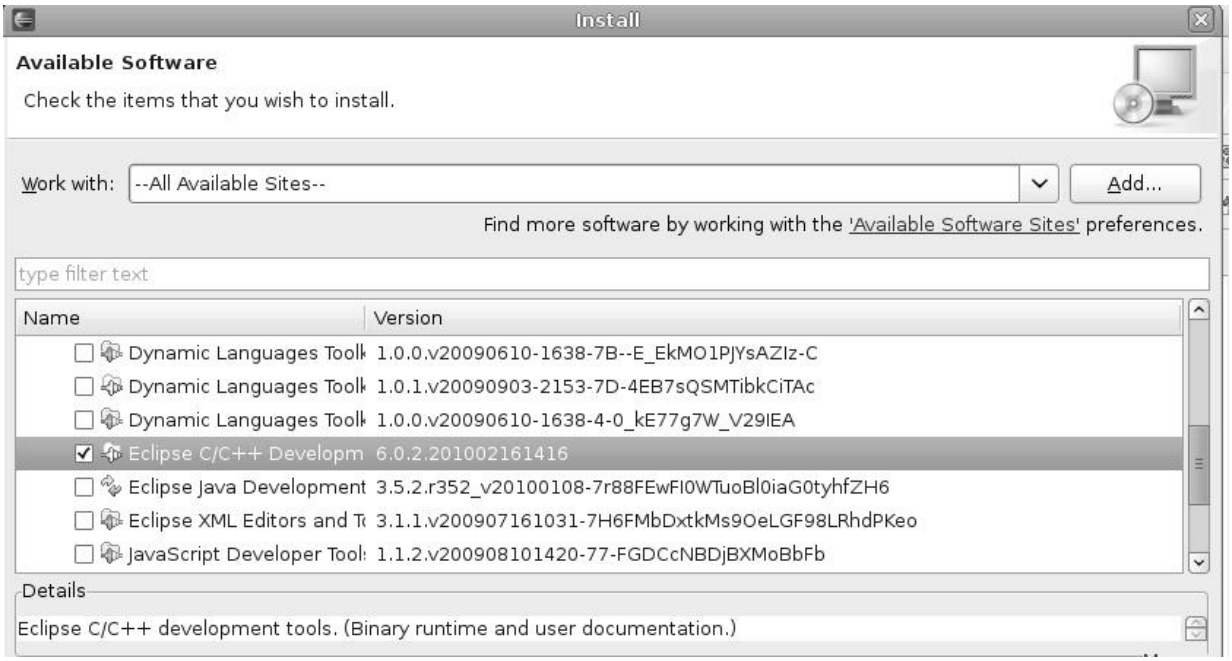
Rys. 3-2 Ekran startowy środowiska Eclipse

Aby rozpocząć pracę z systemem należy kliknąć w ikonę ze strzałką. Po uruchomieniu Eclipse należy zainstalować wtyczkę (ang. Plugin) dla języka C, C++. Wtyczka ta nazywa się CDT (ang. C Development Toolkit). Wtyczkę instaluje się za pomocą wewnętrznego instalatora Eclipse wybierając opcje: Help / Install New Software



Rys. 3-3 Wybór opcji instalacji nowego oprogramowania

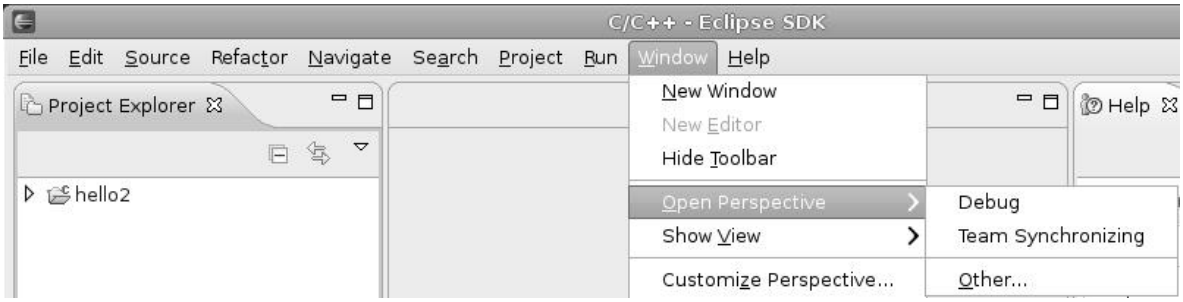
PDF created with pdfFactory trial version www.pdffactory.com



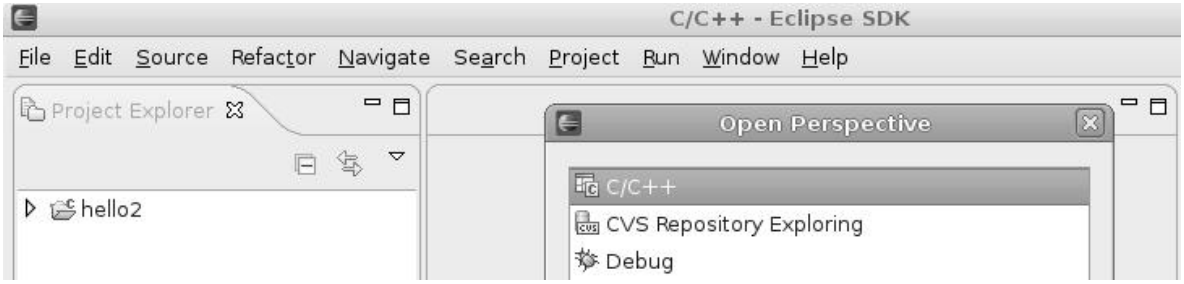
Rys. 3-4 Instalacja wtyczki programowania w C/C++

3.3.2 Przygotowanie środowiska

Środowisko Eclipse jest uniwersalnym narzędziem które może być wykorzystane do różnych celów. Przed przystąpieniem do pracy należy wybrać i ewentualnie skonfigurować tak zwaną perspektywę która określa zestaw okien i narzędzi które maja być widoczne. Perspektywę wybiera się za pomocą opcji **Window / Open Perspective / Other** jak pokazano na poniższym rysunku.



Rys. 3-5 Wybór perspektywy część I

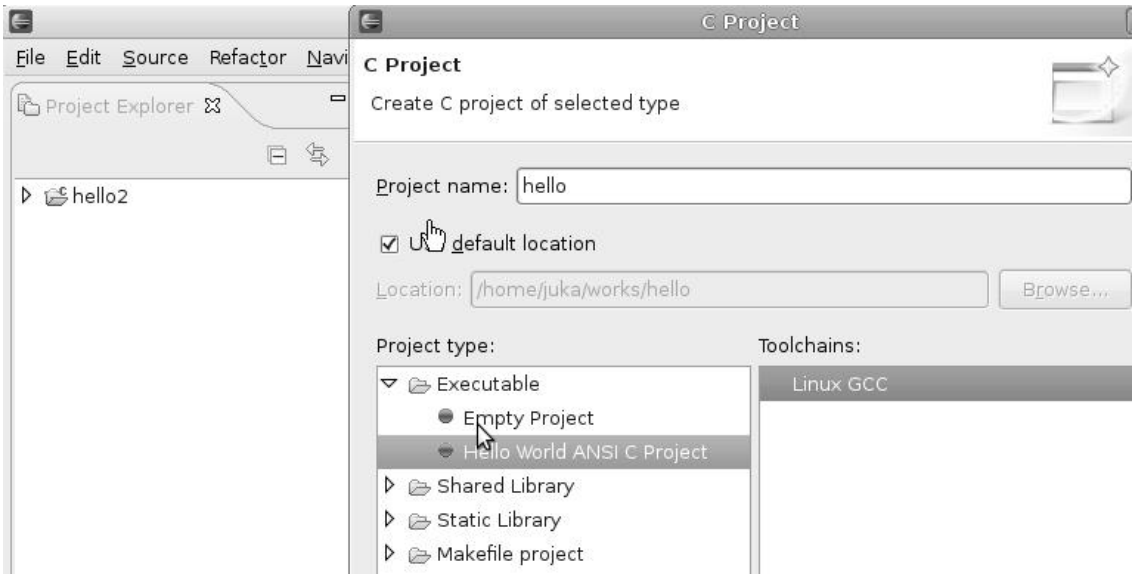


Rys. 3-6 Wybór perspektywy część II

PDF created with pdfFactory trial version www.pdffactory.com

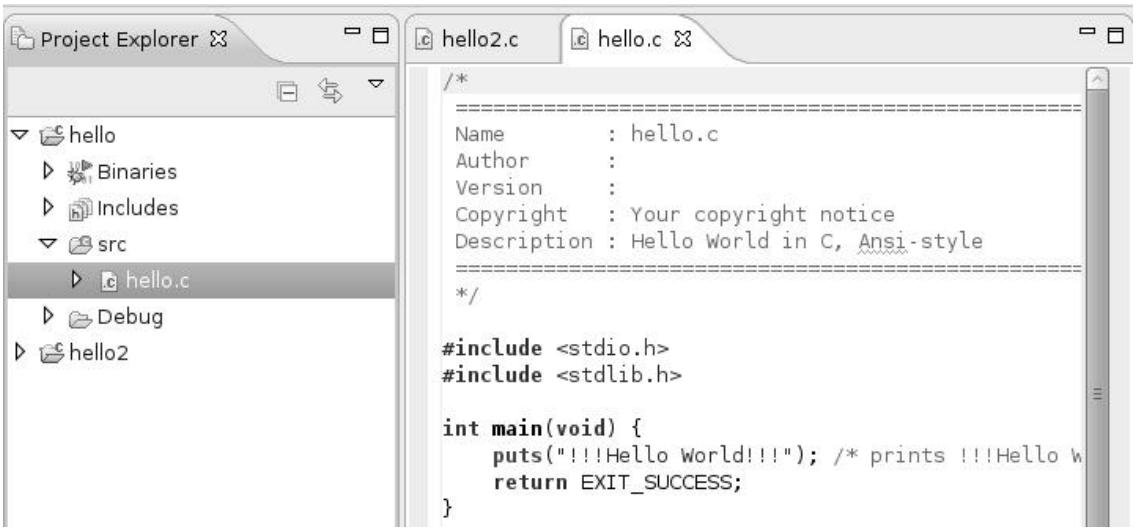
3.3.3 Tworzenie nowego projektu

W środowisku Eclipse aplikacje tworzone są w ramach tak zwanych projektów. Aby utworzyć nowy projekt programu na system docelowy wybieramy opcje: **File/ New / C/C++ Project** po czym pojawia się formatka jak poniżej. W oknie **Project Name** wpisujemy nazwę projektu, wybieramy **Executable / Hello World ANSI C Project** i klikamy w przycisk **Finish**.



Rys. 3-7 Tworzenie nowego projektu

Po naciśnięciu przycisku Finish pojawi się zestaw okien umożliwiający edycję i wykonywanie programów w języku C. Aby uzyskać dostęp do przykładowego programu źródłowego klikamy w nazwę projektu (hello) umieszczoną w lewym oknie i rozwijamy menu do postaci jak poniżej. Klikając dwa razy w ikonę hello.c w prawym oknie pojawi się kod szkieletowy programu hello.c jak pokazano poniżej.

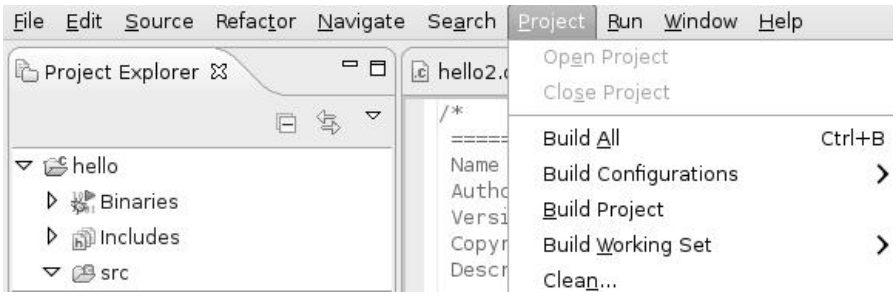


Rys. 3-8 Projekt przykładowy hello.c

3.3.4 Kompilacja i wykonanie projektu

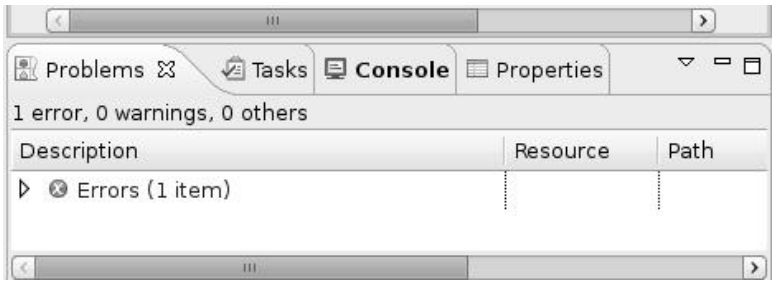
Utworzony przez system szkieletowy kod można poddać edycji zmieniając na przykład napis w funkcji printf. Po zakończeniu edycji należy program zachować klikając w opcje Aby skompilować program należy kliknąć w pozycję Project / Build Project jak pokazano na poniższym rysunku.

