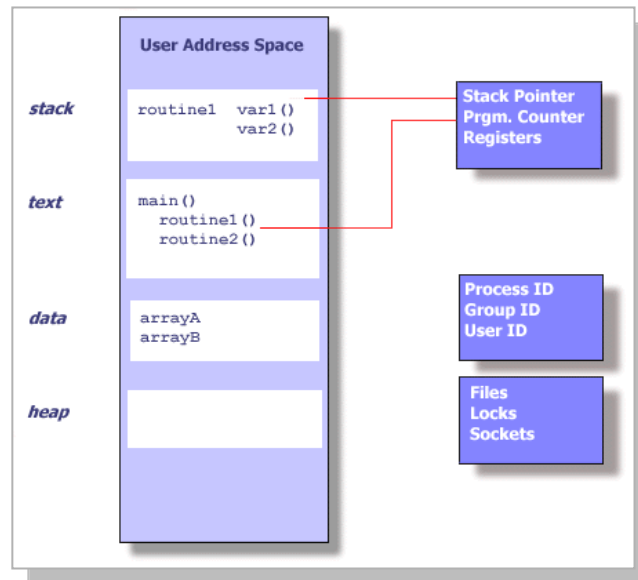


# PODSTAWY OBSŁUGI WĄTKÓW PTHREAD

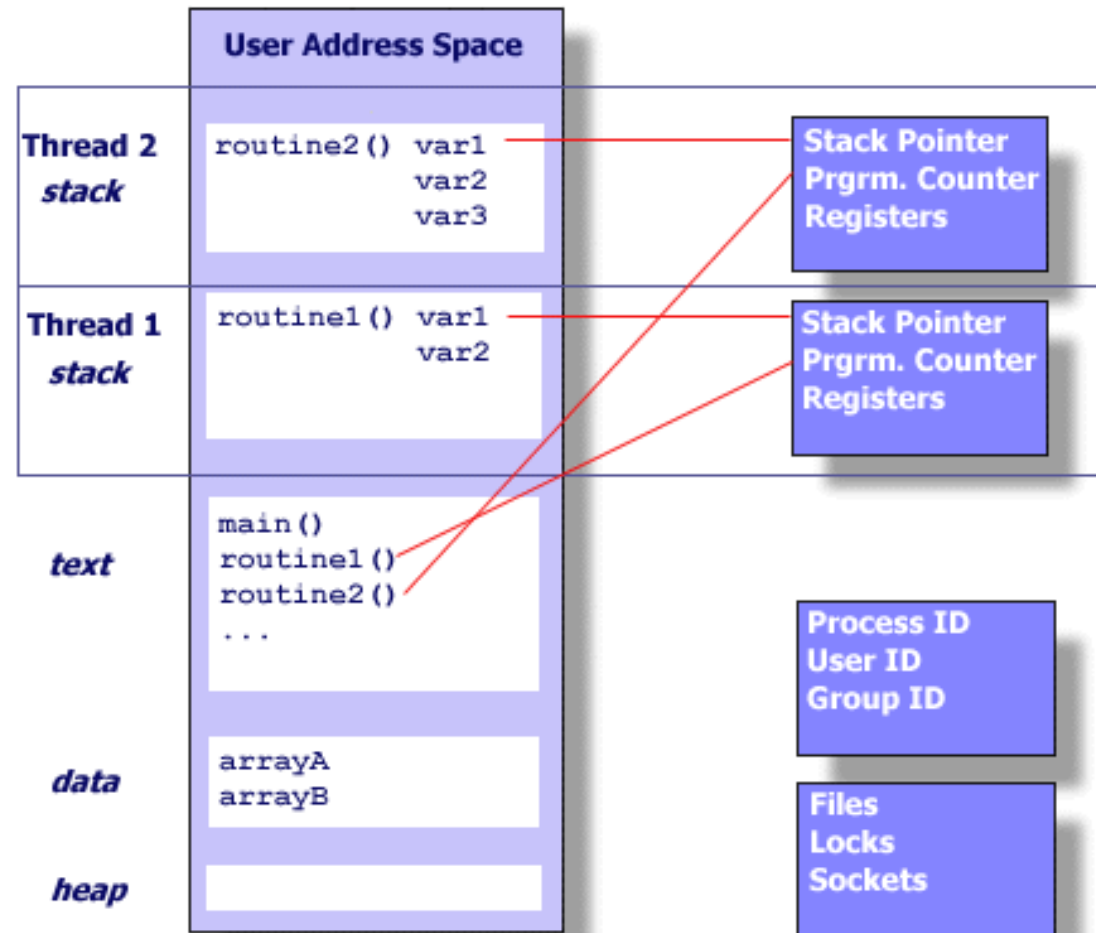
## Co to jest wątek ?

- **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 **podrzedna** 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 wieloprocessorowych, 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 w pozostałych** wątkach 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 wydzielicza "niesforny" wątek. Także blokowanie operacji wejścia/wyjścia nie jest już problemem.

## LINUX: funkcja `clone` (1/2)

- Funkcja `clone` tworzy nową jednostkę wykonawczą (np. proces albo wątek). 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 poziomego 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**.



