Organizacja zajęć

1. Zasady zaliczenia:
   - 2x test: 25%
   - projekt: 40%
   - inne (zadania domowe, aktywność): 10%
2. Zaliczenie

   51-60% dst
   61-70% dst+
   71-80% db
   81-90% db+
   91-100% bdb
3. Dozwolone nieobecności: 2

Organizacja zajęć

Materiały:

- http://www.cs.put.poznan.pl/akobusinska/sop2.html
- http://www.cs.put.poznan.pl/csobaniec/edu/sop/sop2.html
- http://www.cs.put.poznan.pl/dwawrzyniak/
- http://www.cs.put.poznan.pl/mholenko
- http://www.cs.put.poznan.pl/pzierhoffer
- http://www.cs.put.poznan.pl/sop/

Organizacja zajęć

Materiały:

- http://www.cs.put.poznan.pl/akobusinska/sop2.html
- http://www.cs.put.poznan.pl/csobaniec/edu/sop/sop2.html
- http://www.cs.put.poznan.pl/dwawrzyniak/
- http://www.cs.put.poznan.pl/mholenko
- http://www.cs.put.poznan.pl/pzierhoffer
- http://www.cs.put.poznan.pl/sop/

Serwer:

- unixlab.cs.put.poznan.pl

Organizacja zajęć

Materiały:

- http://www.cs.put.poznan.pl/akobusinska/sop2.html
- http://www.cs.put.poznan.pl/csobaniec/edu/sop/sop2.html
- http://www.cs.put.poznan.pl/dwawrzyniak/
- http://www.cs.put.poznan.pl/mholenko
- http://www.cs.put.poznan.pl/pzierhoffer
- http://www.cs.put.poznan.pl/sop/

Serwer:

- unixlab.cs.put.poznan.pl

Opis funkcji:

- www.refcards.com

Środowisko pracy

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

Środowisko pracy

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

Środowisko pracy

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

Wczytywanie parametrów

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

```
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]);
}
```

```
    }
```

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

Otwieranie / tworzenie pliku

- Funkcja:
  `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

- Funkcja:
  `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

- Funkcja:
  `ssize_t write(int fd, const void *buf, size_t count)`
- Parametry:
  - `fd` — deskryptor pliku, z którego następuje odczyt 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); }
```

Przesuwanie wskaźnika bieżącej pozycji

- Funkcja:
  `off_t lseek(int fd, off_t offset, int whence)`
- Parametry:
  - `fd` — deskryptor pliku, z którego następuje odczyt danych,
  - `offset` — wielkość przesunięcia
  - `whence` — odniesienie
    - SEEK_SET
    - SEEK_END
    - SEEK_CUR

Zamykanie/usuwanie pliku

- Funkcja: `int close(int fd)`
- Funkcja: `int unlink(const char *pathname)`

Błędy

- 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:

```
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

- Biblioteki: `<unistd.h>, <sys/types.h>`
- Funkcja: `pid_t fork(void)`
- Przykład:

```
#include <stdio.h>
main(){
    printf (Poczatek\n);
    fork();
    printf (Koniec\n);
```

Proces macierzysty / potomny

```
#include <stdio.h>
main(){
     if (fork()==0)
          printf (“Child\n”);
     else
          printf (“Parent\n”);
```

Identyfikatory procesów

- Funkcja: `pid_t getpid(void)`
- Funkcja: `pid_t getppid(void)`
- Zadanie: Napisz program tworzący dwa procesy. Dla każdego z procesów podaj wartość PID i PPID.

Kolejność wykonania procesów

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
main(){
     fork();
     printf (Hi\n);
     fork();
     printf (Ha\n);
     fork();
     printf (Ho\n);
```

Ilość stworzonych procesów

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
main (){

     fork ();
     fork ();

          if ( fork () == 0)

           fork ();

     fork ();
     }
```

Ilość stworzonych procesów

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
main (){

     fork ();
     fork ();

          if ( fork () != 0)

           exit ();

     fork ();
     }
```

Ilość stworzonych procesów

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
main (){

     fork ();
     fork ();

          if ( fork () != 0)
          if ( fork ()== 0)

           fork();

          else

           exit();
          fork ();

     }
```

