ZAJĘCIA Z SYSTEMÓW OPERACYJNYCH 2 (Programowanie): funkcje POSIX

Author: Arkadiusz D. Danilecki
Version: 1.1
Date: 14.12.2013

Wstęp

Niniejsze opracowanie zawiera jedynie podstawowe informacje o funkcjach dotyczących wątków oraz mechanizmów komunikacji międzyprocesowej (i międzywątkowej) na potrzeby zajęć z systemów operacyjnych. W związku z tym wszelkie informacje dodatkowe student może, a nawet powinien uzyskać samodzielnie przy pomocy przyjaznego polecenia *man*.

W ramach niniejszego opracowania słowo **student** zostało użyte bez intencji sugerowania, że opracowanie dotyczy jedynie studentów płci męskiej i należy je rozumieć jako neutralne słowo oznaczające zarówno studentów, jak i studentki.

Przykłady w języku C obrazujące użycie mechanizmów, oraz kod źródłowy w formacie restructuredText niniejszego opracowania, można znaleźć na stronie autora (http://www.cs.put.poznan.pl/adanilecki/sysop/posix).

W przykładach znak zachęty "#" oznacza polecenia wydawane przez superużytkownika (być może z użyciem poleceni `sudo`), natomiast "$" oznacza polecenia wydawane przez zwykłego użytkownika. W przypadku podawania poleceń należy pominąć znak zachęty. I tak, w przykładzie poniżej, należy wpisać samo `ls`, a nie `$ ls`:

```
$ ls
```

Większość przykładów w tekście wymaga dołączenia do plików odpowiednich plików nagłówkowych. Są one pominięte, ponieważ wystarczy zajrzeć do manuala, by zobaczyć, jakie pliki są potrzebne.

Uwaga edytorska: restructuredText wariuje, kiedy używa się znaku "*", ponieważ znak ten używany jest do zaznaczania fragmentów pogrubionych oraz kursywy. Dlatego też w przykładach kodów piszę np. "char * zm* zamiast "char *zm". Dzieje się tak dlatego, bo parser rst głupieje, gdy dostaje ciąg * w środku bloku kodu, a z kolei samo * w środku bloku kodu sprawia, że głupieje `vi`. Jak dojdę, jak to poprawić, i jak będzie mi się chciało poprawiać, to to zmienię. Proszę o uwagi o odnalezionych błędach lub nieścisłościach na adres adanilecki@cs.put.poznan.pl.

Kolejki komunikatów

Kolejki komunikatów służą do wymiany informacji między procesami. Kolejka istnieje od utworzenia do usunięcia (lub restartu systemu), niezależnie od tego, czy ist-

nieje jakikolwiek proces z niej korzystający. Można wysyłać komunikaty do kolejki (z zadanym priorytetem) za pomocą **mq_send**, można także odbierać komunikaty z kolejki blokującą funkcją **mq_receive**.

Dodatkowe informacje na temat kolejek posix można uzyskać wpisując polecenie:

```
$ man 7 mq_overview
```

Programy kompilujemy z flagą *-lrt*:

```
$ gcc -lrt reader.c -o reader
```

Dostęp z linii poleceń

Superużytkownik (**root**) ma możliwość podmontowania systemu plików **mqueue**, dzięki któremu każdy użytkownik może następnie oglądać utworzone kolejki, rozmiar w bajtach wiadomości w kolejkach itd.

W tym celu superużytkownik powinien wykonać następujące polecenie:

```
# mkdir /dev/mqueue
# mount -t mqueue none /dev/mqueue
```

Ogląd kolejek i ich własności można potem uzyskać przy pomocy zwykłych poleceń dostępnych np. z poziomu basha:

```
$ ls /dev/mqueue
     razem 0
     -rw-r----- 1 szopen users 80 12-13 23:24 pqs
$ cat /dev/mqueue/pqs
     QSIZE:40         NOTIFY:0     SIGNO:0     NOTIFY_PID:0
$ rm /dev/mqueue/pqs
```

Tworzenie i usuwanie kolejek

Każda kolejka posiada własną nazwę. Nazwy te *muszą* zaczynać się od znaku ukośnika na Białystok ("/"), np "/nazwa". Kolejki otwiera się i tworzy podobnie jak zwykłe pliki, tyle, że za pomocą funkcji **mq_open**. W przykładzie poniżej tworzona jest kolejka komunikatów o nazwie "/rumcajs", z domyślnymi atrybutami, otworzona tylko do zapisu. Prawa do dostępu do kolejki mają tylko procesy użytkownika, który utworzył kolejkę, te prawa to odczyt i zapis:

```
mqd_t qd = mq_open( "/nazwa", O_CREAT | O_WRONLY, 0600, NULL );
```

Funkcja **mq_open** zwraca *deskryptor* kolejki, analogicznie jak funkcja **open** zwracała deskryptor pliku. Ten deskryptor (typu **mqd_t**) jest używany potem w funkcjach służących do wysyłania i odbierania wiadomości.