PDF created with pdfFactory trial version www.pdffactory.com



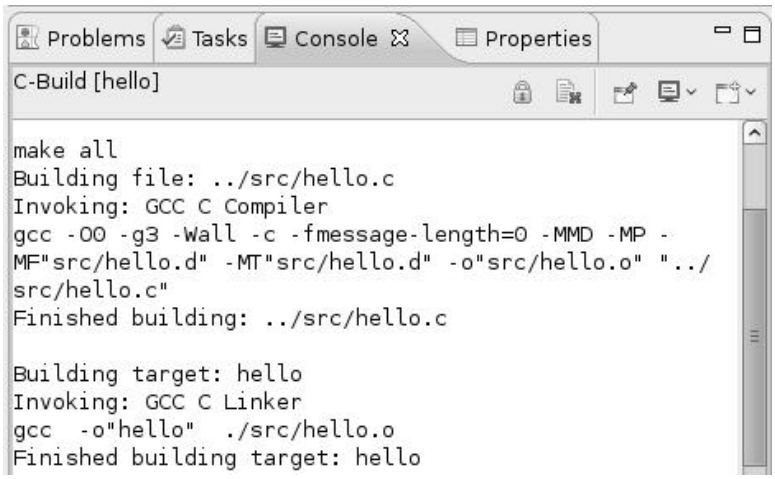
Rys. 3-9 Kompilacja projektu

O ile w kodzie programu są błędy pojawią się one w zakładce Problems dolnego okienka jak pokazano poniżej.



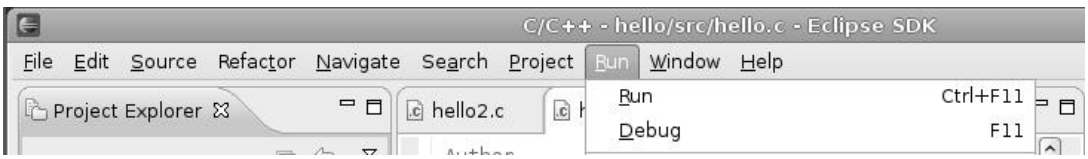
Rys. 3-10 Zakładki błędów, konsoli, zadań i własności.

Przebieg kompilacji można obserwować w zakładce Console. Gdy program będzie poprawny otrzymamy raport kompilacji jak poniżej.

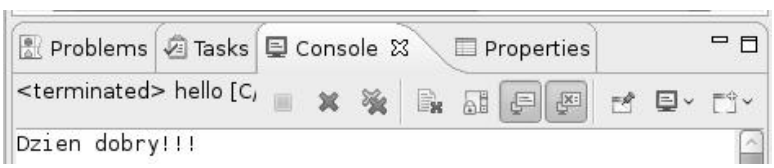


Rys. 3-11 Kompilacja programu przykładowego

Gdy program jest już bezbłędny można go uruchomić klikając w opcję **Run**



Rys. 3-12 Uruchomienie programu hello.c

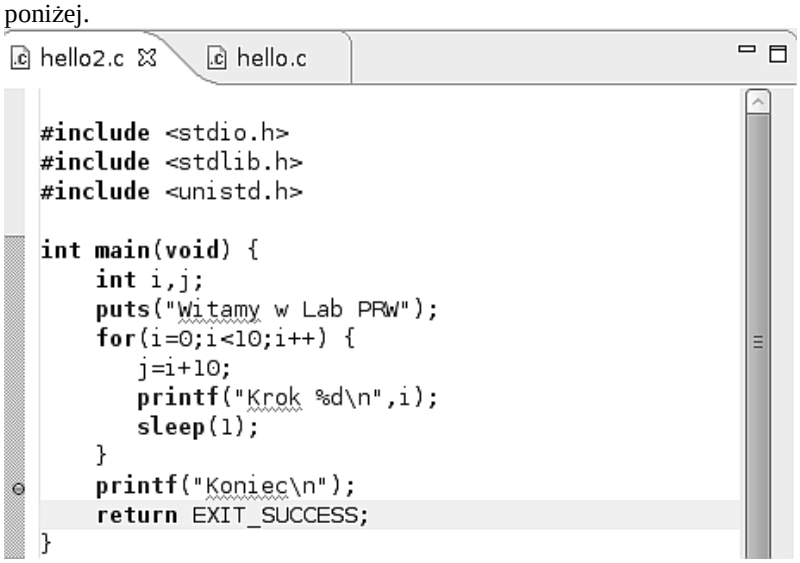


Rys. 3-13 Wyniki działania przykładowego programu hello.c

PDF created with pdfFactory trial version www.pdffactory.com

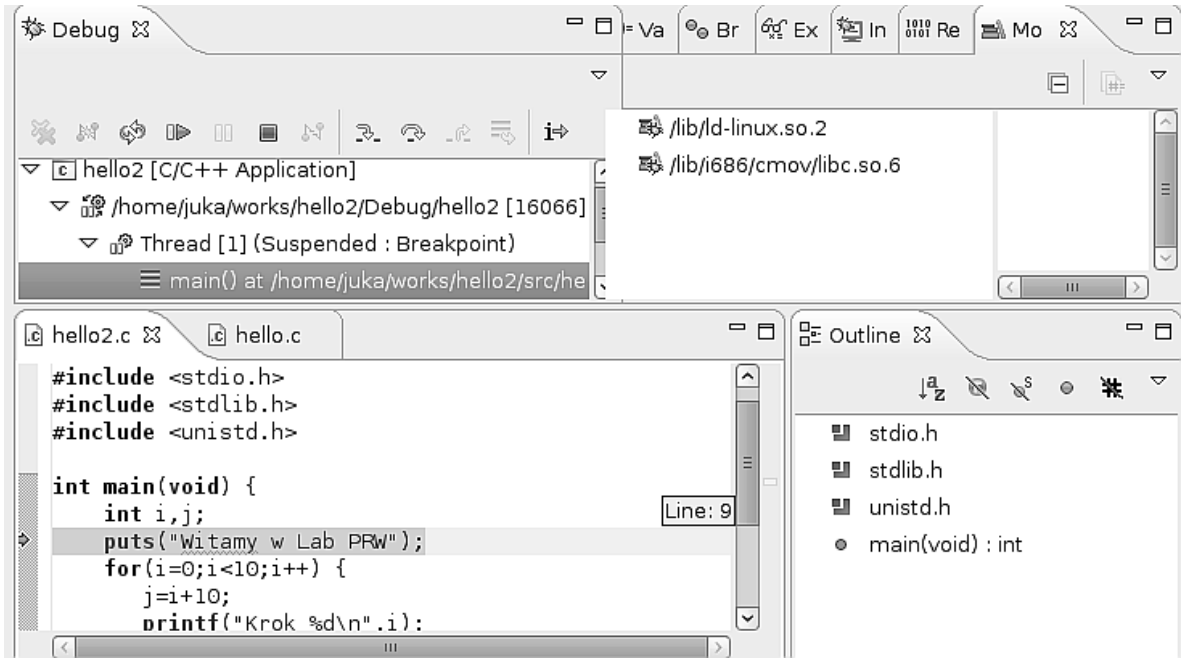
3.3.5 Uruchamianie programów

Środowisko Eclipse może być interfejsem do debuggera gdb. Aby wypróbować jego możliwości utwórzmy nowy projekt o nazwie hello2.c i w oknie edytora napiszmy prosty program zawierający petlę. Program hello2.c pokazano poniżej.



Rys. 3-14 Kod programu hello2

Następnie kompilujemy program w podany uprzednio sposób. Gdy błędy zostaną usunięte przystępujemy do uruchomienia. Można to zrobić wybierając opcję **Run / Debug**, naciskając klawisz F11 lub klikając w ikonę pluskwy. Nastąpi wtedy przełączenie na perspektywę debuggera co pokazuje poniższy ekran.



Rys. 3-15 Perspektywa debuggera dla programu hello2

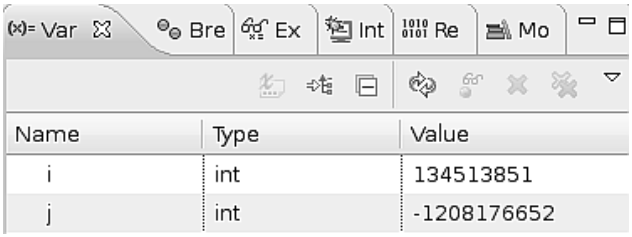
Do sterowania przebiegiem uruchamiania używamy ikonek w oknie debuggera. Najważniejsze podaje poniższa tabela.

PDF created with pdfFactory trial version www.pdffactory.com

	Restart		Uruchom proces od początku
	Resume	F8	Uruchom proces od punktu bieżącego
	Terminate		Zakończ proces
	Step into	F5	Wykonaj krok programu wchodząc do funkcji
	Step over	F6	Wykonaj krok programu nie wchodząc do funkcji
	Run to return	F7	Wyjdź z funkcji
	Suspend		Zawieś proces

Tab. 3-2 Najważniejsze ikony okna programu uruchomieniowego

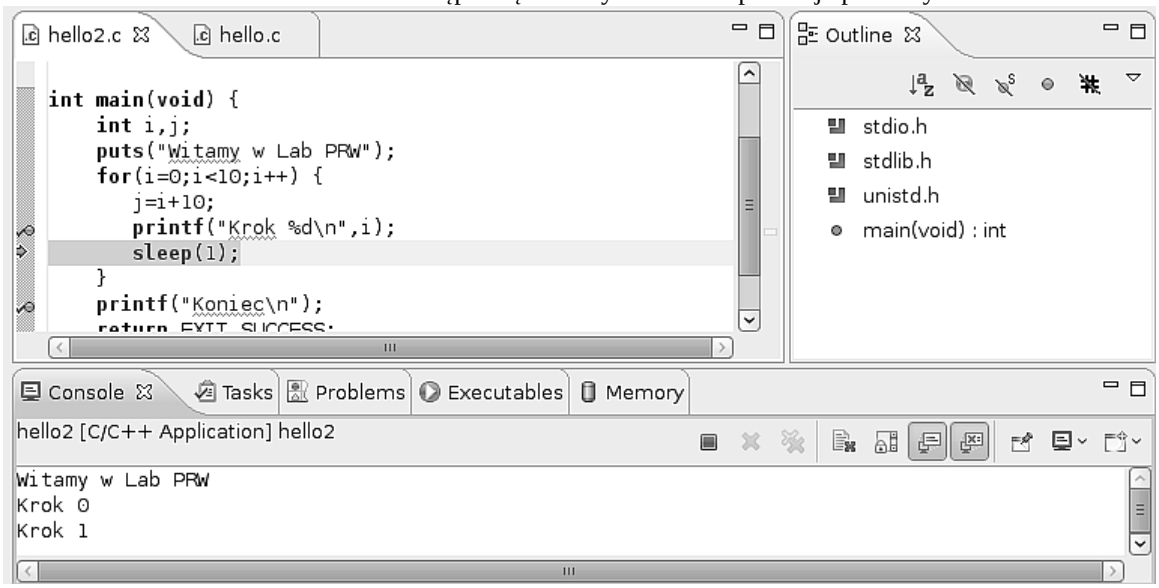
Naciskając ikonę wykonujemy jeden krok programu co pozwala na pracę krokową. Możemy również uruchomić program do najbliższego punktu wstrzymania (*ang. Breakpoint*). Punkt wstrzymania ustawiamy klikając prawym klawiszem myszy na listewkę po lewej stronie okna programu źródłowego. Gdy ustawimy punkt wstrzymania na linię 14 i naciśniemy ikonę program wykona się do linii 14. Wartości zmiennych obserwować można w oknie inspekcji zmiennych (Variables). Debugger posiada znacznie więcej możliwości testowania stanu programu. Dalsze informacje zawarte są w zakładkach umieszczonych z prawej strony głównego okna.



Rys. 3-16 Zakładki inpekcyjne debuggera

- Var – zakładka inspekcji zmiennych (ang *Variables*)
- Bre – zakładka punktów wstrzymania (ang. *Breakpoint*)
- Ex – zakładka wyrażeń (ang expression)
- Int – zakładka interakcji (ang. *Interaction*)
- Re – zakładka rejestrów (ang. *Registers*)

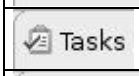
Jeszcze inne możliwości środowiska dostępne są w dolnym oknie co pokazuje poniższy ekran.



Rys. 3-17 Przebieg uruchamiania programu hello2.c – zakładki dolne

Dolne okno zawiera zakładki podane w poniższej tabeli.