Zakończenie procesów

- Funkcja: `void exit(int status)`
- Parametry:
  - `status` — kod wyjścia przekazywany procesowi macierzystemu
- Funkcja: `int kill(pid_t pid, int signum)`
- Parametry:
  - `pid` — identyfikator procesu, do którego adresowany jest sygnał
  - `signum` — numer przesyłanego sygnału
- Przykład: `exit(7), kill(pid,9)`

Odczytanie informacji o sposobie zakończenia potomka

- Biblioteka `<sys/wait.h>`
- Funkcja: `pid_t wait(int *status)`
- Funkcja zwraca identyfikator zakończonego procesu lub -1 w przypadku błędu
- Parametry:
  - `status` — adres słowa w pamięci, w którym umieszczony zostanie status zakończenia

- 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.

Odczytanie informacji o sposobie zakończenia potomka

- Status zakończenia:
  - numer sygnały (mniej znaczące 7 bitów)
  - kod wyjścia (bardziej znaczący bajt będący wartością fn. `exit` wywołanej przez potomka)
- `wait(NULL)`
- `int status; 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

Uruchomianie programu

- Funkcje:
  - `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.

Uruchomianie programu

- Przykłady:
  - `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);`

Łącza

- Przykłady: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 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
- łącze nienazwane (potok) — nie ma nazwy w żadnym katalogu i istnieje tak długo po utworzeniu, jak długo otwarty jest jakiś deskryptor tego łącza.

Tworzenie łącza nienazwanego

- Biblioteki: `<unistd.h>`
- Funkcja: `int pipe(int fd[2])`
- Funkcja zwraca 0 w przypadku poprawnego zakończenia lub -1
- Parametry:
  - `fd` — tablica 2 deskryptorów;
  - `fd[0]` — deskryptor potoku do odczytu
  - `fd[1]` — deskryptor potoku do zapisu

Powielanie deskryptorów (1)

- Funkcja: `int dup (int fd)`
- Powielenie (duplikacja deskryptora)
- Funkcja zwraca numer nowo przydzielonego deskryptora lub -1 w przypadku błędu
  - Parametry:
    - `fd` — deskryptor który ma zostać powielony

Powielanie deskryptorów (2)

- Funkcja: `int dup2 (int oldfd, int newfd)`
  - Powielenie (duplikacja deskryptora) we wskazanym miejscu w tablicy dskryptorów
  - Funkcja zwraca numer nowo przydzielonego deskryptora lub -1 w przypadku błędu
  - Parametry:
    - `oldfd` — deskryptor który ma zostać powielony
    - `newfd` — numer nowoprzydzielonego deskryptora

Tworzenie łącza nazwanego

- Biblioteki: `<sys/types.h>`, `<sys/stat.h>`
  - Funkcja: `int mkfifo(const char *pathname, mode_t mode)`
    - Funkcja zwraca 0 w przypadku poprawnego zakończenia lub -1
    - Parametry:
    - `pathname` — nazwa pliku (w szczególności nazwa ścieżkowa)
    - `mode` — prawa dostępu do nowo tworzonego pliku.

Otwieranie łącza nazwanego

- Funkcja `mkfifo` tworzy plik specjalny typu łącze
  - `mkfifo("kolFIFO", 0600);`
- Funkcja `open` otwiera łącze nazwane
  - `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

- Biblioteki: `<signal.h>`
- Funkcja: `int kill(pid_t pid, int signum)`
- Parametry:
  - `pid` — identyfikator procesu, do którego adresowany jest sygnał
    - `pid > 0` sygnał zostanie wysłany do procesu o identyfikatorze pid,
    - `pid = 0` sygnał zostanie wysłąny 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
- Funkcja zwraca 0 w przypadku poprawnego zakończenia lub -1
- Funkcja: `int raise (int signo)` — wysłanie przez proces sygnału do samego siebie

Określenie sposobu obsługi sygnału

- Funkcja: `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
- Zwaracane wartości:
  - wskaźnik na poprzednio ustawioną funkcję obsługi (lub SIG_IGN, SIG_DFL)
  - SIG_ERR w wypadku błędu

Określenie sposobu obsługi 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

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

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

Oczekiwanie na sygnał

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

Funkcja `alarm`

- Funkcja: `unsigned alarm ( unsigned int sek )`
- Parametry:
  - sek — ilość sekund po których wysyłany jest sygnał SIGALARM
- Działanie:
  - Funkcja wysyła sygnał `SIGALARM` 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.
- Uwaga:
  - jeśli nie otrzymano jeszcze sygnału zamówionego wcześniejszym wywołaniem funkcji `alarm` poprzednie zamówienie zostanie unieważnione.