Gdy zakończymy używać kolejkę, zamykamy ją za pomocą funkcji **mq_close**:

```
mq_close( qd );
```

Usuwanie kolejek odbywa się za pomocą funkcji **mq_unlink**:

```
mq_unlink( "/nazwa" );
```

Atrybuty kolejek

Każda kolejka ma związane ze sobą atrybuty określające maksymalną liczbę wiadomości w kolejce, maksymalny rozmiar wiadomości. Można także dowiedzieć się, ile wiadomości jest aktualnie w kolejce. Można wreszcie ustawić flagę O_NONBLOCK, oznaczającą, że operacje na kolejce mają się nie blokować (patrz poniżej przy funkcjach **mq_send** i **mq_receive**).

Atrybuty są zawarte w strukturze typu **mq_attr**. Pobiera się je funkcją **mq_getattr**, a ustawia funkcją **mq_setattr**. Można także ustawić atrybuty dla kolejki przy jej utworzeniu:

```
struct mq_attr attr = \
        { .mq_flags = 0, .mq_maxmsg = 5, .mq_msgsize=100 };
mqd_t qd = mq_open( "/nazwa", O_CREAT | O_RDWR, 0660, &attr );
mq_getattr( qd, &attr );
printf(" W kolejce jest %d wiadomości\n", attr.mq_curmsgs );
```

Zwykły użytkownik nie może ustawić rozmiarów większych niż zadanych przez systemowe maksima. Są to, standardowo dla linuksa, 10 wiadomości równocześnie w kolejce oraz maksymalny rozmiar 8192 jednego komunikatu. Superużytkownik może ustawić rozmiary większe. Tak więc, programy mające ustawione większą liczbę wiadomości równocześnie mogących być w jednej kolejce muszą być odpalane przez superużytkownika, albo muszą miec ustawiony bit suid i superużytkownik musi być ich właścicielem.

Rozmiary stałych można pobrać z następujących plików:

```
/proc/sys/fs/mqueue/msg_max  maksymalna liczba wiadomości w kolejce
/proc/sys/fs/mqueue/msgsize_max maksymalny rozmiar wiadomości
/proc/sys/fs/mqueue/queues_max maks. liczba kolejek
                          mogących być równocześnie w systemie
```

Superużytkownik może zmienić te wartości w zwykły sposób, np:

```
# echo "20" > /proc/sys/fs/mqueue/msg_max
# cat /proc/sys/fs/mqueue/msg_max
   20
```

Wymiana komunikatów

Do wymiany komunikatów między procesami służą funkcje **mq_send** oraz **mq_receive**. Komunikaty są ciągami bajtów; system nie ingeruje w ich treść i nie wnika, czy dany ciąg bajtów ma jakąś wewnętrzną strukturę. Komunikaty posiadają przypisane priorytety, które można im nadać wysyłając komunikat. Priorytety zmieniają kolejność dostarczania wiadomości przez system; ale nie można ustawić, że czekamy na wiadomość o konkretnym priorytecie. Nie należy więc mylić priorytetów z typami kolejek komunikatów systemu V.

Poniższy kod obrazuje wysłanie komunikatu składającego się dwóch pól, liczby typu **int** oraz ciągu znaków. Komunikat ma przypisany priorytet 5.

```
struct kom_t {
    int liczba;
    char label[10];
```

```
} kom;

strncpy( kom.label, "hejas", 5 );
mq_send( qd, (char * ) &kom, sizeof(struct kom_t), 5);
```

Należy zwrócić uwagę, że nie ma sensu w komunikatach przesyłania wskaźników. Na przykład, gdyby pole **label** było typu **char** *, przesłano by dalej wskaźnik; ponieważ procesy mają rozdzielne przestrzenie adresowe, przesłany wskaźnik u odbiorcy nie odnosiłby się do stringu, ale do jakiegoś obszaru w pamięci.

