Programowanie systemowe i współbieżne

dr hab. inż. Anna Kobusińska, prof. PP

Anna.Kobusinska@cs.put.poznan.pl

Organizacja zajęć

1. Zasady zaliczenia:
   - test: 30%
   - projekt: 60%
   - inne (zadania domowe, aktywność): 10%

Organizacja zajęć

1. Zasady zaliczenia:
   - test: 30%
   - projekt: 60%
   - inne (zadania domowe, aktywność): 10%
2. Zaliczenie
   - 51-60% dst
   - 61-70% dst+
   - 71-80% db
   - 81-90% db+
   - 91-100% bdb

Organizacja zajęć

1. Zasady zaliczenia:
   - test: 30%
   - projekt: 60%
   - inne (zadania domowe, aktywność): 10%
2. Zaliczenie
   - 51-60% dst
   - 61-70% dst+
   - 71-80% db
   - 81-90% db+
   - 91-100% bdb
3. Akceptowane nieobecności: 2

Organizacja zajęć — literatura

- J. S. Gray: Komunikacja między procesami w Unixie. Oficyna Wydawnicza ReadMe
- M. J. Rochkind: Programowanie w systemie Unix dla zaawansowanych. WNT
- Z. Guźlewski, T. Weiss: Programowanie współbieżne i rozproszone w przykładach i zadaniach. WNT
- R. Love: Linux. Programowanie systemowe, Helion

Organizacja zajęć

1. Materiały:
   - http://www.cs.put.poznan.pl/akobusinska/psw.html
   - http://www.cs.put.poznan.pl/csobaniec/edu/psw/
   - http://www.cs.put.poznan.pl/dwawrzyniak/PW/
   - http://www.cs.put.poznan.pl/anstroinski

Organizacja zajęć

1. Materiały:
   - http://www.cs.put.poznan.pl/akobusinska/psw.html
   - http://www.cs.put.poznan.pl/csobaniec/edu/psw/
   - http://www.cs.put.poznan.pl/dwawrzyniak/PW/
   - http://www.cs.put.poznan.pl/anstroinski
2. Serwer: unixlab.cs.put.poznan.pl

Organizacja zajęć

1. Materiały:
   - http://www.cs.put.poznan.pl/akobusinska/psw.html
   - http://www.cs.put.poznan.pl/csobaniec/edu/psw/
   - http://www.cs.put.poznan.pl/dwawrzyniak/PW/
   - http://www.cs.put.poznan.pl/anstroinski
2. Serwer: unixlab.cs.put.poznan.pl
3. Dostęp: PuTTY

Organizacja zajęć

1. Materiały:
   - http://www.cs.put.poznan.pl/akobusinska/psw.html
   - http://www.cs.put.poznan.pl/csobaniec/edu/psw/
   - http://www.cs.put.poznan.pl/dwawrzyniak/PW/
   - http://www.cs.put.poznan.pl/anstroinski
2. Serwer: unixlab.cs.put.poznan.pl
3. Dostęp: PuTTY
4. Kopiowanie: WinSCP

Środowisko pracy

1. Kompilacja:
   - `gcc program.c -o program`
   - `gcc -Wall program.c -o program`
   - `gcc -g program.c -o program`

Środowisko pracy

1. Kompilacja:
   - `gcc program.c -o program`
   - `gcc -Wall program.c -o program`
   - `gcc -g program.c -o program`
2. Uruchomienie: `./program`

Środowisko pracy

1. Kompilacja:
   - `gcc program.c -o program`
   - `gcc -Wall program.c -o program`
   - `gcc -g program.c -o program`
2. Uruchomienie: `./program`
3. Pomoc systemowa:
   - sekcja 2 — funkcje systemowe (`man 2 write`)
   - sekcja 3 — funkcje biblioteczne (`man 3 printf`)

Przekazywanie argumentów

```
int main(int argc, char* argv[])
```

Przekazywanie argumentów

```
int main(int argc, char* argv[])
```

- Przykład

```
#include <stdio.h>
int main(int argc, char* argv[])
{
int i;
printf ("%d\n", argc);
for(i=0; i<argc; i++)
printf("argument %d:  %s\n", i, argv[i]);
}
```

Pliki - biblioteki

```
#include <fcntl.h>

#include <unistd.h>

#include <sys/stat.h>

#include <sys/types.h>
```

Otwieranie / tworzenie pliku

```
int open(const char *pathname, int flags)

int open(const char *pathname, int flags, mode_t
mode)
```

Parametry:

- `pathname` — nazwa pliku (w szczególności nazwa ścieżkowa),
- `flags` — tryb otwarcia:
  - O_WRONLY
  - O_RDONLY
  - O_RDWR
  - O_APPEND
  - O_CREAT
  - O_TRUNC
- `mode` — prawa dostępu

Odczyt pliku

```
ssize_t read(int fd, void *buf, size_t count)
```

Parametry:

- `fd` — deskryptor pliku, z którego następuje odczyt danych,
- `buf` — adres początku obszaru pamięci, w którym zostaną umieszczone odczytane dane,
- `count` — liczba bajtów do odczytu z pliku (nie może być większa, niż rozmiar obszaru pamięci przeznaczony na odczytywane dane).

Zapis pliku

```
ssize_t write(int fd, const void *buf, size_t count)
```

Parametry:

- `fd` — deskryptor pliku, do którego następuje zapis danych,
- `buf` — adres początku obszaru pamięci, zawierającego blok danych do zapisania
- `count` — liczba bajtów do zapisania w pliku

Przykładowe błędne / poprawne wywołania funkcji?

- `char array[10]; read (0,array,10);`

Przykładowe błędne / poprawne wywołania funkcji?

- `char array[10]; read (0,array,10);`
- `int num[3]; read(0,num,sizeof(num)*sizeof(int));`

Przykładowe błędne / poprawne wywołania funkcji?

- `char array[10]; read (0,array,10);`
- `int num[3]; read(0,num,sizeof(num)*sizeof(int));`
- `int num; write(1,num,sizeof(num));`

Przykładowe błędne / poprawne wywołania funkcji?