Funkcja `alarm`

- 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 wysyłąć 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

- Biblioteki: `<sys/types.h>`, `<sys/ipc.h>`, `<sys/msg.h>`
- Funkcja: `int msgget(key_t key, int msgflg)`
  - Funkcja zwraca identyfikator kolejki komunikatów w przypadku poprawnego zakończenia lub -1
  - Parametry:
    - `key` — klucz; liczba, która identyfikuje kolejkę
    - `msgflg` — flagi (prawa dostępu, `IPC_CREAT`, `IPC_EXCL`)

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
- flg — 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

- Funkcja: `int msgsnd(int msgid, struct msgbuf * msgp, int msgs, int msgflg)`
  - funkcja zwraca 0 w przypadku poprawnego zakończenia lub -1
  - wysyłanie komunikatu o zawartości wskazywanej przez `msgp` z ewentualnym blokowaniem procesu w przypadku braku miejsca w kolejce.
  - 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

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];
  };
  ```
- 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

- Funkcja: `int msgrcv(int msgid, struct msgbuf * msgp, int msgs, long msgtyp, int msgflg)`
  - funkcja zwraca rozmiar odebranego komunikatu w przypadku poprawnego zakończenia lub -1
  - odebranie komunikatu oznacza pobranie go z kolejki
  - 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

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

- Funkcja: `int msgctl(int msgid, int cmd, struct msgid_ds * buf)`
  - funkcja zwraca0 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);
```

Utworzenie segmentu pamięci współdzielonej

- Biblioteki: `<sys/types.h>`, `<sys/ipc.h>`, `<sys/shm.h>`
- Funkcja: `int shmget(key_t key, int size, int shmflg)`
  - Funkcja zwraca identyfikator segmentu pamięci współdzielonej w przypadku poprawnego zakończenia lub -1
  - Parametry:
    - `key` — klucz; liczba, która identyfikuje segment pamięci współdzielonej
    - `size` — rozmiar obszaru współdzielonej pamięci w bajtach
    - `shmflg` — flagi (prawa dostępu, `IPC_CREAT`, `IPC_EXCL`)

Przyłączenie segmentu pamięci współdzielonej

- Funkcja: `int shmat(int shmid, const void *shmaddr, int shmflg)`
  - Włączenie segmentu pamięci współdzielonej w przestrzeń adresową procesu. Funkcja zwraca adres segmentu lub -1 w przypadku błędu
  - 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`)

Odłączenie segmentu pamięci współdzielonej

- Funkcja: `int shmdt(const void *shmaddr)`
  - wyłączenie segmentu pamięci współdzielonej z przestrzeni adresowej procesu. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub -1 w przypadku błędu
  - Parametry:
    - `shmaddr` — adres początku obszaru pamięci współdzielonej w przestrzeni adresowej procesu

Przyłączenie segmentu pamięci współdzielonej

- Funkcja: `int shmat(int shmid, const void *shmaddr, int shmflg)`
  - Włączenie segmentu pamięci współdzielonej w przestrzeń adresową procesu. Funkcja zwraca adres segmentu lub -1 w przypadku błędu
  - 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`)

Usunięcie segmentu pamięci współdzielonej

- Funkcja: `int shmctl(int shmid, int cmd, struct shmid_ds *buf)`
  - Usunięcie segmentu pamięci współdzielonej. Funkcja zwraca 0 lub -1 w przypadku błędu
  - `shmctl(shmid, IPC_RMID, NULL)`

Przykłady

- `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;
  ```
  ```

Przykłady

- 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) o 1 gdy jest zajmowany i zwiększany o 1 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

