LINUX
PROGRAMOWANIE SYSTEMOWE
[2/3]

Plan wykładu

- Wątki
- Mechanizmy IPC
  - Kolejki komunikatów
  - Pamięć współdzielona
  - Semafory
- Gniazda

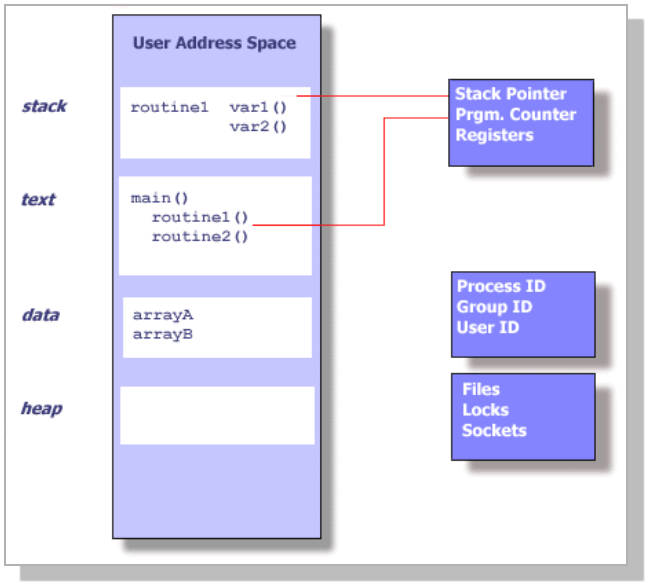
Wątki

- **Wątki**
- Mechanizmy IPC
- Gniazda

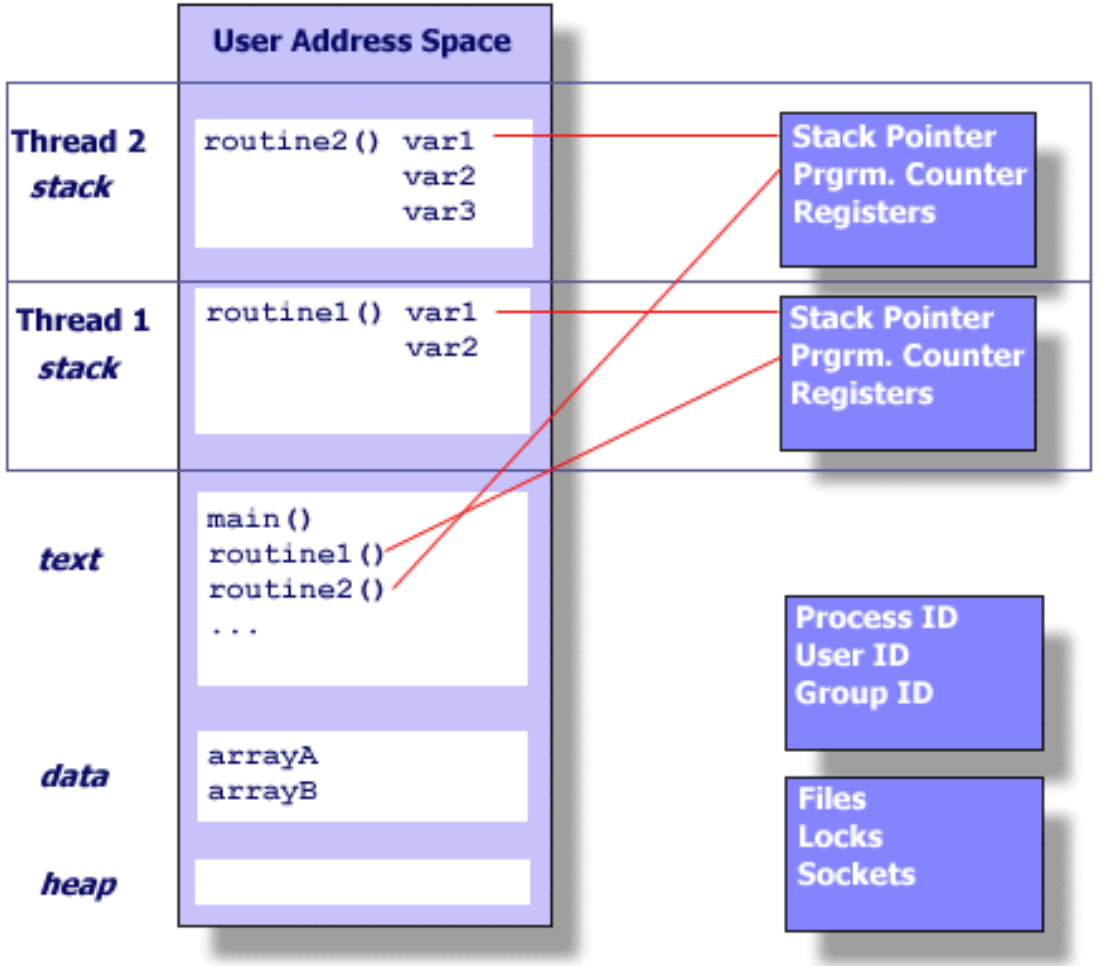
Co to jest wątek ?

- **Wątek** (*ang. thread*) - to jednostka wykonawcza w obrębie **jednego procesu**, będąca ciągiem instrukcji wykonywanym w obrębie tych samych danych (w tej samej przestrzeni adresowej).
- **Wątki** są jednostką **podrzędna** w stosunku do **procesów** – żaden wątek nie może istnieć bez procesu nadrzędnego, ale jeden proces może mieć więcej niż jeden wątek podporządkowany (*uproszczenie*).
- W systemach wieloprocesorowych, a także w systemach z wywłaszczaniem, wątki mogą być wykonywane **współbieżnie**. Równoczesny dostęp do wspólnych danych grozi jednak **utratą spójności danych** i w konsekwencji **błędem działania programu**.

Proces a wątek



Proces jednowątkowy



Proces wielowątkowy

Właściwości wątków (1/2)

- Wątki działają w ramach **wspólnych zasobów procesu**, duplikując jedynie zasoby niezbędne do wykonywania kodu.
- Aby umożliwić niezależne wykonywanie wątków, zarządzają one swoimi egzemplarzami:
  - wskaźnika na stos,
  - rejestrów,
  - informacje dotyczące planowania (np. priorytet),
  - zestawów sygnałów blokowanych i obsługiwanych,
  - danych lokalnych wątku.

Właściwości wątków (2/2)

- Ponieważ wątki **współdzielą zasoby procesu**, więc:
  - zmiany dokonane przez jeden wątek na współdzielonym zasobie (np. zamknięcie otwartego pliku) będą **widoczne dla pozostałych** wątków tego procesu,
  - wskaźniki o tej samej wartości wskazują na **te same dane** (ta sama przestrzeń adresowa),
  - możliwe jest czytanie i pisanie do **tego samego obszaru** pamięci przez różne wątki jednego procesu; wymusza to jawne stosowanie przez programistę **technik synchronizacji**.

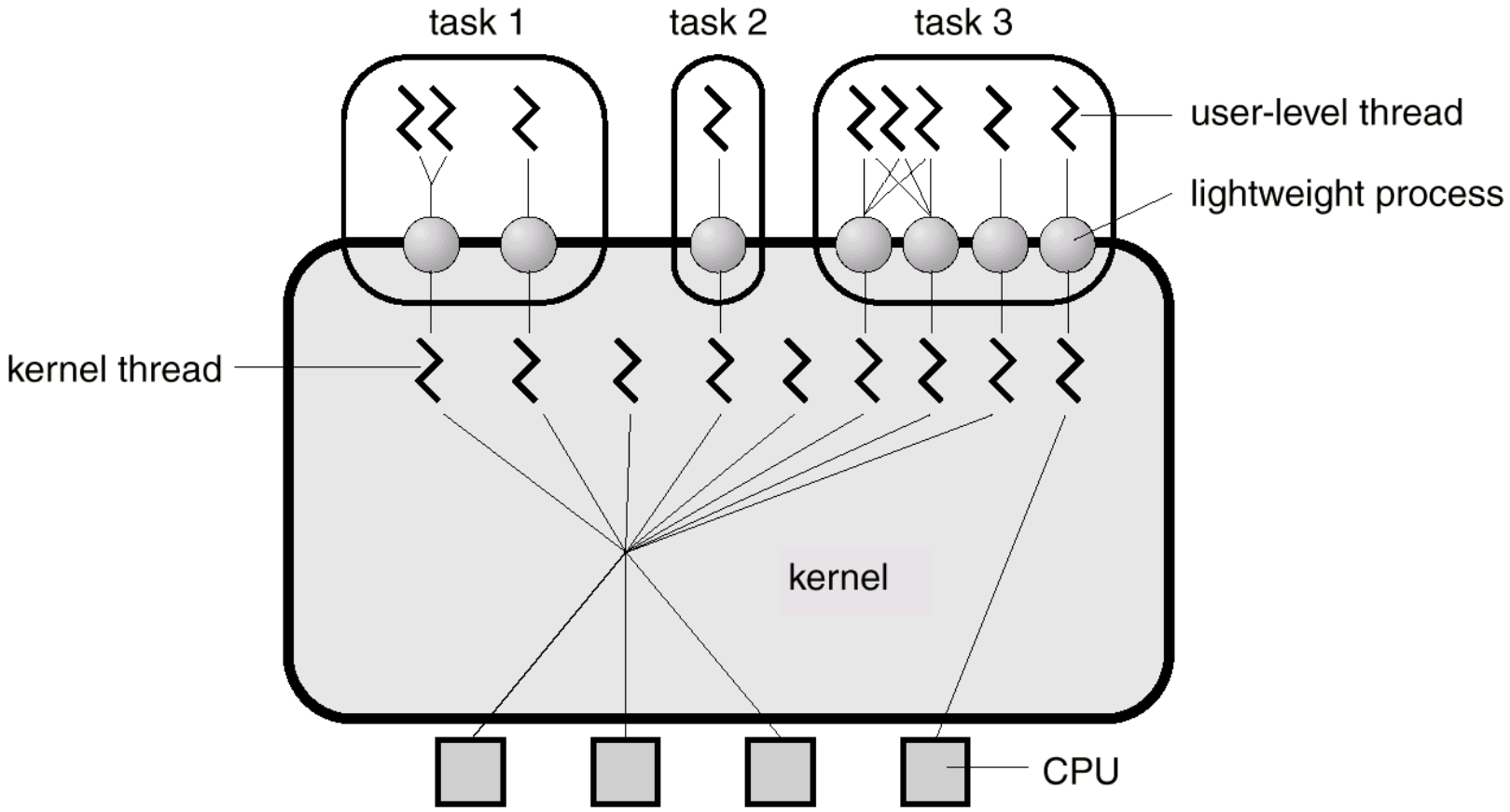
Wątki poziomu użytkownika

- Wątki poziomu użytkownika rezygnują z zarządzania wykonaniem przez jądro i **robią to same**.
- Wątek "rezygnuje" z procesora poprzez bezpośrednie wywołanie żądania wymiany (wysłanie sygnału i zainicjowanie mechanizmu zarządzającego) albo przez odebranie sygnału zegara systemowego.
- Duża szybkość przełączania, ale:
  - **problem "kradzenia" czasu** wykonania innych wątków przez jeden wątek
  - oczekiwanie jednego z wątków na zakończenie **blokującej operacji** wejścia/wyjścia powoduje, że inne wątki tego procesu też tracą swój czas wykonania

Wątki poziomu jądra

- Wątki poziomu jądra są często implementowane poprzez dołączenie do każdego procesu **tabeli** jego **wątków**.
- W tym rozwiązaniu system zarządza każdym wątkiem wykorzystując kwant czasu przyznany dla jego procesu-rodzica.
- Zaletą takiej implementacji jest zniknięcie zjawiska "kradzenia" czasu wykonania innych wątków przez "zachłanny" wątek, bo zegar systemowy tyka niezależnie i system wydziedzicza "niesforny" wątek. Także blokowanie operacji wejścia/wyjścia nie jest już problemem.