## LINUX: funkcja `clone` (2/2)

```
int clone(int (*fn)(void *), void *stack, int flags, void *arg, ...  
        /* pid_t *parent_tid, void *tls, pid_t *child_tid */ );
```

*fn* - funkcja wykonywana przez wątek

*stack* - stos wątku

*flags* - flagi mówiące co jest współdzielone między wątkiem a rodzicem, np.:

- `CLONE_FILES` - rodzic i dziecko współdzielą tablicę otwartych deskryptorów plików
- `CLONE_FS` - rodzic i dziecko współdzielą atrybuty związane z systemem plików
- `CLONE_IO` - dziecko współdzieli kontekst We/Wy rodzica
- `CLONE_NEWIPC` - dziecko otrzymuje nową przestrzeń nazw System V IPC
- `CLONE_NEWPID` - dziecko otrzymuje nową przestrzeń nazw z identyfikatorem procesu
- `CLONE_PARENT` - spraw, by rodzic dziecka był taki sam jak rodzic wywołującego
- `CLONE_SIGHAND` - rodzic i dziecko dzielą dyspozycje sygnałów
- `CLONE_VFORK` - rodzic jest zawieszony dopóki dziecko nie wywoła `exec()` lub `_exit()`
- `CLONE_VM` - rodzic i dziecko współdzielą pamięć wirtualną

# 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**)

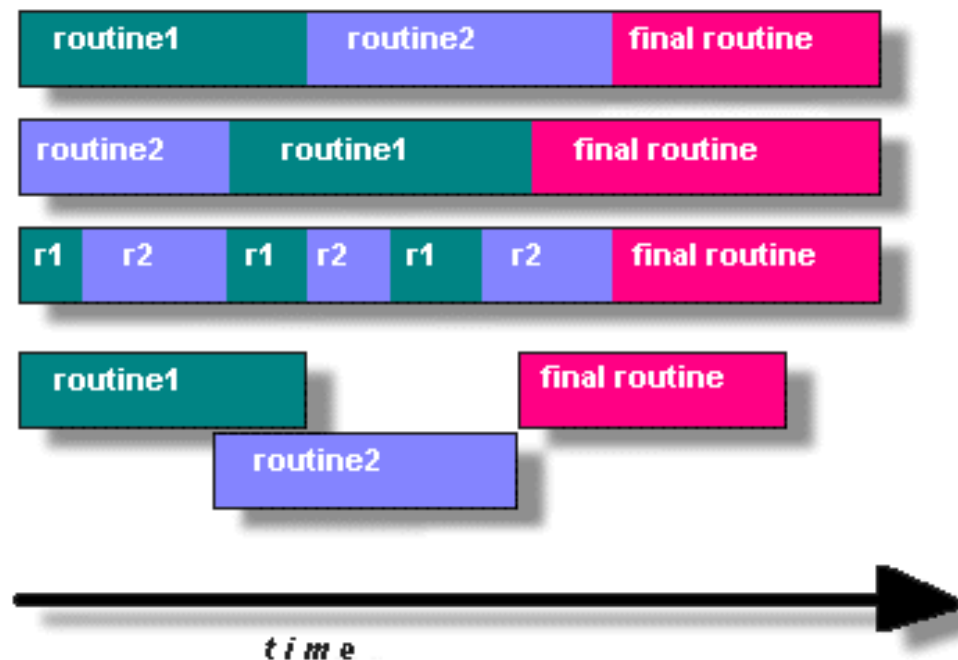
# Tworzenie wątków i procesów

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**

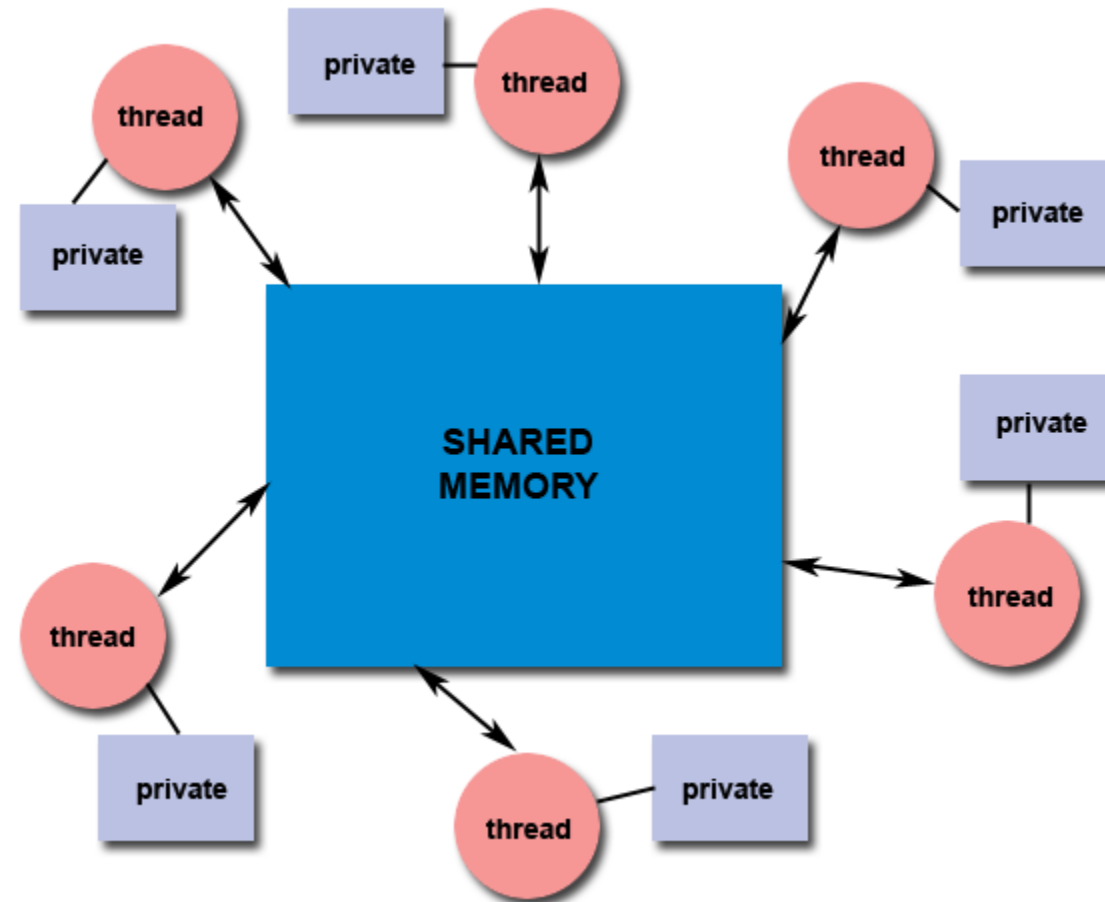
# Wykonanie równoległe

- Stosowanie wątków **ma sens** wtedy, gdy zadania wykonywane w wątkach mogą być **wykonywane** w dużym stopniu **niezależnie**.
- Nie chodzi wyłącznie o przyspieszenie działania, powodem może być rozdzielenie logiki.