- Biblioteki: `<sys/types.h>`, `<sys/ipc.h>`, `<sys/sem.h>`
- Funkcja: `int semget(key_t key, int nsems, int semflg)`
  - Utworzenie zbioru (tablicy semaforów).
  - Funkcja zwraca identyfikator zbioru semaforów w przypadku poprawnego zakończenia lub -1
  - Parametry:
    - key — klucz; liczba, która identyfikuje segment pamięci współdzielonej
    - `nsems` — liczba semaforów w tworzonym zbiorze
    - `semflg` — flagi (prawa dostępu, `IPC_CREAT`, `IPC_EXCL`)

Operacje kontrolne

- Funkcja: `int semctl (int semid, int semnum ,int cmd, union semun ctl_arg)`
  - wykonanie operacji kontrolnych na semaforze lub zbiorze semaforów. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.
  - 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
)
```

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 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 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

- Funkcja: `int semop(int semid, struct sembuf *sops, unsigned nsops)`
  - wykonanie operacji semaforowej; funkcja 0 w przypadku poprawnego zakończenia lub -1 w przypadku błędu
  - jeśli jedna z operacji nie może być wykonana, żadna nie będzie wykonana. (wszystko albo nic)
  - 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)

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);
}
}
```

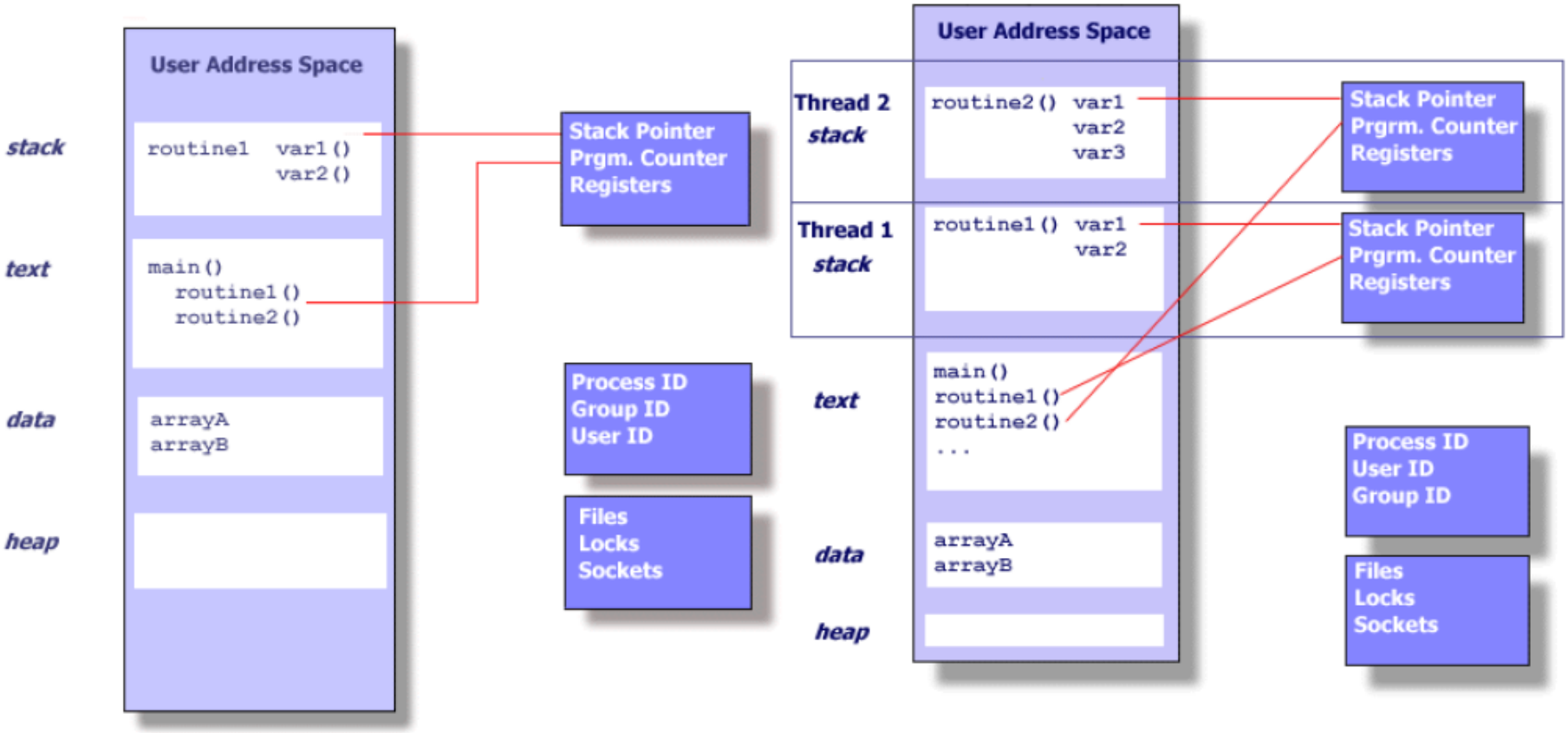
Co to jest wątek

- Niezależny strumień instrukcji, który może być wykonywany jednocześnie z innym strumieniem instrukcji danego procesu
  - „procedura”, która może zostać wykonana niezależnie od głównego programu danego procesu
  - jeśli program (np.. a.out) jest zbiorem pewnej liczby procedur, to każda z tych procedur może zostać wykonana niezależnie oraz na przemian w systemie operacyjnym

Wątek vs. Proces

- Proces jest tworzony przez system operacyjny; duży narzut na jego utworzenie
- Proces zawiera informację o:
  - PID, PGID, UID, GID
  - otoczenie
  - folder roboczy
  - instrukcje programu
  - rejestry, stosy, deskryptory otwartych plików, reakcje na sygnały