Funkcja **mq_send** blokuje się, jeżeli w kolejce komunikatów jest już maksymalna liczba wiadomości. Funkcja przestaje się blokować, kiedy ktoś odbierze komunikat (albo proces otrzyma sygnał). Można ustawić atrybut kolejki O_NONBLOCK - w takim wypadku funkcja nie będzie się blokować, i zwróci błąd EAGAIN. Istnieje także funkcja **mq_timedsend** (zobacz manual).

Odbieranie wykonywane jest za pomocą **mq_receive**. Poniższy przykład odbiera komunikat z kolejki określonej przez deskryptor **qd**:

```
struct kom_t {
    int liczba;
    char label[10];
} kom;

mq_receive( qd, (char * )&kom, 8192, NULL);
```

W przykładzie trzeci argument oznacza maksymalny rozmiar komunikatów, jaki możemy obsłużyć. Argument ten musi być równy lub większy maksymalnemu rozmiarowi wiadomości dla danej kolejki. Liczba 8192 jest, jak pamiętamy, domyślnym maksymalnym rozmiarem wiadomości w systemie linuks.

Funkcja **mq_receive** blokuje się aż do czasu, gdy będzie mogła odebrać komunikat z kolejki. Odbierany jest najstarszy komunikat z komunikatów o największym typie. Priorytet jest zapisywany pod adresem podanym jako czwarty argument. Jeżeli czwarty argument to NULL, to priorytet nie jest zapisywany. Jeszcze raz przypominam, że w czwartym argumencie nie podajemy, na jaki priorytet czekamy.

Przykład poniżej obrazuje sytuację, w którym maksymalny rozmiar komunikatu w kolejce nie jest nam znany a priori. Musimy więc się o nim dowiedzieć za pomocą funkcji **mq_getattr**. Dodatkowo, zapisujemy wartość otrzymanego priorytetu w zmiennej **prio**:

```
struct mq_attr at;

int prio;
int n=mq_receive(pq, (char \*) &kom, at.mq_msgsize+1, &prio);
if (n<0) { perror("receive error"); break;}
printf("Priorytet to %d\n", prio);
```

Funkcja **mq_receive** nie blokuje się, jeżeli ustawimy atrybut O_NONBLOCK dla kolejki.

Pamięć współdzielona

Obiekty pamięci współdzielonej w standardzie POSIX różnią się nieco do pamięci współdzielonej systemu V. Przede wszystkim, obiekty te posiadają *nazwę*, i można

na nich operować za pomocą funkcji analogicznych lub wręcz identycznych jak dla plików.

Programy kompilujemy z flagą *-lrt*:

```
$ gcc -lrt reader.c -o reader
```

Aby uzyskać więcej informacji, należy przejrzeć strony pomocy systemowej:

```
$ man 7 shm_overview
```

Dostęp do pamięci z linii poleceń

Obiekty pamięci współdzielonej znajdują się oczywiście w pamięci, ale istnieje możliwość dostępu do nich tak, jakby to były zwykłe pliki. Interfejs do pseudo-plików reprezentujących obiekty pamięci współdzielonej jest dostępny pod ścieżką */dev/shm*. Nie trzeba tworzyć ani montować tej ścieżki; jest ona (przynajmniej zwykle jest) dostępna domyślnie w systemie linuksowym.

```
$ ls /dev/shm
initrd_exports.sh     pulse-shm-25470420    sem.ADBE_ReadPrefs_szopen
pulse-shm-1063937813  pulse-shm-2804087063  sem.ADBE_REL_szopen
pulse-shm-2148500640  pulse-shm-651255953   sem.ADBE_WritePrefs_szopen
```

W efekcie, można tworzyć, usuwać, lub modyfikować obiekty pamięci współdzielonej tak, jakby to były zwykłe pliki:

```
$ echo "tekst" > /dev/shm/obszar
$ ls -l /dev/shm
razem 2492
-rw-r--r-- 1 root   root       354 12-14 13:26 initrd_exports.sh
-rw-r--r-- 1 szopen users        6 12-14 14:16 obszar
-r-------- 1 szopen users 67108904 12-14 13:27 pulse-shm-1063937813
-r-------- 1 szopen users 67108904 12-14 13:28 pulse-shm-2148500640
-r-------- 1 szopen users 67108904 12-14 13:27 pulse-shm-25470420
-r-------- 1 szopen users 67108904 12-14 13:27 pulse-shm-2804087063
-r-------- 1 szopen users 67108904 12-14 13:28 pulse-shm-651255953
-rw-rw-rw- 1 szopen users       16 12-14 13:36 sem.ADBE_ReadPrefs_szopen
-rw-rw-rw- 1 szopen users       16 12-14 13:36 sem.ADBE_REL_szopen
-rw-rw-rw- 1 szopen users       16 12-14 13:36 sem.ADBE_WritePrefs_szopen
$ cat /dev/shm/obszar
tekst
```