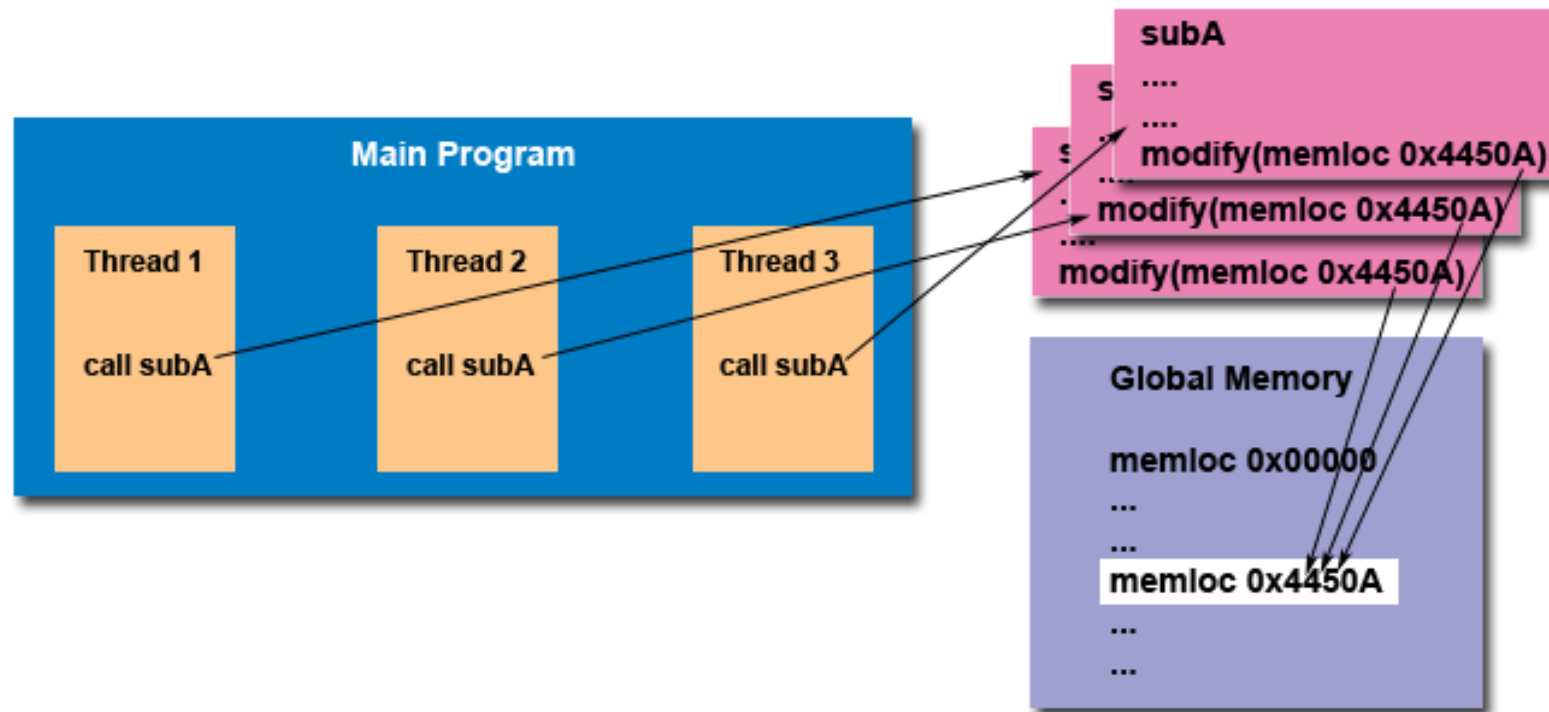
## Model współdzielenia pamięci

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



# Bezpieczna wielowątkowość

Unikanie sytuacji **wyścigu** i jednoczesnej **modyfikacji** danych współdzielonych.

