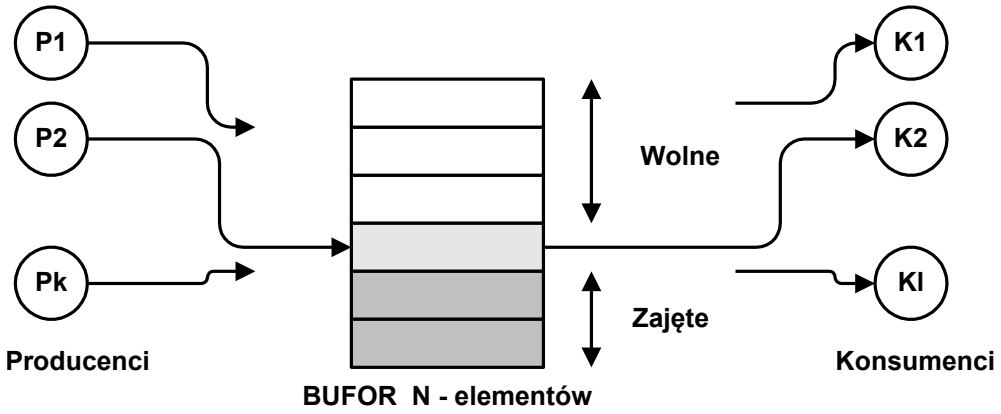
10. Synchronizacja użycia zasobów, Semafory

10.1 Problem producenta i konsumenta

Zagadnienie kontroli użycia jednostek zasobu
W systemie istnieje pula N jednostek zasobu pewnego typu. Procesy mogą pobierać z puli zasoby i je zwracać. Gdy brak jest zasobu a pewien proces będzie próbował go pobrać ulega on zablokowaniu. Proces zostanie odblokowany gdy inny proces zwróci jednostkę zasobu.

1. W systemie istnieją dwie grupy procesów – producenci i konsumenci oraz bufor na elementy.
2. Producenci produkują pewne elementy i umieszczają je w buforze.
3. Konsumenci pobierają elementy z bufora i je konsumują .
4. Producenci i konsumenci przejawiają swą aktywność w nie dających się określić momentach czasu.
5. Bufor ma pojemność na N elementów.



Należy prawidłowo zorganizować pracę systemu:

1. Gdy są wolne miejsca w buforze producent może tam umieścić swój element. Gdy w buforze brak miejsca na elementy producent musi czekać. Gdy wolne miejsca się pojawią producent zostanie odblokowany.
2. Gdy w buforze są jakieś elementy konsument je pobiera. Gdy brak elementów w buforze konsument musi czekać. Gdy jakiś element się pojawi, konsument zostanie odblokowany.

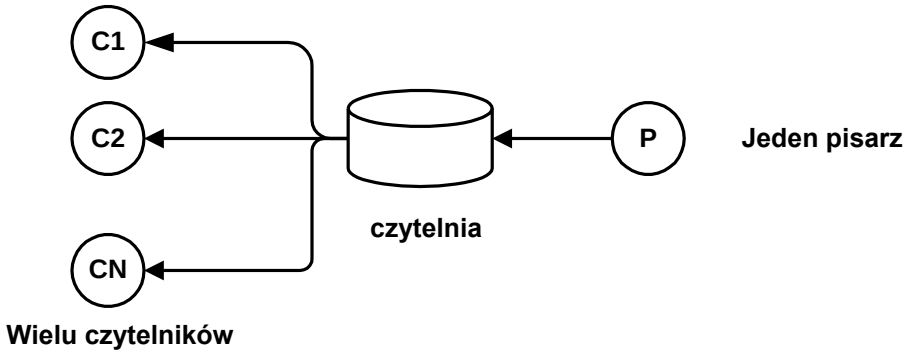
Bufor zorganizowany może być na różnych zasadach:

1. Kolejka FIFO (bufor cykliczny).
2. Kolejka LIFO (stos).

Uwaga:
Umieszczanie / pobieranie elementu z bufora jest sekcją krytyczną.

10.2 Problem czytelników i pisarzy

W systemie istnieją dwie grupy procesów – czytelnicy i pisarze. Czytanie może się odbywać współbieżnie z innymi procesami natomiast pisanie, w celu zapewnienia spójności danych, musi się odbywać na zasadzie wyłączności.



Rysunek 10-1 Problem czytelników i pisarzy

- Czytelnik może wejść do czytelni gdy jest ona pusta lub gdy są tam inni czytelnicy
- Pisarz może wejść do czytelni gdy jest ona pusta

Problem jest uogólnieniem problemu dostępu wielu procesów do bazy danych.

10.3 Semafory

Semafor jest obiektem systemu operacyjnego służącym do kontrolowania dostępu do ograniczonego zasobu. Semafory są szczególnie przydatne w środowisku gdzie wiele procesów lub wątków komunikuje się przez wspólną pamięć.

Definicja semafora.