Tworzenie i usuwanie obiektów pamięci współdzielonej

Obiekty pamięci współdzielonej tworzy się za pomocą funkcji **shm_open**. Zwróćmy uwagę, że nie podajemy rozmiaru obiektu; rozmiar ten można ustalić później za pomocą funkcji **ftruncate**, albo po prostu dynamicznie coś do niego dopisując. Jak poprzednio, nazwa musi zaczynać się od ukośnika na Białystok. Argumenty są analogiczne, jak dla funkcji **open**. Poniżej tworzony jest obiekt o nazwie "/obszar", a jego rozmiar jest zmieniany na 8192 bajty.

```
int sd = shm_open( "/obszar", O_WRONLY | O_CREAT, 0600);
ftruncate( sd, 8192 );
```

Funkcja zwraca nam deskryptor obiektu, który potem możemy używać przy zapisywaniu i odczytywaniu z obiektu.

Kiedy zakończymy używac obiekt, zamykamy go za pomocą funkcji **close** (a nie **shm_close**; nie ma takiej funkcji jak **shm_close**). Usuwanie odbywa się za pomocą **shm_unlink**, przy czym podaje się jako argument nazwę obiektu, a nie deskryptor.

```
close(sd);
shm_unlink( "/obszar" );
```

Komunikacja za pomocą interfejsu plikowego

Po utworzeniu (albo tylko otworzeniu już istniejącego) obiektu pamięci współdzielonej, można na nim operować zwykłymi funkcjami **read** i **write**, o działaniu identycznym, jak dla zwykłych plików.

```
int n= write( sd, "tajna informacja", 17);
if (n < 0) {
    perror(" Błąd przy zapisie");
}
char tablica[17];
n = read( sd, tablica, 17 )
if (n < 0) {
    perror(" Błąd przy odczycie");
} else if (n==0){
    printf("No nic więcej już tutaj nie ma\n");
}
```

Obiekt jest usuwany, gdy zostanie zamknięty przez wszystkie używające go procesy. To znaczy, że nawet jeżeli ktoś usunął obszar, a my ten obiekt mamy jeszcze otwarty, możemy z niego normalnie czytać. Mamy więc normalną uniksową semantykę dostępu do plików.

Mapowanie na przestrzeń adresową procesów

Dostęp przy pomocy funkcji *read* i *write* może wydać się nam niewygodny. Możemy w takim wypadku mapować obszar obiektu na pamięć procesu, analogicznie jak możemy to robić dla wszystkich zwykłych plików. Ważne jest przy tym, że nie istnieje koncepcja obszaru tylko do zapisu, chociaż można utworzyć obszar tylko do odczytu. Przy ustawieniu trybu O_WRONLY, pojawi się błąd EPERM (**Permission denied**) w czasie mapowania obiektu na obszar pamięci. Przy prawach dostępu, bit wykonywalności jest ignorowany, ale możemy go sobie ustawiać do woli.

```
#define ADDR 0
#define LEN 8192
#define OFFSET 0

int sd = shm_open( "/obszar_mmap", O_RDWR | O_CREAT, 0700);
if (sd < 0 ) {
    perror("błąd shm_open");
}
int err = ftruncate( sd, LEN );
```

```
if (err < 0) {
    perror("błąd ftruncate");
}

unsigned int * obszar = (unsigned int * ) mmap( ADDR, LEN, \
                            PROT_WRITE, MAP_SHARED, sd, OFFSET );
if (obszar == (void * ) -1) {
    perror("Błąd mmapa");
}
```

W przykładzie powyżej, PROT_WRITE oznacza zamapowanie do zapisu (PROT_READ oznaczałby mapowanie do odczytu). Flaga MAP_SHARED jest konieczna i oznacza, że mapowany obszar będzie współdzielony między procesami (a więc system musi aktualizować zmiany na bieżąco, a nie dopiero po zamknięciu lub wywołaniu *sync'a*. ADDR oznacza sugerowany adres, gdzie chcielibyśmy podmapować obszar pamięci (system może sugestię zignorować). OFFSET oznacza gdzie, mierząc od początku pliku, zaczyna się mapowany obszar pamięci (OFFSET musi być wielokrotnością rozmiaru strony).