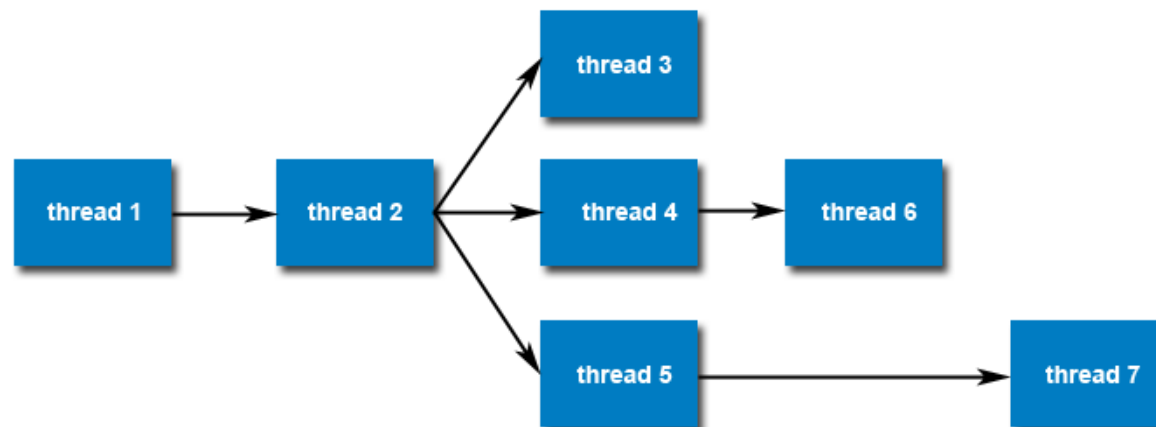


# API Pthread

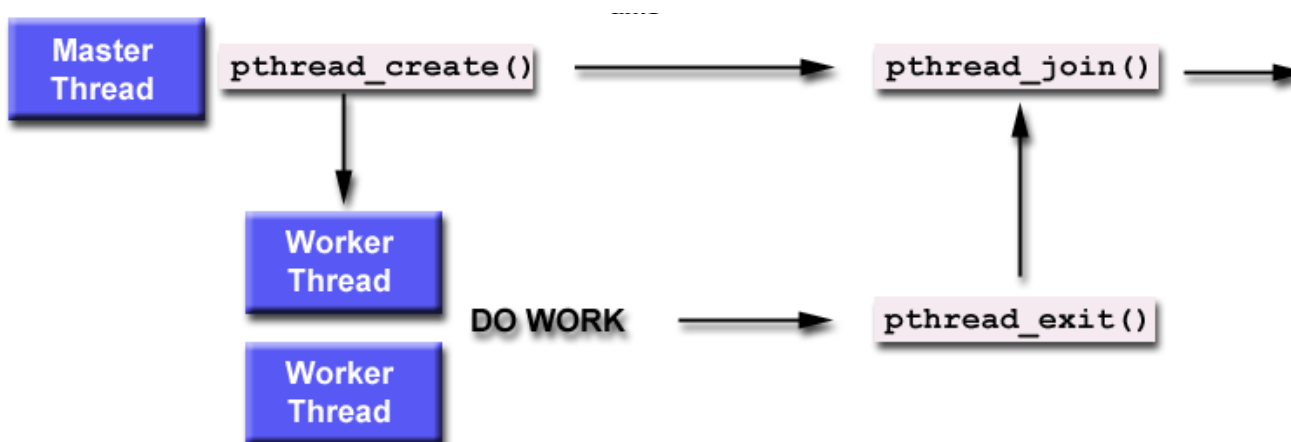
prefiks	Grupa funkcji
pthread_	Wątki
pthread_attr_	Obiekty atrybutów wątków
pthread_mutex_	Muteksy
pthread_mutexattr_	Obiekty atrybutów muteksów
pthread_cond_	Zmienne warunkowe
pthread_condattr_	Obiekty atrybutów warunków
pthread_key_	Klucze

# Zarządzanie wątkami

`pthread_create`  
`pthread_exit`



`pthread_join`





## Funkcja `pthread_create` (1/3)

- `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 wątek zaczyna wykonywać funkcję `start_routine` podając jej `arg` jako argument.
- Nowy wątek 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` prześlemy `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` (2/3)

`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_create` (3/3)

- Nowo utworzony wątek współdzieli z wątkiem głównym m.in.:
  - identyfikator procesu i identyfikator procesu macierzystego;
  - identyfikator grupy procesów i identyfikator sesji;
  - terminal sterujący;
  - dane uwierzytniające proces (identyfikatory użytkownika i grupy);
  - otwarte deskryptory plików;
  - blokady rekordów utworzone przy pomocy `fcntl()`;
  - dyspozycje obsługi sygnałów;
  - informacje związane z systemem plików: `umask`, bieżący katalog roboczy i katalog główny;
  - limity zasobów;
  - zużyty czas procesora (zwrócony przez `times()`) i zużyte zasoby (zwracane przez `getrusage()`);
  - wartość `nice` (ustawiana przez `setpriority()` i `nice()`).
- Wśród atrybutów, które są różne dla każdego wątku są m.in.:
  - identyfikator wątku;
  - maska sygnałów blokowanych;
  - dane specyficzne dla wątku;
  - alternatywny stos sygnałów (`sigaltstack()`);
  - zmienna **`errno`**;
  - zasady i priorytety szeregowania w czasie rzeczywistym;
  - stos (zmienne lokalne i informacje o powiązaniach wywołań funkcji).

## Funkcja `pthread_self`

- `pthread_t pthread_self( void) ;`

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

- `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 działaniewołającego 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 wątek za pomocą funkcji `pthread_join`.

## Funkcja `pthread_join` (1/3)

- `int pthread_join( pthread_t th, void **thread_return );`
- F-cja zawiesza działaniewołającego 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 synchronizację 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`

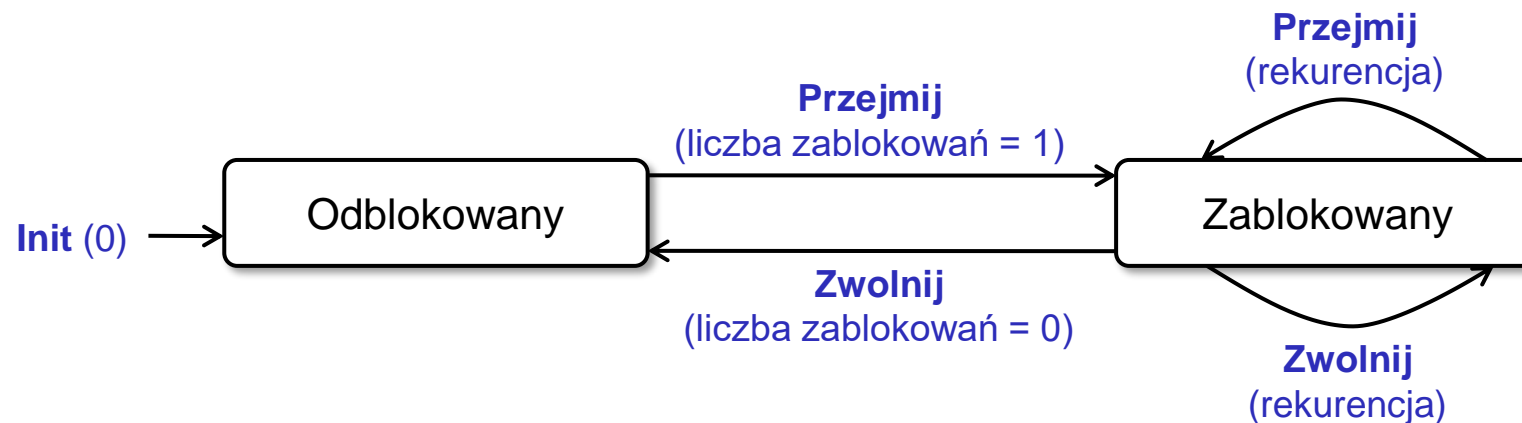
- `int pthread_detach( pthread_t thread );`
- Funkcja odłącza wskazany przez *thread* wątek od jego wątku macierzystego.
- Wątek po odłączeniu działa w trybie, który nie pozwala go synchronizować f-cją `pthread_join` oraz sam zwalania wszystkie zasoby po zakończeniu działania.