Wątki w systemie Solaris 2



Funkcja `clone` (1/2)

- Funkcja **`clone`** tworzy nowy proces. W odróżnieniu od **`fork`**, funkcja ta pozwala procesom potomnym **współdzielić części ich kontekstu** wykonania, takie jak obszar pamięci, tablica deskryptorów plików czy tablica programów obsługi sygnałów, z procesem wywołującym.
- Głównym jej zastosowaniem jest implementacja wątków poziomu jądra. Bezpośrednie użycie funkcji **`clone`** we własnych programach nie jest zalecane. Funkcja ta jest **specyficzna dla Linux-a** i nie występuje w innych systemach uniksowych. Zaleca się stosowanie funkcji z bibliotek implementujących wątki, np. **Pthread**.

Funkcja `clone` (2/2)

```
int clone( int (*fn)(), void **stack, int flags, int
   argc,... /* args */);
```

fn - funkcja wykonywana przez wątek

stack - stos wątku

flags - flagi mówiące co jest współdzielone między wątkiem a rodzicem (np. **CLONE_VM** współdzielenie danych, **CLONE_FS** współdzielenie tablicy plików, **CLONE_FILES** współdzielenie informacji o otwartych plikach, **CLONE_SIGHAND** współdzielenie tablicy sygnałów **CLONE_PID** współdzielenie PID)

argc - liczba parametrów przekazywanych do ***fn***

args - parametry wymagane przez ***fn***

Wątki PTHREAD

- Istnieje wiele odmiennych wersji implementacji wątków dla różnych platform sprzętowych i systemowych.

- W celu ujednolicenia interfejsu programowego wątków dla systemu UNIX organizacja IEEE wprowadziła normę POSIX 1003.1c (POSIX threads czyli **Pthreads**)

- Wątki **PTHREAD** zostały zdefiniowane w postaci zestawu typów i procedur języka **C** (dla **Linuxa** nagłówek **pthread.h** header i biblioteka **pthread**)

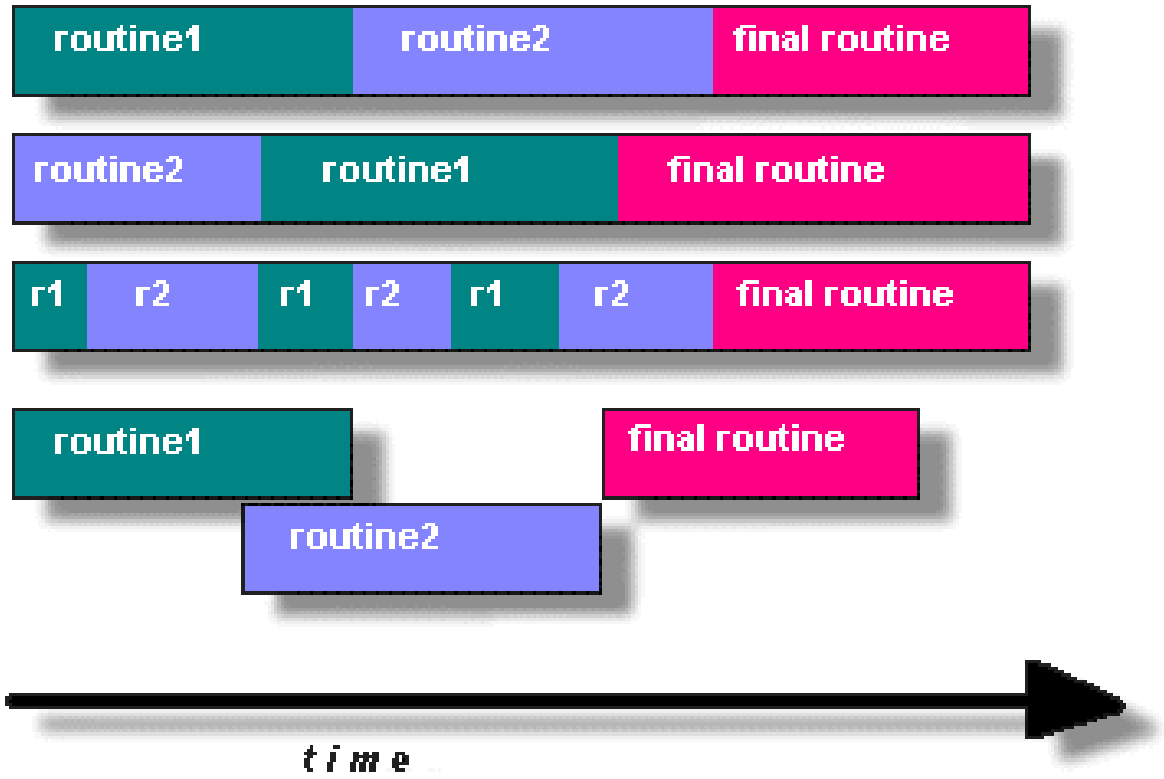
Tworzenie wątków i procesów

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.10
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1.00	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.54	20.78	1.64	0.67	0.90
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	1.07	22.22	2.03	1.26	0.67

50,000 tworzonych procesów/wątków, czas w sekundach, polecenie **`time`**

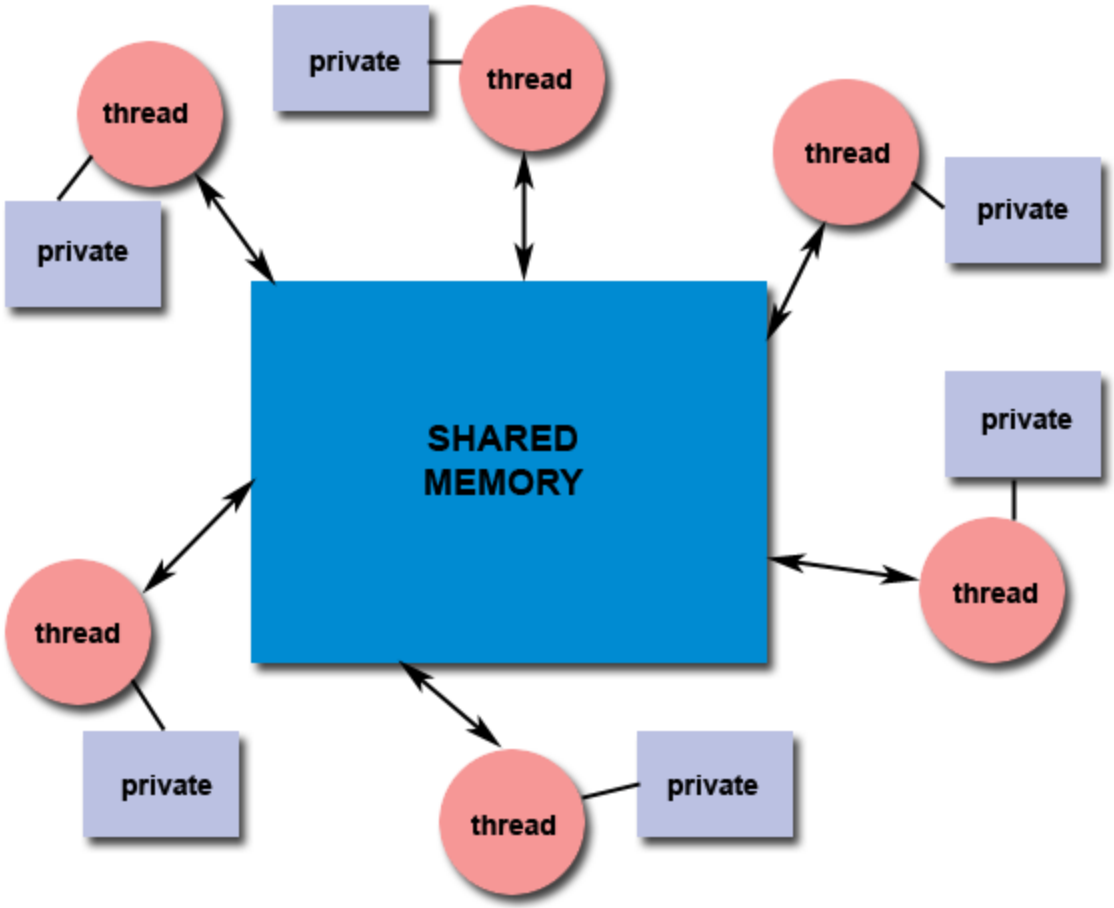
Wykonanie równoległe

Stosowanie wątków **ma sens** wtedy, gdy zadania wykonywane w wątkach mogą być **wykonywane** w dużym stopniu **niezależnie**.