Semafor S jest obiektem systemu operacyjnego z którym związany jest licznik L zasobu przyjmujący wartości nieujemne. Na semaforze zdefiniowane są *atomowe* operacje **`sem_init`**, **`sem_wait`** i **`sem_post`**.

Podano je w poniższej tabeli.

Operacja	Oznaczenie	Opis
Inicjacja semafora S	sem_init(S,N)	Ustawienie licznika semafora S na początkową wartość N >= 0 .
Zajmowanie	sem_wait(S)	• Gdy licznik L semafora S jest dodatni (L > 0) zmniejsz go o 1 (L = L – 1). • Gdy licznik L semafora S jest równy zero (L= 0) zablokuj proces bieżący.
Sygnalizacja	sem_post(S)	• Gdy istnieje jakiś proces oczekujący na semaforze S to odblokuj jeden z czekających procesów. • Gdy brak procesów oczekujących na semaforze S zwiększ jego licznik L o 1 (L=L+1).

Tabela 10-1 Definicja operacji wykonywanych na semaforze

Uwaga!

1. Semafor nie jest liczbą całkowitą na której można wykonywać operacje arytmetyczne .

2. Operacje na semaforach są operacjami atomowymi.

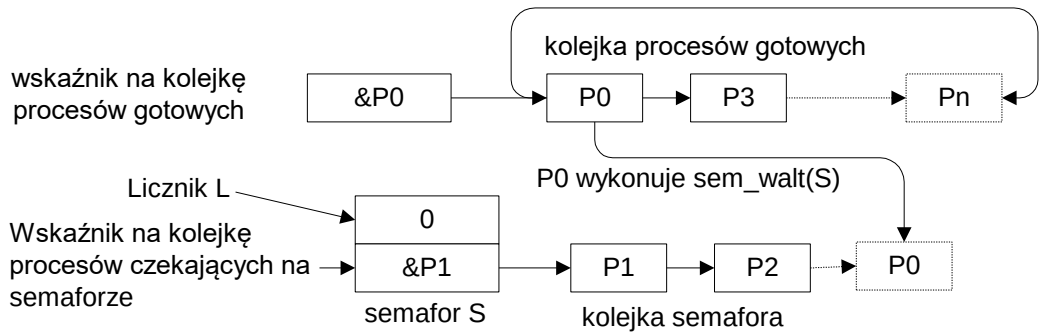
```
sem_wait(S) {
 if(Licznik L semafora S > 0){
  L=L-1;
 } else {
  Zawieś proces bieżący
 }
}


sem_post (S)
{
 if(Istnieje proces czekający na semaforze) {
    Odblokuj jeden z czekających procesów
} else {
    L=L+1;
  }
}
```

Uwaga, operacje:

- **`Zawieś proces bieżący`**
- **`Odblokuj jeden z czekających procesów`**

Nie dają się zaimplementować na poziomie języka C. Muszą być zrealizowane przez system operacyjny.

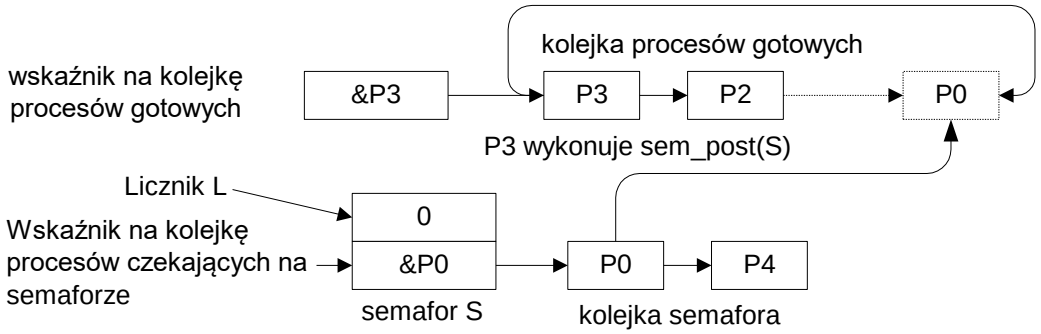
Działanie semafora ilustrują poniższe rysunki. System operacyjny utrzymuje kolejkę procesów gotowych reprezentowanych przez ich deskryptory. Na początku tej kolejki znajduje się proces wykonywany proces bieżący w tym wypadku P0. Semafor składa się z licznika L i kolejki procesów czekających na tym semaforze. Licznik L w tym przykładzie równa się 0. Operacja sem_wait(S) powoduje zawieszenie procesu P0 gdyż brak zasobu semafora (L==0). Polega to na usunięciu procesu P0 z kolejki procesów gotowych i dołączenie go do kolejki semafora S.



Rys. 10-1 Proces P0 wykonuje operację **`sem_wait(S)`**

Po jakimś czasie proces P3 wykonuje operację **`sem_post(S)`** co powoduje odblokowanie jednego z procesów czekających na

semaforze. Jako że pierwszym procesem czekającym w kolejce semafora S jest proces P0 zostanie on usunięty z kolejki semafora i przeniesiony do kolejki procesów gotowych. Za jakiś czas zostanie zaszeregowany do wykonania.