Dygresja: wartość wskaźnika **obszar** rzecz jasna może być różna dla różnych procesów wykonujących mapowanie, mimo, że będą one wskazywać na ten sam obiekt pamięci współdzielonej - pamiętamy, że konkretne wartości wskaźników mają sens dla jednego procesu! Tzn. w jednym procesie obiekt będzie podmapowany pod adres np. b7881000, a w drugim pod adres b7892000 - natomiast oba te adresy wirtualne adresy będą tłumaczone przez system na adres tego samego obiektu pamięci współdzielonej. Jeżeli tego nie rozumiesz, nie przejmuj się. Przeczytaj jeszcze raz, i jeszcze raz, i powtórz sobie z wykładu co było o pamięci wirtualnej. Jeżeli dajesz nie rozumiesz, to wtedy już zacznij się przejmować.

Po wykonaniu mapowania można już używać zmiennej **obszar** tak, jakby była to tablica dynamiczna (lub zmienna innego typu) zaalokowana za pomocą np. **malloc**.

```
obszar[10]=15
```

Po zakończeniu używania obszaru, usuwamy mapowanie, i zamykamy obiekt pamięci współdzielonej:

```
munmap(obszar, LEN);
close( sd );
shm_unlink("/obszar_mmap");
```

Semafory nazwane

Standard POSIX udostępnia dwa rodzaje semaforów: *nazwane* oraz *anonimowe*. Semafory anonimowe poznamy przy okazji omawiania wątków. Obecnie zajmiemy się semaforami nazwanymi: obiektami posiadającymi nazwę w postaci "/nazwa", której pierwszym elementem jest ukośnik, posiadających wartość liczbową, która nigdy nie może być ujemna.

Programy kompiluje się oczywiście z flagą *-lrt* albo, dla odmiany, z flagą *-lpthread*:

```
$ gcc sem.c -o sem -lpthread
$ gcc sem.c -o sem -lrt
```

Aby uzyskać więcej informacji, należy przejrzeć strony pomocy systemowej:

```
$ man 7 sem_overview
```

Tworzenie i usuwanie semaforów

Semafory nazwane tworzy się za pomocą funkcji **sem_open**. W odróżnieniu od semaforów systemu V, funkcja tworzy zawsze tylko jeden semafor, a operacje na nim zawsze podnoszą lub opuszczają semafor o jeden. Przy tworzeniu podajemy także jego wartość początkową. W przykładzie poniżej wartość początkowa semafora wynosi 2, co oznacza, że można go dwa razy opuścić (zanim ktoś inny będzie musiał go podnieść).

```
#include <semaphore.h>

sem_t * s = sem_open("/semafor", O_RDWR | O_CREAT, 0660, 2);
```

Należy zwrócić uwagę na to, że **sem_open** zwraca *wskaźnik* na typ **sem_t**.

Po zakończeniu używaniu semafora, zamykamy go przy pomocy **sem_close**, a usuwamy przy pomocy **sem_unlink**.

```
sem_close( s );
sem_unlink( "/semafor" );
```

Operacje na semaforach

Semafory można podnosić za pomocą **sem_post**, albo obniżyć za pomocą **sem_wait**. Można także pobrać ich wartość za pomocą **sem_getvalue**. Funkcja **sem_wait** blokuje się , jeżeli nie można obniżyć semafora (bo w wyniku obniżenia jego wartość spadłaby poniżej zera, co jest zabronione). Można skorzystać z funkcji **sem_trywait** oraz **sem_timedwait**, jeżeli blokowanie procesu jest niepożądane (*man sem_trywait* dla ciekawych. Na zajęciach nie będziemy korzystać z tej funkcji).

```
sem_post( s );
sem_wait( s );
```

Wątki

Wątki, w odróżnieniu od tradycyjnych procesów, współdzielą przestrzeń adresową. Z założenia wątki współpracują ze sobą i należą do tego samego użytkownika. Dlatego też przełączanie między wątkami jest znacznie tańsze: wystarczy tylko dla każdego wątku zapamiętywać zawartość rejestrów (w tym *ip*), podczas gdy dla przełączenia procesów trzeba, jak pamiętamy, czyścić pamięć podręczną oraz zapisywać i wczytywać tablicę stron danego procesu (mapującą adresy wirtualne na rzeczywiste).

Wątki mogą komunikować się za pomocą zmiennych globalnych. Współdzielą także identyfikator procesów, tablicę otwartych plików, Wątki kończą się, kiedy wywołają funkcję **exit** (wtedy wszystkie wątki procesu są zakończone), zostaną przerwane, albo wywołają **return** z funkcji podanej jako funkcja początkowa wątku (patrz poniżej dla wyjaśnienia). Także kiedy wątek główny wyjdzie z funkcji **main**, wszystkie wątki procesu są kończone (ponieważ wyjście z **main** *de facto* potem prowadzi do wywołania funkcji **exit**). Wreszcie, istnieje funkcja **pthread_exit** kończąca dany wątek.

