

# Programowanie Współbieżne

## Pthread

<http://www.unix.org/version2/whatsnew/threadsref.html>

Oraz strony podręcznika systemowego man

# Wątki

W tradycyjnym modelu w systemie Unix, jeżeli proces wymaga, by część czynności była realizowana przez jakąś inną jednostkę funkcjonalną, to wywołuje funkcję *fork*.

Wady:

- *fork* jest kosztowne, kopiowanie całej zawartości obszaru pamięci przydzielonej procesowi macierzystemu.
- do przekazywania danych po wywołaniu *fork* trzeba używać mechanizmów IPC

# Wątki

Wątki nieraz nazywa się procesami lekkimi (*lightweight processes*), ponieważ nie obciążają tak program jak procesy.

- Utworzenie wątku trwa od 10 do 100 razy szybciej niż procesu.
- wszystkie wątki wykonywane w ramach jednego procesu korzystają ze wspólnej pamięci globalnej. Wymaga to oczywiście synchronizacji
- wspólne instrukcje z których składa się proces
- większość danych też wspólna
- otwarte pliki (tzn. ich deskryptory)
- procedury obsługi sygnałów oraz dyspozycje sygnałów
- bieżący katalog roboczy
- identyfikatory użytkownika i grupy

# Wątki

Wątki różne mają:

- identyfikator wątku
- zbiór rejestrów, włącznie z licznikiem rozkazów oraz wskaźnikiem stosu;
- stos ( w którym są przechowywane zmienne lokalne oraz adresy powrotne);
- zmienna errno;
- maska sygnałów
- priorytet

# Wątki

Większość funkcji dotyczących wątków posixowych zwraca 0 w przypadku sukcesu i dodatni kod błędu w przypadku niepowodzenia.

Gdy używamy wątków trzeba dolinkować

***-lpthread***

# Wątki

- Po uruchomieniu programu funkcją `exec` następuje utworzenie `initial thread` lub `main thread` czyli wątek początkowy.
- Dodatkowe wątki tworzymy za pomocą:

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

```
int pthread_create(pthread_t *tid, const  
pthread_attr_t *attr, void *(func)(void *), void  
*arg);
```

- `tid` – jeżeli uda się utworzyć wątek to tu umieszczony zostanie identyfikator wątku
- `attr` - atrybuty wątku, trzeba unikać alokowania na stosie, gdyż funkcja wołająca może się skończyć zanim wątek się rozpocznie.
- `func` – funkcja z atrybutem `arg` która stanie się wątkiem, tzw. funkcja startowa wątku, jeżeli chcemy więcej to tylko struktura.
- zwraca 0 jeśli ok dodatnią wartość Exxx jeśli wystąpił błąd.

# Wątki

- do każdego wątku wewnątrz procesu można odnosić się za pośrednictwem identyfikatora wątku (threadID). Jest to obiekt typu pthread\_t.
- Każdy wątek ma wiele atrybutów takich jak:
  - priorytet
  - początkowy rozmiar stosu
  - informacje o tym czy jest wątkiem demona
  - itp.
- Zazwyczaj jako atrybutu używa się ustawień domyślnych i wstawia się tam NULL.



# Wątki

Jednokrotne wywołanie funkcji

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

```
int pthread_once (pthread_once_t *control, void  
    (*fun) (void) );
```

- Jednokrotne wywołanie funkcji *fun*.
- Zmienna kontrolna, musi być zainicjowana ***PTHREAD\_ONCE\_INIT***.



# Wątki

Dane specyficzne wątku TSD

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

```
int pthread_key_create(pthread_key_t *key, void  
                      (*destructor)(void));
```

Tworzenie klucza powinniśmy wywołać tylko raz

- **destructor** – jeżeli jest != NULL to *to funkcja o podanym adresie będzie wywołana przy zakończeniu działania wątku z adresem kopii danych specyficznych.*

```
int pthread_key_delete(pthread_key_t key);
```

- Usuwanie klucza

# Wątki

Dane specyficzne wątku TSD

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

```
int pthread_setspecific(pthread_key_t key, void  
    *specific_data);
```

- Ustawienie klucza *key* na dane *specific\_data*.

```
void *pthread_getspecific(pthread_key_t key);
```

- pobranie wskaźnika do danych specyficznych wątku za pomocą klucza *key*

# Wątki

*Czekanie na koniec wątku*

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

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

- jest funkcją podobną do waitpid gdzie czekaliśmy na zakończenie procesu.
- trzeba podać jej **tid** wątku na którego zakończenie chcemy czekać. Nie ma możliwości czekania na dowolny wątek. Jeżeli **status** będzie != NULL to w to miejsce przekazany będzie stan zakończenia wątku.