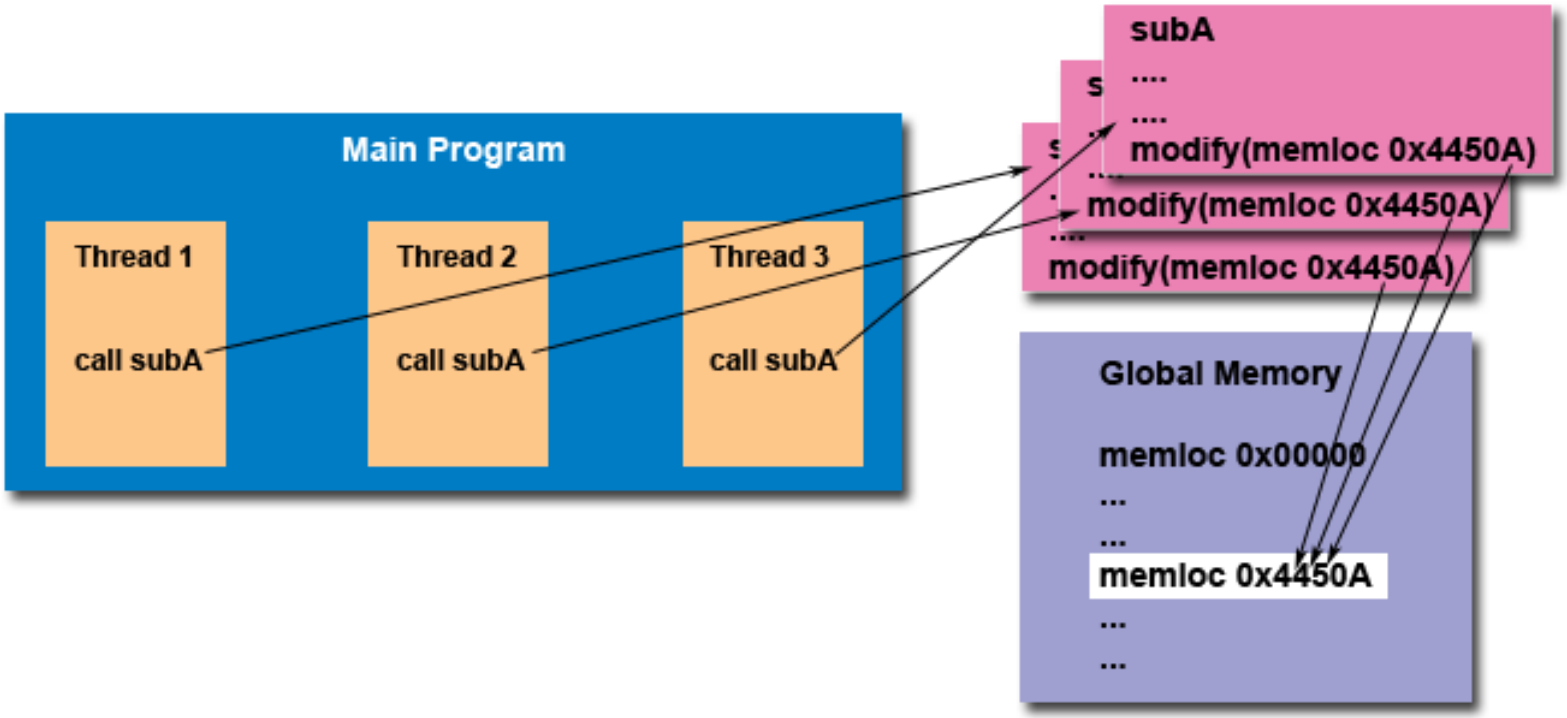
Model współdzielenia pamięci

- Wszystkie wątki w danym procesie mają dostęp do **tej samej globalnej pamięci procesu**.
- Wątki mogą mieć również swoje **prywatne dane**.
- Programista jest odpowiedzialny za ochronę i synchronizację dostępu do danych globalnych.



Bezpieczna wielowątkowość

Unikanie sytuacji **wyścigu** i jednoczesnej **modyfikacji** ogólnie dostępnych danych.



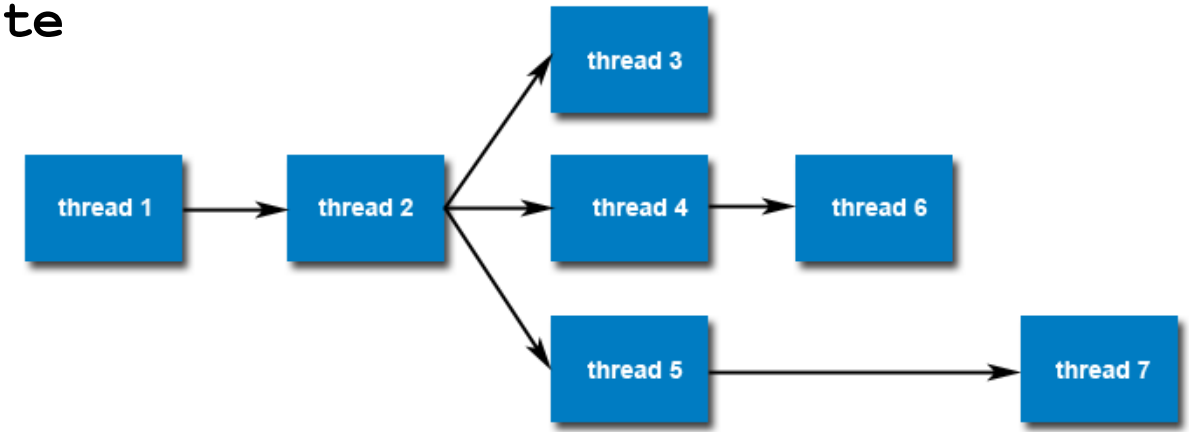
API Pthread

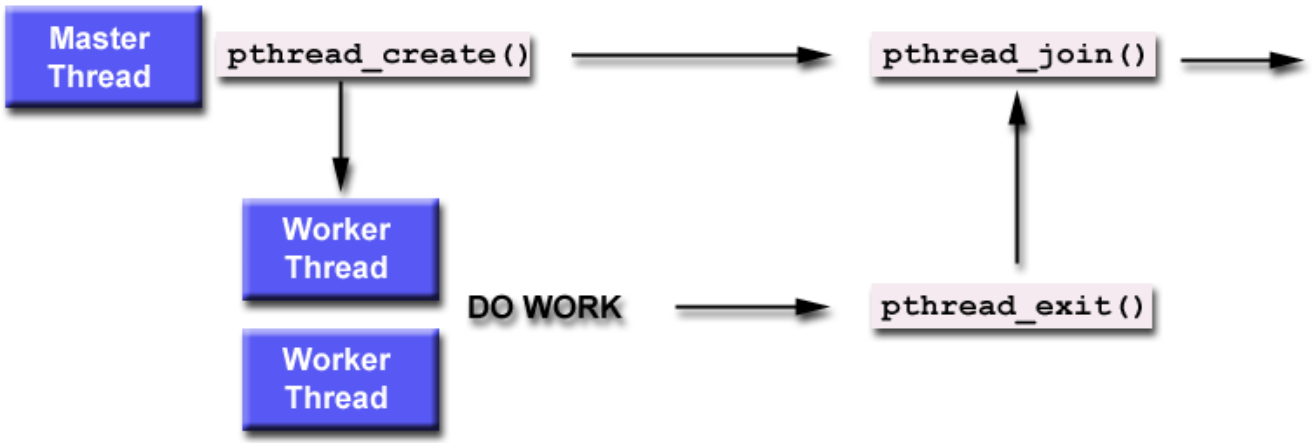
prefiks	Grupa funkcji
`pthread_`	**Wątki**
`pthread_attr_`	**Obiekty atrybutów wątków**
`pthread_mutex_`	**Muteksy**
`pthread_mutexattr_`	Obiekty atrybutów muteksów
`pthread_cond_`	Zmienne warunkowe
`pthread_condattr_`	Obiekty atrybutów warunków
`pthread_key_`	Klucze

Zarządzanie wątkami

`pthread_create`

`pthread_exit`





`pthread_join`

Funkcja `pthread_create` (1/2)

- `int pthread_create( pthread_t *thread, pthread_attr_t attr, void * (*start_routine)(void *), void *arg);`

- F-cja tworzy nowy wątek, który wykonuje się współbieżnie z wątkiem wywołującym. Nowy watek zaczyna wykonywać funkcje ***`start_routine`*** podając jej ***`arg`*** jako argument.
- Nowy watek kończy się przez wywołanie procedury **`pthread_exit`** lub przez powrót z ***`start_routine`***.
- Argument ***`attr`*** określa atrybuty nowego wątku, do których ustalenia służy funkcja **`pthread_attr_init`**. Jeśli jako ***`attr`*** przekażemy ***`NULL`***, to użyte będą atrybuty domyślne (np. możliwość dołączenia).
- Po bezbłędnym wykonaniu f-cja umieszcza identyfikator nowoutworzonego wątku w miejscu wskazywanym przez argument ***`thread`*** i zwraca **0**.

Funkcja `pthread_create` (2/2)

test-pthread-1.c

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

void* print_xs ( void* unused)
{
  while ( 1)
    fputc ( 'x', stderr);
  return NULL;
}

int main ()
{
  pthread_t thread_id;
  pthread_create ( &thread_id, NULL, print_xs, NULL);
  while ( 1)
    fputc ( 'o', stderr);
  return 0;
}
```

Funkcja `pthread_self`

- `pthread_t pthread_self( void);`

  f-cja zwraca **identyfikator wątku**, który wywołał funkcje.

- `int pthread_equal( pthread_t t1, pthread_t t2);`

  Funkcja określa, czy oba identyfikatory odnoszą się do tego samego wątku. Zwracana jest wartość niezerowa jeśli *t1* i *t2* odnoszą się do tego samego wątku lub 0 w przeciwnym wypadku.

Funkcja `pthread_exit`

- **`void pthread_exit( void *retval);`**

  Funkcja kończy działanie wołającego wątku. Wywoływane są po kolei wszystkie funkcje czyszczące określone przez **`pthread_cleanup_push`**. Dopiero po tym wszystkim działanie wątku jest wstrzymywane. Argument ***`retval`*** określa kod zakończenia wątku, który może być odczytany przez inny watek za pomocą funkcji **`pthread_join`**.

Funkcja **pthread_join** (1/3)

- **int pthread_join(pthread_t *th*, void ***thread_return*);**

- F-cja zawiesza działanie wołającego wątku aż do momentu, gdy watek identyfikowany przez ***th*** nie zakończy działania. Jeśli argument ***thread_return*** jest różny od ***NULL*** to kod zakończenia wątku ***th*** zostanie wstawiony w miejsce wskazywane przez ***thread_return***.
- Watek, do którego dołączamy musi być w stanie umożliwiającym dołączanie (nie może być odłączony przez wywołanie **pthread_detach** lub określenie atrybutu **PTHREAD_CREATE_DETACHED** przy jego tworzeniu przez **pthread_create**).

Funkcja **`pthread_join`** (2/3)

- Zasoby wątku (deskryptor wątku i stos) działającego w stanie umożliwiającym dołączenie **nie są** zwalniane dopóki inny watek nie wykona na nim **`pthread_join`**. Dlatego **`pthread_join`** powinien być wykonany dla każdego **nieodłączonego wątku**.
- Co najwyżej **jeden watek** może czekać na zakończenie danego wątku. Wywołanie **`pthread_join`** w momencie, gdy jakiś inny watek już oczekuje na jego zakończenie spowoduje powrót z f-cji z **błędem**.
- Oczekiwanie przez wywołanie funkcji **`pthread_join`** jest tzw. punktem anulowania (jeśli watek zostanie odwołany w czasie oczekiwania, to działanie wątku zostanie zakończone natychmiast bez czekania na synchronizacje z wątkiem ***`th`***).
- W przypadku sukcesu funkcja **`pthread_join`** zwraca **0** i kod zakończenia wątku ***`th`*** jest umieszczany w miejscu wskazywanym przez ***`thread_return`***.

Funkcja `pthread_join` (3/3)

test-pthread-2.c