  - współdzielone biblioteki
  - IPC

Wątek vs. Proces



Wątek vs. Proces

- Wątek tworzony jest wewnątrz procesu i wykorzystuje jego zasoby.
- Wątek do pracy tworzone ma na nowo tylko niezbędne struktury, resztę współdzieli z innymi wątkami/procesem:
  - wskaźnik stosu
  - rejestry
  - ustawienia planowania CPU
  - zestaw obsługi sygnałów
  - dane specyficzne dla wątku (ID wątki, itd.)

Wątek vs. Proces

- Wątek posiada swój własny przepływ w instrukcji działania, dopóki istnieje proces go tworzący zmiany wykonane na zmiennych globalnych są widoczne w innych wątkach, ze wszystkimi tego skutkami.
- Wątki współdzielą tablicę otwartych plików i identyfikator procesu

Pthreads

- Producenci sprzętu i oprogramowania wprowadzili różne wersje wątków — problem z przenaszalnością oprogramowania pomiędzy platformami
- POSIX opracował standard IEEE 1003.1c (1995r.) — standard został wcielony w systemy z rodziny UNIX
- Większość producentów opracowało swoje własne biblioteki obsługi wątków (w tym Windows), aby umożliwić przenoszenie kodu.
- Pthreads to zestaw definicji procedur i wywołań w oparciu o język C
- Implementacja wątków Pthreads została zawarta w bibliotece `Pthreads.h`

Po co?

- Zwiększenie potencjalnej efektywności programu w porównaniu do procesów
- Mniejsze wymagania związane z ich obsługą i tworzeniem, przez co są szybsze i wydajniejsze
- Ponieważ wątki współdzielą ze sobą te same obszary pamięci (zmienne, pliki, itd.) nie wymagany jest nakład na wykorzystanie technik IPC (które są wolne)

Klasy Pthread API

Standard POSIX organizuje funkcje w klasy:

- Zarządzenie wątkami
  - funkcje działające bezpośrednio na wątkach — tworzenie, odłączanie, łączenie.
  - funkcje związane z ustawieniem i sprawdzaniem atrybutów wątków — jonable, scheduling
- Zamki (mutex): funkcje umożliwiające synchronizowanie wątków.
- Zmienne warunkowe (condition variables):
  - synchronizują wątki bez aktywnego oczekiwania w sekcji wejściowej.
  - muszą być wykorzystywane ze zmienną mutex.
  - odblokowywane są po spełnieniu warunku.