Przykład: pthread\_tsd/tsd.c

# Wątki

Pobranie własnego *tida* podobne działanie jak *getpid*:

```
#include <pthread.h>  
  
pthread_t pthread_self(void);
```

# Wątki

Porównanie dwóch tidów:

```
#include <pthread.h>

int pthread_equal(pthread_t thread1, pthread_t
    thread2);
```

Gdy różne zwróci 0.



# Wątki

## Kończenie

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

- Jeżeli wątek nie jest odłączony to jego identyfikator oraz stan końcowy są utrzymywane na wypadek pojawienia się funkcji *pthread\_join* pochodzącej z innego wątku tego samego procesu.
- *status* nie może być adresem lokalnego obiektu.
- Innym przypadkiem kończenia wątku jest zakończenie funkcji startowej procesu.
- Wątek też może skończyć się gdy powracamy z funkcji *main* procesu albo jeden z wątków wywołał funkcję *exit* lub *\_exit*.

# Wątki

Kasowanie wątków:

```
int pthread_cancel(pthread_t tid);
```

- Wywoływana zwykle przez inny wątek np. gdy jeden z wątków znalazł rozwiązanie jakiegoś zadania i chce zakończyć pozostałe, lub wstąpił błąd po którym trzeba zakończyć pozostałe wątki.



# Wątki

Kasowanie wątków:

```
int pthread_setcancelstate(int state, int  
                           *oldstate);
```

- **state** - stan do ustawienia
  - **PTHREAD\_CANCEL\_ENABLE** – wątek pozwala by go skasowano
  - **PTHREAD\_CANCEL\_DISABLE** – wątek broni się przed skasowaniem.
- **oldstate** - jeśli oldstate != NULL, to zostanie przekazany poprzedni stan

# Wątki

Kasowanie wątków:

```
int pthread_setcanceltype(int type, int *oldtype);
```

- **type** - typ do ustawienia
  - **PTHREAD\_CANCEL\_ASYNCHRONOUS** – skasowanie wątku może nastąpić w dowolnej chwili, tryb asynchroniczny
  - **PTHREAD\_CANCEL\_DEFERRED** – skasowanie może wystąpić tylko w punktach anulowania (cancellation points), tryb synchroniczny.
- **oldtype** - jeśli oldtype != NULL, to zostanie przekazany poprzedni typ

# Wątki

Punkt anulowania (cancellation point):

```
int pthread_testcancel ();
```

- Gdy wątek jest w trybie anulowania wątku synchronicznego sprawdza czy wystąpiło żądanie jego anulowania.
- Punkty anulowania (cancelation point) są też jednocześnie sprawdzane przy takich funkcjach jak: *close, creat, fcntl, fsync, msync, nanosleep, open, pause, pthread\_cond\_timedwait, pthread\_cond\_wait, pthread\_join, pthread\_testcancel, read, sigwaitinfo, sigsuspend, sigwait, sleep, system, tcdrain, usleep, wait, waitpid, write*

# Wątki

Obsługa zakończenia:

```
void pthread_cleanup_push(void (*function)
                           void*), void *arg);
```

- Dodaje procedurę obsługi porządkowania. Funkcja *function* z jedynym argumentem *arg* wołana jest w przypadku skasowania wątku. Do każdego wywołania pthread\_cleanup\_push musimy zastosować:

```
void pthread_cleanup_pop(int execute);
```

- Zawsze usuwa funkcję umieszczoną na szczycie stosu funkcji obsługi porządkowania, utrzymywanego przez wywołujący ją wątek. Jeśli argument *execute* jest niezerowy, usuwana funkcja jest najpierw wykonywana.