```
...

int main ()
{
pthread_t thread1_id;
struct char_print_parms thread1_args;

thread1_args.character = 'x';
thread1_args.count = 30000;
pthread_create (&thread1_id, NULL, char_print, &thread1_args);

pthread_join (thread1_id, NULL);

return 0;
}
```

Funkcja `pthread_detach`

- `int pthread_detach( pthread_t thread);`

- Funkcja odłącza wskazany przez ***`thread`*** wątek od jego wątku macierzystego.
- Wątek po odłączeniu działa w trybie, który nie pozwala go synchronizować f-cją **`pthread_join`** oraz sam zwalania wszystkie zasoby po zakończeniu działania.

Funkcja `pthread_cancel`

- `int pthread_cancel( pthread_t thread);`

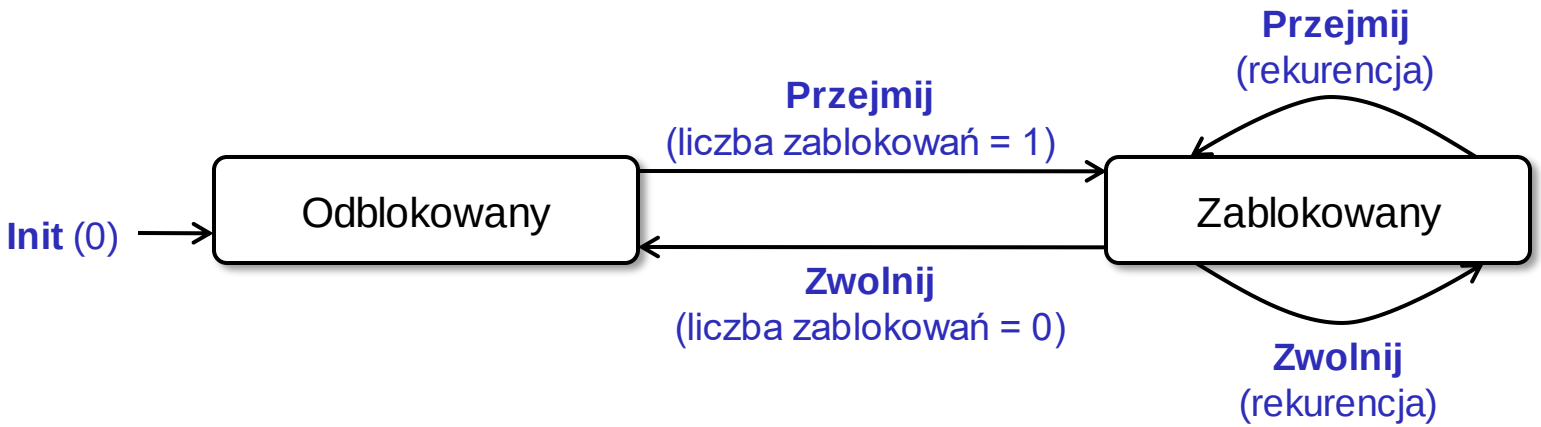
- Funkcja przerywa działanie wskazanego wątku. Anulowany wątek zwraca do `pthread_join` wartość PTHREAD_CANCELED
- Wątek może kontrolować za pomocą f-cji `pthread_setcanceltype` kiedy i czy zostanie anulowany (**anulowanie asynchroniczne** – w każdej chwili; **anulowanie synchroniczne** – żądanie anulowania są kolejkowane, aż do osiągnięcia punktu anulowanie; **brak możliwości anulowania**)
- Bezargumentowe wywołanie `pthread_cancel` tworzy punkt anulowania.

Funkcje `pthread_attr_...`

- Zestaw funkcji do zarządzania obiektami atrybutów wątku.
- F-cje podstawowe: **`pthread_attr_init`**, **`pthread_attr_destroy`**;
- Zarządzanie trybem działania: **`pthread_attr_setdetachstate`**, **`thread_attr_getdetachstate`**;
- Zarządzanie stosem: **`pthread_attr_getstackaddr`**, **`pthread_attr_getstacksize`**, **`pthread_attr_setstackaddr`**, **`pthread_attr_setstacksize`**;
- Zarządzanie szeregowaniem: **`pthread_attr_getschedparam`**, **`pthread_attr_setschedparam`**, **`pthread_attr_getschedpolicy`**, **`pthread_attr_setschedpolicy`**, **`pthread_attr_setinheritsched`**, **`pthread_attr_getinheritsched`**, **`pthread_attr_setscope`**, **`pthread_attr_getscope`**.

Muteksy

Muteksy to rodzaj semaforów binarnych. Muteks może być zablokowany (ma wartość 1) lub odblokowany (ma wartość 0). Jeśli jakieś zadanie zablokuje muteks (nada mu wartość 1), to tylko ono może ten muteks odblokować (nadać mu wartość 0). Z założenia muteksy mogą być rekurencyjne (wielokrotne blokowanie przez jedno zadanie).



Muteksy w **pthread** domyślnie **nie obsługują** rekurencji, dwukrotne zablokowanie muteksu doprowadza do blokady wątku.

Funkcja `pthread_mutex_init`

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

- Funkcja inicjalizuje obiekt ***mutex*** zgodnie z atrybutami przekazanymi przez ***mutexattr***. Jeśli ***mutexattr*** jest ***NULL*** używane są wartości domyślne. Funkcja zawsze zwraca 0.
- Do ustawienia atrybutów muteksu służy zestaw f-cji `pthread_mutexattr_...`

  Mutksy mogą być również zainicjalizowane za pomocą predefiniowanych wartości:
  - `pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;`
  - `pthread_mutex_t recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;`

Funkcja `pthread_mutex_destroy`

- `int pthread_mutex_destroy(pthread_mutex_t *mutex);`

- Funkcja niszczy obiekt ***mutex*** i zwalnia zasoby z nim związane (funkcja zwraca 0).
- Zwalniany ***mutex*** nie może być zablokowany, w przeciwnym przypadku zwracany jest błąd EBUSY.

Funkcja `pthread_mutex_lock`

- `int pthread_mutex_lock(pthread_mutex_t *mutex);`

- Funkcja zajmuje dany ***mutex***. Jeśli jest on wolny zostaje zajęty i przypisany wątkowi wołającemu i **`pthread_mutex_lock`** kończy działanie natychmiast. Jeśli ***mutex*** jest zajęty przez jakiś inny watek **`pthread_mutex_lock`** zawiesza działanie wątku aż do momentu, kiedy ***mutex*** zostanie zwolniony.
- Jeśli ***mutex*** jest już zajęty przez wątek wołający to zachowanie funkcji zależy od rodzaju mutexu. Jeśli ***mutex*** dopuszcza rekurencje to funkcja kończy działanie poprawnie zapisując sobie ilość wywołań funkcji (głębokość rekurencji - potem trzeba wywołać tyle samo razy **`pthread_mutex_unlock`** żeby zwolnić ***mutex***), jeśli zaś nie dopuszcza to doprowadza do blokady wątku.

Funkcja **pthread_mutex_trylock**

- **int pthread_mutex_trylock(pthread_mutex_t *mutex);**

- Funkcja **pthread_mutex_trylock** zachowuje się podobnie jak **pthread_mutex_lock**, jednak nie jest blokującą (zwraca **EBUSY** w przypadku gdy ***mutex*** jest zajęty).

Funkcja `pthread_mutex_unlock`

- `int pthread_mutex_unlock( pthread_mutex_t *mutex);`

- Funkcja **`pthread_mutex_unlock`** zwalnia dany ***`mutex`***. ***`mutex`*** musi być wcześniej zajęty przez wołający proces. Jeśli ***`mutex`*** jest nierekurencyjny to zawsze wraca do stanu zwolnionego, jeśli jest rekurencyjny, to zmniejszana jest głębokość rekurencji. Jedynie gdy głębokość jest zero ***`mutex`*** zostaje faktycznie zwolniony.

Sekcja krytyczna bez muteksu (1/2)

test-mutex-1.c [1/2]

```
...
int globalVar;

void *thFunction( void *arg)
{
  int i,j;
  for ( i=0; i<100000; i++ ){
    j=globalVar;
    j=j+1;
    globalVar=j;
  }
  return NULL;
}
...
```

Sekcja krytyczna bez muteksu (2/2)

test-mutex-1.c [2/2]

```
...

int main( void) {

  pthread_t th;
  int i;

 globalVar = 0;

  pthread_create( &th, NULL, thFunction, NULL);

  for ( i=0; i<1000000; i++)
     globalVar = globalVar + 1;

  pthread_join ( th, NULL );
  printf( "\nWartość mojej zmiennej globalnej to %d\n",globalVar);
  return 0;
}
```

Sekcja krytyczna z muteksem (1/2)

test-mutex-2.c [1/2]