Rys. 10-2 Proces P3 wykonuje operację sem_post(S)

Niezmiennik semafora

Aktualna wartość licznika L semafora S spełnia następujące warunki:

- Jest nieujemna czyli: L >= 0
- Jego wartość wynosi: L= N - Liczba_operacji sem_wait(S) + Liczba_operacji sem_post(S). N jest wartością początkową licznika.

Semafor binarny

W semaforze binarnym wartość licznika przyjmuje tylko dwie wartości: 0 i 1.

Rodzaje semaforów

Wyróżniamy następujące rodzaje semaforów:

1. Semafor ze zbiorem procesów oczekujących (*ang. Blocked- set Semaphore*) – Nie jest określone który z oczekujących procesów ma być wznowiony.

2. Semafor z kolejką procesów oczekujących (*ang. Blocked-queue Semaphore*) – Procesy oczekujące na semaforze umieszczone są w kolejce FIFO.

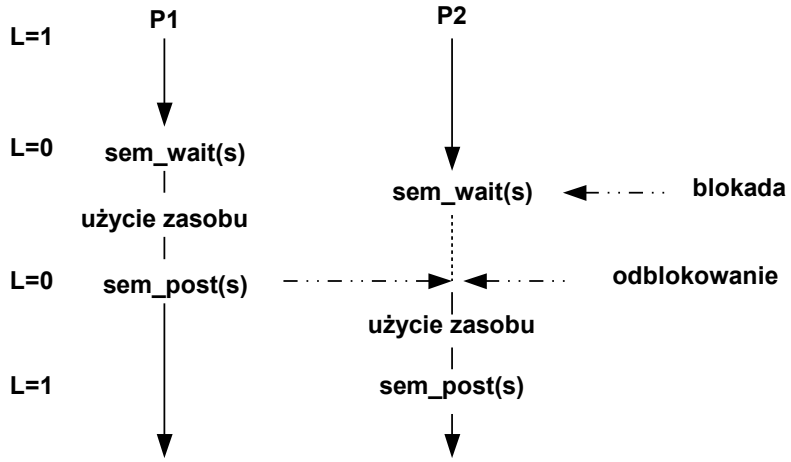
Uwaga!
Pierwszy typ semafora nie zapewnia spełnienia warunku zagłodzenia.

10.4 Zastosowania semaforów

Implementacja wzajemnego wykluczania

```
sem_t S;                          // Deklaracja semafora
sem_init(S,1);                    // Inicjacja semafora

Proces1 {                         Proces2 {
 do {                              do {
   Sekcja_lokalna;                   Sekcja_lokalna;
   sem_wait(S);                      sem_wait(S);
   Sekcja_krytyczna;                 Sekcja_krytyczna;
   sem_post(S);                      sem_post(S);
 } while(1);                       } while(1);
}                                 }
```

Przykład 10-1 Zastosowania semafora do ochrony sekcji krytycznej



Rys. 10-3 Przebieg operacji blokowania

Rozwiązanie problemu producenta – konsumenta

```
#define BufSize 8                    // Bufor ma 8 elementów
RecType Buffer[BufSize];             // Bufor na elementy
sem_t   Mutex;                       // Ochrona bufora
sem_t   Puste;                       // Wolne bufory
sem_t   Pelne;                       // Zajete bufory
int     count;                       // Wskaźnik bufora

// Kod wątku producenta ---------- // Kod wątku konsumenta -------
producent(void) {                    konsument(void) {
 RecType x;                           RecType x;
 do {                                 do {
   ...                                  ...
  produkcja rekordu x;                  ...
  // Czekaj na wolny bufor             // Czekaj na element
  sem_wait(Puste);                     sem_wait(Pelne);
  sem_wait(Mutex);                     sem_wait(Mutex);
    // Umieść element x w buforze      // Pobierz element x z bufora
    Buffer[count] = x;                 count--;
    count++;                           x = Buffer[count];
  sem_post(Mutex);                     sem_post(Mutex);
  // Pojawił się nowy element          // Zwolnij miejsce w buforze
  sem_post(Pelne);                     sem_post(Puste);
                                       konsumpcja rekordu x;
 } while(1);                             ...
}                                     } while(1);
                                     }
main(void) {
 count = 0;
 sem_init(Puste,BufSize);            // Inicjacja semafora Puste
 sem_init(Pelne,0);                  // Inicjacja semafora Pelne
 sem_init(Mutex,1);                  // Inicjacja semafora Mutex
 pthread_create(…,producent,..);     // Start K wątków producenta
 ..
 pthread_create(…,konsument,..);     // Start L wątków konsumenta
  ..
}
```

Program 10-1 Rozwiązanie problemu producenta – konsumenta za pomocą semaforów.