# 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 watek 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 przezwołający 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 mutexu (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 mutexu (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;
}
```

## Zmienne warunkowe (1/2)

Istnieje możliwość otrzymania informacji o zmianie wspólnego zasobu.

- `int pthread_cond_signal(pthread_cond_t *cond);`
- `int pthread_cond_broadcast(pthread_cond_t *cond);`
- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
- `int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);`
- Funkcje `...wait` i `...timedwait` usypiają wątek w oczekiwaniu na zmianę stanu zmiennej **cond** wywołaną przez `...signal` albo `...broadcast` w innym wątku.
- W ten sposób możemy informować o tym, że wątek producenta udostępnia gotową porcję danych konsumentowi.
- `...timedwait` działa tak samo jak `...wait` tylko dodany jest maksymalny czas uśpienia.
- `...timedwait` i `...wait` wywołują z automatu `...mutex...lock` na wskazanym muteksie.
- `...signal` wybudzi jeden dowolny czekający wątek, `...broadcast` wszystkie.

## Zmienne warunkowe (2/2)

```
int global = 0;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

static void *threadFunc(void *arg)
{
    while(1) {
        sleep(1);
        pthread_mutex_lock(&mtx);
        global++;
        pthread_mutex_unlock(&mtx);
        pthread_cond_signal(&cond);
        // pthread_cond_broadcast(&cond);
    }
    return NULL;
}

int main() {
    pthread_t thr;
    int s = pthread_create(&thr, NULL, threadFunc, NULL);

    while(1) {
        pthread_cond_wait(&cond, &mtx);
        printf("%d\n", global);
        pthread_mutex_unlock(&mtx);
        if(global == 3) break;
    }

    pthread_cancel(thr); printf("canceled\n");
    pthread_join(thr, NULL); printf("joined\n");
    return 0;
}
```

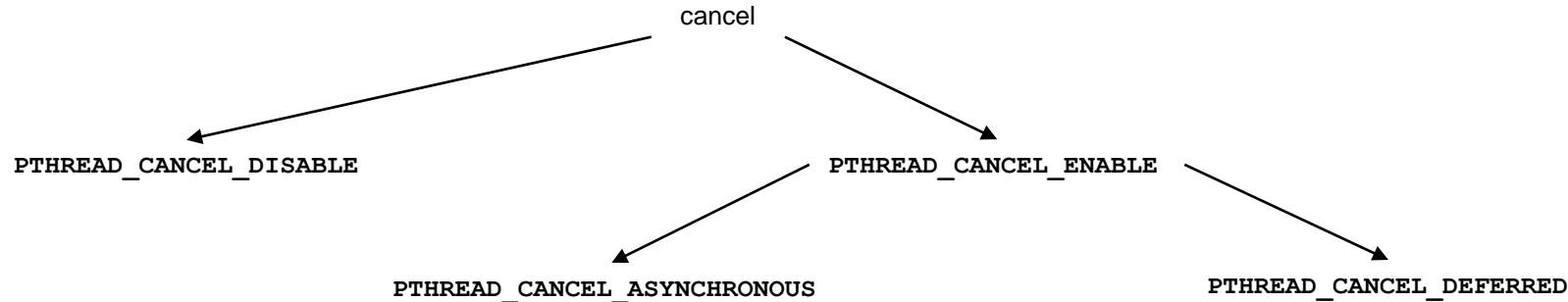
```
$ ./cond
```