```
...
int globalVar;
pthread_mutex_t fastMutex=PTHREAD_MUTEX_INITIALIZER;

void *thFunction( void *arg) {
  int i,j;
  for ( i=0; i<100000; i++ ) {
    pthread_mutex_lock( &fastMutex);
    j=globalVar;
    j=j+1;
    globalVar=j;
    pthread_mutex_unlock( &fastMutex);
  }
  return NULL;
}
```

Sekcja krytyczna z muteksem (2/2)

test-mutex-2.c [2/2]

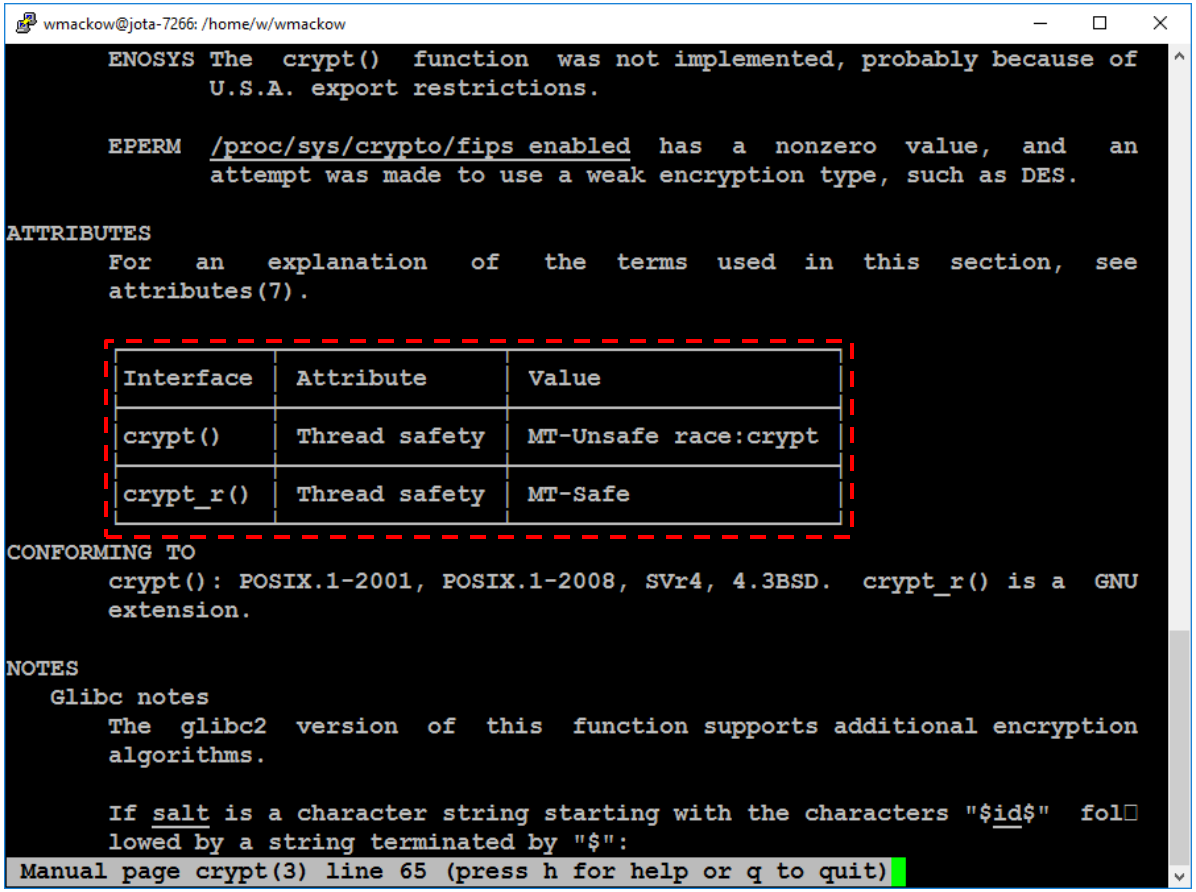
```
...

int main( void) {

  pthread_t th;
  int i;

  pthread_create( &th, NULL, thFunction, NULL);

  for ( i=0; i<1000000; i++){
    pthread_mutex_lock( &fastMutex);
    globalVar=globalVar+1;
    pthread_mutex_unlock( &fastMutex);
  }
  pthread_join ( th, NULL );
  printf("\nWartość mojej zmiennej globalnej to %d\n",globalVar);
  return 0;
}
```

Wielowątkowość a funkcje biblioteczne

Nie wszystkie funkcje biblioteczne mogą być bezpiecznie używane w aplikacjach wielowątkowych. W niektórych sytuacjach ich użycie może doprowadzić np. do sytuacji wyścigu i wzajemnego nadpisywania sobie danych. Często udostępniane są wersje funkcji bezpieczne wątkowo.

```
char *crypt(
  const char *key,
  const char *salt);
```

```
char *crypt_r(
  const char *key,
  const char *salt,
  struct crypt_data *data);
```



Mechanizmy IPC

- Wątki
- **Mechanizmy IPC**
  - Kolejki komunikatów
  - Semafory
  - Pamięć współdzielona
- Gniazda

Identyfikatory i klucze

System V wprowadził trzy rodzaje mechanizmów komunikacji międzyprocesowej: **kolejki komunikatów**, **zestawy semaforów** i **pamięć współdzieloną**.

- Każdy z zasobów IPC ma określony **klucz** oraz **identyfikator** (32-bitowe nieujemne liczby całkowite).
- **Klucz** stanowi pewnego rodzaju "nazwę" zasobu, na podstawie której możemy uzyskać jego identyfikator. Nazwa ta musi być unikalna w systemie (w obrębie danego typu zasobu), oraz muszą ją znać wszystkie procesy korzystające z danego zasobu.
- **Identyfikator** jest przydzielany przez system podczas otwierania zasobu (odpowiednik "deskryptora pliku") - musimy go przekazać do każdej funkcji systemowej operującej na danym zasobie.

- Dostęp do zasobów IPC kontrolowany jest analogicznie jak dla plików - każdy zasób jest własnością pewnego **użytkownika i grupy**, oraz ma określone prawa **odczytu i zapisu** dla użytkownika, grupy i innych (bit określający prawa do wykonywania jest ignorowany).

Dostępne funkcje

	Kolejki komunikatów	**Semafory**	**Pamięć współdzielona**
Plik nagłówkowy	**<sys/msg.h>**	**<sys/sem.h>**	**<sys/shm.h>**
Funkcja systemowa tworzenia lub otwierania	**msgget**	**semget**	**shmget**
Funkcja systemowa operacji sterujących	**msgctl**	**semctl**	**shmctl**
Funkcje operacji na obiektach IPC	**msgsnd** **msgrcv**	**semop**	**shmat** **shmdt**

Polecenia konsoli związane z IPC: **ipcs, ipcrm**

Mechanizmy IPC – kolejki komunikatów

- Wątki
- **Mechanizmy IPC**
  - **Kolejki komunikatów**
  - Pamięć współdzielona
  - Semafory
- Gniazda

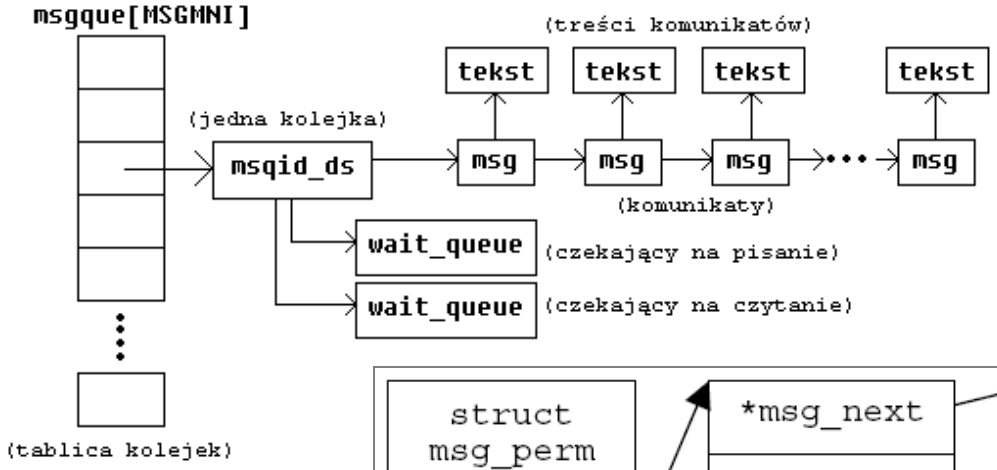
Kolejki komunikatów (1/2)

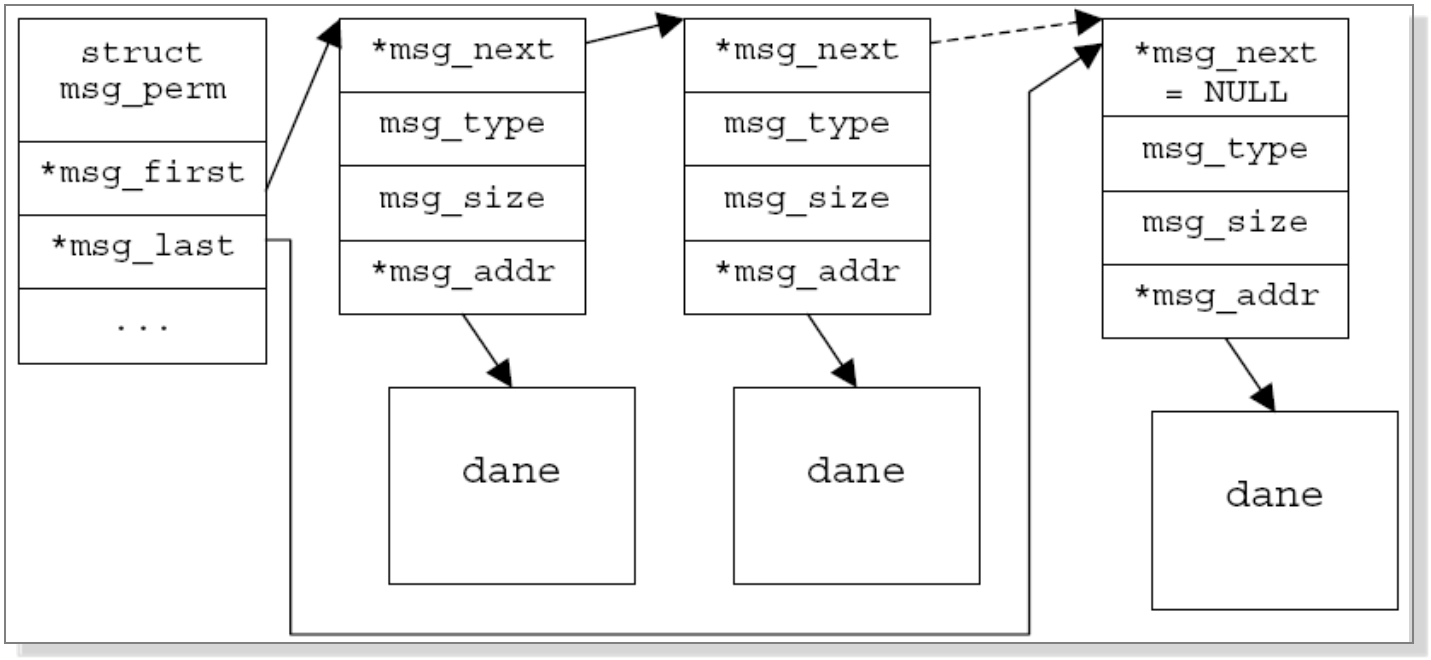
- Kolejki komunikatów umożliwiają przesyłanie pakietów danych, nazywanych komunikatami, pomiędzy różnymi procesami. Komunikat jest zbudowany jako struktura:

```
struct msgbuf{
  long mtype; //typ komunikatu (>0)
  char mtext[1]; //treść komunikatu
}
```

- Komunikat ma określony **typ i długość**. Typ komunikatu, pozwalający określić rodzaj komunikatu, nadaje proces inicjujący komunikat.
- Komunikaty są umieszczane w kolejce w kolejności ich wysyłania.
- Nadawca może wysyłać komunikaty nawet wówczas, gdy żaden z potencjalnych odbiorców nie jest gotów do ich odbioru (komunikaty są buforowane).
- Przy odbiorze komunikatu odbiorca może oczekiwać na pierwszy przybyły komunikat lub na pierwszy komunikat określonego typu.
- Komunikaty w kolejce są przechowywane nawet po zakończeniu procesu nadawcy tak długo, aż nie zostaną odebrane lub kolejka nie zostanie zlikwidowana.

Kolejki komunikatów (2/2)





Funkcja **msgget**

- **int msgget(key_t *key*, int *msgflg*);**

- F-cja zwraca identyfikator kolejki związanej z kluczem ***key***. Jeżeli jako klucz podamy **IPC_PRIVATE** albo nie istnieje kolejka o podanym kluczu (a we flagach ustawimy **IPC_CREAT**) to zostanie stworzona nowa kolejka.
- Znaczniki **IPC_CREAT** i **IPC_EXCL** przekazywane parametrem ***semflg*** pełnią tę samą rolę w obsłudze kolejek komunikatów, co **O_CREAT** i **O_EXCL** w parametrze ***mode*** funkcji systemowej **open**.

Funkcja `msgctl` (1/2)

- `int msgctl( int msqid, int cmd, struct msqid_ds *buf);`

- F-cja wykonuje operację określoną przez parametr ***cmd*** na kolejce komunikatów o identyfikatorze ***msqid***.
- Możliwe wartości ***cmd***:
  - **IPC_STAT** - kopiowanie informacji ze struktury kontrolnej kolejki komunikatów ***msqid*** pod adres ***buf***.
  - **IPC_SET** - zapis wartości niektórych pól struktury **`msqid_ds`** wskazywanej przez parametr ***buf*** do struktury kontrolnej kolejki komunikatów ***msgid***.
  - **IPC_RMID** - usunięcie kolejki komunikatów i skojarzonej z nią struktury danych.

Funkcja `msgctl` (2/2)

Zawartość struktury **`msqid_ds`**:

```
struct msqid_ds {
  struct ipc_perm msg_perm;   /* Własności i uprawnienia */
  time_t          msg_stime;  /* Czas ostatniego msgsnd() */
  time_t          msg_rtime;  /* Czas ostatniego msgrcv() */
  time_t          msg_ctime;  /* Czas ostatniej zmiany */
  unsigned long   msg_cbytes; /* Bieżąca liczba bajtów w
                                 kolejce*/
  msgqnum_t       msg_qnum;   /* Bieżąca liczba komunikatów w
                                 kolejce */
  msglen_t        msg_qbytes; /* Maksymalna liczba dostępnych
                                 bajtów w kolejce */
  pid_t           msg_lspid;  /* PID ostatniego msgsnd() */
  pid_t           msg_lrpid;  /* PID ostatniego msgrcv() */

};
```

Funkcja **msgsnd**

- `int msgsnd( int msqid, const void *msgp, size_t msgsz, int msgflg);`

- F-cja służy do wysyłania komunikatu ***msgp*** o długości ***msgsz*** do kolejki ***msqid***. Komunikat musi rozpoczynać się polem typu **long int** określającym typ komunikatu, po którym umieszczone zostaną pozostałe bajty wiadomości. Przykładowo może być to struktura:

```
struct mymsg {
  long int mtype; /* message type */
  char mtext[1]; /* message text */
}
```

- **W celu wysłania lub odebrania komunikatu, proces powinien zaalokować strukturę danych!**

Funkcja **msgrcv** (1/2)

- **ssize_t msgrcv(int *msqid*, void **msgp*, size_t *msgsz*, long int msgtyp, int *msgflg*);**

- F-cja odczyta komunikat z kolejki wskazanej przez ***msqid*** do struktury ***msgbuf*** wskazywanej przez ***msgp*** usuwając odczytany komunikat z kolejki.
- Parametr ***msgsz*** określa maksymalny rozmiar (w bajtach) pola ***mtext*** struktury wskazywanej przez parametr ***msgp***. Dłuższe komunikaty są obcinane (tracone) jeżeli flaga ustawiona jest na **MSG_NOERROR**, w przeciwnym przypadku komunikat nie jest usuwany z kolejki.

Funkcja **msgrcv** (2/2)

Parametr ***msgtyp*** określa rodzaj komunikatu w następujący sposób:

- Jeśli jest **równy 0**, to czytany jest pierwszy dostępny komunikat w kolejce (czyli najdawniej wysłany).
- Jeśli ma wartość **większą niż 0**, to z kolejki odczytywany jest pierwszy komunikat wskazanego typu, chyba że w parametrze ***msgflg*** zostanie ustawiony znacznik **MSG_EXCEPT**, kiedy to z kolejki zostanie odczytany pierwszy komunikat o typie innym niż podany w ***msgtyp***.
- Jeśli msgtyp ma wartość **mniejszą niż 0**, to z kolejki zostanie odczytany pierwszy komunikat o najniższym numerze typu, o ile jest on mniejszy lub równy wartości bezwzględnej ***msgtyp*** .

Użycie kolejki komunikatów (1/2)

test-msg.c [1/2]

```
...
int main()
{
  key_t key = ftok( ".", 'z');
  int id = msgget( key, IPC_CREAT | 0600);
  struct { long type; char a[10]; } data;
  int r;
  data.type = 2; strcpy( data.a, "hello");
  msgsnd( id, ( struct msgbuf*)&data,
          sizeof( data) - 4, 0);
  data.type = 1; strcpy( data.a, "world");
  msgsnd( id, ( struct msgbuf*)&data,
          sizeof( data) - 4, 0);
...
```

Użycie kolejki komunikatów (2/2)

test-msg.c [2/2]

```
...
for(;;){
      r = msgrcv( id, ( struct msgbuf*)&data,
                      sizeof(data) - 4,
                      -2,
                      IPC_NOWAIT);
      if(r<0)
         break;
      puts(data.a);
   }

   msgctl( id, IPC_RMID, NULL);
}

...
```

Mechanizmy IPC – pamięć współdzielona

- Wątki
- **Mechanizmy IPC**
  - Kolejki komunikatów
  - **Pamięć współdzielona**
  - Semafory
- Gniazda

Pamięć współdzielona

- Pamięć współdzielona jest specjalnie utworzonym **segmentem wirtualnej przestrzeni adresowej**, do którego dostęp może mieć wiele procesów. Jest to najszybszy sposób komunikacji pomiędzy procesami.
- Podstawowy schemat korzystania z pamięci współdzielonej wygląda następująco: jeden z procesów tworzy segment pamięci współdzielonej, dowiązuje go powodując jego **odwzorowanie w bieżący obszar danych procesu**, opcjonalnie zapisuje w stworzonym segmencie dane.
- Następnie, w zależności od praw dostępu inne procesy mogą odczytywać i/lub zapisywać wartości w pamięci współdzielonej.

Funkcja **shmget**

- `int shmget( key_t key, size_t size, int shmflg);`

- Funkcja **shmget** służy do tworzenia segmentu pamięci współdzielonej i do uzyskiwania dostępu do już istniejących segmentów pamięci.
- W drugim przypadku wartością parametru ***size*** może być 0, ponieważ rozmiar segmentu został już wcześniej zadeklarowany przez proces, który go utworzył.

Funkcja **shmctl**

- **int shmctl(int *shmid*, int *cmd*, struct shmid_ds *buf);**

- Funkcja odpowiada funkcji **msgctl**. Przy próbie usunięcia segmentu odwzorowanego na przestrzeń adresową procesu system odpowiada komunikatem o błędzie.
- Możliwe wartości ***cmd***:
  - **IPC_STAT** - pozwala uzyskać informację o stanie pamięci współdzielonej
  - **IPC_SET** - pozwala zmienić parametry segmentu pamięci
  - **IPC_RMID** - pozwala usunąć segment pamięci współdzielonej z systemu

Funkcja **shmat**

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

- F-cja **dołącza** segment pamięci wspólnej o deskryptorze ***shmid*** **do przestrzeni adresowej procesu**, który ją wywołał. Adres, pod którym segment ma być widoczny jest przekazywany parametrem ***shmaddr***.
- Jeśli ***shmaddr*** jest równy **NULL**, wówczas system sam wybierze odpowiedni (nieużywany) adres, pod którym segment będzie widoczny.
- W wyniku poprawnego wykonania f-cja zwraca adres początku obszaru odwzorowania segmentu.

Funkcja **shmdt**

- **int shmdt(const void *_shmaddr_);**

- Funkcja **shmdt** wyłącza segment pamięci wspólnej odwzorowany pod adresem podanym w ***shmaddr*** z przestrzeni adresowej procesu wywołującego tę funkcję.
- Przekazany funkcji w parametrze ***shmaddr*** adres musi być równy adresowi zwróconemu wcześniej przez wywołanie **shmat**.

Użycie pamięci współdzielonej

test-shm.c

```
...
int main()
{
   key_t key = ftok( ".", 'z');
   int id = shmget( key, 1024, IPC_CREAT | 0600);
   char * base = shmat( id, NULL, 0);
   if( !base) return;

   printf( "Zawartość obszaru: %s\n", base);
   sprintf( base, "tu byłem (proces %d)", getpid());
   printf( "Zmieniłem na: %s\n", base);