10.5 Rozwiązanie problem czytelników i pisarzy za pomocą semaforów

Rozwiązanie z możliwością zagłodzenia pisarzy

- Czytelnik może wejść do czytelni gdy jest ona pusta lub gdy są tam inni czytelnicy
- Pisarz może wejść do czytelni gdy jest ona pusta

Może się tak zdarzyć że zawsze jakiś czytelnik jest w czytelni co doprowadzi do zagłodzenia pisarzy.

```
sem_t mutex ;  // Kontrola dostępu do reader_count
sem_t db;      // Kontrola dostępu do czytelni
int reader_count;   // Liczba czytelników w czytelni

Reader(){
  while (TRUE) {
     sem_wait(mutex);     // Blokada dostępu do  reader_count
     reader_count = reader_count + 1;

     // Pierwszy czytelnik blokuje dostęp do czytelni  pisarzom
     if (reader_count == 1)
          sem_wait(db);
     // Zwolnienie dostępu do reader_count
     sem_post(mutex);
     read_db();           // Czytelnik czyta
     sem_wait(mutex);     // Blokada dostępu  do reader_count
     reader_count = reader_count - 1;

     // Gdy nie ma innych czytelników to wpuszczamy pisarzy
     if (reader_count == 0)
          sem_post(db);
     sem_post(mutex); // Zwolnienie dostępu do reader_count
}

Writer() {
  while (TRUE) {
     create_data();    // Pisarz zastanawia się
     sem_wait(db);     // Pisarz czeka na dostęp do czytelni
     write_db();       // Pisarz w czytelni - pisze
     sem_post(db);     // Pisarz zwalnia dostęp do czytelni
  }
}
```

```
main()  {
  sem_init(mutex,1);
  sem_init(db,1);
  ....
}
```

Program 10-2 Problem czytelników i pisarzy – rozwiązanie za pomocą semaforów

Rozwiązanie z możliwością zagłodzenia czytelników

- Czytelnik musi czekać gdy są w czytelni lub czekają jacyś pisarze

Rozwiązanie poprawne

- 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 też pisarzy.

```
#define PLACES 8          // Liczba wolnych miejsc w czytelni
sem_t wolne ;         // Liczba wolnych miejsc w czytelni
sem_t wr;             // Kontrola dostępu do czytelni

Reader()
{
  while (TRUE) {
     sem_wait(wolne);     // Czekanie na miejsca w czytelni
     read_db();           // Czytelnik w czytelni - czyta
     sem_post(wolne);     // Zwolnienie miejsca w czytelni
}

Writer() {
  while (TRUE) {
     create_data();       // Pisarz zastanawia się
     sem_wait(wr);        // Zablokowanie dostępu dla pisarzy
     // Wypieranie czytelników z czytelni
     for(j=1;j<=places;j++)
          sem_wait(wolne);
     write_db();          // Pisarz w czytelni – pisze
     // Wpuszczenie czytelników
     for(j=1;j<= PLACES;j++)
          sem_post(wolne);
     sem_post(wr);        // Odblokowanie pisarzy
  }
}

main()  {
  sem_init(wolne,PLACES);
  sem_init(wr,1);
  ....
}
```

Program 10-3 Rozwiązanie problemu czytelników i pisarzy z ograniczoną liczbą miejsc w czytelni

11. Semafory w standardzie POSIX

Wyróżnione są tu dwa typy semaforów:

1. Semafory nienazwane
2. Semafory nazwane

Semafory nienazwane nadają się do synchronizacji wątków w obrębie jednego procesu. Dostęp do semafora nienazwanego następuje poprzez jego adres. Może on być także użyty do synchronizacji procesów o ile jest umieszczony w pamięci dzielonej.

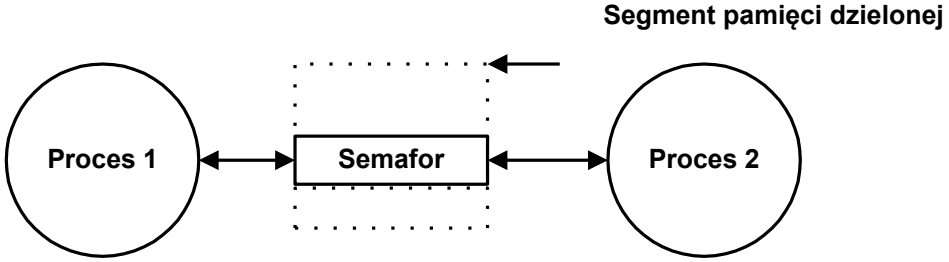
Dostęp do semaforów nazwanych następuje poprzez nazwę. Ten typ semaforów bardziej nadaje się synchronizacji procesów niż wątków. Semafory nienazwane działają szybciej niż nazwane.

sem_open	Utworzenie semafora nazwanego
sem_init	Inicjacja semafora
sem_wait	Czekanie na semaforze
sem_trywait	Pobranie zasobu
sem_timedwait	Czekanie z przeterminowaniem
sem_post	Sygnalizacja
sem_getvalue	Pobranie wartości licznika semafora – funkcja
sem_close	Zamknięcie semafora
sem_unlink	Kasowanie semafora

Tab. 11-1 Operacje semaforowe w standardzie POSIX

Semafory nienazwane

Dostęp do semafora nienazwanego następuje po adresie semafora. Stąd nazwa semafor nienazwany.