Przykład pthread\_cancel/watek.c

# Atrybuty Wątków

Inicjacja struktury atrybutów wartościami domyślnymi:

```
int pthread_attr_init(pthread_attr_t* attr);
```

Usuwanie struktury atrybutów:

```
int pthread_attr_destroy(pthread_attr_t* attr);
```



# Atrybuty Wątków

## Kontekst szeregowania

```
int pthread_attr_setscope(pthread_attr_t* attr, int scope);
```

```
int pthread_attr_getscope(pthread_attr_t* attr, int *scope);
```

- **scope** – kontekst
  - **PTHREAD\_SCOPE\_SYSTEM** - tryb jądra
  - **PTHREAD\_SCOPE\_PROCESS** – tryb użytkownika (w Linuxie niedostępny)

# Atrybuty Wątków

Czy można czekać na zakończenie wątku pthread\_join

```
int pthread_attr_setdetachstate(pthread_attr_t* attr, int  
detachstate);
```

```
int pthread_attr_getdetachstate(pthread_attr_t* attr, int  
*detachstate);
```

- **detachstate** – stan
  - **PTHREAD\_CREATE\_JOINABLE** - (domyślnie) można czekać, zasoby zakończonego wątku nie będą zwalniane.
  - **PTHREAD\_CREATE\_DETACHED** – nie można czekać, gdyż zasoby zostaną automatycznie zwolnione po zakończeniu tego wątku



# Atrybuty Wątków

## Polityka szeregowania

```
int pthread_attr_setschedpolicy(pthread_attr_t* attr, int policy);
```

```
int pthread_attr_getschedpolicy(pthread_attr_t* attr, int *policy);
```

- **policy** – polityka szeregowania (więcej: man sched\_setscheduler )
  - **SCHED\_OTHER** - (domyślnie) zwykły tryb szeregowania.
  - **SCHED\_RR** – czasu rzeczywistego, algorytm Round-Robin
  - **SCHED\_FIFO** – czasu rzeczywistego, algorytm kolejki FIFO

```
int sched_yield();
```

- Proces zostanie przeniesiony na koniec kolejki swojego statycznego priorytetu i uruchomiony zostanie kolejny proces

# Atrybuty Wątków

## Polityka szeregowania

- `int pthread_attr_setschedparam(pthread_attr_t* attr, int policy, const struct schedparam* param);`
- `int pthread_attr_getschedparam(pthread_attr_t* attr, int *policy, struct schedparam* param);`
- **param** – parametry szeregowania, w szczególności priorytet (domyślnie – 0)
  - `int sched_priority`
- **policy** – polityka szeregowania
  - **SCHED\_OTHER** - (domyślnie) zwykły tryb szeregowania.
  - **SCHED\_RR** – czasu rzeczywistego, algorytm Round-Robin
  - **SCHED\_FIFO** – czasu rzeczywistego, algorytm kolejki FIFO

# Atrybuty Wątków

Tryb ustalania atrybutów wątku

```
int pthread_attr_setinheritsched(pthread_attr_t* attr, int  
    inherit);
```

```
int pthread_attr_getinheritsched(pthread_attr_t* attr, int  
    *inherit);
```

- *inherit* – tryb
  - *PTHREAD\_INHERIT\_SCHED* - Atrybuty są kopiowane z atrybutów przodka wątku.
  - *PTHREAD\_EXPLICIT\_SCHED* – (domyślna) Atrybuty są pobierane z danej struktury przy tworzeniu.

# Muteksy

- Muteksy i zmienne warunku zdefiniowano w normie Posix.1 razem z opisem wątków.
- Można je używać do synchronizowania różnych wątków tego samego procesu.
- Posix umożliwia też synchronizowanie za pomocą muteksów lub zmiennych warunku dla wielu procesów jeżeli te elementy są umieszczone we wspólnej pamięci procesów

# Muteksy

mutekst w posixie to zmienna typu

*pthread\_mutex\_t*

Jeżeli będzie typu *static* musimy nadać wartość początkową

***PTHREAD\_MUTEX\_INITIALIZER***

```
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Jeżeli jest alokowany dynamicznie można go zainicjować w trakcie uruchamiania programu za pomocą *pthread\_mutex\_init*.



# Muteksy

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

```
int pthread_mutex_lock(pthread_mutex_t *mptr);
```

```
int pthread_mutex_trylock(pthread_mutex_t *mptr);
```

```
int pthread_mutex_unlock(pthread_mutex_t *mptr);
```

- Zwracają 0 jeśli ok, Exxx jeśli błąd.
- **lock** – blokuje proces który ją wywołał do czasu zwolnienia muteksu przez inny proces
- **trylock** – w razie zablokowania zwróci błąd EBUSY.
- **unlock** – odblokowuje muteks
- Jeżeli kilka wątków będzie zablokowanych na muteksie to obudzony będzie ten o najwyższym priorytecie (więcej w standardzie Posix.1)

# Muteksy