- `char array[10]; read (0,array,10);`
- `int num[3]; read(0,num,sizeof(num)*sizeof(int));`
- `int num; write(1,num,sizeof(num));`
- `int *num; write(1,num,sizeof(num));`

Odczyt całego pliku

```
while((n=read(fd, buf, 20)) > 0)
{ write(1, buf, n); }
```

Wskaźnik bieżącej pozycji

```
off_t lseek(int fd, off_t offset, int whence)
```

Parametry:

- `fd` — deskryptor pliku
- `offset` — wielkość przesunięcia
- `whence` — odniesienie
  - SEEK_SET
  - SEEK_END
  - SEEK_CUR

Zamykanie/usuwanie pliku

```
int close(int fd)

int unlink(const char *pathname)
```

Obsługa błędów

- Szczegółowy kod błędu można odczytać badając wartość globalnej zmiennej `errno` typu `int`.
- Obsługa błędów — funkcja `perror` (bada wartość zmiennej errno i wyświetla tekstowy opis błędu, który wystąpił
- Przykład:

```
# include <errno.h>
fd = open("przyklad.txt", O_RDONLY);
if (fd == -1)
{
printf("Kod: %d\n", errno);
perror("Otwarcie pliku");
exit(1);
}
```

Funkcja getopt

```
#include <stdio.h>
#include <unistd.h>
int main (int argc, char *argv[]){
        int opt;
        opterr = 0;       //disable error massages
        while ((opt = getopt(argc, argv, ":if:lr")) != -1){
                switch (opt){
                case 'i':
                case 'l':
                case 'r':
                  printf("Option: %c\n", opt);
                break;
                case 'f':
                  printf("Filename: %s\n", optarg);
                break;
                case ':':
                  printf("Option %c needs a value\n", optopt);
                break;
                case '?':
                  printf("Unknown option: %c\n", optopt);
                break;
                }
        }
        for (; optind < argc; optind++)
                printf("Argument: %s\n", argv[optind]);
        exit(0);
}
```

Tworzenie procesów

```
pid_t fork(void)
```

Tworzenie procesów

```
pid_t fork(void)
```

- Przykład

```
#include <stdio.h>
#include <unistd.h>
    int main(){
        printf ("Begin\n");
        fork();
        printf ("End\n");
        return 0;
    }
```

Proces macierzysty i potomny

```
#include <stdio.h>
#include <unistd.h>
main(){
    if (fork()==0)
        printf ("Child\n");
    else
        printf ("Parent\n");
    return 0;
}
```

Identyfikatory procesów

```
pid_t getpid(void)

pid_t getppid(void)
```

Identyfikatory procesów

```
pid_t getpid(void)

pid_t getppid(void)
```

- Napisz program tworzący proces macierzysty i potomny. Dla każdego z proesów podaj wartość PID i PPID.

Kolejność wykonania procesów

```
#include <stdio.h>
#include <unistd.h>
int main(){
    fork();
    printf (Hi\n);
    fork();
    printf (Ha\n);
    fork();
    printf (Ho\n);
    return 0;
}
```

Liczba tworzonych procesów

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main (){
    fork ();
    fork ();
        if ( fork () == 0)
         fork ();
    fork ();
    return 0;
}
```

Liczba tworzonych procesów

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main (){
    fork ();
    fork ();
        if ( fork () != 0)
         exit ();
    fork ();
    return 0;
}
```

Liczba tworzonych procesów

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main (){
    fork ();
    if ( fork () == 0)
        fork ();
    else
        if (fork ()!=0)
         if (fork()==0)
           fork();
        fork();
    return 0;
}
```

Zakończenie procesu

```
void exit(int status)
```

Zakończenie procesu

```
void exit(int status)
```

- Parametr
  - `status` — kod wyjścia przekazywany procesowi macierzystemu
- Przykład: exit(7);

Zakończenie procesu

```
void exit(int status)
```

- Parametr
  - `status` — kod wyjścia przekazywany procesowi macierzystemu
- Przykład: exit(7);

```
int kill(pid_t pid, int signum)
```

Zakończenie procesu

```
void exit(int status)
```

- Parametr
  - `status` — kod wyjścia przekazywany procesowi macierzystemu
- Przykład: exit(7);

```
int kill(pid_t pid, int signum)
```

- Parametry
  - `pid` — identyfikator procesu, do którego adresowany jest sygnał
  - `signum` — numer przesyłanego sygnału
- Przykład: kill(pid, 9);

Status zakończenia

```
#include <sys/wait.h>

pid_t wait(int *status)
```

Status zakończenia

```
#include <sys/wait.h>

pid_t wait(int *status)
```

- Parametr
  - `status` — adres słowa w pamięci, w którym umieszczony zostanie status zakończenia
- Funkcja zwraca identyfikator zakończonego procesu lub -1 w przypadku błędu

Status zakończenia

```
#include <sys/wait.h>

pid_t wait(int *status)
```

- Parametr
  - `status` — adres słowa w pamięci, w którym umieszczony zostanie status zakończenia
- Funkcja zwraca identyfikator zakończonego procesu lub -1 w przypadku błędu

- Jesli wywołanie funkcji `wait` nastąpi przed zakonczeniem potomka, przodek zostaje zawieszony w oczekiwaniu na to zakonczenie.
- Po zakończeniu potomka następuje wyjście procesu macierzystego z funkcji `wait`.
- Pod adresem wskazanym w parametrze znajduje się status zakończenia.

Status zakończenia

- Status zakończenia:
  - numer sygnału (mniej znaczące 7 bitów)
  - kod wyjścia (bardziej znaczący bajt będący wartością fn. `exit` wywołanej przez potomka)

```
int status;
...
if (fork()!=0)
wait(NULL)

...
if (fork()!=0)
wait(&status)

printf( %x, status)
printf( %04x, status)
```

Proces zombi / sierota

- Sierota — proces potomny, którego przodek się już zakończył
- Zombi — proces potomny, który zakończył swoje działanie i czeka na przekazanie statusu zakończenia przodkowi
  - System nie utrzymuje procesów zombi, jeśli przodek ignoruje sygnał SIGCLD
- Zadanie: Stwórz proces sierotę i proces zombi

Wykonanie programu

```
int execl(const char *path,const char *arg,...)

int execlp(const char *file,const char *arg,...)

int execle(const char *path,const char *arg ,..., char *const
envp[])

int execv(const char *path, char *const argv[])

int execvp(const char *file, char *const argv[])

int execve(const char *file, char *const argv[], char *const
envp[])
```

Wykonanie programu

```
int execl(const char *path,const char *arg,...)

int execlp(const char *file,const char *arg,...)

int execle(const char *path,const char *arg ,..., char *const
envp[])

int execv(const char *path, char *const argv[])

int execvp(const char *file, char *const argv[])

int execve(const char *file, char *const argv[], char *const
envp[])
```

- Parametry:
  - `path` --- nazwa ścieżkowa pliku z programem,
  - `file` --- nazwa pliku z programem,
  - `arg` --- argument linii poleceń
  - `argv` --- wektor (tablica) argumentów linii poleceń
  - `envp` --- wektor zmiennych środowiskowych.

Wykonanie programu

- `execl ("/bin/ls", "ls", "-a", NULL);`

Wykonanie programu

- `execl ("/bin/ls", "ls", "-a", NULL);`
- `execlp("ls", "ls", "-a", NULL);`

Wykonanie programu

- `execl ("/bin/ls", "ls", "-a", NULL);`
- `execlp("ls", "ls", "-a", NULL);`
- ```
  char *const av[]={"ls", "-a", NULL};
  execv ("/bin/ls", av);
  ```
  ```