Deklaracja semafora
Przed użyciem semafora musi on być zadeklarowany jako obiekt typu sem_t a pamięć używana przez ten semafor musi zostać mu jawnie przydzielona.

O ile semafor ma być użyty w różnych procesach powinien być umieszczony w wcześniej zaalokowanej pamięci dzielonej.

1. Utworzyć segment pamięci za pomocą funkcji **shm_open**.
2. Określić wymiar segmentu używając funkcji f**trunc**.
3. Odwzorować obszar pamięci wspólnej w przestrzeni danych procesu - **mmap**.

Inicjacja semafora - funkcja sem_init
Przed użyciem semafor powinien być zainicjowany.

```
int sem_init(sem_t *sem, int pshared, unsigned
value)
```

sem Identyfikator semafora (wskaźnik na strukturę w pamięci)
pshared Gdy wartość nie jest zerem semafor może być umieszczony w pamięci dzielonej i dostępny w wielu procesach
value Początkowa wartość semafora

Funkcja zwraca 0 gdy sukces, -1 gdy błąd.

Czekanie na semaforze – funkcja sem_wait
Funkcja ta odpowiada omawianej wcześniej operacji sem_wait(S). Prototyp funkcji jest następujący:

```
int sem_wait(sem_t *sem)
```

sem Identyfikator semafora

Gdy licznik semafora jest nieujemny funkcja zmniejsza go o 1. W przeciwnym przypadku proces bieżący jest zawieszany. Zawieszony proces może być odblokowany przez procedurę sem_post wykonaną w innym procesie lub sygnał. Funkcja zwraca 0 gdy sukces, -1 gdy błąd.

Ograniczone czasowo czekanie na semaforze – funkcja sem_timedwait

Funkcja ta jest wersją operacji sem_wait(S). Prototyp funkcji jest następujący:

```
int sem_timedwait(sem_t *sem, struc timespec
timeout)
```

sem Identyfikator semafora
timeout Specyfikacja przeterminowania – czas absolutny

Gdy licznik semafora jest nieujemny funkcja zmniejsza go o 1. W przeciwnym przypadku proces bieżący jest zawieszany.

Zawieszony proces może być odblokowany przez procedurę sem_post wykonaną w innym procesie, sygnał, lub system operacyjny, gdy po upływie czasu timeout zawieszony proces nie zostanie inaczej odblokowany.

Funkcja zwraca 0 gdy sukces, -1 gdy błąd.
Gdy wystąpił timeout kod błędu errno wynosi ETIMEDOUT

```
struct  timespec {
   time_t tv_sec;  // Sekundy
   long   tv_nsec; // nanosekundy
}
```

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

main(){
  struct timespec tm;
  sem_t sem;
  int i=0;
  sem_init( &sem, 0, 0);
  do {
        clock_gettime(CLOCK_REALTIME, &tm);
        tm.tv_sec += 1;   i++;
        printf("i=%d\n",i);
        if (i==10) {
            sem_post(&sem);
        }

  } while(sem_timedwait(&sem,&tm) == -1 );
  printf("Semafor zwolniony po %d probach\n", i);
  return;
}
```

Program 11-1 Przykład wykorzystania semafora z przeterminowaniem

Nieblokujące pobranie zasobu

Funkcja ta podobna jest do operacji sem_wait(S). Prototyp funkcji jest następujący:

```
int sem_trywait(sem_t *sem)
```

sem Identyfikator semafora

Gdy licznik semafora jest nieujemny funkcja zmniejsza go o 1. W przeciwnym przypadku funkcja zwraca –1 i proces bieżący nie vjest zawieszany.

Funkcja zwraca 0 gdy sukces, -1 gdy błąd.

Sygnalizacja na semaforze – funkcja sem_post
Funkcja ta odpowiada omawianej wcześniej operacji sem_post(S). Prototyp funkcji jest następujący:

```
int sem_post(sem_t *sem)
```

sem Identyfikator semafora

Jeśli jakikolwiek proces jest zablokowany na tym semaforze przez wykonanie funkcji **sem_wait** zostanie on odblokowany. Gdy brak procesów zablokowanych licznik semafora zwiększany jest o 1. Funkcja zwraca 0 gdy sukces, -1 gdy błąd.

Pobranie wartości licznika semafora – funkcja sem_getvalue

sem_getvalue(sem_t *sem, int *sval)

sem Identyfikator semafora
sval Wartość licznika semafora

Funkcja zwraca 0 gdy sukces, -1 gdy błąd.

Uwaga!
Nie stosować konstrukcji pobierz, testuj, działaj. Musi to być operacja atomowa.

Kasowanie semafora – funkcja sem_destroy
Semafor kasuje się przy pomocy funkcji :