```
//przykład pthread_mutex/mutex.c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

long licznik = 0;

void *zwieksz(void *ile_razy)
{
    int ii;
    int ile = *(int*)ile_razy;
    printf("jestem sobie wątek 0x%lx\n",pthread_self());
    for (ii=0;ii<ile;ii++)
    {
        pthread_mutex_lock(&mutex);
        licznik++;
        pthread_mutex_unlock(&mutex);
    }
    printf("wątek 0x%lx skonczył\n",pthread_self());
    return NULL;
}
```



# Muteksy

```
int main(int argc, char *argv[])
{
    int ile_razy1 = 3000000;
    int ile_razy2 = 2000000;
    unsigned long tid1, tid2;

    if (pthread_create(&tid1, NULL, zwieksz, &ile_razy1))
        perror("blad pthread1");
    else
        printf("stworzylismy watek1: 0x%lx\n", (unsigned long int) tid1);
    if (pthread_create(&tid2, NULL, zwieksz, &ile_razy2))
        perror("blad pthread2");
    else
        printf("stworzylismy watek2: 0x%lx\n", (unsigned long int) tid2);
    /* jezeli chcemy miec pewność że że wynik będzie po zakończeniu
    wątków to musimy na nie poczekać */
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("licznik: %ld\n", licznik);
    exit(0);
}
```

# Muteksy

```
/* pięciu filozofów z wykorzystaniem mutexów 5f.c*/
```

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

```
#define IL_F 5
#define IL_ITER 10
```

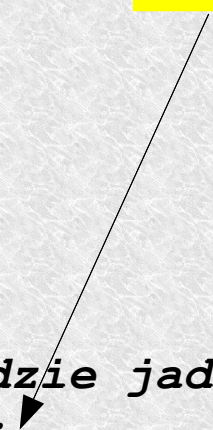
```
pthread_mutex_t paleczka[IL_F];
pthread_t filozofy[IL_F];
```

```
int los(float _min, float _max)
{
    return (int) (rand() * (_max - _min + 1) / INT_MAX + _min );
}
```

# Muteksy

```
void * filozof (void *arg)
{
    int nr = *((int *) arg);
    int ii;
    srand(pthread_self());
    pthread_cleanup_push (&free, arg);
    for (ii = 0; ii < IL_ITER; ii++)
    {
        printf ("Filozof nr %d (%d raz) będzie jadł ...\n", nr, ii+1);
        pthread_mutex_lock (&paleczka[nr]);
        pthread_mutex_lock (&paleczka[(nr + 1) % IL_F]);
        printf ("Filozof nr %d (%d raz) je...\n", nr, ii+1);
        sleep (los(1,5));
        printf ("Filozof nr %d (%d raz) po jedzeniu...\n", nr, ii+1);
        pthread_mutex_unlock (&paleczka[(nr + 1) % IL_F]);
        pthread_mutex_unlock (&paleczka[nr]);
        printf ("Filozof nr %d (%d raz) myśli...\n", nr, ii+1);
        sleep (los(1,5));
    }
    pthread_cleanup_pop (1);
    return NULL;
}
```

Możliwość blokady!!!



# Muteksy

```
int main ()
{
    int ii, *nr;

    for (ii = 0; ii < IL_F; ii++)
        pthread_mutex_init (&paleczka[ii], NULL);

    for (ii = 0; ii < IL_F; ii++)
    {
        nr = (int *) malloc (sizeof (int));
        *nr = ii;
        pthread_create (&filozofy[ii], NULL, &filozof, nr);
    }
    for (ii = 0; ii < IL_F; ii++)
        pthread_join (filozofy[ii], NULL);
    return 0;
}
```

# Zmienne warunkowe

Mutex służy do ryglowania, natomiast zmienna warunku do oczekiwania i sygnalizowania.

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

```
int pthread_cond_wait(pthread_cond_t *cptr,  
                      pthread_mutex_t *mptr);
```

```
int pthread_cond_signal(pthread_cond_t *cptr);
```



# Zmienne warunkowe

Zwykle instrukcje powiadamiania o zmiennej warunku i sprawdzania jej wyglądają tak:

```
pthread_mutex_t mutex =  
PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t cond =  
PTHREAD_COND_INITIALIZER;  
  
pthread_mutex_lock(&mutex);  
//nadanie warunkowi wartości  
//true;  
pthread_cond_signal(&cond);  
pthread_mutex_unlock(&mutex)  
;
```

```
pthread_mutex_lock(&mutex);  
while (warunek nie jest  
      prawdziwy)  
    pthread_cond_wait(&cond, &mutex);  
pthread_mutex_unlock(&mutex);
```

# Zmienne warunkowe

By zapobiec blokadzie po sygnale można zmodyfikować część wysyłającą sygnał tak by funkcja *signal* była poza muteksem. Jednak najlepiej by po *signal* było od razu zwalnianie muteksu

```
int mozna_puscic;  
pthread_mutex_lock(&mutex);  
mozna_puscic = (war == 1);  
pthread_mutex_unlock(&mutex);  
  
if (mozna_puscic)  
    pthread_cond_signal(&cond);
```



# Zmienne warunkowe

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

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int war = 0;

void *sygnal(void *cos)
{
    printf("sygnal START - tid = 0x%lx\n",pthread_self());
    sleep(2);
    pthread_mutex_lock(&mutex);
    printf("sygnal LOCK - tid = 0x%lx\n",pthread_self());
    war = 1;
    printf("sygnal PRZED SIGNAL - tid = 0x%lx\n",pthread_self());
    pthread_cond_signal(&cond);
    printf("sygnal PO SIGNAL - tid = 0x%lx\n",pthread_self());
    sleep(3); //do póki nie opuścimy zwolnimy mutexu drugi proces nadal
jest zablokowany ale już przeszedł warunek wait.
    pthread_mutex_unlock(&mutex);
    printf("sygnal STOP - tid = 0x%lx \n",pthread_self());
    return NULL;
}
```

# Zmienne warunkowe

```
void *czekaj(void *ile_razy)
{
printf("czekaj START - tid = 0x%lx\n",pthread_self());
pthread_mutex_lock(&mutex);
printf("czekaj LOCK - tid = 0x%lx\n",pthread_self());
while(!war)
{
printf("czekaj w while przed wait tid = 0x%lx\n",pthread_self());
sleep(1);
pthread_cond_wait(&cond,&mutex);
printf("czekaj w while po wait tid = 0x%lx\n",pthread_self());
}
printf("czekaj modyfikacja warunku tid = 0x%lx\n",pthread_self());
war = 0;
pthread_mutex_unlock(&mutex);
printf("czekaj STOP tid = 0x%lx \n",pthread_self());
return NULL;
}
```

# Zmienne warunkowe

```
int main(int argc, char *argv[])
{
    unsigned long tid1, tid2;

    if (pthread_create(&tid1, NULL, sygnal, NULL) )
        perror("blad pthread1");
    else
        printf("stworzylismy watek1: 0x%lx\n", (unsigned long int) tid1);
    if (pthread_create(&tid2, NULL, czekaj, NULL) )
        perror("blad pthread2");
    else
        printf("stworzylismy watek2: 0x%lx\n", (unsigned long int) tid2);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    exit(0);
}
```

# Zmienne warunkowe

*pthread\_cond\_signal* budzi tylko jeden z wątków.

Gdy chcemy obudzić wszystkie czekające wątki na danym warunku użyjemy

```
int pthread_cond_broadcast (pthread_cond_t *cptr);
```



# Zmienne warunkowe

Gdy nie chcemy w nieskończoność czekać na zajście warunku możemy zastosować czekanie warunkowe

```
int pthread_cond_timedwait(pthread_cond_t *cptr,  
    pthread_mutex_t *mptr, const struct timespec  
    *abstime);
```

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

**abstime** to struktura opisująca kiedy funkcja ma skończyć czekanie. Jest to czas bezwzględny nie przyrostowy. I liczony jest od 1970.01.01 00:00:00.



# Zmienne warunkowe

Muteks lub zmienna warunku może być inicjowana za pomocą

```
int pthread_mutex_init(pthread_mutex_t *mptr,  
const pthread_mutexattr_t *attr);  
int pthread_cond_init(pthread_cond_t *cptr, const  
pthread_condattr_t *attr);
```

Jeżeli drugi argument **attr** jest NULL to brane są wartości domyślne.