Wykonanie programu

- `execl ("/bin/ls", "ls", "-a", NULL);`
- `execlp("ls", "ls", "-a", NULL);`
- ```
  char *const av[]={"ls", "-a", NULL};
  execv ("/bin/ls", av);
  ```
- ```
  char *const av[]={"ls", "-a", NULL};
  execvp ("ls", av);
  ```

Wykonanie polecenia podanego jako parametry programu

- Zadanie: podać czas wykonania polecenia podanego jako parametr wejściowy programu (polecenie może mieć dowolną liczbę przełączników, np. ./run find /usr -type f -size +100k -mtime -30)

Wykonanie polecenia podanego jako parametry programu

- Zadanie: podać czas wykonania polecenia podanego jako parametr wejściowy programu (polecenie może mieć dowolną liczbę przełączników, np. ./run find /usr -type f -size +100k -mtime -30)
- Należy skorzystać z funkcji: gettimeofday()

Wykonanie polecenia podanego jako parametry programu

- Zadanie: podać czas wykonania polecenia podanego jako parametr wejściowy programu (polecenie może mieć dowolną liczbę przełączników, np. ./run find /usr -type f -size +100k -mtime -30)
- Należy skorzystać z funkcji: gettimeofday()
- execvp(argv[1], argv+1);

Łącza

- ograniczona liczba bloków z danymi — łącza mają rozmiar: 4KB - 8KB w zależności od konkretnego systemu
- dostęp sekwencyjny (nie ma możliwości przemieszczania wskaźnika bieżącej pozycji, nie wywołuje się fn. lseek)
- dane odczytane z łącza są z niego usuwane
- proces jest blokowany w fn. read na pustym łączu, jeśli jest otwarty jakiś deskryptor tego łącza do zapisu
- proces jest blokowany w fn. write, jeśli w łączu nie ma wystarczającej ilości wolnego miejsca do zapisania całego bloku.
- przepływ strumienia — dane są odczytywane w kolejności, w której były zapisane

Rodzaje łączy

- łącze nienazwane (potok) — nie ma nazwy w żadnym katalogu i istnieje tak długo po utworzeniu, jak długo otwarty jest jakiś deskryptor tego łącza.
- łącze nazwane (kolejka FIFO) — ma dowiązanie w systemie plików, co oznacza, że jego nazwa jest widoczna w jakimś katalogu i może ona służyć do identyfikacji łącza

Tworzenie łącza nienazwanego

```
#include <unistd.h>

int pipe(int fd[2])
```

Tworzenie łącza nienazwanego

```
#include <unistd.h>

int pipe(int fd[2])
```

- Parametr
  - `fd` — tablica 2 deskryptorów;
  - `fd[0]` — deskryptor potoku do odczytu
  - `fd[1]` — deskryptor potoku do zapisu
- Funkcja zwraca 0 w przypadku poprawnego zakończenia lub -1

Powielanie deskryptorów (I)

```
int dup (int fd)
```

Powielanie deskryptorów (I)

```
int dup (int fd)
```

- Parametr
  - `fd` — deskryptor który ma zostać powielony
- Funkcja zwraca numer nowo przydzielonego deskryptora lub -1 w przypadku błędu

Powielanie deskryptorów (II)

- `int dup2 (int oldfd, int newfd)`

Powielanie deskryptorów (II)

- `int dup2 (int oldfd, int newfd)`

- Parametr
  - `oldfd` — deskryptor który ma zostać powielony
  - `newfd` — numer nowoprzydzielonego deskryptora
- Powielenie (duplikacja deskryptora) we wskazanym miejscu w tablicy dskryptorów
- Funkcja zwraca numer nowo przydzielonego deskryptora lub -1 w przypadku błędu

Tworzenie łącza nazwanego

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode)
```

Tworzenie łącza nazwanego

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode)
```

- Funkcja zwraca 0 w przypadku poprawnego zakończenia lub -1
- Parametr
  - `pathname` — nazwa pliku (w szczególności nazwa ścieżkowa)
  - `mode` — prawa dostępu do nowo tworzonego pliku.

Otwieranie łącza

```
mkfifo("kolFIFO", 0600);

open("kolFIFO", O_RDONLY);
```

Otwieranie łącza

```
mkfifo("kolFIFO", 0600);

open("kolFIFO", O_RDONLY);
```

- Funkcja `open` musi być wywołana w trybie komplementarnym
- Polecenie systemowe `mkfifo`

Obsługa sygnałów

1. Sygnały są obsługiwane w sposób asynchroniczny
2. Reakcja procesu na otrzymany sygnał:
   - Wykonanie akcji domyślnej (najczęściej zakończenie procesu z ewentualnym zrzutem zawartości segmentów pamięci na dysk)
   - Zignorowanie sygnału
   - Przechwycenie sygnału tj. podjęcie akcji zdefiniowanej przez użytkownika
3. Lista sygnałów: `kill -l`, `man 7 signal`

Wysłanie sygnału

```
#include <signal.h>

int kill(pid_t pid, int signum)
```

Wysłanie sygnału

```
#include <signal.h>

int kill(pid_t pid, int signum)
```

- Funkcja zwraca 0 w przypadku poprawnego zakończenia lub -1
- Parametry:
  - `pid` — identyfikator procesu, do którego adresowany jest sygnał
    - `pid > 0` sygnał zostanie wysłany do procesu o identyfikatorze pid,
    - `pid = 0` sygnał zostanie wysłany do grupy procesów do których należy proces wysyłający,
    - `pid = -1` sygnał zostanie wysłany do wszystkich procesów oprócz wysyłającego i procesu INIT,
    - `pid < -1` oznacza procesy należące do grupy o identyfikatorze `-pid`.
  - `signum` — numer przesyłanego sygnału

Wysłanie sygnału

```
int raise(int signo)
```

Wysłanie sygnału

```
int raise(int signo)
```

- Wysłanie przez proces sygnału do samego siebie

Obsługa sygnału

```
void *signal(int signum, void *f())
```

Obsługa sygnału

```
void *signal(int signum, void *f())
```

- Parametry:
  - `signum` numer sygnału, którego obsługa ma zostać zmieniona
  - `f` może obejmować jedną z trzech wartości:
    - `SIG_DFL` — (wartość 0) standardowa reakcja na sygnał
    - `SIG_IGN` — (wartość 1) ignorowanie sygnału
    - Wskaźnik do funkcji - wskaźnik na funkcję, która będzie uruchomiona w reakcji na sygnał
- Funkcja zwraca:
  - wskaźnik na poprzednio ustawioną funkcję obsługi (lub `SIG_IGN, SIG_DFL`)
  - `SIG_ERR` w wypadku błędu