```
int sem_destroy(sem_t *sem)
```

sem Identyfikator semafora

Gdy jakieś procesy są zablokowane na tym semaforze, zostaną one odblokowane i operacja **sem_destroy** zakończy się błędem.

Semafory nazwane

Semafory nazwane identyfikowane są w procesach poprzez ich nazwę.
Na semaforze nazwanym operuje się tak samo jak na semaforze nienazwanym z wyjątkiem funkcji otwarcia i zamknięcia semafora.

Otwarcie semafora – funkcja sem_open

Aby użyć semafora nazwanego należy uzyskać do niego dostęp poprzez funkcję:

```
sem_t *sem_open(const char *sem_name, int oflags,
[int mode, int value])
```

sem_name	Nazwa semafora, powinna się zaczynać od znaku „/”.
oflags	Flagi trybu tworzenia i otwarcia: O_RDONLY, O_RDWR, O_WRONLY. Gdy semafor jest tworzony należy użyć flagi O_CREAT
mode	Prawa dostępu do semafora – takie jak do plików. Parametr jest opcjonalny.
value	Początkowa wartość semafora. Parametr jest opcjonalny

Funkcja zwraca identyfikator semafora. Semafor widoczny jest w katalogu /dev/sem. Funkcja tworzy semafor gdy nie był on wcześniej utworzony i otwiera go.

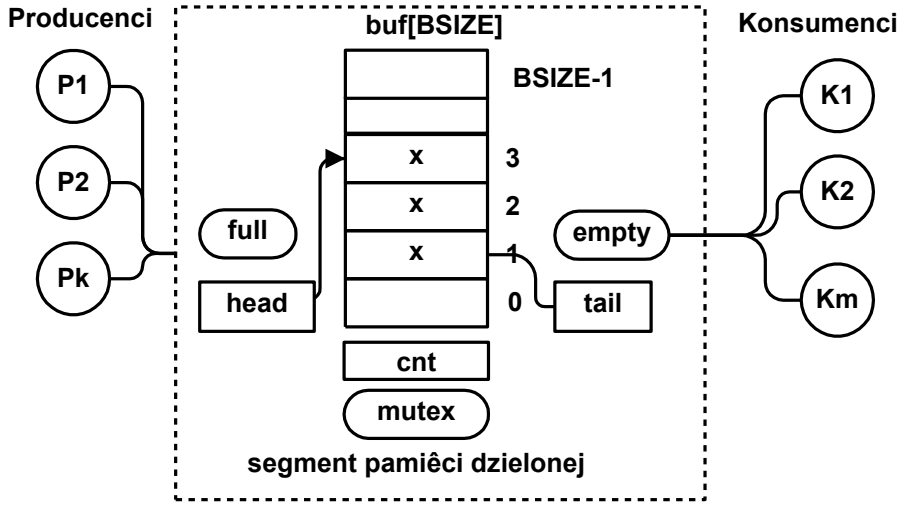
Zamykanie semafora nazwanego – funkcja sem_close

Semafor zamyka się poprzez wykonanie funkcji:

```
int sem_close( sem_t * sem )
```

Funkcja zwalnia zasoby semafora i odblokowuje procesy które są na nim zablokowane. Funkcja zwraca 0 gdy sukces, -1 gdy błąd.

11.1 Rozwiązanie problemu producenta i konsumenta – semafory nienazwane



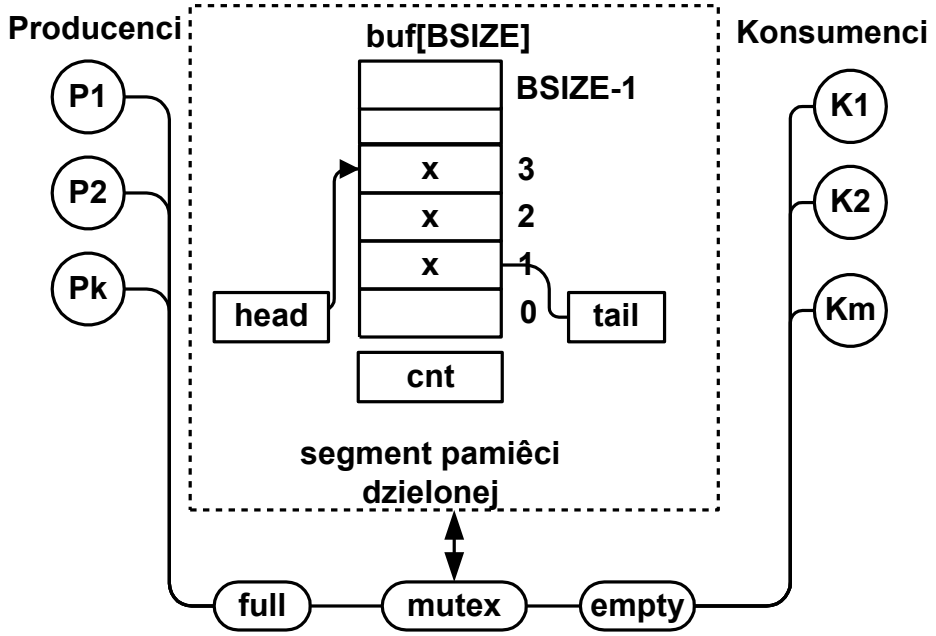