Programy kompilujemy z flagą *-lpthread*:

```
$ gcc -lpthread watki.c -o watki
```

Aby uzyskać więcej informacji, należy przejrzeć strony pomocy systemowej:

```
$ man 7 pthreads
```

W szczególności, można obejrzeć pomoc systemową dla **pthread_rwlock_init**, **pthread_kill**, **pthread_cancel**, **pthread_exit** oraz **pthread_once**.

Tworzenie wątków

Utworzenie wątków odbywa się za pomocą funkcji **pthread_create**. Musimy przy tym podać wskaźnik do funkcji, od której wykonania rozpocznie działanie wątek. Wyjście z tej funkcji kończyć będze wywołanie wątku. Funkcja powinna być typu **void * (*)(void *)** (tj. zwraca wskaźnik na **void** oraz bierze jeden argument będący wskaźnikiem na **void**) W najprostszym przypadku, w którym chcemy utworzenia wątków z atrybutami domyślnymi i nie przekazujemy wątkom żadnych argumentów, odpowiedni kod wygląda w taki sposób:

```
void * start_func(void * argument)
{
    printf("Hello worldzie\n");
    return 0;
}

int main()
{
    pthread_t watek;
    int err = pthread_create( &watek, NULL, start_func, NULL);
    if (err<0){
        perror("Nie udało się utworzyć wątku:");
    }
}
```

Nie chcemy, by wątek główny zakończył działanie, zanim skończą pracę wszystkie inne wątki (Skutkowałoby to ich przedwczesnym zakończeniem). Dlatego też można wywołać funkcję **pthread_join**, która blokuje się do czasu, aż wątek podany jako argument się zakończy.

```
pthread_join( watek, 0 );
```

Wątki posiadają swoje własne identyfikatory. Identyfikatory te, typu **pthread_t**, wątek może otrzymać wywołując funkcję **pthread_self**.

Wątki można przerywać za pomocą **pthread_cancel**, o ile wątki są utworzone z atrybutem zezwalającym na ich przerywanie (domyślnie, o ile dobrze pamiętam, można przerywać). Wątki nie są przerywane natychmiast, nawet w trybie *asynchronous* (tzn. patrz dalej dla wyjaśnienia). W trybie domyślnym (*deferred*) wątki są przerywane po dojściu do dokładnie zdefiniowanych *punktów przerwania*. Punktami przerwania są np. wywołania funkcji z dobrze zdefiniowanego zbioru. Dla trybu *asynchronous* (trzeba go sobie ustawić za pomocą **pthead_setcanceltype**) wątek **może** być przerwany natychmiast, ale wcale nie musi.

Przekazywanie argumentów, zwracanie wartości

Do wątków można spokojnie przekazywać wartości dowolnego typu. Wątki mogą też zwracać dowolne wartości, z jednym zastrzeżeniem: wątek zwraca *wskaźnik*, więc

ten wskaźnik musi wskazywać adres pamięci istniejący po zakończeniu wątku. Oznacza to, że nie można zwracać adresów zmiennych lokalnych. Dopuszczalne jest więc zwracanie wskaźników do zmiennych globalnych oraz alokowanych dynamicznie.

Poniższy kod demonstruje przekazywanie wartości do wątku oraz zwracanie wartości z wątku.

```
int * start_func(int * arg)
{
    printf("Dostałem %d\n", * arg);
    int * x = (int * ) malloc(sizeof(int));
    * x = 10;
    return x;
}

int main()
{
    pthread_t watek;
    int x = 5;
    int err = pthread_create( &watek, NULL, \
                    (void * ( * ) (void * ))start_func, &x);
    if (err<0){
        perror("Nie udało się utworzyć wątku:");
    }
    int * retval;

    pthread_join( watek, (void ** ) &retval );

    printf("Zwrócono: %d\n", * retval);
    free(retval);
}
```

Zwróćmy uwagę, że **retval** jest typu wskaźnik na *int*, i dodatkowo należy i tak pobrać jego adres; dzieje się tak, ponieważ do **retval** ma zostać zapisany adres pamięci, w którym wątek ulokował zwracaną przez siebie wartość. **pthread_join** musi więc zmodyfikowac wartość zmiennej **retval**, a nie wartość obszaru pamięci wskazywanej przez **retval** (zwracanie wartości odbywa się tak, że jeżeli wątek zwraca **wsk**, to koncepcyjnie chcemy mieć przypisanie **retval = wsk**; dlatego nie możemy przekazać **retval** przez wartość).