Obsługa sygnału

- Nie można przechwytywać, ani ignorować sygnałów `SIGKILL` i `SIGSTOP`.
- Gdy w procesie macierzystym ustawiony jest tryb ignorowania sygnału `SIGCLD` to po wywołaniu funkcji `exit` przez proces potomny, proces zombi nie jest zachowywany i miejsce w tablicy procesów jest natychmiast zwalniane

Przykład użycia funkcji signal

```
void (*f)();

f=signal(SIGINT,SIG_IGN); //ignorowanie sygnału
SIGINT

signal(SIGINT,f); //przywrócenie poprzedniej reakcji
na syg.

signal(SIGINT,SIG_DFL); //ustaw.  standardowej
reakcji na syg.

void moja_funkcja() {
printf(Został przechwycony sygnał\n");
exit(0);
}

main(){
signal(SIGINT,moja_funkcja); //przechwycenie sygnału
}
```

Przykład użycia funkcji signal

```
void obsluga(int signo) {
printf("Odebrano sygnał %d\n", signo);
}

int main() {
signal(SIGINT, obsluga);
while(1) ; //pętla nieskończona
}
```

Oczekiwanie na sygnał

```
void *pause()
```

Oczekiwanie na sygnał

```
void *pause()
```

- Działanie:
  - Zawiesza wywołujący proces aż do chwili otrzymania dowolnego sygnału.
  - Najczęściej sygnałem, którego oczekuje pause jest sygnał pobudki `SIGALRM`.
  - Jeśli sygnał jest ignorowany przez proces, to funkcja `pause` też go ignoruje.

Funkcja alarm

```
unsigned alarm ( unsigned int sek )
```

Funkcja alarm

```
unsigned alarm ( unsigned int sek )
```

- Parametry:
  - `sek` — ilość sekund po których wysyłany jest sygnał SIGALRM
- Działanie:
  - Funkcja wysyła sygnał `SIGALRM` po upływie czasu podanym przez użytkownika.
- Wynik:
  - Jeśli w momencie wywołania oczekiwano na dostarczenie sygnału zamówionego wcześniejszym wywołaniem funkcji, zwracana jest liczba sekund pozostała do jego wygenerowania.
  - W przeciwnym wypadku zwracane jest 0.

Funkcja alarm

Funkcja `alarm`

- Uwaga:
  - Jeśli nie otrzymano jeszcze sygnału zamówionego wcześniejszym wywołaniem funkcji `alarm` poprzednie zamówienie zostaje unieważnione
  - Procesy wygenerowane przez funkcję fork() mają wartości swoich alarmów ustawione na 0
  - Procesy utworzone przez funkcję exec będą dziedziczyły alarm razem z czasem pozostałym do zakończenia odliczania

Mechanizmy IPC — kolejki komunikatów

- komunikat to pakiet informacji przesyłany pomiędzy różnymi procesami
- ma określony typ i długość
- nadawca może wysyłać komunikaty, nawet wtedy, gdy żaden z potencjalnych odbiorców nie jest gotowy do ich odbioru
- komunikaty są buforowane w kolejce oczekiwania na odebranie
- odbiorca może oczekiwać na komunikat danego typu
- komunikaty w kolejce są przechowywane nawet po zakończeniu procesu nadawcy (tak długo, aż nie zostaną odebrane, lub kolejka nie zostanie zlikwidowana)

Tworzenie kolejki komunikatów

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg)
```

Tworzenie kolejki komunikatów

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg)
```

- Parametry:
  - `key` — klucz; liczba, która identyfikuje kolejkę
  - `msgflg` — flagi (prawa dostępu, `IPC_CREAT`, `IPC_EXCL`)
- Wynik:
  - Funkcja zwraca identyfikator kolejki komunikatów w przypadku poprawnego zakończenia lub -1

Tworzenie kolejki komunikatów

- `key` (wartość całkowita typu `key_t`) — jest odnośnikiem do konkretnego obiektu w ramach danego rodzaju mechanizmów lub stałą `IPC_PRIVATE`
  - `IPC_PRIVATE` — nie jest flagą tylko szczególną wartością `key_t`; jeśli wartość ta jest użyta jako parametr `key`, to system zawsze będzie próbował utworzyć nową kolejkę
- `msgflg` — określa prawa dostępu do nowo tworzonego obiektu reprezentującego mechanizm IPC
  - opcjonalnie połączone (sumowane bitowo) z flagami:
    - `IPC_CREAT` — w celu utworzenia obiektu, jeśli nie istnieje,
    - `IPC_EXCL` — w celu wymuszenia zgłoszenia błędu, gdy obiekt ma być utworzony, ale już istnieje

Wysłanie komunikatu

```
int msgsnd(int msgid, struct msgbuf * msgp, int msgs,
int msgflg)
```

Wysłanie komunikatu

```
int msgsnd(int msgid, struct msgbuf * msgp, int msgs,
int msgflg)
```

- Parametry:
  - `msgid` — identyfikator kolejki komunikatów,
  - `msgp` — wskaźnik na bufor z treścią i typem komunikatu do wysłania,
  - `msgs` — rozmiar fragmentu bufora, zawierającego właściwą treść komunikatu,
  - `msgflg` — flagi specyfikujące zachowanie się funkcji w warunkach nietypowych
- Działanie:
  - wysyłanie komunikatu o zawartości wskazywanej przez `msgp` z ewentualnym blokowaniem procesu w przypadku braku miejsca w kolejce.
- Funkcja zwraca:
  - funkcja zwraca 0 w przypadku poprawnego zakończenia lub -1

Wysłanie komunikatu

- `msgflg` — flagi specyfikujące zachowanie się funkcji w warunkach nietypowych
  - `IPC_NOWAIT` — jeśli kolejka jest pełna, to wiadomość nie jest zapisywana do kolejki, a sterowanie wraca do procesu
  - 0 — proces jest wstrzymywany tak długo, aż zapis wiadomości nie będzie możliwy

Struktura komunikatu

```
struct msgbuf {
long mtype;
char mtext[1];
};
```

Struktura komunikatu

```
struct msgbuf {
long mtype;
char mtext[1];
};
```

- Znaczenie poszczególnych pól:
  - `mtype` — typ komunikatu; przy odbiorze możliwe jest wybieranie z kolejki komunikatów określonego rodzaju (wartość większa od 0),
  - `mtext` — treść komunikatu (może posiadać dowolną strukturę)
  - `msgs` — jest rozmiarem pola mtext

Odbiór komunikatu

- `int msgrcv(int msgid, struct msgbuf * msgp, int msgs, long msgtyp, int msgflg)`

Odbiór komunikatu

- `int msgrcv(int msgid, struct msgbuf * msgp, int msgs, long msgtyp, int msgflg)`