PDF created with pdfFactory trial version www.pdffactory.com

Nazwa ang.	Nazwa polska	Znaczenie
	Konsola	Pokazuje wyniki działania programu
	Tasks	Pokazuje wątki
	Problemy	Pokazuje komunikaty o błędach
	Pliki wykonywalne	Pokazuje programy
	Pamięć	Pokazuje zawartość pamięci

Tab. 3-3 Zakładki debuggera w dolnej części ekranu

PDF created with pdfFactory trial version www.pdffactory.com

3.4 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. Aplikacja składa się z dwóch programów - `pierwszy` i `drugi`. 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 Rys. 3-18. 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
```

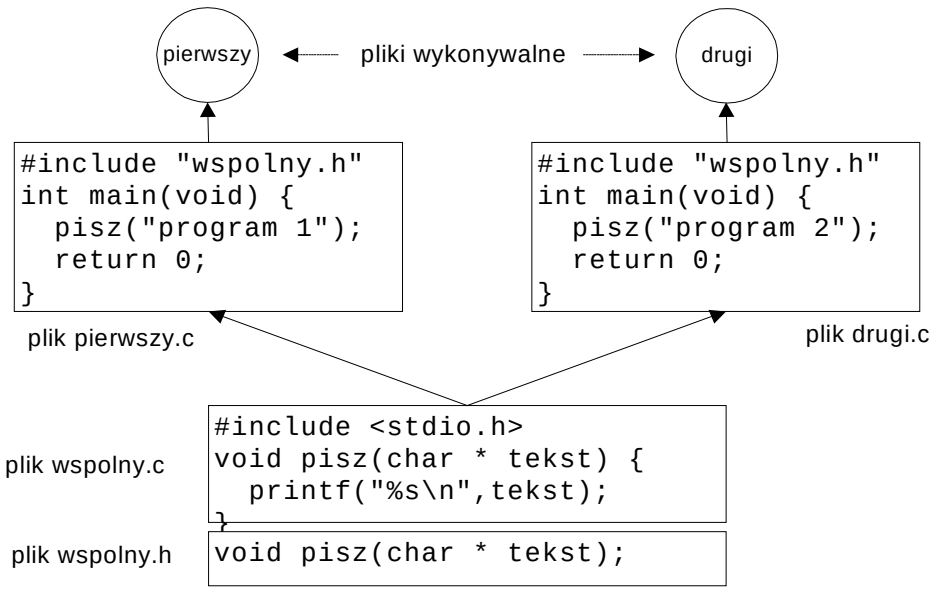
Analogiczny efekt osiągnąć można tworząc plik definicji `makefile` dla narzędzia `make` a następnie pisząc z konsoli polecenie `make`. Narzędzie `make` opisane jest obszernie w literaturze [7], [6] i dokumentacji systemu. Plik `makefile` dla powyższego przykładu pokazany jest poniżej. Należy zauważyć że plik `makefile` i pliki źródłowe muszą być umieszczone w jednym folderze. Po wpisaniu polecenia make system szuka w folderze bieżącym pliku o nazwie `Makefile` a następnie `makefile`. Gdy chcemy aby miał inną nazwę wpisujemy ja jako parametr polecenia `make:`

```
make –f nazwa_pliku.
```

```
# 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 3-19 Plik `makefile` dla aplikacji składającej się z dwóch plików

Natomiast wyniki działania polecenia `make` pokazuje Przykład 3-20.



Rys. 3-18 Aplikacja składająca się z dwóch programów

PDF created with pdfFactory trial version www.pdffactory.com

```
juka@Ulinux-wirt:~/prog/lab/make$ ls
drugi.c  makefile  pierwszy.c  wspolny.c  wspolny.h
juka@Ulinux-wirt:~/prog/lab/make$ make
gcc -o pierwszy pierwszy.c wspolny.c
gcc -o drugi drugi.c wspolny.c
juka@Ulinux-wirt:~/prog/lab/make$ ls
drugi  drugi.c  makefile  pierwszy  pierwszy.c  wspolny.c  wspolny.h
```

Przykład 3-20 Działanie polecenia `make`

Plik definicji `makefile` składa się z zależności i reguł. Zależność podaje jaki plik 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. Nawiązując do omawianego przykładu występuje tam definiująca taką 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
```

Należy zwrócić uwagę że powyższa linia zaczyna się niewidocznym znakiem tabulacji. W plikach `makefile` umieszczać można linie komentarza poprzez umieszczenie na pierwszej pozycji takiej linii znaku #. Gdy do programu `make` wpiszemy parametr będący nazwą pewnej zależności można spowodować wykonanie reguły odpowiadające tej zależności. Do poprzedniego pliku makefile dodać można regułę o nazwie `archiw` wykonania archiwizacji plików źródłowych co pokazuje Przykład 3-21. Wpisanie polecenia make archiw spowoduje wykonanie archiwum plików źródłowych i zapisanie ich w pliku `prace.tgz`. Narzędzie `make` ma znacznie więcej możliwości ale nie będą one potrzebne w dalszej części laboratorium i zostaną pominięte.

```
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
archiw: pierwszy.c drugi.c wspolny.c wspolny.h
      tar -cvf prace.tar *.c *.h makefile
      gzip prace.tar
      mv prace.tar.gz prace.tgz
```

Przykład 3-21 Plik `make` z opcją archiwizacji plików źródłowych

PDF created with pdfFactory trial version www.pdffactory.com

4. Tworzenie procesów

4.1 Wstęp

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 <process.h>
#include <unistd.h>
void main(void){
  int pid,status;
  if((pid = fork()) == 0) { /* Proces potomny ---*/
      printf(" Potomny = %d \n",getpid());
      sleep(30);
      exit(0);
  }
 /* Proces macierzysty */
 printf("Macierzysty = %d \n",getpid());
 pid = wait(&status);
 printf("Proces %d zakończony status: %d\n",pid,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.

PDF created with pdfFactory trial version www.pdffactory.com

```
// Ilustracja działania funkcji execl – uruchomienie programu potomny
#include <stdio.h>
#include <process.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,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 Zadania

4.3.1 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

Następnie wprowadź do otoczenia nowy parametr MOJPAR i nadaj mu wartość wczytywaną z klawiatury i przetestuj czy zmiana została wprowadzona poprawnie.

Zmień priorytet procesu za pomocą polecenia nice i sprawdź efekty.

4.3.2 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:

PDF created with pdfFactory trial version www.pdffactory.com

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

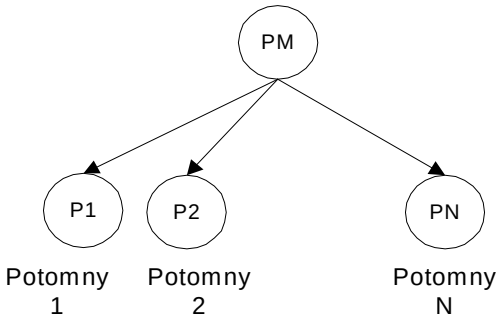
Aby wykonać zadanie do procesu procm1 należy przekazać informacje:
- Ile ma być procesów potomnych
- Ile kroków ma wykonać każdy z procesów potomnych.

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

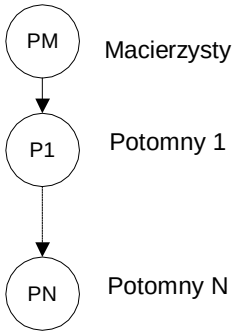
Np. wywołanie procm1 10 11 12 oznacza że należy utworzyć 2 procesy potomne i maja one wykonac 11 i 12 kroków. Proces macierzysty ma wykonać 10 kroków. Na zakończenie procesu potomnego powinien on wykonać funkcję `exit(NR)` przekazując jako kod powrotu swój numer identyfikacyjny procesowi macierzystemu. 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.



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

4.3.3 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.

PDF created with pdfFactory trial version www.pdffactory.com

4.3.4 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` 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
```

4.3.5 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.

4.3.6 Znajdowanie liczb pierwszych

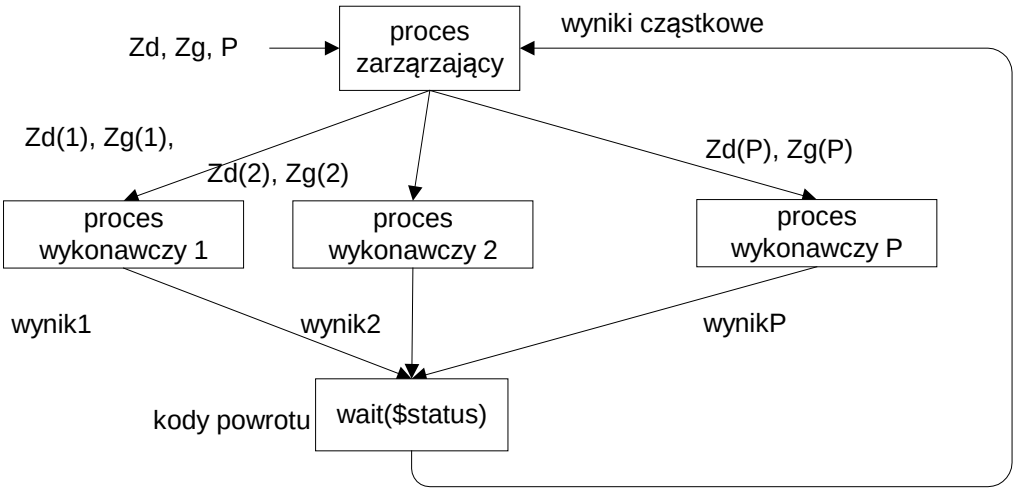
Napisz program który ma znajdować liczby pierwsze w zadanym przedziale [Zd,...,Zg]. Liczba jest pierwsza gdy dzieli się przez 1 i przez siebie samą. Prymitywny algorytm sprawdzania, czy dana liczba n jest liczbą pierwszą dany jest poniżej:

```
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 `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

PDF created with pdfFactory trial version www.pdffactory.com

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,numerP,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. 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 i 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
- Zakres górny przedziału
- Liczbę procesów wykonawczych

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

PDF created with pdfFactory trial version www.pdffactory.com

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];
  fd = open(argv[1],O_RDONLY);
  do {
     rd = read(fd,buf,80);
     printf("%s",buf);
  } while(rd == 80);
  close(fd);
}
```

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

PDF created with pdfFactory trial version www.pdffactory.com

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 Zadania

5.4.1 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:

PDF created with pdfFactory trial version www.pdffactory.com

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.

5.4.2 Program kopiowania plików – użycie strumieni

Napisz program f`copy` który kopiuje pliki używając funkcji standardowej biblioteki wejścia wyjścia. Program ma być uruchamiany poleceniem f`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: f`open()`, `fread()`, f`write()`, f`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. Sprawdź funkcją feof czy wystąpił koniec pliku `file1` . Gdy nie przejdź do kroku 4. Gdy plik się skończył to zamknij pliki i zakończ program.

5.4.3 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

PDF created with pdfFactory trial version www.pdffactory.com

5.4.4 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.

5.4.5 Równoległe znajdowanie liczb pierwszych – komunikacja przez wspólny plik

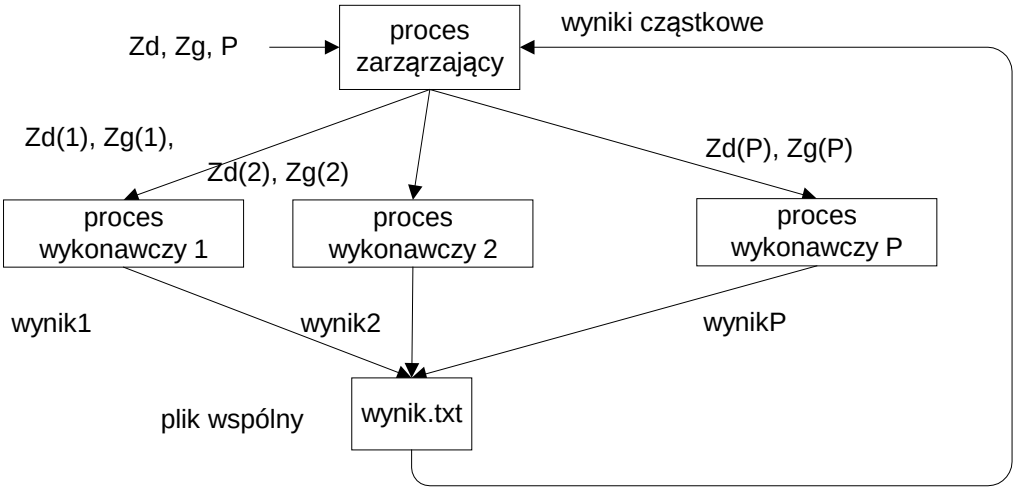
Napisz program który ma znajdować liczby pierwsze w zadanym przedziale [Zd,...,Zg]. Liczba jest pierwsza gdy dzieli się przez 1 i przez siebie samą. Prymitywny algorytm sprawdzania, czy dana liczba n jest liczbą pierwszą dany jest poniżej:

```
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.txt. Każdy z procesów wykonawczych ma dodać do pliku jedną linię zawierającą:

```
numer_procesu   ile_pierwszych
```

Po zakończeniu procesów wykonawczych proces macierzysty odczytuje poszczególne linie 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.txt`.
2. Program zarządzający dzieli przedział [Zd,...,Zg] na P podprzedziałów. Następnie tworzy procesy potomne używając funkcji `fork i execl("./licz","licz",pocz,kon,plik,numerP,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.

PDF created with pdfFactory trial version www.pdffactory.com

Tak więc, proces wykonawczy powinien mieć postać: `licz granica_dolna granica_górna nazwa_pliku numer_procesu`

3. Proces zarządzający czeka na zakończenie wykonawczych i odczytuje z pliku wynik.txt dane o znalezionych liczbach liczb pierwszych, oblicza sumę która ma być wyprowadzona na konsolę i czas obliczeń.

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

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)`. Proszę narysować wykres pokazujący zależność czasu obliczeń od liczby procesów.