Semafory nienazwane posixowe a wątki

W połączeniu z wątkami, można wygodnie (stosunkowo wygodnie) używać semaforów nienazwanych. Wymaga to zainicjowania pewnego obszaru pamięci jako semafora. Ma to zastosowanie np. wtedy, gdy chcemy związać po jednym semaforze z każdą zmienną współdzieloną. Wygodnie jest wtedy stworzyć np. strukturę, której jednym polem jest strzegący dostępu semafor, a drugim polem jest sama zmienna współdzielona, do której dostępu strzeżemy.

Inicjalizacji semaforów nienazwanych (anonimowych) dokonuje się za pomocą funkcji **sem_init**. W przykładzie poniżej najpierw alokujemy dynamicznie strukturę, którą przekazujemy do wątku jako argument. W strukturze pierwsze pole, typu **sem_t**, będzie naszym semaforem. Jak widać, korzystamy z semafora za pomocą poznanych

już wcześniej funkcji **sem_post** i **sem_wait**, a po wszystkim usuwamy go za pomocą **sem_destroy**.

Drugi argument funkcji **sem_init** powinien mieć wartość 0, jeżeli semafor nienazwany będzie używany przez wątki. Jeżeli semafor będzie używany przez procesy, powinien mieć wartość różną od zera. W tym drugim przypadku semafor musi być utworzony w segmencie pamięci współdzielonej systemu V, albo w obiekcie pamięci współdzielonej POSIX.

Trzeci argument funkcji **sem_init** to wartość początkowa semafora.

```
struct pshared {
    sem_t guard;
    char label[48];
    int data;
};

void * producent(struct pshared * comm)
{
    sem_wait( &(comm->guard));
    printf("KONIEC\n");
    return 0
}

void * consument(struct pshared * comm)
{
    sleep(1);
    sem_post( &(comm->guard));
    printf("KONIEC\n");
    return 0
}

#define FLAG 0
#define INITVAL 0
int main()
{
    pthread_t thr_prod, thr_cons;
    pthread_attr_t attr;

    struct pshared * c= \
            (struct pshared * )malloc(sizeof(struct pshared));

    err=sem_init( &(c->guard)), FLAG, INITVAL);
    pthread_attr_init(&attr);

    pthread_create( &thr_prod, &attr, \
                (void * ( * )(void * )) producent, (void * )c);
    pthread_create( &thr_prod, &attr, \
                (void * ( * )(void * )) consument, (void * )c);

    pthread_join(thr_prod, 0);
    pthread_join(thr_cons, 0);
```

```
    sem_destroy( &(c->guard) );
    free(pshared);
}
```

Zwróćmy uwagę na kolejność sprzątania w kodzie: najpierw usuwamy semafor za pomocą funkcji **sem_destroy**, a dopiero potem zwalniamy zaalokowaną pamięć.

Zamki (muteksy)

Zamki, albo inaczej w pongliszu (polish english) muteksy, to po prostu odmiana semaforów, które mogą być podniesione, albo opuszczone - to znaczy mają tylko dwie wartości, 0 i 1. Zamki można próbować zamknąć za pomocą **pthread_mutex_lock**. Operacja ta powoduje zablokowanie, jeżeli zamek jest już zamknięty. Otworzyć zamek można za pomocą **pthread_mutex_unlock**. Standard nie definiuje, co się stanie, jeżeli spróbujemy otworzyć otwarty zamek. Pod linuksem się to udaje, ale zazwyczaj to oznacza błędnie napisany kod - zastosowanie zamków to przecież dostęp do sekcji krytycznej, w której zamykamy zamek przed dostępem, i otwieramy po dostępie.

Istnieją funkcje inicjalizacji zamków, ale nie pamiętam jak się nazywają. Łatwiej zrobić to tak:

```
pthread_mutex_t zamek = PTHREAD_MUTEX_INITIALIZER
```

W przykładzie powyżej **zamek** jest zmienną **globalną**. Używa się potem to cudo tak:

```
void section_critical()
{
        pthread_mutex_lock(&mut);
        printf("Jestem w sekcji krytycznej %x\n", pthread_self());
        sleep(1);
        printf("Wychodzę z sekcji krytycznej %x\n", pthread_self());
        pthread_mutex_unlock(&mut);
}
```

Usuwa się potem zamek za pomocą **pthread_mutex_destroy**

```
pthread_mutex_destroy( &mut);
```

Zmienne warunkowe

Wątki mogą także używać zmiennych warunkowych. Jak pamiętamy z wykładu, cel jest taki: powiedzmy, że zmienna współdzielona X ma wartości od 0 do n, co symbolizuje np. ilość pewnych zasobów (np. chcemy, by naraz n wątków tylko komunikowało się przez sieć, żeby sieć się nie zapchała).