```
#include <sys/mman.h>
#include <fcntl.h>
#include <semaphore.h>
#define BSIZE     4   // Rozmiar bufora
#define LSIZE    80   //  Dlugosc linii
typedef struct {
    char buf[BSIZE][LSIZE];
    int head;
    int tail;
    int cnt;
    sem_t mutex;
    sem_t empty;
    sem_t full;
} bufor_t;

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));
      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);
    }
    shm_unlink("bufor");
    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);
 shm_unlink("bufor");
 sem_close(&(wbuf->mutex));
 sem_close(&(wbuf->empty));
 sem_close(&(wbuf->full));
 return 0;
}
```

Przykład 11-1 Rozwiązanie problemu producenta i konsumenta za pomocą semaforów nienazwanych

11.2 Rozwiązanie problemu producenta i konsumenta – semafory nazwane




```
#include <sys/mman.h>
#include <fcntl.h>
#include <semaphore.h>
#define BSIZE     4  // Rozmiar bufora
#define LSIZE    80  // Dlugosc linii

typedef struct { // Obszar wspólny
    char buf[BSIZE][LSIZE];
    int head;
    int tail;
    int cnt;
} bufor_t;

main(int argc,char *argv[]) {
 int i,stat,k, pid, size, fd,res;
 bufor_t  *wbuf ;
 char c;
 sem_t *mutex;
 sem_t *empty;
 sem_t *full;

 // Utworzenie segmentu --------------------------
 shm_unlink("bufor");
 if((fd=shm_open("bufor", O_RDWR|O_CREAT , 0774)) == -1){
   perror("open"); exit(-1);
 }
 printf("fd: %d\n",fd);
```

```
size = ftruncate(fd, BSIZE);
if(size < 0) {perror("trunc"); exit(-1); }
// Odwzorowanie segmentu fd w obszar pamieci procesow
wbuf = ( bufor_t *)mmap(0,BSIZE, 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;

// Utworzenie semaforw ------------------------
mutex = sem_open("mutex",O_CREAT,S_IRWXU,1);
empty = sem_open("empty",O_CREAT,S_IRWXU,BSIZE);
full  = sem_open("full",O_CREAT,S_IRWXU,0);