5.4.6 Baza danych „hotel”

Napisz prostą bazę danych obsługującą hotel. Baza danych zawarta będzie w pliku ”`hotel.bin`” który zawiera rekordy:

```
#define NSIZE 30
typedef struct {
      int wolny;
      int numer;
      char nazwisko[NSIZE];
      char imie[NSIZE];
} pokoj_t;
```

Baza danych powinna realizować następujące funkcje:

Zameldowanie gościa w pierwszym wolnym pokoju, podaje się imię i nazwisko. Pole wolny = 0, do pól imię i nazwisko kopiujemy wymagane wartości.	`melduj(bazaf)`
Wymeldowanie gościa z danego pokoju. Przy wymeldowaniu podaje się pokój. Pole wolny = 1, pola imię i nazwisko są zerowane.	`wymeld(bazaf)`
Podanie informacji kto jest zameldowany w danym pokoju. Dla danego pokoju wyprowadzamy imię i nazwisko gościa.	`info(bazaf)`
Podanie jaka osoba zameldowana jest w danym pokoju. Przeszukujemy rekordy bazy i szukamy rekordu w którym pole nazwisko posiada podaną wartość.	`znajdz(bazaf)`
Wyprowadzenie na konsolę listy pokojów z informacją kto je zajmuje i ile jest wolnych pokojów.	`lista(bazaf)`
Inicjacja bazy danych, zapis pustych rekordów z informacja że pokój jest wolny. Bazę inicjujemy na zadaną liczbę pokojów.	`init(bazaf,POKOJOW)`

```
#define MELD      1
#define WYMELD    2
#define KTO       3
#define LISTA     4
#define ZNAJDZ    5
#define WYJSCIE   6
#define POKOJOW  10
```


PDF created with pdfFactory trial version www.pdffactory.com

```
char bazaf[] = "hotel.bin";
int main(int argc, char*argv[]) {
   if(Czy istnieje plik bazy danych) {
       // Plik nie istnieje - inicjacja bazy
       init(bazaf,POKOJOW);
   }
   do {
     opt = menu();
         switch(opt) {
         case MELD   : melduj(bazaf);     break;
         case WYMELD : wymeld(bazaf);     break;
         case KTO    : info(bazaf);       break;
         case LISTA  : lista(bazaf)       break;
         case ZNAJDZ : znajdz(bazaf);    break;
         case WYJSCIE: _exit(0);
         default     :;
     }

   } while(stop != 1);
   return 0;
}
```

Przykład 5-4 Szkic procesu bazy danych

Przy starcie program powinien sprawdzić czy istnieje plik bazy danych (można użyć funkcji `stat`). Gdy brak pliku należy go utworzyć wypełniając go pustymi rekordami gdzie pole `wolny`=1, pole `numer=1,2,...` a pola `nazwisko` i `imię` powinny zawierać łańcuch pusty W procedurach realizujących poszczególne funkcje należy zastosować blokowanie pliku bazy danych. Do tego celu użyć można funkcji `lockf`. Proszę wypróbować blokadę uruchamiając program hotel z dwóch konsol.

PDF created with pdfFactory trial version www.pdffactory.com

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

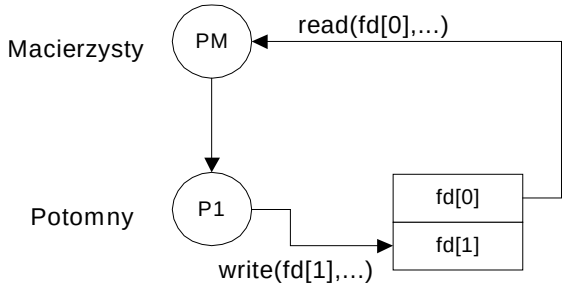
Funkcja zwraca:
0 gdy sukces
-1 gdy operacja się nie udała.

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.

```
#include <unistd.h>
#include <stdlib.h>
 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 do 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]);
}
```

Przykład 6-1 Proces potomny przesyła wiadomość do macierzystego poprzez łącze

PDF created with pdfFactory trial version www.pdffactory.com



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

1. 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 procesu piszącego do pliku FIFO podano poniżej.

PDF created with pdfFactory trial version www.pdffactory.com

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#define FIFO MyFIFO

main() { // Zapis -------------
  int fdes,res,i=0;
  static char c;
  res = mkfifo("MyFIFO",O_RDWR | O_CREAT);
  if(res < 0) { perror("mkfifo");  }
  printf("mkfifo - wynik %d\n",res);
  fdes = open("MyFIFO",O_WRONLY);
  if(fdes < 0) { perror("Open");
    exit(0);
  }
  do {
       c = '0' + (i++)%10;
       printf("Zapis: %c \n", c);
       res = write(fdes, &c, 1);
  } while(i<10);
  close(fdes);
}
```

Przykład 6-5 Zapis do pliku FIFO

6.3 Funkcja select

Funkcja `select` powoduje zablokowanie procesu bieżącego do czasu wystąpienia gotowości lub błędu na którymś z deskryptorów. Odblokowuje się również wtedy gdy upłynie zadany okres oczekiwania (timeout).

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *errorfds, struct timeval * timeout)
```

Gdzie:

`nfds`	liczba deskryptorów plików (maksymalne FD_SETSIZE)
`readfds`	maska deskryptorów plików do odczytu (gotowość odczytu)
`writefds`	maska deskryptorów plików do zapisu (gotowość zapisu)
`errorfds`	maska deskryptorów plików dotyczących błędów
`timeout`	maksymalny okres zablokowania

Po wykonaniu funkcja ponownie ustawia maski bitowe `readfds`, `writefds`, `errorfds` zgodnie z wynikiem operacji czyli ustawia bity zmiennych `readfds, writefds, errorfds` na których wystąpiła gotowość.

Funkcja zwraca:
> 0 – liczbę deskryptorów na których wystąpiła gotowość
0 – gdy zakończenie na przeterminowaniu
- 1 – gdy błąd

Funkcje operujące na maskach bitowych:
fd_set - typ zdefiniowany w `<sys/time.h>`. Bit i ustawiony na 1 gdy deskryptor i obecny w zbiorze.

`void FD_ZERO( fd_set *fdset)`	Zerowanie zbioru `fdset`
`void FD_SET( int fd, fd_set fdset)`	Włączenie deskryptora `fd` do zbioru `fdset`
`int FD_ISSET( int fd, fd_set *fdset)`	Testowanie czy `fd` włączony do zbioru `fdset`
`void FD_CLR( int fd, fd_set *fdset)`	Wyłączenie `fd` ze zbioru `fdset`

Tab. 6-1 Funkcje operujące na maskach bitowych

PDF created with pdfFactory trial version www.pdffactory.com

W poniższym przykładzie proces macierzysty tworzy dwa procesy potomne. Procesy te przesyłają do procesu macierzystego napisy „Proces 1” i „Proces 2". Proces macierzysty wykorzystuje funkcję `select` do badania gotowości deskryptorów.

```
#include <sys/select.h>
#define SIZE 9
char msg[2][SIZE] = {"Proces 1","Proces 2"};

void main(void) {
 int rura[2][2];
 int i,pid,numer,bajtow, j = 0;;
 fd_set set,set1;
 char buf[SIZE];
 FD_ZERO(&set);
 printf("set: %x\n",set);
 for(i=0;i<2;i++) {
    pipe(rura[i]);
    FD_SET(rura[i][0],&set);
 }

 for(i=0;i<2;i++) { // Uruchamianie procesów potomnych
   if((pid = fork(0)) == 0) {
    potom(i,rura[i]);
    exit(0);
   }
 }

 // Macierzysty -------------
 do {
  set = set1;
  numer = select(FD_SETSIZE,&set,NULL,NULL,NULL);
  for(i=0;i<2;i++) {
        if(FD_ISSET(rura[i][0],&set)) {
              bajtow = read(rura[i][0],buf,SIZE);
              printf("Z: %d otrzymano: %s\n",i,buf);
        }
  }
  j++;
 } while(j<5);
}

int potom(int nr,int rura[2]) {
  int i;
  for(i=0;i<5;i++) {
     printf("Potomny: %d pisze: %s do: %d\n",nr, msg[nr],   rura[1]);
     write(rura[1],msg[nr],SIZE);
     sleep(1);
  }
  return(1);
}
```

Przykład 6-6 Sprawdzanie gotowości deskryptorów przy użyciu funkcji `select`

6.4 Zadania

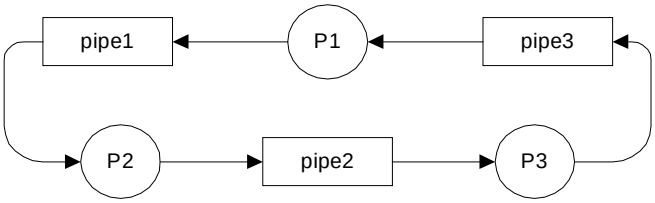
6.4.1 Prosta komunikacja przez łącza nienazwane

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.

PDF created with pdfFactory trial version www.pdffactory.com

6.4.2 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

6.4.3 Wykorzystanie funkcji popen

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

6.4.4 Zapis i odczyt z kolejki FIFO

Napisz program czytający znaki z kolejki FIFO które zapisuje tam program podany w Przykład 6-5.

6.4.5 Łącza nazwane - klient i serwer komunikatów

Korzystając z kolejek FIFO napisz prosty system komunikacyjny składający się z programów pisz i czytaj. W programie pisz jako parametry podaje się łańcuchy zamknięte w apostrofach. Łańcuchy te są wpisywane do pliku FIFO. Odczytuje je stamtąd program czytaj. Programy powinny się zachowywać jak w poniższym przykładzie.

```
$pisz ‘tekst 1’ ‘tekst 2’ ‘tekst 3’
$czytaj
tekst 1
tekst 2
tekst 3
$
```

6.4.6 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.

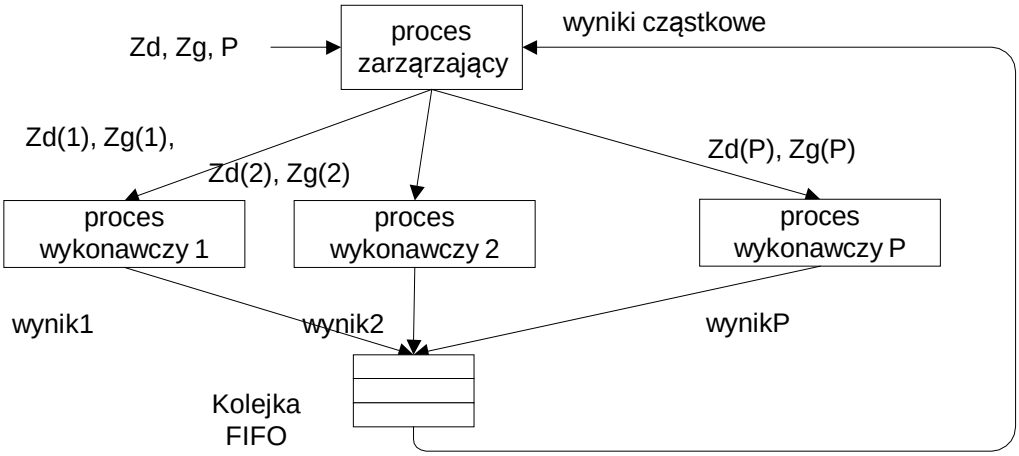
PDF created with pdfFactory trial version www.pdffactory.com

6.4.7 Znajdowanie liczb pierwszych

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 do kolejki komunikatów.



Rys. 6-1 Znajdowanie liczb pierwszych – komunikacja poprzez kolejkę FIFO

Zadanie powinno być rozwiązane w następujący sposób:

3. Program zarządzający tworzy kolejkę FIFO
4. 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
   ```

5. Proces zarządzający czeka na zakończenie wykonawczych i odczytuje z pliku FIFO dane o
6. 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)`.

PDF created with pdfFactory trial version www.pdffactory.com

6.4.8 Znajdowanie liczb pierwszych – wersja równoważąca obciążenia

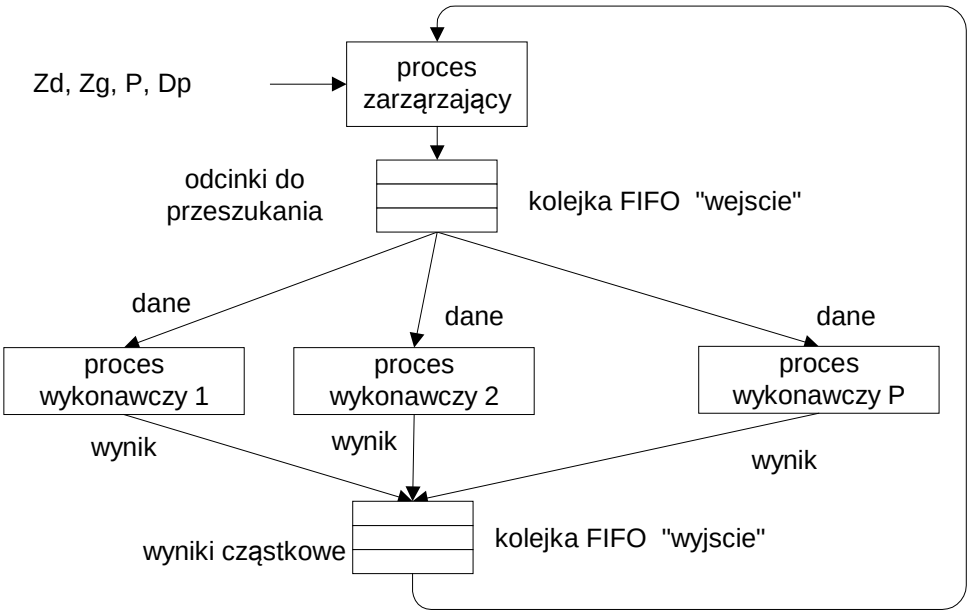
Poprzednia wersja programu znajdowania liczb pierwszych nie pozwalała na poprawne rozwiązanie problemu równoważenia obciążenia. Znaczy to tyle że procesy przeszukujące przedziały zawierające mniejsze liczby kończyły działanie wcześniej pozostawiając procesor niewykorzystany. Należy więc rozwiązać problem 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. 6-2 Znajdowanie liczb pierwszych – wersja równoważąca obciażenia

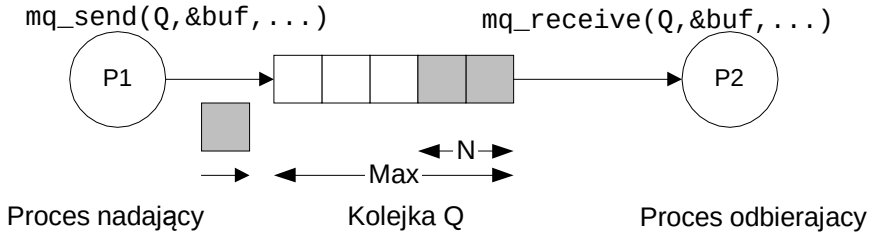
Należy wykonać eksperyment zmierzający do okreslenia optymalnej długości przedziału i liczby procesów.

PDF created with pdfFactory trial version www.pdffactory.com

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 ”/”

PDF created with pdfFactory trial version www.pdffactory.com

- 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.