Konwencja nazw funkcji

Prefix	Funkcja
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Atrybuty wątków
pthread_mutex_	Mutexey
pthread_mutexattr_	Atrybuty obiektów typu mutex
pthread_cond_	Zmienne warunkowe- Condition variables
pthread_condattr_	Atrybuty obiektów typu condition variables
pthread_key_	Thread-specific data keys

Kompilacja

- Kompilacja programu z wątkami:
  `gcc -lpthread program.c -o program`
- Dodatkowe informacje o wątkach:
  `man 7 pthreads`
- Biblioteka `<pthread.h>`

Tworzenie wątków

- Funkcja: `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);`
- Funkcja zwraca 0 jeśli wątek został stworzony, inna wartość – numer błędu
- Parametry:
  - `thread` — unikalny identyfikator wątka
  - `attr` — zestaw opakowanych atrybutów (rozmiar stosu, jego adres, priorytet, stan), NULL- domyślne
    - Jeżeli chcemy przekazać wątkowi inne atrybuty — funkcja `int pthread_attr_init(pthread_attr_t *attr);`
  - `start_routine` — wskaźnik na funkcję od której wątek rozpocznie działanie
  - `arg` — wskaźnik do obszaru z argumentami do przekazania do funkcji startującej wątek

Tworzenie wątków

```
#include<stdio.h>
#include<pthread.h>
pthread_mutex_t mtx;
void* work(void* arg) {
    printf("[thraed] Thread is working ...\n");
    sleep(3);
    return 0;
}
int main() {
    pthread_t th;
    printf("Creating thread ...\n");
    pthread_create(&th,NULL,work,NULL);
    printf("Thread is running ...\n");
    sleep(3);
    printf("Thread ended ...\n");
    return 0;
}
}
```

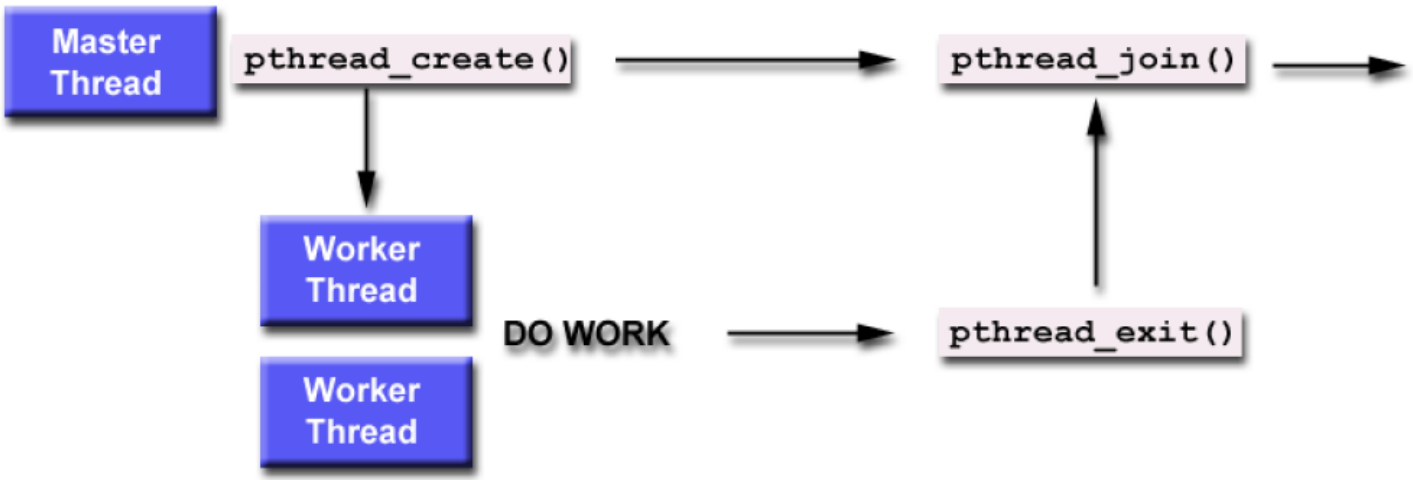
Kończenie wątku

- Wątek powraca ze swojej funkcji
- Wątek wywołuje procedurę `void pthread_exit(void *status);`
- Zakończenie wątku przez inny wątek za pomocą procedury `int pthread_cancel(pthread_t thread);`
- Kończący się proces, który zamyka wszystkie swoje wątki

- Funkcja `pthread_exit()` nie zamyka deskryptorów do plików, należy to zrobić samemu
- Zakończenie wątku działającego w funkcji `main()`nie powoduje zatrzymania innych wątków danego procesu