```
1
2
3
canceled
joined
```

# Funkcja `pthread_cancel`

- `int pthread_cancel( pthread_t thread );`
- Funkcja przerywa działanie wskazanego wątku. Anulowany wątek zwraca do `pthread_join` wartość `PTHREAD_CANCELED`
- Reakcję wątku na próbę jego przerywania za pomocą `pthread_cancel` możemy kontrolować funkcjami:
  - `pthread_setcancelstate(int state, int *oldstate);`  
czy anulowany: `PTHREAD_CANCEL_DISABLE` i `PTHREAD_CANCEL_ENABLE`
  - `pthread_setcanceltype(int type, int *oldtype);`  
jak anulowany: `PTHREAD_CANCEL_ASYNCHRONOUS` i `PTHREAD_CANCEL_DEFERRED`

# Funkcja `pthread_cancel`

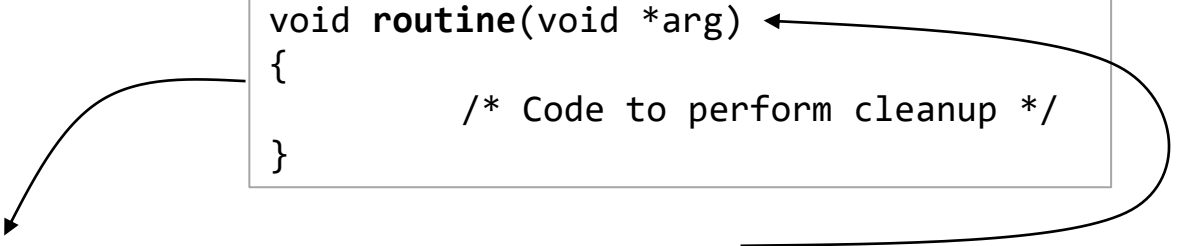


- **PTHREAD\_CANCEL\_ASYNCHRONOUS** - anulowanie asynchroniczne – w każdej chwili
  - **PTHREAD\_CANCEL\_DEFERRED** - anulowanie synchroniczne – żądanie anulowania są kolejkowane, aż do osiągnięcia punktu anulowania
  - **PTHREAD\_CANCEL\_DISABLE** - brak możliwości anulowania
- 
- Bezargumentowe wywołanie `pthread_testcancel` tworzy punkt anulowania.
  - POSIX określa, że punktami anulowania są również miejsca wywołania funkcji: `accept()`, `close()`, `connect()`, `creat()`, `fsync()`, `msgrcv()`, `msgsnd()`, `nanosleep()`, `open()`, `pause()`, `poll()`, `pthread_join()`, `read()`, `recv()`, `sem_wait()`, `send()`, `sleep()`, `system()`, `wait()`, `write()`, ... (itd. kompletna lista w `man 7 pthreads`)



# Funkcja `pthread_cancel`

```
void routine(void *arg)
{
    /* Code to perform cleanup */
}
```



- `void pthread_cleanup_push(void (*routine)(void*), void *arg);`
- `void pthread_cleanup_pop(int execute);`
- Wymuszone przerwanie działania wątku może doprowadzić do problemów, np. w sytuacji gdy przerywany wątek znajdował się w sekcji krytycznej (nie zwolni blokady).
- Mamy możliwość zarejestrowania funkcji czyszczących, które pomogą rozwiązać takie problemy (podobne do `atexit` na poziomie procesów).
- Zarejestrowane funkcje uruchamiane są jeżeli:
  - wątek jest przerywany przez wywołanie `pthread_cancel` (wszystkie funkcje „cleanup” ze stosu),
  - wątek jest zakończony przez wywołanie `pthread_exit` (wszystkie funkcje „cleanup” ze stosu),
  - w wątku wywołujemy `pthread_cleanup_pop` z niezerowym argumentem (funkcja „cleanup” ze szczytu stosu).
- Nie są uruchamiane po wywołaniu `return` !

# Funkcja `pthread_cancel`

```
static void cleanupHandler(void *arg)
{
    printf("cleanup %d\n", *((int*)arg));
}

static void *threadFunc(void *arg)
{
    int test = pthread_self();
    pthread_cleanup_push(cleanupHandler, (void*)&test);
    sleep(10);
    pthread_cleanup_pop(1);
    return NULL;
}

int main() {
    pthread_t thr;
    void *res;
    int s;

    s = pthread_create(&thr, NULL, threadFunc, NULL);
    sleep(2);
    pthread_cancel(thr); printf("canceled\n");
    pthread_join(thr, NULL); printf("joined\n");
    return 0;
}
```

```
$ ./cleanup
```

```
canceled
cleanup 690398976
joined
```

W przykładzie nigdy nie dochodzi do jawnego wywołania `pthread_cleanup_pop` (wątek wcześniej jest przerywany), jednak POSIX wymaga, żeby w danym zakresie leksykalnym obydwie funkcje (push i pop) były wywoływane parami.

# Dane specyficzne dla wątku 1/4

```
int getTick()
{
    ...
}

static void *threadFunc(void *arg)
{
    printf("Thread: %d\n", getTick());
    printf("Thread: %d\n", getTick());
    return NULL;
}

int main(void)
{
    pthread_t thr;

    printf("Main: %d\n", getTick());

    pthread_create(&thr, NULL, threadFunc, NULL);
    pthread_join(thr, NULL);

    printf("Main: %d\n", getTick());

    return 0;
}
```

```
$ ./specificData
```

```
Main: 1
Thread: 1
Thread: 2
Main: 2
```

Chcemy napisać funkcję **getTick**, która po każdym wywołaniu będzie zwracać kolejną wartość całkowitą (począwszy od 1).

Licznik ma się zmieniać niezależnie dla każdego wątku, który będzie wywoływał **getTick** !!!

## Dane specyficzne dla wątku 2/4

```
int getTick()
{
    int counter = 0;
    return ++counter;
}
```

```
$ ./specificData
```

```
Main: 1
Thread: 1
Thread: 1
Main: 1
```

```
int getTick()
{
    static int counter = 0;
    return ++counter;
}
```

```
$ ./specificData
```

```
Main: 1
Thread: 2
Thread: 3
Main: 4
```

## Dane specyficzne dla wątku 3/4

```
int counter = 0;

int getTick()
{
    return ++counter;
}
```

```
$ ./specificData
```

```
Main: 1
Thread: 2
Thread: 3
Main: 4
```

```
int getTick()
{
    int *counter = malloc(sizeof(int));
    *counter = 0;
    *counter = *counter + 1;
    return *counter;
}
```

```
$ ./specificData
```

```
Main: 1
Thread: 1
Thread: 1
Main: 1
```

# Dane specyficzne dla wątku 4/4

```
static pthread_key_t   counterKey;
static pthread_once_t counterOnce = PTHREAD_ONCE_INIT;

static void freeMemory(void *buffer)
{
    free(buffer);
}

static void createKey(void)
{
    pthread_key_create(&counterKey, freeMemory);
}

int getTick()
{
    int *counter;
    // pierwszy wątek wołający funkcję utworzy klucz dla danych specyficznych (jednorazowe wywołanie funkcji createKey)
    pthread_once(&counterOnce, createKey);