   shmdt( base);
}
...
```

Mechanizmy IPC - semafory

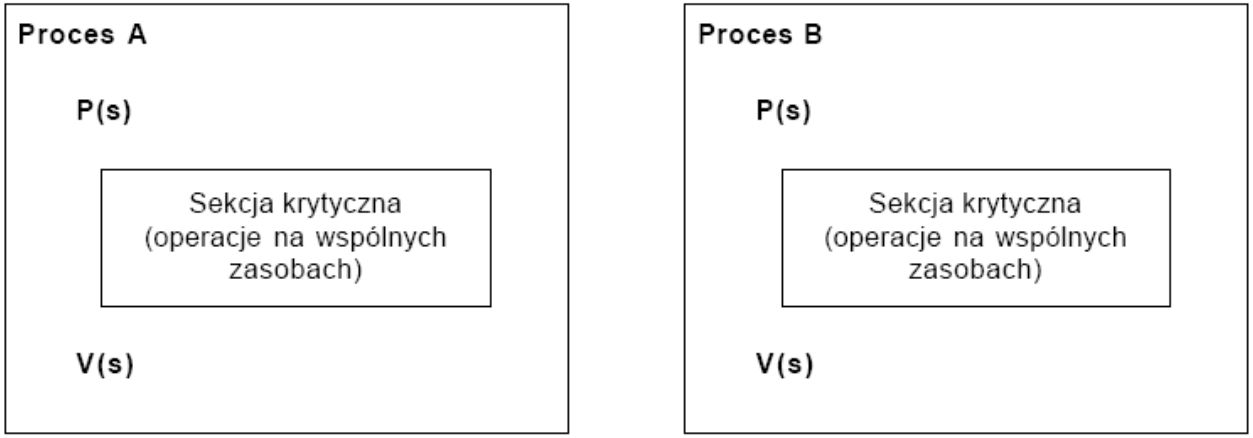
- Wątki
- **Mechanizmy IPC**
  - Kolejki komunikatów
  - Pamięć współdzielona
  - **Semafory**
- Gniazda

Semafory (1/2)

- Semafory są strukturami danych wspólnie użytkowanymi przez kilka procesów. Najczęściej znajdują one zastosowanie w **synchronizowaniu działania kilku procesów** korzystających ze wspólnego zasobu, przez co zapobiegają niedozwolonemu wykonaniu operacji na określonych danych jednocześnie przez większą liczbę procesów.

- Podstawowym rodzajem semafora jest semafor binarny, przyjmujący dwa stany:
  - opuszczony (zamknięty) - wówczas proces, który napotyka semafor musi zawiesić swoje działanie do momentu podniesienia semafora (opuść - **P**),
  - podniesiony (otwarty) - proces może kontynuować działanie (podnieś - **V**)

Semafory (2/2)



W **IPC** zdefiniowano jedynie strukturę semafora i zbiór funkcji do operacji na nich, bez określania jakie wartości semafora odpowiadają jakim jego stanom. Możliwe są dwie konwencje:

Podejście 1	**Podejście 2**
opuszczony ma **wartość =0**	opuszczony ma **wartość >0**
podniesiony ma **wartość >0**	podniesiony ma **wartość = 0**

Funkcja **semget**

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

- F-cja zwraca identyfikator zestawu semaforów, skojarzonego z parametrem ***key***. Jeśli ***key*** ma wartość **IPC_PRIVATE** lub, gdy z wartością ***key*** nie jest skojarzony żaden istniejący zestaw semaforów, a w parametrze ***semflg*** został przekazany znacznik **IPC_CREAT** to tworzony jest nowy zestaw złożony z ***nsems*** semaforów.
- Znaczniki **IPC_CREAT** i **IPC_EXCL** przekazywane parametrem ***semflg*** pełnią tę samą rolę w obsłudze semaforów, co **O_CREAT** i **O_EXCL** w parametrze ***mode*** funkcji systemowej **open.**

Funkcja **semctl** (1/2)

- **int semctl(int *semid*, int *semnum*, int *cmd*, union semun *arg);**

- F-cja wykonuje operację sterującą określoną przez ***cmd*** na zestawie semaforów ***semid*** lub na ***semnum***-tym semaforze tego zestawu (numeracja od 0).
- W zależności o wydanego polecenia ***cmd*** argument ***arg*** jest różnie interpretowany:

```
union semnum{
  int val;
  struct semid_ds *buf;
  unsigned short *array;
};
```

Funkcja **`semctl`** (2/2)

Możliwe wartości ***`cmd`***:

- **IPC_STAT** Kopiowanie informacji ze struktury kontrolnej zestawu semaforów do struktury wskazywanej przez ***`arg.buf`***
- **IPC_SET** - modyfikuje wybrane ustawienia zestawu semaforów
- **IPC_RMID** - usuwa zestaw semaforów z systemu
- **GETVAL** - zwraca wartość semafora (***`semval`***), wskazywanego jako ***`semnum`***
- **SETVAL** - nadaje wartość semaforowi o numerze ***`semnum`***
- **GETPID** - zwraca wartość ***`sempid`***
- **GETNCNT** - pobranie liczby procesów oczekujących na to, aż semafor wskazywany przez ***`semnum`*** zwiększy swoją wartość
- **GETZCNT** - pobranie liczby procesów oczekujących na to, aż semafor wskazywany przez ***`semnum`*** osiągnie wartość zero
- **GETALL** - pobranie bieżących parametrów całego zestawu semaforów i zapisanie uzyskanych wartości w tablicy wskazanej czwartym argumentem funkcji
- **SETALL** - zainicjowanie wszystkich semaforów z zestawu wartościami przekazanymi w tablicy określonej przez wskaźnik przekazany czwartym argumentem funkcji

Funkcja **semop** (1/3)

- `int semop( int semid, struct sembuf *sops, size_t nsops);`

- Wykonanie operacji semaforowej. Może być ona wykonywana jednocześnie na kilku semaforach w tym samym zestawie identyfikowanym przez ***semid***.
- ***sops*** to wskaźnik na adres tablicy operacji semaforowych, a ***nsops*** to liczba elementów tej tablicy.
- Każdy element tablicy opisuje jedną operację semaforową i ma następującą strukturę:

```
struct sembuf {
  short sem_num; /* numer semafora - od 0 */
  short sem_op;  /* operacja semaforowa */
  short sem_flg; /* flagi operacji */
};
```

Funkcja **semop** (2/3)

- Pole ***sem_op*** zawiera wartość, która zostanie dodana do zmiennej semaforowej pod warunkiem, że zmienna semaforowa nie osiągnie w wyniku tej operacji wartości **mniejszej od 0**. Dodatnia liczba całkowita oznacza zwiększenie wartości semafora (co z reguły oznacza zwolnienie zasobu), ujemna wartość ***sem_op*** oznacza zmniejszenie wartości semafora (próbę pozyskania zasobu).

- Funkcja **semop** podejmuje próbę wykonania wszystkich operacji wskazywanych przez ***sops***. Gdy chociaż jedna z operacji nie będzie możliwa do wykonania nastąpi blokada procesu lub błąd wykonania funkcji **semop**, zależnie od ustawienia flagi **IPC_NOWAIT** i żadna z operacji semaforowych zdefiniowanych w tablicy ***sops*** nie zostanie wykonana.

Funkcja **semop** (3/3)

	podejście 1 (0-semafor opuszczony)		**podejście 2 (0-semafor podniesiony)**			
operacja:	- opuść semafor (jeżeli opuszczony to czekaj aż ktoś go podniesie i natychmiast opuść)	- podnieś	- opuść semafor (jeżeli opuszczony to czekaj aż ktoś go podniesie i natychmiast opuść)	- opuść	- czkaj aż ktoś podniesie	- podnieś
`nsops`	`1`	`1`	`2`	`1`	`1`	`1`
`sem_op`	`-1`	`+1`	`0` `+1`	`+1`	`0`	`-1`

Użycie semaforów (1/2)

test-sem.c [1/2]

```
...

struct sembuf sb;

int main()
{
  key_t key = ftok( ".", 'z');
  int id = semget( key, 1, IPC_CREAT | 0600);
  int i = 0;
  int pid;

  struct semid_ds buf;

  semctl( id, 0, SETVAL, 1);
  semctl( id, 0, IPC_STAT, &buf);
  pid = fork();

...
```

Użycie semaforów (2/2)

test-sem.c [2/2]

```
...
    for(i=0;i<3;i++){
      sb.sem_num = 0; sb.sem_op  = -1; sb.sem_flg = 0;
      semop( id, &sb, 1);
      printf( "%d: Korzystam\n", getpid());
      sleep( rand()%3 + 1);
      printf( "%d: Przestałem korzystać\n", getpid());
      sb.sem_op = 1;
      semop( id, &sb, 1);
    }
    if( pid){
      wait( NULL);
      semctl( id, IPC_RMID, 0);
    }
}
```

Semafory POSIX

Nazwane

Semafor mają unikalne w ramach systemu nazwy i mogą być używane przez współpracujące procesy

```
sem_t *sem_open(const char *name,
  int oflag);
sem_t *sem_open(const char *name,
  int oflag, mode_t mode,
  unsigned int value);
```

Nienazwane

Semafory nie mają nazwy, dostęp do nich to dostęp do odpowiednio zainicjowanej zmiennej globalnej. Stąd najprościej używać je w wątkach jednego procesu. Jeżeli mają być użyte w różnych procesach, to procesy te muszą współdzielić pamięć (!).

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

```
int sem_post(sem_t *sem);

int sem_wait(sem_t *sem);
```

```
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

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

Gniazda

- Wątki
- Mechanizmy IPC
- **Gniazda**

Podstawowe funkcje

wspólne

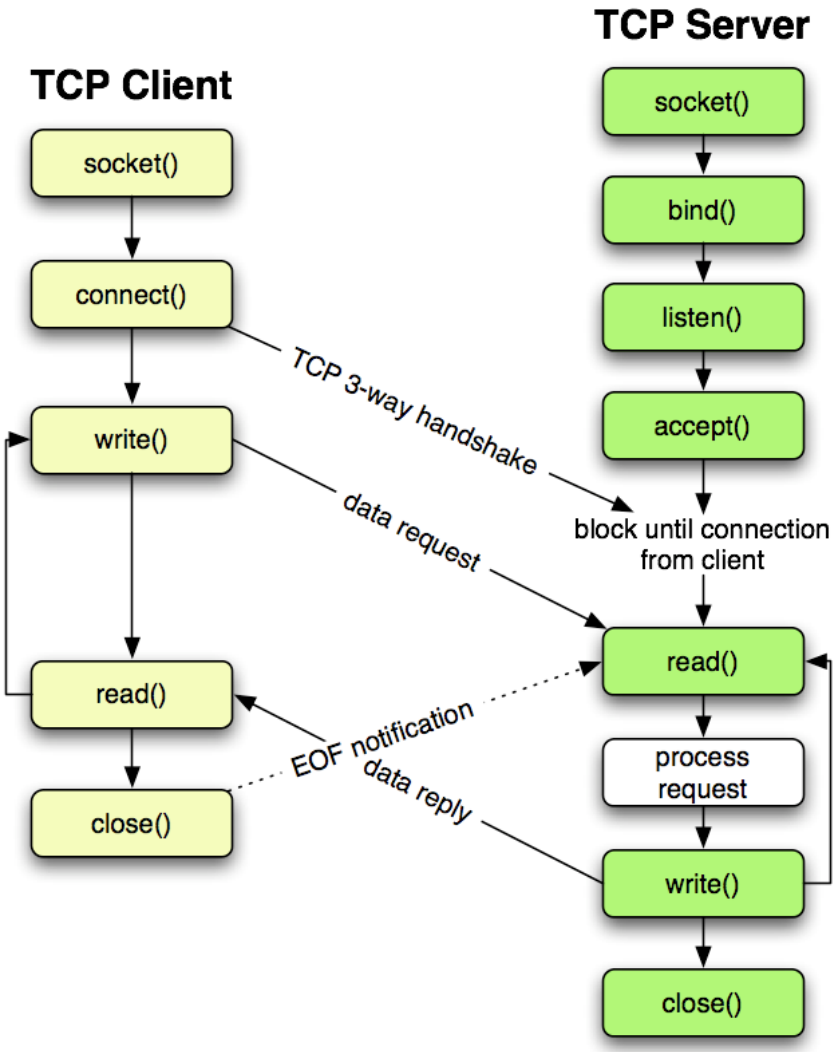
- **`socket(2)`** – tworzenie gniazda
- **`close(2)`** – usuwanie gniazda
- **`read(2)`** – czytanie z gniazda
- **`write(2)`** – pisanie do gniazda
- **`recv(2)`** – czytanie z gniazda (dodatkowe opcje, np. z flagą **MSG_PEEK** nie usuwa wiadomości z kolejki)
- **`send(2)`** – pisanie do gniazda

serwer

- **`bind(2)`** – nadanie gniazdu serwera adresu (lokalnego lub internetowego)
- **`listen(2)`** – konfigurowanie gniazda w celu przyjmowania połączeń
- **`accept(2)`** – akceptowanie połączenia i tworzenie dla niego nowego gniazda

klient

- **`connect(2)`** – tworzenie połączenia między dwoma gniazdami



TCP Socket Programming and Concurrent Servers, http://www.eg.bucknell.edu/~cs363/wordpress/?page_id=88