Zobaczmy jakbyśmy to robili, gdybyśmy nie mieli zmiennych warunkowych, a tylko same zamki. Zajmujemy zamek, sprawdzamy, czy wartość zmiennej współdzielonej X jest większa od zera. Jeżeli jest, to super: możemy korzystać z sieci. Zmniejszamy wartość zmiennej X, zwalniamy zamek. Jeżeli wartość zmiennej X jest równa 0, to musimy poczekać. Musimy zwolnić zamek, bo inaczej żaden inny proces nie będzie mógł zwiększyć wartości zmiennej X. Zwalniamy zamek. Po chwili znowu zajmujemy zamek, sprawdzamy wartość zmiennej, dalej jest 0, więc znowu zwalniamy i tak

w kółko: niepotrzebnie co chwilę sprawdzamy wartość tej zmiennej, bo w międzyczasie nikt jej nie modyfikuje: co więcej, możemy właśnie przeszkadzać komuś powiększyć wartość zmiennej. Wygodniej by było mieć mechanizm, w którym byśmy czekali do czasu, aż ktoś wreszcie zmienił wartość X, nie zajmując czasu procesora i nie przeszkadzając innym.

Zmienne warunkujące są rozwiązaniem. Funkcja **pthread_cond_wait** zwalnia zamek i blokuje wątek do czasu, aż ktoś go powiadomi (za pomocą **pthread_cond_signal**), że coś się zmieniło i warto zerknąć na sekcję krytyczną. W takiej sytuacji wątek atomowo się odblokowuje i zajmuje znowu zamek. Tak więc ze zmienną warunkową *zawsze* musi być związany jakiś zamek.

Używa się toto następująco. Załóżmy, że mamy producenta i konsumenta, i dwie zmienne warunkowe (zdefiniowane jako globalne): **cond** służy do sygnalizowania konsumentowi, że coś się zmieniło ze zmienną współdzieloną "a", a **cond2** służy do sygnalizowania producentowi.

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER
pthread_cond_t cond2 = PTHREAD_COND_INITIALIZER
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER
```

Następnie, gdzieś w kodzie mamy dostęp do sekcji krytycznej (tutaj konsument):

```
pthread_mutex_lock( &mut );
while (a<=0) pthread_cond_wait( &cond, &mut);
a--;
pthread_cond_signal( &cond2 );
pthread_mutex_unlock( &mut );
```

Analogicznie, inny wątek (producenta) powiększałby zmienną **a** w ramach swojej sekcji krytycznej i powiadamiał konsumenta za pomocą **pthread_cond_signal**.

Zwróćmy uwagę, że a) przed **pthread_cond_wait** nie zwalniamy zamka, b) po **pthread_cond_wait** nie zamykamy znowu zamka (ta funkcja robi za nas jedno i drugie) b) pamiętamy za to o zwolnieniu zamka po wyjściu z sekcji krytycznej.

Prywatny obszar pamięci

Wątki mogą posiadać prywatne obszary pamięci. Można użyć do tego funkcji **pthread_key_create** i następnie **pthread_setspecific**. Jest to jednak dość upierdliwe, chociaż bardziej eleganckie (jak dla kogo) i przenośne. Jeżeli dane chcemy mieć naprawdę prywatne i niewidoczne dla innych wątków, to możemy je trzymać w obszare zadeklarowanym jako prywatny.

Pod linuksem robi się to tak:

```
__thread int x;
```

Od tej pory zmienna **x** jest prywatna dla każdego wątku.

Mechanizm ten jest **nieprzenośny**, tj. może być dostępny nawet nie na wszystkich architekturach wspieranych przez gcc (i linuksa). Tym bardziej może nie być dostępny dla innych systemów operacyjnych i kompilatorów.

W takim wypadku musimy skorzystać z funkcji **pthread_once**, oraz wspomnianych już **pthread_key_create**, **pthread_get_specific** i **pthread_set_specific**. Pomoc systemowa zawiera przykłady użycia tych funkcji.