```
// Kompilacja: cc wyslij.c -o wyslij -lrt
#include <stdio.h>
#include <mqueue.h>
main(int argc, char *argv[]) {
     int kom, res;
     mqd_t  mq;
     struct mq_attr attr;
     /* Utworzenie kolejki komunikatow -----------------*/
     attr.mq_msgsize = sizeof(kom);
     attr.mq_maxmsg = 1;
     attr.mq_flags = 0;
     mq=mq_open("/Kolejka", O_RDWR | O_CREAT , 0660, &attr );
     kom = atoi(argv[1]);
     res = mq_send(mq,&kom,sizeof(kom),10);
     printf("Wyslano komunikat: %d\n",kom);
     mq_close(mq);
     mq_unlink("Kolejka");
}
```

Przykład 7-1 Proces wysyłający komunikaty do kolejki

```
#include <stdio.h>
#include <mqueue.h>
main(int argc, char *argv[]) {
     int i, kom, res;
     unsigned int prior;
     mqd_t  mq;
     struct mq_attr attr;
     /* Utworzenie kolejki komunikatow -----------------*/
     attr.mq_msgsize = sizeof(kom);
     attr.mq_maxmsg = 1;
     attr.mq_flags = 0;
     mq=mq_open("/Kolejka", O_RDWR | O_CREAT , 0660, &attr );
     res = mq_receive(mq,&kom,sizeof(kom),&prior);
     printf("Otrzymano komunikat: %d\n",kom);
     mq_close(mq);
     mq_unlink("Kolejka");
}
```

Przykład 7-2 Proces 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 długośc_kolejki – 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

PDF created with pdfFactory trial version www.pdffactory.com

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:

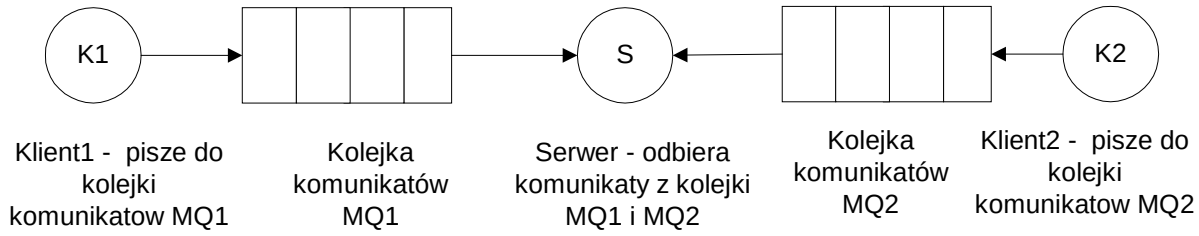
```
   for(i=0;i<10;i++) {
       msg.pnr = nr;
       msg.type = PROD;
       sprintf(msg.text,"Producent %d krok %d",nr,i);
       // Przeslanie komunikatu do kolejki
       res = mq_send(mq,&msg,sizeof(msg),priority);
       ......
       sleep(1);
   }
```

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 Odbiór komunikatów z wielu źródeł

Napisz kod serwera oraz 2 klientów. Klient 1 wysyła komunikaty do kolejki komunikatów MQ1 (funkcja `mq_send`). Klient 2 wysyła komunikaty do kolejki MQ2. Rozwiąż ten problem wykorzystując możliwość generacji sygnału przy zmianie stanu kolejki komunikatów MQ1 (funkcja `mq_notify`) lub funkcję `select`.



Rys. 7-2 Serwer odbiera komunikaty z kolejek MQ1 i MQ2

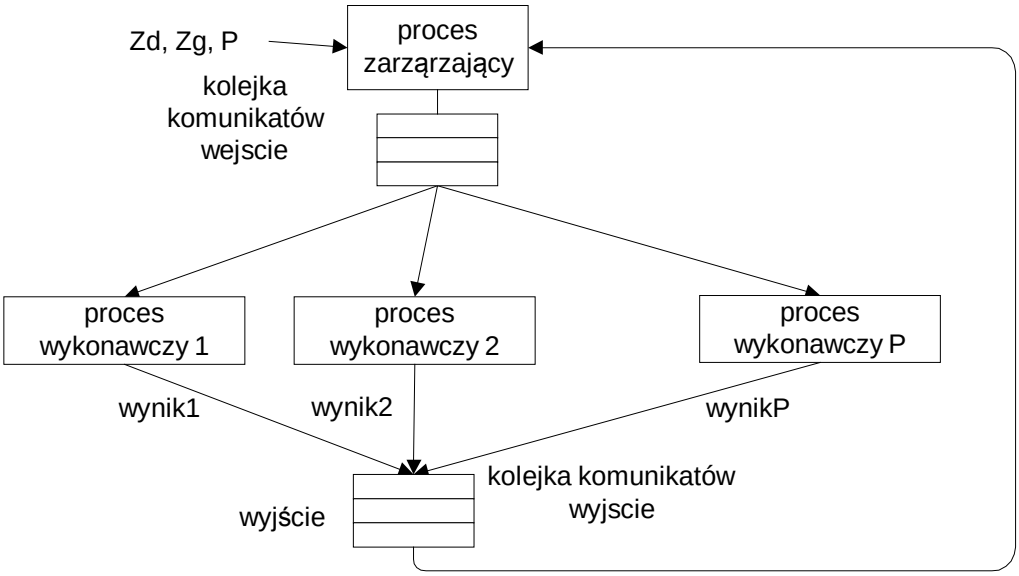
PDF created with pdfFactory trial version www.pdffactory.com

7.2.3 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.

7.2.3.1 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

PDF created with pdfFactory trial version www.pdffactory.com

- Zakres górny przedziału
- Liczbę procesów wykonawczych

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

7.2.4 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.

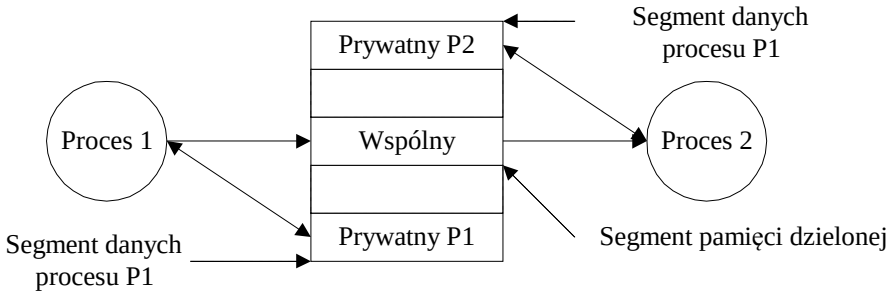
PDF created with pdfFactory trial version www.pdffactory.com

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.



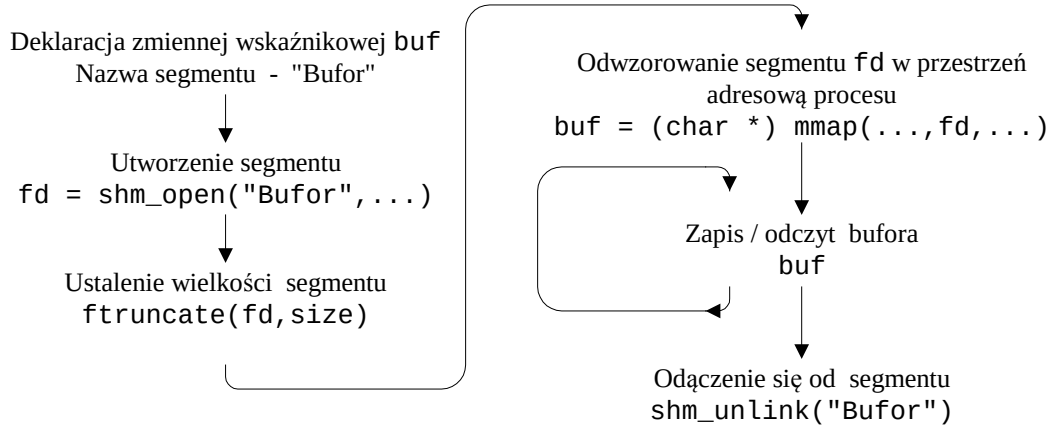
Rys. 8-1 Procesy P1 i P2 komunikuję się poprzez wspólny obszar pamięci

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.

PDF created with pdfFactory trial version www.pdffactory.com

```
// 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`

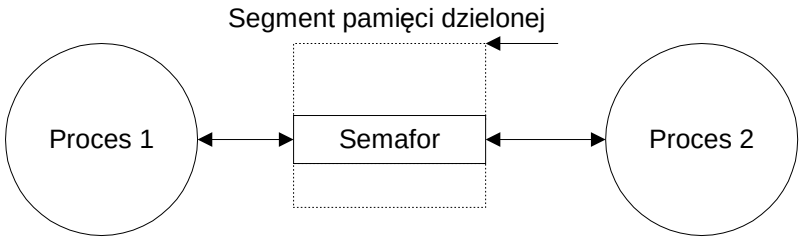
PDF created with pdfFactory trial version www.pdffactory.com

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()`.

PDF created with pdfFactory trial version www.pdffactory.com

```
// Program producenta-konsumenta (C) J. Ulasiewicz 2013
// Demonstruje dzialanie pamieci dzielonej i semaforow
// Kompilacja gcc prod_kons.c -o prod_kons -lrt
#include <sys/mman.h>
#include <fcntl.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.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;

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);
       printf("Producent - cnt:%d head: %d tail: %d\n",
               wbuf-> cnt,wbuf->head,wbuf->tail);
       sem_wait(&(wbuf->empty));
       sem_wait(&(wbuf->mutex));
```


PDF created with pdfFactory trial version www.pdffactory.com

```
       sprintf(wbuf->buf[wbuf->head],"Komunikat %d",i);
       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++) {
     printf("Konsument - cnt: %d odebrano %s\n",wbuf->cnt
             ,wbuf->buf[wbuf->tail]);
     sem_wait(&(wbuf->full));
     sem_wait(&(wbuf->mutex));
     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 Rozwiązanie problemu producenta i konsumenta za pomocą semaforów nienazwanych

8.3 Zadania

8.3.1 Szukanie liczb pierwszych w przedziale – komunikacja przez pamięć dzieloną

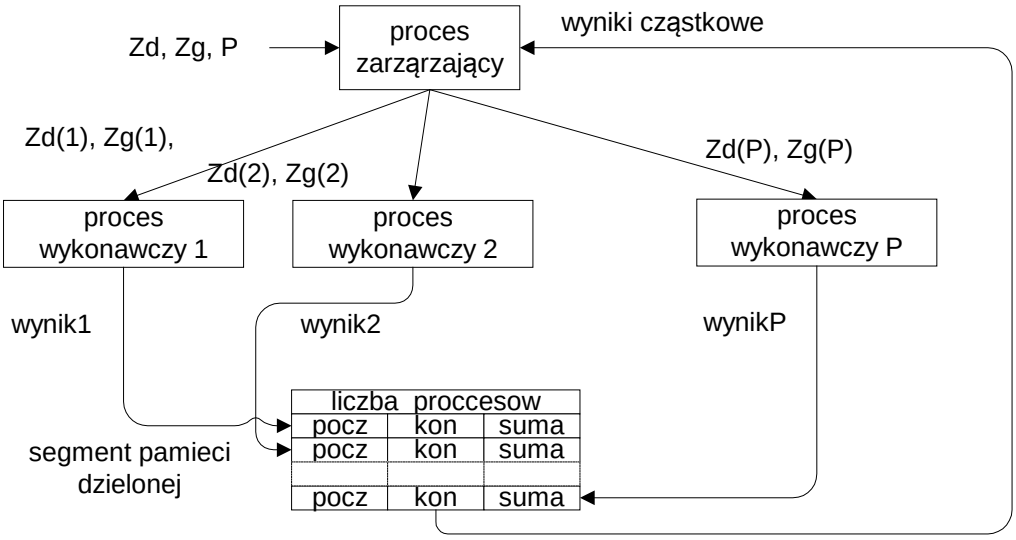
Napisz program który ma znajdować liczby pierwsze w zadanym przedziale [Zd,...,Zg]. Obliczenia można przyspieszyć dzieląc zakres [Zd,...,Zg] na P podprzedziałów [Zd(1),...,Zg(1)], [Zd(2),...,Zg(2)],..., [Zd(P),...,Zg(P)] gdzie P jest liczbą dostępnych procesorów. W każdym z podprzedziałów [Zd(i),...,Zg(i)] 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.

PDF created with pdfFactory trial version www.pdffactory.com



Rys. 8-4 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`. 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:

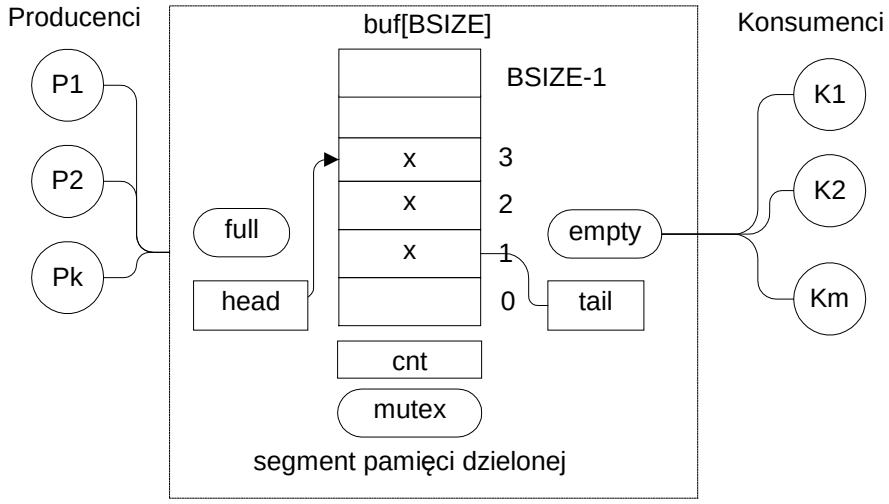
- Zakres dolny przedziału
- Zakres górny przedziału
- Liczbę procesów wykonawczych

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

8.3.2 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.

PDF created with pdfFactory trial version www.pdffactory.com



Rys. 8-5 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 dlugosc_bufora` – 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.

8.3.3 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
```