- Parametry:
  - `msgid` — identyfikator kolejki komunikatów,
  - `msgp` — wskaźnik do obszaru pamięci zawierającego treść komunikatu,
  - `msgs` — rozmiar właściwej treści komunikatu,
  - `msgtyp` — typ komunikatu, jaki ma być odebrany
  - `msgflg` — flagi specyfikujące zachowanie się funkcji w warunkach nietypowych
- Działanie:
  - odebranie komunikatu oznacza pobranie go z kolejki
- Funkcja zwraca:
  - funkcja zwraca rozmiar odebranego komunikatu w przypadku poprawnego zakończenia lub -1

Odbiór komunikatu

- `msgtyp` — typ komunikatu, jaki ma być odebrany
  - `msgtyp >0` — wybierany jest komuniakt o typie msgtyp
  - `msgtyp <0` — wybierany jest komuniakt o o najmniejszej wartości typu, mniejszej lub równej bezwzględnej wartości z msgtyp
  - `msgtyp=0` —typ komunikatu nie jest brany przy pobraniu
- `msgflg` — flagi specyfikujące zachowanie się funkcji w warunkach nietypowych
  - `IPC_NOWAIT` — jeśli w kolejce nie ma komunikatu, to zwracana jest wartość -1
  - 0 — proces jest wstrzymywany tak długo, aż do czasu pojawienia się komunikatu

Zarządzanie kolejką komunikatów

- `int msgctl(int msgid, int cmd, struct msgid_ds * buf)`

Zarządzanie kolejką komunikatów

- `int msgctl(int msgid, int cmd, struct msgid_ds * buf)`

- Funkcja zwraca:
  - funkcja zwraca 0 w przypadku poprawnego zakończenia lub -1
- Parametry:
  - `msgid` — identyfikator kolejki komunikatów,
  - `cmd` — stałą określająca rodzaj operacji
  - `buf` — wskaźnik na zmienną strukturalną, przez którą przekazywane są parametry operacji
- Rodzaje operacji:
  - `cmd = IPC_STAT` — pozwala uzyskać informację o stanie (atrybuty) kolejki komunikatów
  - `cmd = IPC_SET` — modyfikacja atrybutów kolejki komunikatów (identyfikator właściciela, grupy, prawa dostępu, itd)
  - `cmd = IPC_RMID` — usunięcie kolejki komunikatów

Polecenia systemowe

- `ipcs`
- `ipcs -q`
- `ipcrm -q id`

Przykład wykorzystania funkcji

```
mid = msgget(0x123, 0600 | IPC_CREAT);

struct msgbuf
{
long type;
char text[1024];
} my_msg;

strcpy(my_msg.text, "Text");
my_msg.type =5;

msgsnd(mid, &my_msg, strlen(my_msg.text)+1, 0);
msgrcv(mid, &my_msg, 1024, 5, 0);

msgctl(mid, IPC_RMID, NULL);
```

Pamięć współdzielona

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflg
```

Pamięć współdzielona

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflg
```

- Parametry:
  - `key` — klucz; liczba, która identyfikuje segment pamięci współdzielonej
  - `size` — rozmiar obszaru współdzielonej pamięci w bajtach
  - `shmflg` — flagi (prawa dostępu, `IPC_CREAT`, `IPC_EXCL`)
- Wynik:
  - funkcja zwraca identyfikator segmentu pamięci współdzielonej w przypadku poprawnego zakończenia lub -1

Przyłączenie segmentu pamięci współdzielonej

```
int shmat(int shmid, const void *shmaddr, int shmflg)
```

Przyłączenie segmentu pamięci współdzielonej

```
int shmat(int shmid, const void *shmaddr, int shmflg)
```

- Parametry:
  - `shmid` — identyfikator obszaru pamięci współdzielonej
  - `shmaddr` — adres w przestrzeni adresowej procesu, pod którym ma być dostępny segment pamięci współdzielonej (wartość `NULL` oznacza wybór adresu przez system)
  - `shmflg`— flagi specyfikujące sposób przyłączenia (`SHM_RDONLY`, `SHM_RND`, `0`)
- Wynik:
  - Włączenie segmentu pamięci współdzielonej w przestrzeń adresową procesu.
  - Funkcja zwraca adres segmentu lub -1 w przypadku błędu

Przykład

```
char *addr;

addr = (char *)shmat(shmid,NULL,0);shmdt(addr);

addr = (int *)shmat(shmid,NULL,0);

for (i=0; i<MAX; i++)

*(addr+i)=i*i;
```

Odłączenie segmentu pamięci współdzielonej

```
int shmdt(const void *shmaddr)
```

Odłączenie segmentu pamięci współdzielonej

```
int shmdt(const void *shmaddr)
```

- Parametry:
  - `shmaddr` — adres początku obszaru pamięci współdzielonej w przestrzeni adresowej procesu
- Wynik:
  - Wyłączenie segmentu pamięci współdzielonej z przestrzeni adresowej procesu.
  - Funkcja zwraca 0 w przypadku poprawnego zakończenia lub -1 w przypadku błędu

Usunięcie segmentu pamięci współdzielonej

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf)

shmctl(shmid, IPC_RMID, NULL)
```

Usunięcie segmentu pamięci współdzielonej

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf)

shmctl(shmid, IPC_RMID, NULL)
```

- Parametry:
  - `shmaddr` — adres początku obszaru pamięci współdzielonej w przestrzeni adresowej procesu
- Wynik:
  - Funkcja zwraca 0 lub -1 w przypadku błędu

Przykład

- Stworzenie, przyłączenie:

```
int shmid, i;
int *buf;
shmid = shmget(45281, MAX*sizeof(int),
IPC_CREAT|0600);
buf = (int*)shmat(shmid, NULL, 0);
```

Przykład

- Stworzenie, przyłączenie:

```
int shmid, i;
int *buf;
shmid = shmget(45281, MAX*sizeof(int),
IPC_CREAT|0600);
buf = (int*)shmat(shmid, NULL, 0);
```

- Zapis:

```
for (i=0; i<10000; i++)
buf[i%MAX] = i;
```

Przykład

- Stworzenie, przyłączenie:

```
int shmid, i;
int *buf;
shmid = shmget(45281, MAX*sizeof(int),
IPC_CREAT|0600);
buf = (int*)shmat(shmid, NULL, 0);
```

- Zapis:

```
for (i=0; i<10000; i++)
buf[i%MAX] = i;
```

- Odczyt:

```
for (i=0; i<10000; i++)
printf("Numer:  %5d Wartosc:  %5d\n", i, buf[i%MAX]);
```

Semafory

- Wykorzystanie semaforów zapobiega niedozwolonemu wykonaniu operacji na określonych danych jednocześnie przez większą liczbę procesów
- Przez odpowiednie wykorzystywanie semaforów można zapobiec sytuacji w której wystąpi zakleszczenie (ang. deadlock) lub zagłodzenie (ang. starvation)
- Semafor binarny — będący tylko w stanie podniesienia lub opuszczenia $S \in \{0, 1\}$
- Semafor uogólniony — możliwe wiele stanów $S \in \{0, 1, ..., \infty\}$
- Definicja — Edgar Dijkstra

Semafory cd.