// Utworzenie procesow ------------------------
if(fork() == 0) { // Producent
   for(i=0;i<10;i++) {
     // printf("Producent: %i\n",i);
     sem_wait(empty);
     sem_wait(mutex);
     printf("Producent - cnt:%d head: %d tail: %d\n",
             wbuf-> cnt,wbuf->head,wbuf->tail);
     sprintf(wbuf->buf[wbuf->head],"Komunikat %d",i);
     wbuf-> cnt ++;
     wbuf->head = (wbuf->head +1) % BSIZE;
     sem_post(mutex);
     sem_post(full);
     sleep(1);
   }
   shm_unlink("bufor");
   exit(i);
}
```

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

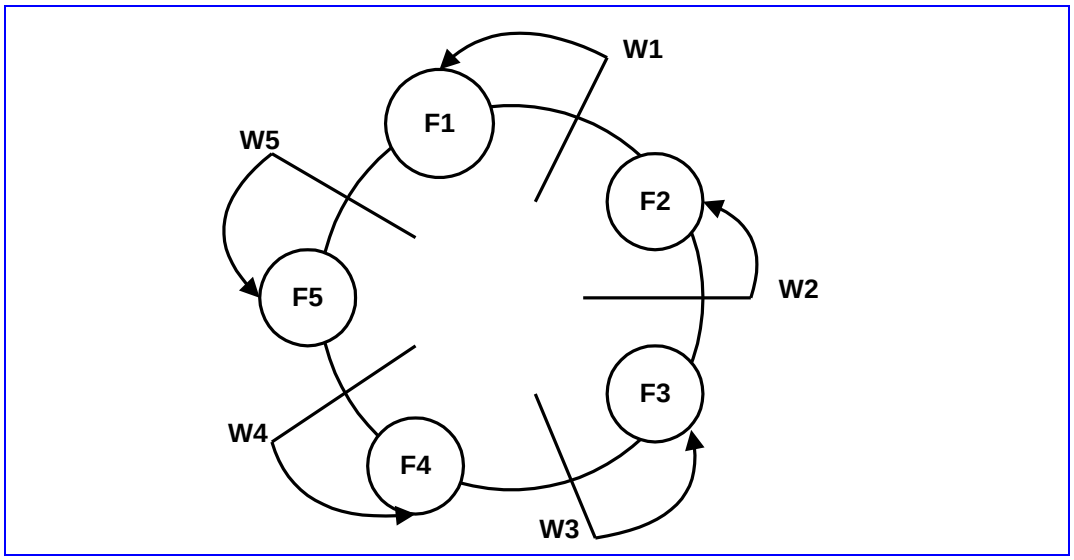
Przykład 11-2 Rozwiązanie problemu producenta i konsumenta za pomocą semaforów nazwanych

11.3 Problem pięciu ucztujących filozofów

Przy okrągłym stole siedzi pięciu filozofów. Zajmują się oni naprzemiennie tylko dwoma czynnościami - myśleniem i jedzeniem. Do jedzenia filozof potrzebuje dwu widelców. Gdy filozof otrzyma dwa widelce je, a następnie odkłada obydwa widelce. Problem polega na takim zorganizowaniu pracy filozofów aby spełnione były warunki:

- Filozof je wtedy gdy zdobędzie dwa widelce.
- Dwu filozofów nie może trzymać tego samego widelca.
- Każdy z filozofów musi się w końcu najeść (nie zostanie zagłodzony).

Dodatkowym wymaganiem jest efektywność rozwiązania. Znaczy to że nie należy blokować aktywności filozofa gdy nie jest to konieczne dla spełnienia poprzednich warunków.



Rys. 11-1 **Ilustracja problemu pięciu ucztujących filozofów**

A. Rozwiązanie z możliwością blokady:

1. Filozof czeka aż będzie wolny lewy widelec i podnosi go.
2. Filozof czeka aż będzie wolny prawy widelec i podnosi go.
3. Filozof je.
4. Filozof odkłada obydwa widelce.
5. Filozof myśli.

Jeżeli w pewnej chwili każdy z filozofów podniesie lewy widelec i będzie czekał na prawy nigdy go nie otrzyma gdyż algorytm przewiduje zwolnienie widelców po zakończeniu jedzenia. Nastąpi zakleszczenie.

B. Rozwiązanie z możliwością zagłodzenia:

1. Filozof czeka aż będą wolne oba widelce i podnosi je (musi to być operacja niepodzielna).
2. Filozof je.
3. Filozof odkłada obydwa widelce.
4. Filozof myśli.

Może się tak zdarzyć że filozof będzie miał bardzo żarłocznych sąsiadów z których w każdej chwili jeden z nich je. W takim przypadku filozof zostanie zagłodzony.

C. Rozwiązanie poprawne

1. Potrzebny jest arbiter zewnętrzny (nazywany lokajem) który dba o to aby jednej chwili najwyżej czterech filozofów konkurowało o widelce.
2. Dalej postępujemy jak w przypadku A.

Poniższy program ilustruje problem pięciu filozofów i jego rozwiązanie. Program można uruchomić z parametrem 0 lub 1 który nadaje wartość zmiennej wybor. Gdy wybor==0 program będzie podatny na zakleszczenie. Gdy wybor==1 program będzie ilustrował poprawne rozwiązanie gdyż o widelce konkurować będzie tylko 4 filozofów. Występująca w programie funkcja **`sched_yield()`** powoduje wywołanie funkcji szeregującej i w efekcie przełączenie wątku.

```
// Problem pieciu filozofow z semaforami
// kompilacja:
// gcc filozofy.c -o filozofy -lrt -lpthread
// Uruchomienie
// ./filozofy 0 - wersja z mozliwoscia zakleszczenia
// ./filozofy 1 - wersja z lokajem
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define N 10 /* Liczba prob      */
#define P 5  /* Liczba  filozofow*/

typedef   enum { False=0, True=1 } bool ;

sem_t lokaj;
sem_t widelec[P];
bool Wybor ;

void *filozof(void *ptr) {
    int res,i, k = *((int *) ptr);
    printf("start filozofa %d\n",k);
    for(i = 1; i <= N; i++) {
        printf("Filozof %d mysli krok %d\n", k, i);
        if(Wybor) sem_wait(&lokaj);
        sem_wait(&widelec[k]) ;
        sem_wait(&widelec[(k+1) % P]) ;
        printf("Filozof %d je krok %d\n", k, i);
        res = sched_yield();
        sem_post(&widelec[k]) ;
        sem_post(&widelec[(k+1) % P]) ;
        if(Wybor) {
            sem_post(&lokaj) ;
        }
    }
    pthread_exit(0);
}

int main(int argc, char * argv[]) {
    int i, targ[P];
    pthread_t thread[P];
    if(argc < 2) {
    printf("podaj argument 0-brak lokaja, 1-jest\n");
           return 0;
    }
    sem_init(&lokaj, 0, P-1);
    Wybor = atoi(argv[1]); /* Lub 0 gdy nie ma lokaja */
    printf("Wybor=%s\n",(Wybor?"true":"false"));
    for(i=0;i<P;i++) {
        sem_init(&widelec[i], 0, 1);
    }
    for(i=0;i<P;i++) {
```

```
        targ[i] = i;
        pthread_create(&thread[i], NULL, &filozof,
        (void *) &targ[i]);
    }
    for(i=0;i<P;i++) {
        pthread_join(thread[i], NULL);
    }
    for(i=0;i<P;i++) {
        sem_destroy(&widelec[i]);
    }
    sem_destroy(&lokaj);
    return 0;
}
```

Przykład 11-3 Problem 5 filozofów, program filozofy.c