PDF created with pdfFactory trial version www.pdffactory.com

```
} bufor_t

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

Rozwiązanie podane zostało w pracy [1].

PDF created with pdfFactory trial version www.pdffactory.com

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

Różnice pomiędzy stylami komunikacji podaje [10]. W komunikacji UDP każdy komunikat adresowany jest oddzielnie a ponadto zachowywane są granice przesyłanych komunikatów. W komunikacji bezpołączeniowej stosowane są następujące funkcje:

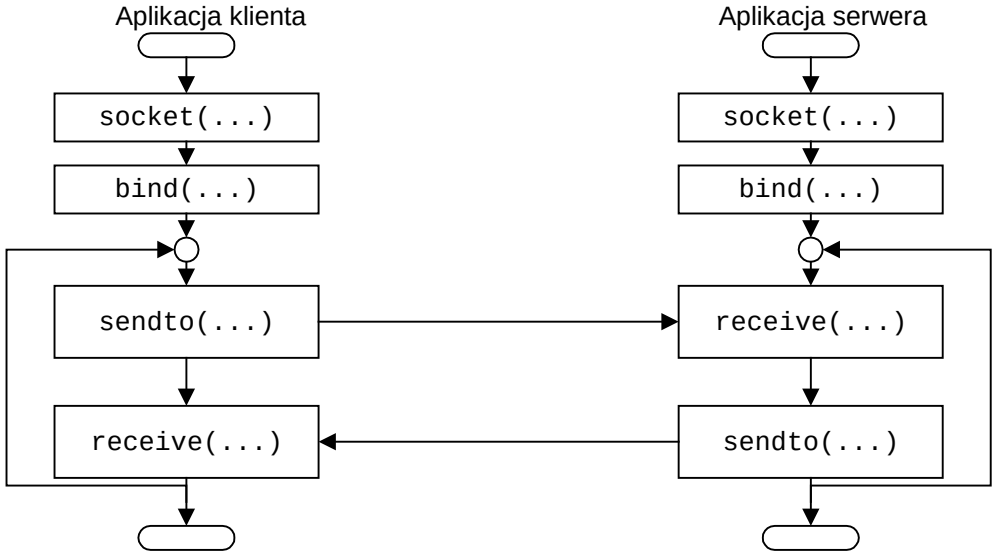
`socket`	Utworzenie gniazdka
`bind`	Powiązanie z gniazdkiem adresu IP
`sendto`	Wysłanie datagramu do odbiorcy
`recfrom`	Odbiór datagramu
`htons, htonl`	Konwersja formatu lokalnego liczby do formatu sieciowego (s – liczba krótka, s – liczba długa)
`ntohs, ntohl`	Konwersja formatu sieciowego liczby do 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 podręczniku ich parametry i znaczenia. Kolejność działań podejmowanych przez klienta i serwera podana jest poniżej a ich współpracę pokazuje Rys. 9-1.

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

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



Rys. 9-1 Przebieg komunikacji bezpołączeniowej

PDF created with pdfFactory trial version www.pdffactory.com

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

PDF created with pdfFactory trial version www.pdffactory.com

```
// 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>
 #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.

PDF created with pdfFactory trial version www.pdffactory.com

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
```

Przykładowo 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

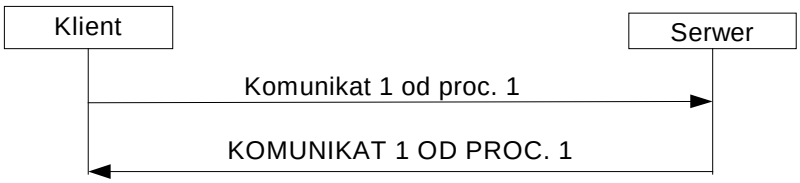
```
$./udp_cli 192.168.0.160
```

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

Proces serwera

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 klienta do serwera przesyłane następujące rodzaje komunikatów:

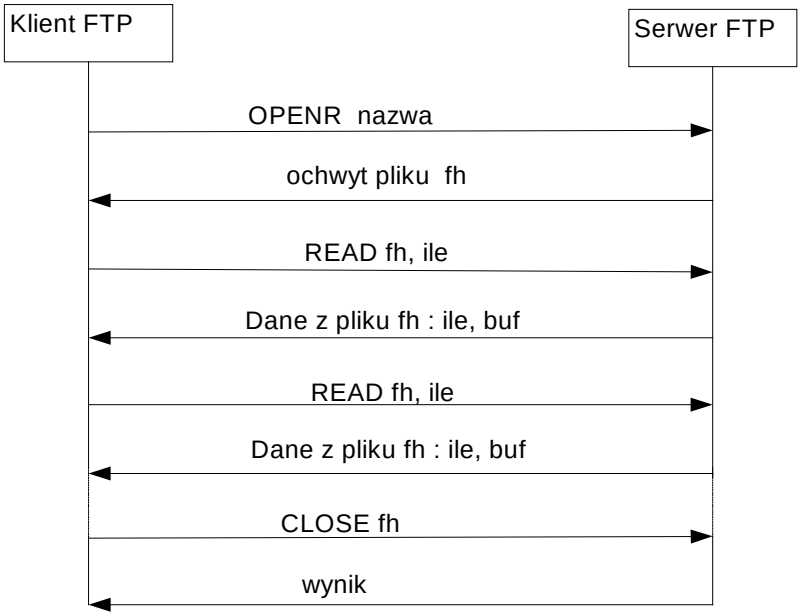
PDF created with pdfFactory trial version www.pdffactory.com

```
#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 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 argumenty programów podajemy nazwę pod którą rejestruje się serwer. Przedstawiony wyżej serwer jest iteracyjnym serwerem stanowym.

Rozszerzenia:

- Dodaj funkcję zapisu 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 Rozproszona aplikacja „hotel”

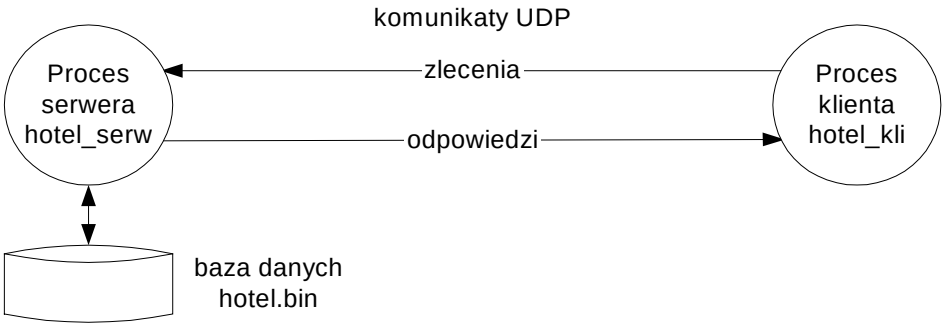
Napisz prostą rozproszoną aplikacją obsługującą hotel. Wykorzystaj wyniki pracy uzyskane we wcześniejszym ćwiczeniu 5.4.6. Baza danych zawarta będzie w pliku ”`hotel.bin`” który zawiera rekordy:

PDF created with pdfFactory trial version www.pdffactory.com

```
typedef struct {
      int wolny;  // 1 gdy pokoj wolny 0 gdy nie
      int numer;  // Numer pokoju
      char nazwisko[NSIZE]; // Nazwisko goscia
      char imie[NSIZE];     // Imie goscia
} pokoj_t;
```

Aplikacja powinna się składać z procesu serwera `hotel_serw` obsługującego plik bazy danych i odbierającej zlecenia klienta oraz procesu klienta `hotel_kli` wysyłającego zlecenia do procesu serwera. Komunikacja pomiędzy procesami odbywać się powinna za pomocą komunikatów UDP. Można przyjąć następujący ich format:

```
typedef struct {
  int cmd;    // Polecenie
  int wolny;
  int numer;
  char nazwisko[NSIZE];
  char imie[NSIZE];
  int  wynik; // Wynik polecenia
} kom_t;
```



Rys. 9-2 Schemat rozproszonej aplikacji hotel

Aplikacja powinna realizować następujące funkcje:

Funkcja	Komunikat	Funkcja
Zameldowanie gościa w pierwszym wolnym pokoju, podaje się imię i nazwisko. Pole wolny = 0, do pól imię i nazwisko kopiujemy wymagane wartości z komunikatu `msg`.	`MELDUJ`	`melduj(bazaf,&msg)`
Wymeldowanie gościa z danego pokoju. Przy wymeldowaniu podaje się pokój. Pole wolny = 1, pola imię i nazwisko są zerowane.	`WYMELD`	`wymeld(bazaf,&msg)`
Podanie informacji kto jest zameldowany w danym pokoju. Dla danego pokoju wyprowadzamy imię i nazwisko gościa.	`INFO`	`info(bazaf,&msg)`
Sprawdzenie jaka osoba zameldowana jest w danym pokoju. Przeszukujemy rekordy bazy i szukamy rekordu w którym pole nazwisko posiada podaną wartość.	`ZNAJDZ`	`znajdz(bazaf,&msg)`
Wyprowadzenie na konsolę listy pokojów z informacją kto je zajmuje i ile jest wolnych pokojów.	`LISTA`	`info(bazaf,&msg)`
Inicjacja bazy danych, zapis pustych rekordów z informacja że pokój jest wolny. Bazę inicjujemy na zadaną liczbę pokojów.	-	`init(bazaf,POKOJOW)`

PDF created with pdfFactory trial version www.pdffactory.com

```
int main(int argc, char*argv[]) {
   struct sockaddr_in adr_moj, adr_cli;
   int s, i, slen=sizeof(adr_cli),snd, rec, blen=sizeof(kom_t);
   int opt,res,stop=0;
   // Utworzenie gniazdka ----
   s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
   if(s < 0) blad("socket");
   printf("Gniazdko %d utworzone\n",s);
   // Ustalenie adresu IP gniazdka
   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)
       blad("bind");

   do { // Odbior komunikatu -------------------
      rec = recvfrom(s, &msg, blen, 0, &adr_cli, &slen);
      if(rec < 0) blad("recvfrom()");
      switch(msg.cmd) {
       case MELD     : melduj(bazaf,&msg);  break;
       case WYMELD   : wymeld(bazaf,&msg);  break;
       case KTO      : info(bazaf,&msg);     break;
       case ZNAJDZ   : znajdz(bazaf,&msg);  break;
       case WYJSCIE  : _exit(0);
       default       :   ;
     }
     // Odpowiedz ---------
     snd = sendto(s, &msg, blen, 0, &adr_cli, slen);
     if(snd < 0) blad("sendto()");
   } while(stop != 1);
   return 0;
}
```

Przykład 9-3 Szkic procesu serwera aplikacji hotel_udp

```
#include "hotel_wsp.h"
pokoj_t pokoj;
kom_t msg;

struct sockaddr_in adr_moj, adr_serw, adr_x;


int menu(void) {
  int opcja;
  printf("\n1 - meldowanie\n");
  printf("2 - wymeldowanie\n");
  printf("3 - kto w pokoju\n");
  printf("4 - lista\n");
  printf("5 - znajdz\n");
  printf("6 - wyjscie\n");
  scanf("%d",&opcja);
  return opcja;
}

int main(int argc, char*argv[]) {
   int opt,res,stop=0;
   int s, i, slen=sizeof(adr_serw), snd, blen=sizeof(kom_t),rec;
   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));
```