- Obszar programu składający się z instrukcji które może wykonywać tylko określona liczba procesów = sekcja krytyczna
- Operacje wykonywane na semaforze są atomowe.
- Semafor można traktować jako licznik, który jest zmniejszany (zamykany) gdy jest zajmowany i zwiększany gdy jest zwalniany (podnoszony)
- Semafor trzeba zainicjować wywołując operację podniesienia

Operacje semaforowe

- Semafor jest pewną całkowitą liczbą nieujemną S.
- Opuszczenie semafora jest równoważne wykonaniu instrukcji:
  - jeśli $S > 0$ to $S = S - 1$,
  - w przeciwnym razie wstrzymaj działanie procesu próbującego opuścić semafor
- Podniesienie semafora:
  - jeśli są procesy wstrzymane przy próbie opuszczenia semafora S to wznów jeden z nich,
  - w przeciwnym wypadku $S = S + 1$

Struktury związane z obiektem semafora

- Struktura opisująca obiekt będący semaforem: `semid_ds`
- W systemie Unix/Linux występują tzw. zestawy semaforów. Każdy semafor z tego zestawu posiada strukturę z nim związaną:

```
struct sem{
ushort semval
pid_t sempid
ushort semncnt
ushort semzcnt
}
```

Utworzenie tablicy semaforów

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg)
```

Utworzenie tablicy semaforów

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg)
```

- Parametry:
  - `key` — klucz; liczba, która identyfikuje segment pamięci współdzielonej
  - `nsems` — liczba semaforów w tworzonym zbiorze
  - `semflg` — flagi (prawa dostępu, `IPC_CREAT`, `IPC_EXCL`)
- Wynik:
  - Utworzenie zbioru (tablicy semaforów).
  - Funkcja zwraca identyfikator zbioru semaforów w przypadku poprawnego zakończenia lub -1

Operacje kontrolne

```
int semctl (int semid, int semnum ,int cmd, union
semun ctl_arg)
```

Operacje kontrolne

```
int semctl (int semid, int semnum ,int cmd, union
semun ctl_arg)
```

- Parametry:
  - `semid` — identyfikator zestawu semaforów
  - `semnum` — numer semafora w identyfikowanym zbiorze, na którym ma być wykonana operacja
  - `cmd` — specyfikacja wykonywanej operacji kontrolnej
  - `semun` — unia

```
union semun (
int val;
struct semid_ds *buf;
unsigneda short *array
)
```

- Wynik:
  - Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Operacje kontrolne

Operacje kontrolne

- tradycyjne operacje IPC (`semnum = 0`)
  - `IPC_STAT` — zwraca wartości struktury `semid_ds` dla semafora (lub zestawu) o identyfikatorze semid, informacja jest umieszczana w strukturze wskazywanej przez 4 argument
  - `IPC_SET` — modyfikuje wartości struktury `semid_ds`
  - `IPC_RMID` usuwa zestaw semaforów o identyfikatorze semid z systemu

Operacje kontrolne

Operacje kontrolne

- operacje na pojedynczym semaforze (dotyczą semafora określonego przez `semnum`)
  - `GETVAL` — zwraca wartość semafora (`semval`)
  - `SETVAL` — ustawia wartość semafora w strukturze
  - `GETPID` — zwraca wartość `sempid`
  - `GETNCNT` — zwraca `semcnt`
  - `GETZCNT` — zwraca `semzcnt`

Operacje kontrolne

Operacje kontrolne

- operacje na wszystkich semaforach (`semnum = 0`):
  - `GETALL` — umieszcza wszystkie wartości semaforów w tablicy podanej jako 4 argument
  - `SETALL` — ustawia wszystkie wartości zgodnie z zawartością tablicy podanej jako 4 argument

Definiowanie wartości zmiennej semaforowej

```
semctl(semid, 2, SETVAL, 10)

short tab[6]={4,1,1,1,1,1};
semctl(semid, 0, SETALL, tab)
```

Wykonanie operacji na tablicy semaforów

```
int semop(int semid, struct sembuf *sops, unsigned
nsops)
```

Wykonanie operacji na tablicy semaforów

```
int semop(int semid, struct sembuf *sops, unsigned
nsops)
```

- Parametry:
  - `semid` — identyfikator zestawu semaforów
  - `sops` — adres tablicy struktur, w której każdy element opisuje opercję na jednym semaforze w zbiorze
  - `nsops` — rozmiar tablicy adresowanej przez `sops` (liczba elementów `sembuf` w tablicy)
- Wynik:
  - Jeśli jedna z operacji nie może być wykonana, żadna nie będzie wykonana. (wszystko albo nic)
  - Wykonanie operacji semaforowej; funkcja 0 w przypadku poprawnego zakończenia lub -1 w przypadku błędu

Struktura opisująca operację na semaforze

```
struct sembuf{
     short sem_num;
     short sem_op;
     short sem_flg;
}
```

Struktura opisująca operację na semaforze

```
struct sembuf{
     short sem_num;
     short sem_op;
     short sem_flg;
}
```

- Znaczenie poszczególnych pól:
  - `sem_num` — numer semafora w zbiorze (numeracja od 0),
  - `sem_op` — wartość dodawana do zmiennej semaforowej,
  - `sem_flg` — flagi operacji (`IPC_NOWAIT` — wykonanie bez blokowania)

Operacja opuszczenia semafora

```
static struct sembuf buf;

void opusc(int semid, int semnum){

buf.sem_num = semnum;
buf.sem_op = -1;
buf.sem_flg = 0;
if (semop(semid, &buf, 1) == -1){
  perror("Opuszczenie semafora");
  exit(1);
}
}
```

Operacja podniesienia semafora

```
static struct sembuf buf;

void podnies(int semid, int semnum){

buf.sem_num = semnum;
buf.sem_op = 1;
buf.sem_flg = 0;
if (semop(semid, &buf, 1) == -1){
  perror("Podnoszenie semafora");
  exit(1);
}
}
```

Wątki — informacje ogólne

- Część programu wykonywana współbieżnie w obrębie jednego procesu; w jednym procesie może istnieć wiele wątków.
- Wątki działające w danym procesie współdzielą przestrzeń adresową i inne struktury systemowe; procesy posiadają niezależne zasoby.
- Elementy wspólne procesu i wątków:
  - PID, PPID, lista otwartych plików, uprawnienia, blokady plików, obsługa sygnałów, limity zasobowe
- Elementy unikalne wątków:
  - identyfikator wątku, stos, zmienna errno, maska sygnałów
- Zmienne globalne i statyczne oraz zalokowane dynamicznie obszary pamięci są współdzielone przez wątki

Wątki — informacje ogólne

Z pozostałymi wątkami danego procesu wątek współdzieli wszystkie podstawowe zasoby, które są przydzielone procesowi:

- segment kodu programu — w każdym wątku możliwe jest wywołanie dowolnej dostępnej funkcji zdefiniowanej w programie procesu,
- segment danych — w każdym wątku przez cały czas działania procesu dostępne są zmienne alokowane w segmencie danych (zmienne globalne oraz statyczne),
- tablice otwartych plików procesu — deskryptory otwartych plików dostępne są w każdym wątku niezależnie od tego, w którym wątku zostały utworzone,
- tablice sygnałów — reakcja na sygnał zdefiniowana w dowolnym wątku obowiązuje w każdym wątku, który dany sygnał otrzyma,
- informacje o systemie plików — zmiana korzenia drzewa katalogów (chroot) lub bieżącego katalogu roboczego (chdir) w jednym z wątków dotyczy wszystkich pozostałych.