Funkcja `socket` (1/2)

- `int socket(int domain, int type, int protocol);`

AF_UNIX, **AF_LOCAL** - komunikacja lokalna
AF_INET - protokół IPv4
AF_INET6 - protokół IPv6
…

SOCK_STREAM - komunikacja połączeniowa
SOCK_DGRAM – komunikacja bezpołączeniowa
…

Zazwyczaj dla konkretnej pary **domain** i **type** istnieje tylko jeden protokół, aby go użyć argument **protocol** ustawiamy na 0

Funkcja zwraca deskryptor utworzonego gniazda

Funkcja `socket` (2/2)

test-local.c [1/2]

```
...

int lsfd;
lsfd = socket (AF_UNIX, SOCK_STREAM, 0);

...
```

AF_UNIX

test-inet.c [1/2]

```
...

int isfd;
isfd = socket (AF_INET, SOCK_STREAM, 0);

...
```

AF_INET

Funkcja **bind** (1/3)

- **int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);**

Deskryptor utworzonego gniazda

```
struct sockaddr_un {
 unsigned short sun_family; /* AF_UNIX */
char sun_path[108];
}
```
AF_UNIX

```
struct sockaddr_in {
  short   sin_family;  /* AF_INET */
  u_short   sin_port;   /* 16-bit port number */
  struct in_addr   sin_addr;
  char   sin_zero[8];   /* unused */
};

struct in_addr {
  u_long   s_addr;   /*32-bit net id */
};
```
AF_INET

Jeżeli przyjmujemy połączenia na każdym interfejsie to:
sin_addr.s_addr = INADDR_ANY;

Funkcja `bind` (2/3)

test-local.c [2/2]

```
...
#define SERV_PATH "./serv.path"

struct sockaddr_un serv_addr;

erv_addr.sun_family = AF_UNIX;
strcpy(serv_addr.sun_path, SERV_PATH);

bind(lsfd, (struct sockaddr *)&serv_addr, SUN_LEN(&serv_addr));
...
```

`AF_UNIX`

```
#define SUN_LEN(su) \
  (sizeof(*(su)) - sizeof((su)->sun_path) + strlen((su)->sun_path))
#endif
```

Funkcja **bind** (3/3)

test-inet.c [2/2]

```
...
#define SERV_PORT 5232
#define SERV_HOST_ADDR "82.145.73.240"

struct addrinfo* res;
```

Host name, IP4 or IP6 address — Service name or port — hints — result

```
hostinfo = getaddrinfo(SERV_HOST_ADDR, SERV_PORT, NULL, &res);

bind (isfd, res->ai_addr, res->ai_addrlen));
...
```

AF_INET

Automatycznie ustawiane pole z **AF_INET** albo **AF_INET6**

Tworzenie połączenia

- **`int listen(int sockfd, int backlog);`**

  Określa, że gniazdo będzie oczekiwało na połączenia. Parametr **`backlog`** określa maksymalną długość kolejki przychodzących zgłoszeń. Zwraca **0** w przypadku sukcesu.

- **`int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`**

  Wyciąga pierwsze żądanie połączenia z kolejki oczekujących połączeń, tworzy nowo podłączone gniazdo o tych samych właściwościach co **sockfd** i alokuje nowy deskryptor pliku dla gniazda (nowy deskryptor jest zwracany). Pod adres **addr** jest wpisywany adres łączącej się jednostki (przekazany przez warstwę komunikacyjną). **Jeżeli w kolejce brak połączeń funkcja blokuje proces.**

- **`int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`**

  Inicjalizuje połączenie między gniazdem klienta a wskazanym adresem serwera. Zwraca **0** w przypadku sukcesu.

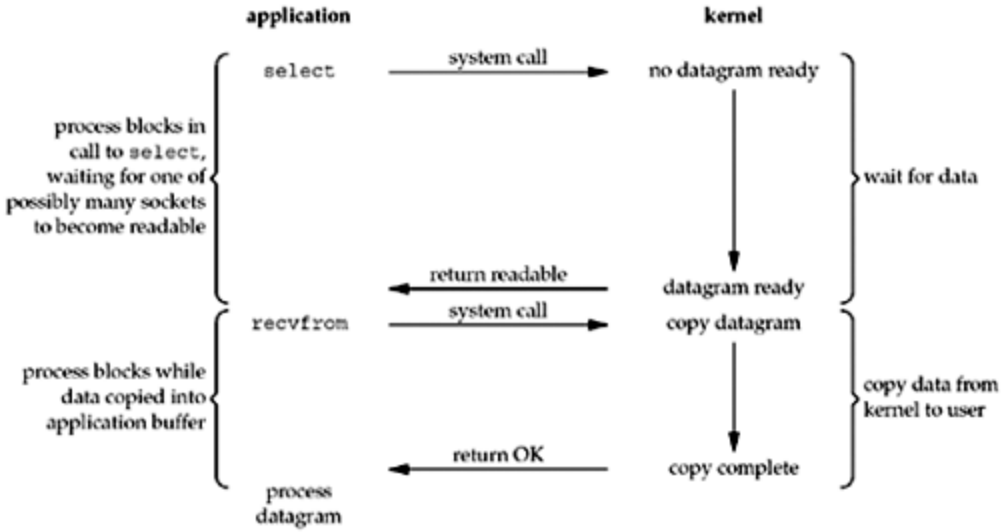
Obsługa wielu połączeń

- wątki
- procesy

Wykorzystanie niezależnych procesów/wątków dla serwera oczekującego na połączenia oraz do obsługi poszczególnych połączeń – nieefektywne wykorzystanie zasobów!

- select (POSIX)
- poll (POSIX)
- ppoll (Linux)
- epoll (Linux)

Monitorowanie w jednym procesie/wątku wielu otwartych połączeń (deskryptorów) w oczekiwaniu na zmiany (I/O Multiplexing Model).



poll() (1/2)

```
#include <poll.h>
int poll (struct pollfd *fdarray, unsigned long nfds, int timeout);
/* Returns: count of ready descriptors, 0 on timeout, –1 on error */

struct pollfd {
  int     fd;       /* descriptor to check */
  short   events;   /* events of interest on fd */
  short   revents;  /* events that occurred on fd */
};
```

Constant	Input to *events* ?	Result from *revents* ?	Description
POLLIN	•	•	Normal or priority band data can be read
POLLRDNORM	•	•	Normal data can be read
POLLRDBAND	•	•	Priority band data can be read
POLLPRI	•	•	High-priority data can be read
POLLOUT	•	•	Normal data can be written
POLLWRNORM	•	•	Normal data can be written
POLLWRBAND	•	•	Priority band data can be written
POLLERR		•	Error has occurred
POLLHUP		•	Hangup has occurred
POLLNVAL		•	Descriptor is not an open file

poll() (2/2)

test-poll.c

```
...
struct pollfd fds[2];                          // The structure for two events

fds[0].fd = sock1;
fds[0].events = POLLIN;                        // Monitor sock1 for input

fds[1].fd = sock2;
fds[1].events = POLLOUT;                       // Monitor sock2 for output

int ret = poll( &fds, 2, 10000 );              // Wait 10 seconds - repeat in a loop

if ( ret == -1 )                               // Check if poll actually succeed
    /* report error and abort */
else if ( ret == 0 )
    /* timeout; no event detected */
else
{
    if ( pfd[0].revents & POLLIN ) {           // event detected, zero it out to reuse the structure
        pfd[0].revents = 0;
        /* input event on sock1, read data from pfd[0].fd*/
    }
    if ( pfd[1].revents & POLLOUT ) {
        pfd[1].revents = 0;
        /* output event on sock2, write data to pfd[1].fd */
    }
}
...
```


Inny, kompletny schemat: https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_71/rzab6/poll.htm

Łącza strumieniowe `socketpair` (1/2)

- `int socketpair(int domain, int type, int protocol, int socket_vector[2]);`
- F-cja tworzy parę sprzężonych gniazd do komunikacji za pomocą tzw. łączy strumieniowych (ang. *stream pipe*). Łącze służy do komunikacji dwustronnej.
- Dostępna jest jedynie lokalna domena **AF_UNIX**. Typy to: **SOCK_STREAM**, **SOCK_DGRAM** i **SOCK_SEQPACKET**.
- Jeżeli nie wystąpił błąd (zwrócona jest warość 0) to do wektora `socket_vector` jest zapisana para deskryptorów gniazd strumieniowych domeny UNIX.
- Komunikacja możliwa jedynie miedzy spokrewnionymi procesami na jednej maszynie. Jedynie QNX pozwala na komunikację między procesami w różnych węzłach.

Łącza strumieniowe `socketpair` (2/2)

test-socketpair.c

```
#define DATA1 "abcde"
#define DATA2 "12345"

...

int sockets[2], child;
char buf[1024];

if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) == 0) {

  if ((child = fork()) > 0) {    /* This is the parent. */
    close(sockets[0]);
    read(sockets[1], buf, sizeof(buf));
    printf("parent - write: %s, read: %s\n", DATA2, buf);
    write(sockets[1], DATA2, sizeof(DATA2));
    close(sockets[1]);
  } else {                       /* This is the child. */
    close(sockets[1]);
    write(sockets[0], DATA1, sizeof(DATA1));
    read(sockets[0], buf, sizeof(buf));
    printf("child - write: %s, read: %s\n", DATA1, buf);
    close(sockets[0]);
  }
}
```

```
parent - write: 12345, read: abcde
child - write: abcde, read: 12345
```