PDF created with pdfFactory trial version www.pdffactory.com

```
  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);
  }

  do {
        opt = menu();
        switch(opt) {
        case MELD: // Meldowanie ---
         msg.cmd = MELD;
         printf("Podaj imie: \n");
         scanf("%s",(char *)&msg.imie);
         …
         send_rec(s, &msg, &msg);
         break;

        case WYMELD:
        case KTO:      ... ; break;
        case LISTA:    ... ; break;
        case ZNAJDZ:   ... ; break;
        case WYJSCIE:  _exit(0);
        default:               ;
    }

  } while(stop != 1);
  return 0;
}

int send_rec(int s, kom_t *snd, kom_t *rcv)
// Wyslanie komunikatu snd do gniazdka s snd
// odbior w rcv, funkca zwraca 1 gdy ok
{  int res;
   int slen, blen=sizeof(kom_t);
   slen=sizeof(adr_serw);
   res = sendto(s, snd, blen, 0, &adr_serw, slen);
   if(res < 0) blad("sendto()");
   res = recvfrom(s, rcv, blen, 0, &adr_x, &slen);
   if(res < 0) blad("recvfrom()");
   return 1;
}
```

Przykład 9-4 Szkic procesu klienta aplikacji hotel_udp

9.2.4 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;
```


PDF created with pdfFactory trial version www.pdffactory.com

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`

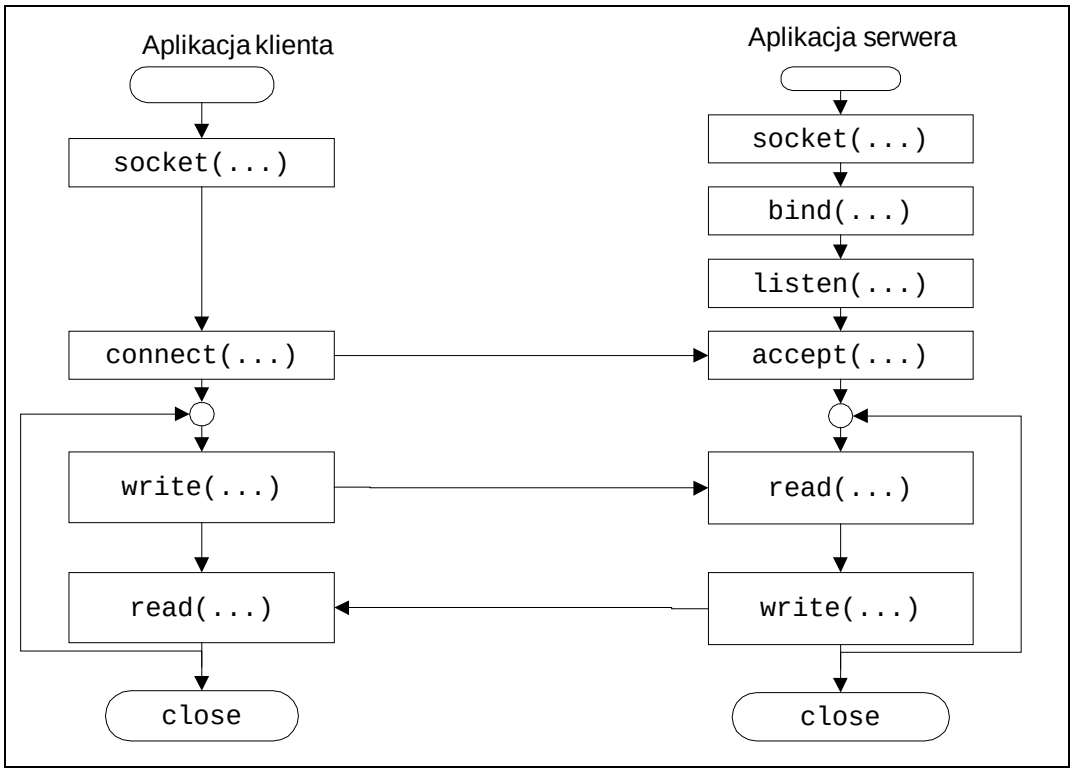
Serwer:

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

PDF created with pdfFactory trial version www.pdffactory.com



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;
```


PDF created with pdfFactory trial version www.pdffactory.com

```
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

PDF created with pdfFactory trial version www.pdffactory.com

```
// 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");
```


PDF created with pdfFactory trial version www.pdffactory.com

```
   // 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  );
}
```

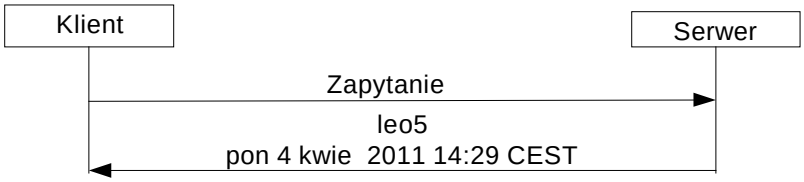
Przykład 10-2 Klient `tcp_cli.c` w trybie z kontrolą połączenia

10.2 Zadania

10.2.1 Przesyłanie komunikatów pomiędzy komputerami – uzyskiwanie czasu i nazwy zdalnego komputera

Serwer odbiera komunikaty wysyłane przez klientów i podaje swoja nazwę i czas bieżacy
. Procesy klienta i serwera uruchamiane są niezależnie z linii poleceń.

```
typedef struct  {
   int  typ;          // typ komunikatu
   int  nr_pol;       // Numer polaczenia (dla wersji polaczeniowej)
   char nazwa[SIZE1]; // nazwa komputera
   char czas[SIZE2];  // Data i czas lokalny w postaci łańcucha
 } mss_t;
```



Rys. 10-1 Współpraca klienta i serwera

Proces serwera

Serwer wykonuje następujące kroki:

- Utworzenie gniazdka, wejście w tryb nasłuchu i oczekiwanie na połączenia
- Obiór zleceń klientów.
- Odpowiedź na zlecenia klientów polegająca na podaniu nazwy komputera (funkcja `hostname`) i daty i czasu bieżącego (funkcja `ctime`).

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
- Połączenie się z serwerem
- Wysyłanie komunikatu z zapytaniem do serwera. Pole `typ` ma zawierać 1.
- Odbiór i wyświetlanie odpowiedzi serwera.

PDF created with pdfFactory trial version www.pdffactory.com

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. Numer połączenia podać w polu `nr_pol` struktury.

10.2.3 Klient i serwer usługi FTP

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

PDF created with pdfFactory trial version www.pdffactory.com

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

10.2.4.1 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.

PDF created with pdfFactory trial version www.pdffactory.com

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

PDF created with pdfFactory trial version www.pdffactory.com

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) .

PDF created with pdfFactory trial version www.pdffactory.com

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>
#define NUM_THREADS 2
#define KROKOW      4
pthread_t tid[NUM_THREADS]; // Tablica identyfikatorow watkow

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);
   }
   return (void*) 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 *) &status);
             printf("Watek %d zakonczony\n",status );
   }
   return  0;
}
```

Przykład 12-1 Program 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.

PDF created with pdfFactory trial version www.pdffactory.com

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

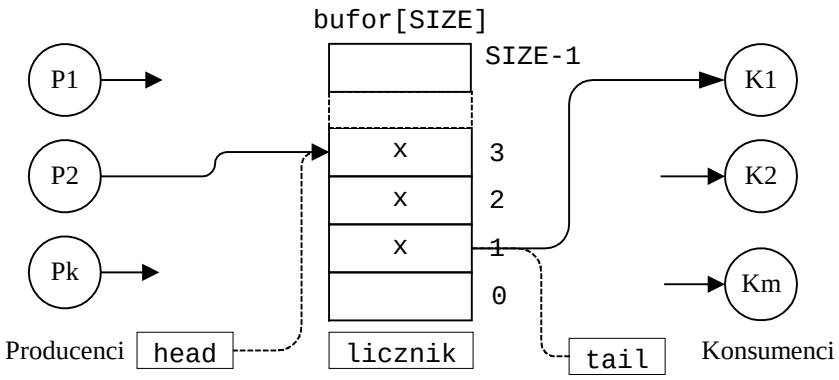
12.3.1 Problem producenta i konsumenta

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`. Struktura zawierająca bufor i zmienne pomocnicze dana jest poniżej.

```
typedef struct {
    char buf[SIZE][LSIZE];
    int head;
    int tail;
    int cnt;
    pthread_cond_t empty;
    pthread_cond_t full;
    pthread_mutex_t mutex;
} buf_t;
```

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

PDF created with pdfFactory trial version www.pdffactory.com



Rys. 12-1 Problem producenta i konsumenta

12.3.2 Szukanie liczb pierwszych w aplikacji wielowątkowej

Liczba jest pierwsza gdy dzieli się przez 1 i przez siebie samą. 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 watki. 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 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
} par_t
```

Wyniki działania wątków (liczba liczb pierwszych w zakresie) powinna być zwracana jako kod powrotu wątku.

12.3.3 Szukanie liczb pierwszych w aplikacji wielowątkowej – równomierne obciążenie wątków

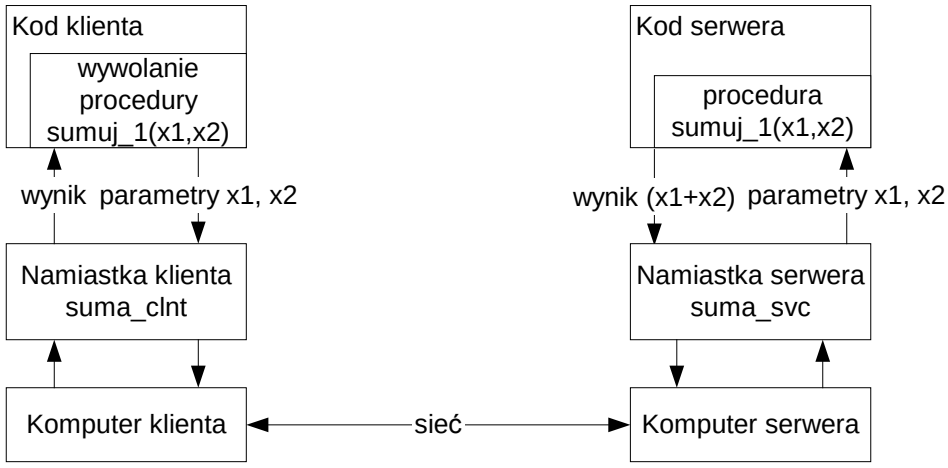
W poprzednim przykładzie wątki liczące wyższe zakresy liczb są bardziej obciążone i kończą się później. Dokonaj takiej modyfikacji programu aby problem ten był rozwiązany przy równomiernym obciążeniu procesorów. Można postępować w taki sposób że wątki robocze zgłaszają swoją gotowość wątkowi sterującemu 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.

PDF created with pdfFactory trial version www.pdffactory.com

13. Zdalne wywoływanie procedur

13.1 Podstawy

Zdalne wywoływanie procedur RPC (*ang. Remote Procedures Call*) jest jedną z podstawowych metod stosowanych w przetwarzaniu rozproszonym i równoległym. Celem niniejszego ćwiczenia jest zapoznanie się ze środowiskiem Sun RPC opisanym w [5] i metodami tworzenia w tym środowisku aplikacji równoległych i rozproszonych. Aby upewnić się że środowisko jest odpowiednio skonfigurowane należy sprawdzić czy dostępna jest program `rpcgen` (polecenie: `$which rpcgen`) i czy uruchomiony jest program `portmap` (polecenie: `$sin | grep portmap`). Dla ilustracji sposobu posługiwania się metodą rozpatrzony zostanie przykład zdalnego wywołania procedury sumowania dwu liczb.



Rys. 13-1 Przebieg zdalnego wywołania procedury sumowania dwu liczb

Pierwszą czynnością jest utworzenie pliku `suma.x` definicji interfejsu zawierającego definicję procedury `sumuj` i potrzebnych argumentów co pokazuje Przykład 13-1.

```
struct integers{
  int x1;
  int x2;
};

typedef struct integers intargs;

program PROG3 {
  version WERS1 {
    int sumuj(intargs) = 1;
    } = 1;
} =  0x30000005;
```

Przykład 13-1 Plik definicji interfejsu `suma.x`

Następnie program należy skompilować prekompilatorem `rpcgen` jak poniżej:

```
$rpcgen suma.x
```

Prekompilator wyprodukuje następujące pliki:

`suma.h`	Plik nagłówkowy
`suma_svc.c`	Namiastka serwera
`suma_clnt.c`	Namiastka klienta
`suma_xdr.c`	Plik konwersji danych

Tabela 13-1 Pliki generowane przez prekomplilator rpcgen

W pliku `suma.h` zawarte są prototypy funkcji `sumuj` widzianej po stronie klienta i serwera co pokazuje Przykład 13-2.

PDF created with pdfFactory trial version www.pdffactory.com

```
struct integers {
      int x1;
      int x2;
};
typedef struct integers integers;

extern  int * sumuj_1(intargs *, CLIENT *);
extern  int * sumuj_1_svc(intargs *, struct svc_req *);
```

Przykład 13-2 Fragment pliku suma.h

Kolejnym krokiem jest implementacja funkcji `sumuj` po stronie serwera. W tym celu należy utworzyć plik `sserwer.c` co pokazuje Przykład 13-3.