Wątki — informacje ogólne

- Konsekwencje:
  - Wątki wymagają mniej zasobów do działania i krótszy jest czas ich tworzenia.
  - Wystarczy pamiętać tylko wartości rejestrów, nie trzeba czyścić pamięci podręcznej oraz wczytywać tablicy stron
  - Mogą komunikować się za pomocą zmiennych globalnych
  - Przekazanie dowolnie dużej ilości danych wymaga przesłania jedynie wskaźnika
- Funkcje z biblioteki pthreads zwracają 0 w przypadku poprawnego wykonania. Wartość większa od zera oznacza numer błędu.

Kompilacja programu z wątkami

1. Kompilacja:
   - `$ gcc -lpthread program.c -o program`
2. Przełącznik `-lpthread` powoduje załączenie biblioteki `pthread` oraz zdefiniowanie makra `_REENTRANT`
3. Dodatkowe informacje:
   - `$ man 7 pthreads`
4. Wyświeltenie listy procesów:
   - `$ ps x`
   - `$ ps -L x`
   - `$ ps m x`

Tworzenie wątków

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const
pthread_attr_t *attr, void * (*start_func)(void *),
void *arg);
```

- Znaczenie poszczególnych pól:
  - `thread` — wskaźnik do wątku,
  - `attr` — parametry tworzonego wątku,
  - `start_func` — wskaźnik na definiowną funkcję, która będzie wykonywana jako nowy wątek; funkcja powinna zwracać wskaźnik na `void` i mieć jeden argument w postaci wskaźnika na `void`
  - `arg` — wskaźnik na rzeczywisty parametr przekazywany do funkcji

Tworzenie wątków

- Jeżeli wartość zwracana przez funkcję jest wskaźnikiem ale nie na typ `void` , należy zastosować rzutowanie: `(void * (*) ())`
- Aby przekazać kilka parametrów do funkcji należy zdefiniować odpowiednią strukturę.

Przykład

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

void *hello_world(void *arg) {
   char *s = (char *) arg;
   printf("Hello %s!\n", s);
   return 0;
}

int main(int argc, char* argv[]) {
   pthread_t hello_world_thread;
   int result = pthread_create(&hello_world_thread, NULL,
                               hello_world, (void *) argv[0]);
   if (result != 0) {
      perror("Could not create thread.");
   }
   //sleep(3);
   printf("Main thread exiting.\n");
   return 0;
}
```

Kończenie działania wątków

- Zakończy się funkcja
- Zostanie wykonane wywołanie funkcji `pthread_exit`
- Wątek zostanie anulowany za pomocą funkcji `pthread_cancel`
- Zakończy się proces macierzysty wątku
- Jeden z wątków wykona funkcję `exec`

Zarządzanie wątkami

```
int pthread_exit(void *status);
```

- Znaczenie poszczególnych pól:
  - `status` — wskaźnik na wartość stanu wątku,
- Wskaźnik jest zwracany jeśli kończony wątek nie jest wątkiem odłączonym (`detached`)
- Wartość `status` może być odczytana przez inny wątek, który zsynchronizuje się z nim wywołaniem `pthread_join`
- Wątek zwalnia swoje zasoby (tylko swoje, nie swojego procesu)

Zarządzanie wątkami

```
int pthread_cancel(pthread_t th);
```

- Znaczenie poszczególnych pól:
  - `th` — identyfikator wątku,
- Zakończenie wykonywania innego wątku. Zatrzymywanie wątku można blokować funkcją: `pthread_setcancelstate()`

Oczekiwanie na zakończenie innego wątku

```
int pthread_join(pthread_t th, void **status);
```

- Znaczenie poszczególnych pól:
  - `th` — identyfikator wątku na który czekamy,
  - `status` — wskaźnik na statyczne miejsce w pamięci, w którym zostanie zapisany stan zakończenia wątku, na który proces czekał
- Wartości reprezentujące informacje o stanie to wartość, którą przekazuje się funkcji `pthread_exit` lub wartość zwracana w chwili gdy kod funkcji osiągnie instrukcję `return`
- Jeśli drugi argument funkcji `pthread_join` ma wartość NULL, informacje o stanie zostaną zignorowane

Przykład

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

void* worker(){
  printf("thread\n");
  sleep(3);
  return NULL;
}

int main(){
  pthread_t th;
  pthread_create(&th, NULL, worker, NULL);
  printf("main program\n");
  pthread_join(th, NULL);
}
```

Przykład

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

void* worker(void *s){
  int i; for (i=1; i<=10;i++)
  printf("%s:  %d\n", (char *)s, i);
  return NULL;
}

int main(){
  pthread_t th1, th2;
  pthread_create(&th1, NULL, worker, (void*)"thread 1");
  pthread_create(&th2, NULL, worker, (void*)"thread 1");
  pthread_join(th1, NULL);
  pthread_join(th2, NULL);
}
```

Przykład

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

int* worker(int* info){
  printf("Argument:  %d\n", *info);
  int *x=malloc(sizeof(int));
  *x=10;
  return x;
}

int main(){
  pthread_t th;
  int x=3;
  pthread_create(&th, NULL, (void *) worker, &x);
  int *retVal;
  pthread_join(th, (void **) &retVal);
  printf("Return value:  %d\n", *retVal);
  free(retVal);
  return 0;
}
```

Synchronizacja wątków

- Wykorzystywane mechanizmy:
  - zamki (ang. mutex) — binarne semafory (mogą znajdować się w jednym z dwóch stanów: podniesiony lub opuszczony)
  - zmienne warunkowe (ang. conditional variable) — pozwalają na kontrolowane zasypianie i budzenie wątków w momencie zajścia określonego zdarzenia. Wraz z zamkami umożliwiają one implementowanie tzw. monitorów.
    - dostarczają mechanizmu, który pozwala czekać aż pewien współdzielony zasób osiągnie pożądany stan lub do sygnalizowania, że osiągnął już stan, którym może być zainteresowany inny wątek
    - każda zmienna warunkowa jest związana z muteksem, który chroni stan zasobu.
  - semafory nienazwane standardu POSIX
- Wszystkie wymienione mechanizmy nie wymagają tworzenia obiektów widocznych dla innych procesów.

Inicjowanie zamka