Oczekiwanie na zakończenie

- Funkcja: `int pthread_join(pthread_t th, void **status);`
  - zatrzymuje wątek wykonujący, aż wątek `th` zakończy swoje działanie
  - po zakończeniu się wątku th wątek czekający otrzymuje status zakończenia przekazany przez funkcję `pthread_exit()`



Zarządzanie wątkami

- Unikalny identyfikator wątka odczytujemy funkcją `pthread_self()`
- Zakończenie wątku przez inny wątek: `int pthread_cancel(pthread_t thread);`
- Blokowanie zatrzymania wątku: `int pthread_setcancelstate(int state, int *oldstate);`
- Funkcja: `int pthread_cancel(pthread_t thread);`
- Jeśli wątek domyślnie został utworzony jako dołączony, może wyłączyć możliwość oczekiwania na niego funkcją: `int pthread_detach(pthread_t th)`
- Taki wątek kontynuuje pracę niezależnie od wątku głównego — jego zasoby są zwalniane po zakończeniu wątku

Przekazywanie parametrów do wątków

- Funkcja `pthread_create()` umożliwia przekazanie tylko jednego parametru do tworzonego wątku
- Jeśli wymagamy przekazania większej ilości parametrów musimy zadeklarować strukturę i przekazać do niej wskaźnik w funkcji `pthread_create()`
- Wszystkie parametry muszą zostać przekazane przez referencję oraz zrzutowanie do postaci `(void *) param`
- Zwracanie parametrów:

```
pthread_exit(&result);
...
pthread_join(th,&return_val);
printf("Thread ended with value %d\n",
return_val);
```

Przekazywanie parametrów do wątków

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

#define NUM_THREADS 5
void *PrintHello(void *threadid){
printf("Hello World!  It's me, thread %d!\n", threadid);

}


int main (int argc, char *argv[]){
pthread_t threads[NUM_THREADS];
int rc,t;
for(t=0; t<NUM_THREADS; t++){
    printf("In main:  creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
}
pthread_exit(NULL);
}
```

Przekazywanie parametrów do wątków

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 8

char *messages[NUM_THREADS];
struct thread_data
{
   int thread_id;
   int sum;
   char *message;
};
struct thread_data thread_data_array[NUM_THREADS];
```

Przekazywanie parametrów do wątków

```
void *PrintHello(void *threadarg){
int taskid, sum;
char *hello_msg;

struct thread_data *my_data;
   sleep(1);
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;

   hello_msg = my_data->message;
   printf("Thread %d:  %s Sum=%d\n", taskid, hello_msg, sum);
   pthread_exit(NULL);
}
```

Przekazywanie parametrów do wątków

```
int main(int argc, char *argv[])
{
pthread_t threads[NUM_THREADS];
int *taskids[NUM_THREADS];
int rc, t, sum;
sum=0;
messages[0] = "English:  Hello World!";
messages[1] = "French:  Bonjour, le monde!";
messages[2] = "Spanish:  Hola al mundo";
messages[3] = "Klingon:  Nuq neH!";
messages[4] = "German:  Guten Tag, Welt!";
messages[5] = "Russian:  Zdravstvytye, mir!";
messages[6] = "Japan:  Sekai e konnichiwa!";
messages[7] = "Latin:  Orbis, te saluto!";
for(t=0;t<NUM_THREADS;t++) {
   sum = sum + t;
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   thread_data_array[t].message = messages[t];
   printf("Creating thread %d\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello, (void
*),&thread_data_array[t]);
}
pthread_exit(NULL);
```

Zamki (ang. mutex) — mechanizm synchronizacji

- Zamki są binarnymi semaforami
- Na zamku można wykonywać dwa działania:
  - Lock – blokowanie
  - Unlock – odblokowanie
- Zablokowanie zamku oznacza przejęcie jego na własność oraz wyjście z sekcji wejściowej i możliwość wejścia do sekcji krytycznej
  - Tylko jeden wątek może zablokować mutex w danej chwili

Tworzenie i usuwanie zamków

- Tworzenie zamka: `int pthread_mutex_init(`
  `pthread_mutex_t *mutex,`
  `const pthread_mutexattr_t *mutexattr);`

  Wywołanie:

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

- Usunięcie zamka:
  `pthread_mutex_destroy((pthread_mutex_t *mutex)`

Operacje na zamkach

- Nowo utworzony mutex jest odblokowany
- Zajęcie zamka:
  `int pthread_mutex_lock(pthread_mutex_t *mutex);`
  - jeśli mutex jest odblokowany to polecenie go zablokuje i umożliwi wątkowi wykonanie następnej operacji
  - jeśli mutex jest zablokowany to wątek będzie czekał na jego odblokowanie
- Zwolnienie zamka:
  `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
  - Jeśli mutex był odblokowany, lub jeśli wątek nie był jego właścicielem to wykonanie tej instrukcji spowoduje błąd

Operacje na zamkach

- Jeśli wykorzystywanych jest więcej niż jeden mutex do blokowania zmiennych dzielonych może dojść do wystąpienia blokady
- Próba zapobiegania blokadzie:
  `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
  - Jeśli mutex jest zablokowany to funkcja ta zwraca błąd EBUSY, co umożliwia napisanie protokołu zwrócenia zablokowanego pierwszego mutexu aby nie doszło do blokady

Operacje na zamkach

```
#include<stdio.h>
#include<pthread.h>
pthread_mutex_t mtx;
void* work(void* arg) {
   int id = pthread_self();
   int i;
   for(i=0;i<10;i++) {
       printf("[%d] Waiting ...\n",id);
       pthread_mutex_lock(&mtx);
       printf("[%d] Critical section ...  \n",id);
       sleep(1);
       printf("[%d] Exiting ...  \n",id);
       pthread_mutex_unlock(&mtx);
       usleep(100000);
   }
}
int main() {
   pthread_t th1, th2;
   pthread_mutex_init(&mtx,NULL);
   pthread_create(&th1,NULL,work,NULL);
   pthread_create(&th2,NULL,work,NULL);
   pthread_join(th1,NULL);
   pthread_join(th2,NULL);
}
```

Zmienne warunkowe (ang. condition variables)

- Zamki synchronizują w oparciu o kontrolę dostępu do danych, a zmienne warunkowe synchronizują wątki w oparciu o aktualną wartość danych
  Zmienne warunkowe pozwalają na kontrolowane zasypianie i budzenie procesów w momencie zajścia określonego zdarzenia.
- Bez zmiennych warunkowych trzeba by było zatrzymywać proces w aktywnym czekaniu co zajmuje czas procesora
- Zmienne warunkowe wykorzystujemy **ZAWSZE** w połączeniu z mutexami !!! — jeśli tego nie zrobimy może dojść do zakleszczenia

Operacje na zmiennych warunkowych

- Zmienna warunkowa musi zostać zadeklarowana jako typ: `pthread_condt_t cond;`
- Zajęcie zamka: `pthread_cond_init(&cond,attr);`
  - `attr` — atrybut określający czy `cond` może być wykorzystywana przez wątki w innych procesach (domyślnie inicjujemy wartością NULL)
- Dealokacji zmiennej warunkowej: `pthread_cond_destroy(cond);`

Budzenie wątków

- Budzi co najwyżej jednego wątku oczekującego na wskazanej zmiennej warunkowej:
```
int pthread_cond_signal(pthread_cond_t *cond);
```
- Budzenie wszystkich wątków uśpionych na wskazanej zmiennej warunkowej:
```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- Oczekiwanie bezwarunkowo do czasu odebrania sygnału budzącego:
```
int pthread_cond_wait(pthread_cond_t *cond,
pthread_mutex_t *mutex);
```
- Ogranicza maksymalnego czasu oczekiwania:
```
int pthread_cond_timedwait(pthread_cond_t *cond,
            pthread_mutex_t *mutex,
            const struct timespec *abstime);
```

Zmienne warunkowe w synchronizacji wątków

- wątek oczekujący:

```
pthread_cond_t c;
pthread_mutex_t m;
...
pthread_mutex_lock(&m); /* zajęcie zamka */
pthread_cond_wait(&c, &m); /* oczekiwanie na
zmiennej warunkowej */
pthread_mutex_unlock(&m); /* zwolnienie zamka */
```

- wątek budzący:

```
pthread_cond_signal(&c); /* sygnalizacja zmiennej
warunkowej */
```