```
// Kompilacja: gcc sserwer.c suma_svc.c suma_xdr.c -o sserwer -lrpc

#include "suma.h"

int *sumuj_1_svc(intargs* arg, struct svc_req * s) {
  static int result;
   result = arg->x1 + arg->x2;
   return &result;
}
```

Przykład 13-3 Plik serwera `sserwer.c` zawierający implementację zdalnie wywoływanej procedury `sumuj`

Tak otrzymany plik należy skompilować do postaci wykonywalnej:

```
$gcc sserwer.c suma_svc.c suma_xdr.c -o sserwer –lrpc
```

W wyniku kompilacji otrzymujemy plik serwera `sserwer` który należy uruchomić na maszynie pełniącej funkcję serwera.

```
$./sserwer &
```

Uruchomiony program powinien się zarejestrować w łączniku `portmap` co można sprawdzić poleceniem `rpcinfo` co pokazuje `Przykład 13-4`

```
$rpcinfo -p
program vers proto   port
    100000    4   tcp    111  portmapper
    ...
    100000    2     0    111  portmapper
    ...
 805306373    1   udp  65297
 805306373    1   tcp  65485 <- to jest nasza procedura zdalna
```

Przykład 13-4 Działanie polecenia `rpcinfo`

Znając numer procedury można korzystając z programu `rpcinfo –t nazwa_komputera numer_procedury` sprawdzić jej gotowość co pokazano poniżej.

```
$rpcinfo –t localhost  805306373
program 805306373 version 1 ready and waiting
```

Kolejnym etapem jest utworzenie kodu klienta który wywołuje procedurę sumuj. Kod takiego programu o nazwie `sklient` zawiera Przykład 13-5.

```
// Kompilacja: gcc sklient.c suma_clnt.c suma_xdr.c -o sklient -lrpc
#include "suma.h"

int main(int argc, char *argv[]) {
  CLIENT *c1;
  integers arg;
  int *sum;
  char* host = "localhost"; // Zmienic nazwe gdy serwer na innym komputerze
```


PDF created with pdfFactory trial version www.pdffactory.com

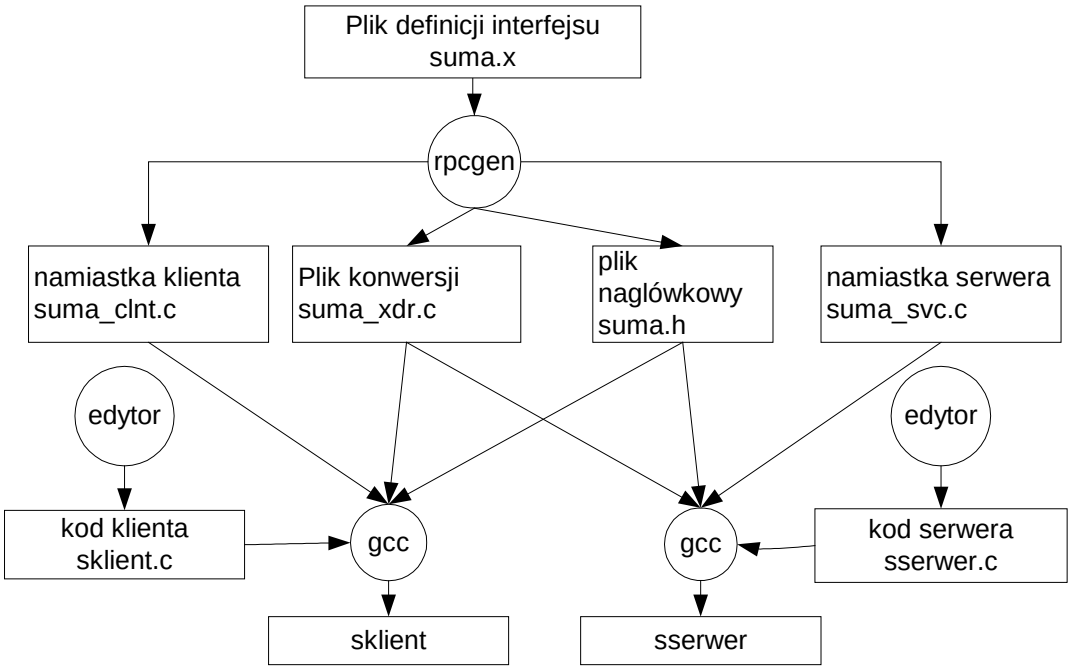
```
  if(argc < 3) {
     printf("Uzycie: ssuma l1 l2\n");
     exit(0);
  }
  c1 = clnt_create(host, PROG3, WERS1, "tcp");
  if (c1 == NULL) {
     clnt_pcreateerror(host);
     return -1;
  }
  arg.x1 = atoi(argv[1]);
  arg.x2 = atoi(argv[2]);
  sum = (int*) sumuj_1(&arg, c1);
  if(sum == NULL) {
    clnt_perror(c1, "blad wywolania RPC");
    return -1;
  }
  printf(" %d + %d = %d \n",arg.x1,arg.x2,*sum);
  clnt_destroy(c1);
  return 0;
}
```

Przykład 13-5 Plik klienta `sklient.c` zawierający zdalnie wywołanie procedury `sumuj`

Po utworzeniu programu należy go skompilować w podany poniżej sposób:

```
gcc sklient.c suma_clnt.c suma_xdr.c -o sklient -lrpc
```

Następnie należy program uruchomić pisząc `$./sklient`. W podanym przykładzie klient i serwer wykonywane są na jednym komputerze. Aby przetestować wywołanie zdalne należy w przykładzie zmienić nazwę zmiennej host na adres IP lub nazwę komputera na którym wykonywany jest serwer. Przebieg tworzenia aplikacji sumowania podaje Rys. 13-2.



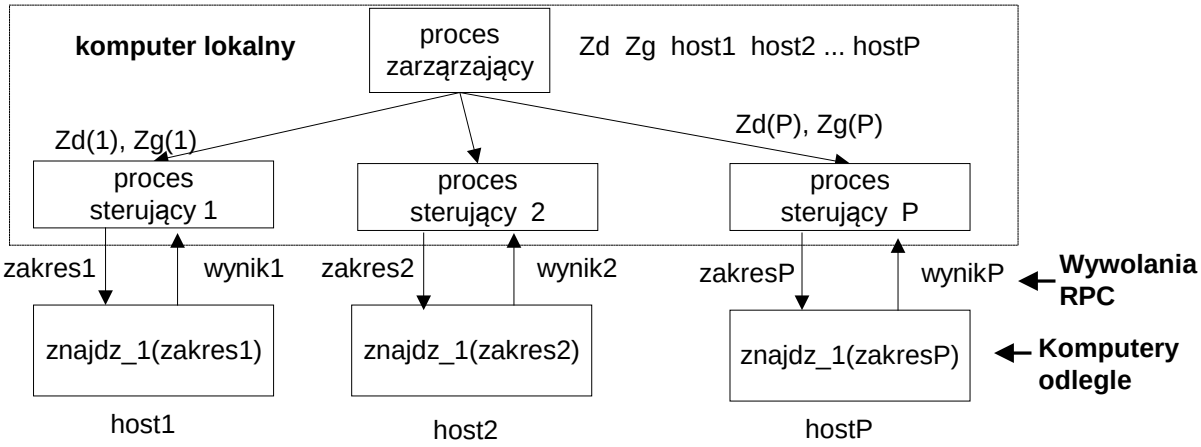
Rys. 13-2 Tworzenie aplikacji sumowania dwu liczb

PDF created with pdfFactory trial version www.pdffactory.com

13.2 Zadania

13.2.1 Równoległe znajdowanie liczb pierwszych z wykorzystaniem RPC – wersja ze współbieżnymi procesami

Napisz program który ma w sposób równoległy znajdować liczby pierwsze w zadanym przedziale [Zd,...,Zg] tak jak w poprzednim rozdziale. 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 komputerów. W każdym z podprzedziałów [Zd(i),...,Zg(i)] możemy znajdować liczby pierwsze niezależnie a więc obliczenia wykonane mogą być równolegle na różnych węzłach. Na poszczególnych węzłach należy uruchomić serwery RPC wykonujące procedurę znajdowania liczb pierwszych w przedziale.



Rys. 13-1 Równoległe znajdowanie liczb pierwszych z wykorzystaniem RPC

Zadanie powinno być rozwiązane w następujący sposób:
1. Tworzymy plik definicji interfejsu `lpierw.x` tak jak poniżej:

```
struct  zakres {
   int odl;   // Poczatek zakresu
   int dol;   // Koniec zakresu
   int numer; // Numer przedzialu
};
program PROG1 {
   version WERS1 {
     int znajdz(zakres) = 1;
   } = 1;
} = 0x30000004;
```

Plik kompilujemy za pomocą prekompilatora `rpcgen` otrzymując pliki: `lpierw.h, lpierw_svc.c, lpierw_clnt.c lpierw_xdr.c`.

2. Implementujemy funkcję `int znajdz(zakres)` która w podanym zakresie znajduje liczbę liczb pierwszych i zwraca te liczbę jako kod powrotu. Następnie tworzymy serwer `lserw` który uruchamiany na kolejnych węzłach.

3. Tworzymy aplikację klienta. Program ten powinien mieć następujące argumenty: zakres dolny przedziału, zakres górny przedziału, nazwy lub adresy kolejnych węzłów na których mają być prowadzone obliczenia:
   `lklient zd zg host1 host2 ... hostP`
   Aplikacja klienta działa w następujący sposób:
   - Dzieli przedział zd zg na P podprzedziałów
   - Tworzy P procesów potomnych z których wywoływana jest funkcja `znajdz(zakres)`
   - Czeka na zakończenie procesów potomnych i odbiera status
   - Podaje wynik końcowy i czas obliczeń

4. Uruchamiamy aplikacje klienta i wykonujemy testy.

Należy rozwiązać problem przekazywania wyników z procesów sterujących do procesu zadządzającego. Należy określić czasy obliczeń dla jednego, dwóch, czterech i ośmiu węzłów. Proszę narysować wykres pokazujący zależność czasu obliczeń od liczby węzłów sieci.

PDF created with pdfFactory trial version www.pdffactory.com

14. System pogawędki sieciowej - IRC

14.1 Sformułowanie problemu

Posługując się poznanymi wcześniej mechanizmami komunikacji miedzy procesowej można zaimplementować system pogawędki sieciowej. System taki składa się z procesu serwera usługi oraz programów klientów. Z funkcjonalnego punktu widzenia działanie systemu jest następujące:

1. Na jednym z komputerów uruchamiany jest serwer usługi: `irc_serwer &.`
2. Program `irc_klient` uruchomiony może być z dowolnego komputera który widzi komputer serwera. Program klienta powinien umożliwiać:
   - Zgłoszenie się (logowanie) do systemu, przy zgłoszeniu podaje się osobisty identyfikator (ang. *Nickname*)
   - Po zgłoszeniu się klient może obserwować wszystkie komunikaty przesyłane w systemie
   - Przesłać własny komunikat do systemu
   - Wymeldować się z system

Przykładowo komunikaty mogą być postaci:

[Jurek] > Co u was słychać ?
[Zenek] > Mam dużo pracy
[Mundek] > A ja mam dużo czasu
.....

14.2 Wymagania

Aby zaprojektować aplikacje trzeba rozwiązać kilka problemów projektowych:
1. Określić sposób komunikacji międzyprocesowej – komunikaty UDP, komunikacja połączeniowa, RPC
2. Określić który proces wysyła komunikaty a który je odbiera oraz określić liczbę procesów aplikacji klienta i serwera
3. Określić format przesyłanych komunikatów.
4. Rozwiązać problem oddzielenia okna odbiorczego i nadawczego. Jeżeli nie tworzy się oddzielnych okien należy zapewnić aby w trakcie pisanie własnego komunikatu nie przeszkadzały komunikaty od innych klientów i aby tych komunikatów nie utracić.
5. Określić jakie struktury danych powinien przechowywać serwer.

Należy wziąć pod uwagę fakt że serwer ma obsługiwać wielu klientów i powinien być zawsze dostępny. Stąd jego działanie nie może być uzależnione od błędnego działania sieci lub klienta. Pierwszym stąd wnioskiem jest wymaganie aby serwer nie wykonywał operacji Send. Jak wiadomo operacja ta może spowodować zablokowanie wykonującego ją procesu.

14.3 Format komunikatów

Chyba najprościej jest gdy klient wysyła komunikaty do serwera. Powinny być zaimplementowane przynajmniej następujące typy komunikatów wysyłane od klienta do serwera:

LOG_IN	Zgłoszenie się do systemu
ASK_DATA	Zapytanie czy są nowe dane
MY_DATA	Wysłanie nowych danych
LOG_OUT	Wylogowanie się z systemu

Z kolei od serwera do klienta powinny być przesyłane dwa typy komunikatów

NEW_DATA	Przesłanie nowych danych
NO_DATA	Brak nowych danych

Komunikat powinien zawierać przynajmniej następujące pola:
- Typ komunikatu
- Numer węzła z którego wysłano komunikat
- Kolejny numer komunikatu
- Identyfikator nadawcy
- Tekst komunikatu

PDF created with pdfFactory trial version www.pdffactory.com

Może być więc następująca struktura:

```
typedef stuct {
    int mtyp;               // Typ komunikatu
    int node;               // Numer węzła
    int numer;              // Numer komunikatu
    char nick[N_SIZE];      // Identyfikator nadawcy
    char text[T_SIZE];      // Tekst komunikatu
      .....                 // inne dane
} irc_msg;
```

Literatura

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

PDF created with pdfFactory trial version www.pdffactory.com