```
int pthread_mutex_init(pthread_mutex_t *mutex, const
pthread_mutexattr_t *attr);

pthread_mutex_t m;
pthread_mutex_init(&m, NULL);
```

- Znaczenie poszczególnych pól:
  - `mutex` — wskaźnik na mutex,
  - `attr` — wskaźnik na strukturę `pthread_mutexattr_t` opisującą początkowe atrybuty zamka
- Zamek z domyślnymi atrybutami można utworzyć przekazując pusty wskaźnik

Operacje na zamkach

```
int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- `mutex` — wskaźnik na mutex
- `pthread_mutex_lock` - zajęcie zamka (opuszczenie semafora). Funkcja jest blokująca jeżeli zamek przed jej wywołaniem był zajęty przez inny wątek
- `pthread_mutex_trylock` - zajęcie wolnego zamka. Próba zajęcia już zajętego zamka kończy się zasygnalizowaniem błędu (kod błędu EBUSY)

Operacje na zamkach

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- `mutex` — wskaźnik na mutex
- `pthread_mutex_unlock` - zwolnienie zamka (podniesienie semafora). Zwolnienia powinien dokonać wątek, który zajął zamek
- `pthread_mutex_destroy` - usunięcie zamka

Inicjowanie zmiennej warunkowej

```
int pthread_cond_init(pthread_cond_t *cond, const
pthread_condattr_t *attr);

pthread_cond_t cond;
pthread_cond_init(&cond, NULL);
```

- Znaczenie poszczególnych pól:
  - `cond` — wskaźnik na zmienną warunkową,
  - `attr` — wskaźnik na strukturę `pthread_condattr_t` opisującą początkowe atrybuty zmiennej warunkowej
- Zmienną warunkową z domyślnymi atrybutami można utworzyć przekazując pusty wskaźnik

Usypianie wątku na zmiennej warunkowej

```
int pthread_cond_wait(pthread_cond_t *cond,
pthread_mutex_t *mutex);
```

- Znaczenie poszczególnych pól:
  - `cond` — wskaźnik na zmienną warunkową,
  - `mutex` — wskaźnik na skojarzony mutex,

- `pthread_cond_wait` - oczekuje bezwarunkowo do czasu odebrania sygnału budzącego (będącego wewnętrznym mechanizmem synchronizacyjnym biblioteki pthread)

Usypianie wątku na zmiennej warunkowej

```
int pthread_cond_timedwait(pthread_cond_t *cond,
pthread_mutex_t *mutex, const struct timespec
*abstime);
```

- Znaczenie poszczególnych pól:
  - `abstime` — jeśli wartość wskazywana przez `*abstime` zostanie osiągnęta lub przekroczona, funkcja się przeterminuje (zakończy działanie przed czasem)
- `pthread_cond_timedwait` - oczekiwanie na wybudzenie ograniczone czasowo. Przekroczenie czasu oczekiwania sygnalizowane jest błędem ETIMEDOUT

Wysyłanie sygnału

```
int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

- `cond` — wskaźnik na zmienną warunkową,
- `pthread_cond_signal` – wysłanie sygnału budzącego do wątków oczekujących na wskazanej zmiennej warunkowej. Sygnał powoduje wybudzenie co najmniej jednego wątku
- `pthread_cond_broadcast` - wysłanie sygnałów budzących do wszystkich wątków oczekujących na wskazanej zmiennej warunkowej.

Zmienne warunkowe — uwagi

- Usypianie wątków wymaga użycia jednocześnie zmiennej warunkowej i zamka.
- Przed zaśnięciem zamek musi być już zajęty.
- Zaśnięcie oznacza atomowe zwolnienie zamka i rozpoczęcie oczekiwania na sygnał budzący.
- Obudzenie wątku powoduje ponowne zajęcie zamka.

Zastosowania zmiennej warunkowej do synchronizacji wątków

```
pthread_cond_t c;

pthread_mutex_tm;

...
pthread_mutex_lock(&m); // zajęcie zamka

pthread_cond_wait(&c, &m); // oczekiwanie na zmiennej
warunkowej

pthread_mutex_unlock(&m); // zwolnienie zamka
```

Przykład

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

int* worker(int* info){
int i; int tid=*(int*)info;
for(i=0;i<5;i++)
  printf("Thread:  %d\n is entering critical section\n", tid);
  pthread_mutex_lock(&mx);
  printf("Thread:  %d\n is in critical section\n", tid);
  sleep(5);
  pthread_mutex_unlock(&mx);
  printf("Thread:  %d\n leaving critical section\n", tid);
  sleep(2);
  return NULL;
}

int main(){
  pthread_t th1, th2, th3;
  int w1=1, w2=2, w3=3;
  pthread_mutex_init(&mx, NULL);
  pthread_create(&th1, NULL, (void *) worker, &w1);
  pthread_create(&th2, NULL, (void *) worker, &w2);
  pthread_create(&th3, NULL, (void *) worker, &w3);
  pthread_join(th1, NULL);
  pthread_join(th2, NULL);
  pthread_join(th3, NULL);
  return 0;
}
```

Przykład

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

static int buf_full;
static int buf = 0;
static pthread_mutex_t mtx;
static pthread_cond_t cnd_empty, cnd_full;

void put(int x){
   pthread_mutex_lock( &mtx );
   if ( buf_full )
      pthread_cond_wait( &cnd_empty, &mtx );
   buf = x;
   buf_full = 1;
   pthread_cond_signal( &cnd_full );
   pthread_mutex_unlock( &mtx );
}

int get(){
   int x;
   pthread_mutex_lock( &mtx );
   if ( !  buf_full)
       pthread_cond_wait( &cnd_full, &mtx );
   x = buf;
   buf_full = 0;
   pthread_cond_signal( &cnd_empty );
   pthread_mutex_unlock( &mtx );
   return x;
}
```

Przykład

```
void *send1(void* p){
    put(1);
}

void* send2(void* p){
put(2);
}

void* receive(void* p){
int x;
x = get(); printf("%d\n", x); fflush(stdout);
x = get(); printf("%d\n", x); fflush(stdout);
x = get(); printf("%d\n", x); fflush(stdout);
}

int main(){
pthread_t tsnd_1, tsnd_2, trcv;
buf_full = 1;
pthread_create(&tsnd_1, NULL, send1, NULL);
pthread_create(&tsnd_2, NULL, send2, NULL);
pthread_create(&trcv, NULL, receive, NULL);
pthread_join(trcv, NULL);
return 0;
}
```