    // próbujemy pobrać pamięć powiązaną z kluczem (dla każdego wątku niezależna/specyficzna)
    counter = pthread_getspecific(counterKey);
    // jeżeli jest to pierwsze wywołanie funkcji dla danego wątku, to pthread_getspecific zwróci null, wtedy zaalokujemy pamięć i zwiążemy ją z kluczem
    if (counter == NULL)
    {
        counter = malloc(sizeof(int));
        *counter = 0;
        pthread_setspecific(counterKey, counter);
    }

    *counter = *counter + 1;
    return *counter;
}
```

```
$ ./specificData
```

```
Main: 1
```

```
Thread: 1
```

```
Thread: 2
```

```
Main: 2
```

Wykorzystujemy funkcje:

- pthread\_key\_create
- pthread\_getspecific
- pthread\_set\_specific
- pthread\_once

# Funkcja `pthread_atfork`

```
void prepare() { printf("\tatfork callback - prepare, pid: %d, ppid: %d, tid: %ld\n", getpid(), getppid(), pthread_self()); }
void parent() { printf("\tatfork callback - parent, pid: %d, ppid: %d, tid: %ld\n", getpid(), getppid(), pthread_self()); }
void child() { printf("\tatfork callback - child process, pid: %d, ppid: %d, tid: %ld\n", getpid(), getppid(), pthread_self()); }

static void *threadFunc(void *arg)
{
    pthread_atfork(prepare, parent, child);
    printf("main process, new thread, pid: %d, ppid: %d, tid: %ld\n", getpid(), getppid(), pthread_self());
    sleep(1);
    if( !fork()) {
        printf("main process, new thread, pid: %d, ppid: %d, tid: %ld\n", getpid(), getppid(), pthread_self());
    } else {
        printf("child process, new thread, pid: %d, ppid: %d, tid: %ld\n", getpid(), getppid(), pthread_self());
    }
    return NULL;
}

int main() {
    pthread_t thr;
    printf("parent process, main thread, pid: %d, ppid: %d, tid: %ld\n", getpid(), getppid(), pthread_self());
    int s = pthread_create(&thr, NULL, threadFunc, NULL);
    pthread_join(thr, NULL);
    return 0;
}
```

**prepare** – przed forkiem, w wątku wywołującym

**parent** – po forku, w wątku wywołującym

**child** – po forku, w wątku nowego procesu

```
$ ./atfork
```

```
parent process, main thread, pid: 1441, ppid: 253, tid: 139871435192128
main process, new thread, pid: 1441, ppid: 253, tid: 139871435187968
    atfork callback - prepare, pid: 1441, ppid: 253, tid: 139871435187968
    atfork callback - parent, pid: 1441, ppid: 253, tid: 139871435187968
child process, new thread, pid: 1441, ppid: 253, tid: 139871435187968
    atfork callback - child process, pid: 1443, ppid: 1441, tid: 139871435187968
main process, new thread, pid: 1443, ppid: 1441, tid: 139871435187968
```

# Wątki i sygnały 1/3

```
int counter = 0;

void handler(int sigNo) {
    counter++;
    printf("\tsignal %d, pid: %d, ppid: %d, tid: %ld\n", sigNo, getpid(), getppid(), pthread_self());
}

static void *threadFunc(void *arg)
{
    while(counter < 2);
    printf("puffff ...\n");
    return NULL;
}

int main() {
    pthread_t thr;
    struct sigaction act;
    act.sa_handler = handler;
    act.sa_flags = 0;
    sigemptyset(&(act.sa_mask));
    sigaction(SIGINT, &act, NULL);

    printf("parent process, main thread, pid: %d, ppid: %d, tid: %ld\n", getpid(), getppid(), pthread_self());
    int s = pthread_create(&thr, NULL, threadFunc, NULL);

    while(counter < 4);
    pthread_cancel(thr);
    pthread_join(thr, NULL);
    return 0;
}
```

```
$ ./sigaction
```

```
parent process, main thread, pid: 1615, ppid: 253, tid: 140613593556800
^C    signal 2, pid: 1615, ppid: 253, tid: 140613593556800
^Cpuffff ...
    signal 2, pid: 1615, ppid: 253, tid: 140613593556800
^C    signal 2, pid: 1615, ppid: 253, tid: 140613593556800
^C    signal 2, pid: 1615, ppid: 253, tid: 140613593556800
```

- Obsługa sygnałów jest współdzielona.
- Zmiana obsługi w jednym wątku jest widoczna w innych wątkach.
- Akcja jest wykonywana jeden raz w ramach procesu.



## Wątki i sygnały 2/3

```
...

static void *threadFunc(void *arg)
{
    while(counter < 3);
    printf("puffff ...\n");
    exit(0);
}

int main() {
    pthread_t thr;
    struct sigaction act;
    act.sa_handler = handler;
    act.sa_flags = 0;
    sigemptyset(&(act.sa_mask));
    sigaction(SIGINT, &act, NULL);

    printf("parent process, main thread, pid: %d, ppid: %d, tid: %ld\n", getpid(), getppid(), pthread_self());
    int s = pthread_create(&thr, NULL, threadFunc, NULL);

    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    pthread_sigmask(SIG_BLOCK, &set, NULL);

    while(counter < 4);
    pthread_cancel(thr);
    pthread_join(thr, NULL);
    return 0;
}
```

```
$ ./sigaction
```

```
parent process, main thread, pid: 1626, ppid: 253, tid: 139838118266688
^C      signal 2, pid: 1626, ppid: 253, tid: 139838118262528
^C      signal 2, pid: 1626, ppid: 253, tid: 139838118262528
^C      signal 2, pid: 1626, ppid: 253, tid: 139838118262528
puffff ...
```

- Maska blokowanych sygnałów jest związana z wątkiem.
- Maskę możemy modyfikować za pomocą funkcji **pthread\_sigmask**

# Wątki i sygnały 3/3

```
...

static void *threadFunc(void *arg)
{
    while(1) {
        sleep(1);
    };
    return NULL;
}

int main() {
    pthread_t thr;
    struct sigaction act;
    act.sa_handler = handler;
    act.sa_flags = 0;
    sigemptyset(&(act.sa_mask));
    sigaction(SIGINT, &act, NULL);

    printf("parent process, main thread, pid: %d, ppid: %d, tid: %ld\n", getpid(), getppid(), pthread_self());
    int s = pthread_create(&thr, NULL, threadFunc, NULL);

    while(counter < 3){
        sleep(1);
        pthread_kill(thr, SIGINT);
    };
    pthread_cancel(thr);
    pthread_join(thr, NULL);
    return 0;
}
```

```
$ ./sigaction
```

```
parent process, main thread, pid: 1655, ppid: 253, tid: 140487624591168
    signale 2, pid: 1655, ppid: 253, tid: 140487624587008
    signale 2, pid: 1655, ppid: 253, tid: 140487624587008
    signale 2, pid: 1655, ppid: 253, tid: 140487624587008
    signale 2, pid: 1655, ppid: 253, tid: 140487624587008
```

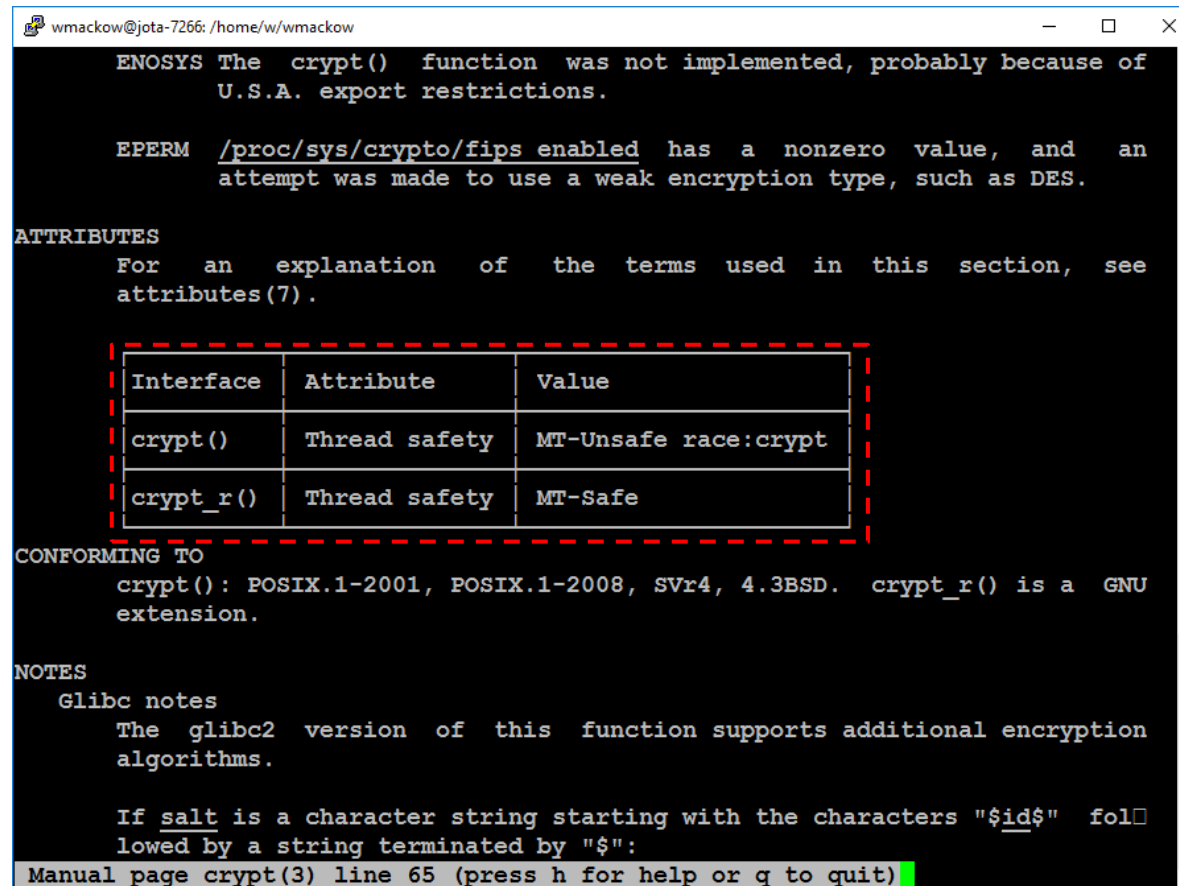
- Sygnały można kierować do konkretnego wątku z innego wątku w tym samym procesie (funkcja `pthread_kill`).

# 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);
```



```
wrmackow@jota-7266: /home/w/wrmackow
```

ENOSYS The crypt() function was not implemented, probably because of U.S.A. export restrictions.

EPERM /proc/sys/crypto/fips enabled has a nonzero value, and an attempt was made to use a weak encryption type, such as DES.

ATTRIBUTES

For an explanation of the terms used in this section, see attributes(7).

Interface	Attribute	Value
crypt()	Thread safety	MT-Unsafe race:crypt
crypt_r()	Thread safety	MT-Safe

CONFORMING TO

crypt(): POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD. crypt\_r() is a GNU extension.

NOTES

Glibc notes

The glibc2 version of this function supports additional encryption algorithms.

If salt is a character string starting with the characters "\$id\$" followed by a string terminated by "\$":

Manual page crypt(3) line 65 (press h for help or q to quit)