Distributed Systems
RESEARCH GROUP

Systemy Operacyjne 2

mechanizmy posix

POLITECHNIKA POZNAŃSKA
Poznan University of Technology

Distributed Systems
RESEARCH GROUP

Wątki

- w odróżnieniu od procesów współdzielą przestrzeń adresową
- należą do tego samego użytkownika
- są tańsze od procesów: wystarczy pamiętać tylko wartości rejestrów, nie trzeba czyścić pamięci podręcznej oraz wczytywać tablicy stron
- mogą komunikować się za pomocą zmiennych globalnych
- współdzielą tablicę otwartych plików i identyfikator procesu
- watki kończą się, kiedy wywołajaą funkcję **exit** (wtedy wszystkie watki procesu zostają zakończone), zostana przerwane, albo wywołają **return** z funkcji podanej jako funkcja początkowa wątku
- jak wątek główny wyjdzie z procedury **main** wszystkie wątki potomne są kończone
- procedura **pthread_exit** kończy wywołujący wątek

Wątki

Kompilacja programu z wątkami

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

Dodatkowe informacje o wątkach

```
$ man 7 pthreads
```

Distributed Systems
RESEARCH GROUP

Tworzenie wątków

Wątki tworzy się za pomocą procedury **pthread_create**

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

`thread` – wskaźnik do wątku

`attr` – parametry tworzonego wątku

`start_routine` – wskaźnik do funkcji wywoływanej w wątku
`void * (*) (void *)` (tj. zwraca wskaźnik na `void` oraz bierze jeden argument będący wskaźnikiem na `void`)

`arg` – parametry funkcji

`return` – 0 jeśli się powiodło, inaczej numer błędu

Distributed Systems
RESEARCH GROUP

Tworzenie wątków - przykład

```
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(”blad”);
      }
      sleep(2);
      return 0;
}
```

Distributed Systems
RESEARCH GROUP

Synchronizacja wątków

Do synchronizacji wątków służy funkcja **pthread_join**. Wątek wywołujący tą procedurę będzie oczekiwał na zakończenie wyspecyfikowanego wątku.

```
int pthread_join(pthread_t thread, void **retval);
```

`thread` – wątek na który czekamy

`**retval` – wartość zwracana przez wątek

`return` – 0 jeśli się powiodło, inaczej numer błędu

Distributed Systems
RESEARCH GROUP

Synchronizacja wątków - przykład

```
void* worker(void* info)
{
        int i;
        for(i=0; i<10; i++)
        {
                sleep(1);
                printf(”thread\n”);
        }
        return NULL;
}
int main()
{
        pthread_t th;
        int i;
        pthread_create(&th, NULL, worker, NULL);
        for(i=0; i<10; i++)
        {
                sleep(1);
                printf(”main program\n”);
        }
        pthread_join(th, NULL);
        return 0;
}
```

Distributed Systems
RESEARCH GROUP

parametry i zwracanie wartości - wątki

Do wątków można przekazywać parametry dowolnego typu oraz zwracać dowolne wartości, z jednym zastrzeżeniem: watek zwraca wskaźnik, więc ten wskaźnik musi wskazywać adres pamięci istniejący po zakończeniu wątku. Zatem w tym celu wykorzystuje się zmienne globalne lub alokowane dynamicznie.

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

Distributed Systems
RESEARCH GROUP

parametry i zwracanie wartości - wątki

```
int main()
{
        pthread_t watek;
        int x = 3;
        int err = pthread_create (&watek, NULL, (void
                        * ( * ) (void *))start_func, &x);
        if (err<0){
                printf("error\n");
        }
        int * retVal;
        pthread_join(watek, (void **) &retVal);
        printf("zwrocono: %d\n",  * retVal);
        free(retVal);
        return 0;
}
```

Distributed Systems
RESEARCH GROUP

Zarządzanie wątkami

Wątek może oczekiwać na zakończenie innego wątku funkcją:

```
int pthread_join(pthread_t th, void **retval);
```

Jest to funkcja analogiczna do funkcji `wait` oczekującej na zakończenie procesu potomnego. Wskaźnik `retval` zostanie zainicjowany wartością zwróconą przez wątek.

Wykonanie wątku można zakończyć w dowolnym momencie wywołaniem

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

Wartość `retval` może być odczytana przez inny wątek, który zsynchronizuje się z nim wywołaniem `pthread_join`.

Distributed Systems
RESEARCH GROUP

Zarządzanie wątkami

Wątek można zatrzymać z poziomu innego wątku funkcją `pthread_cancel`:

```
int pthread_cancel(pthread_t thread);
```

Zatrzymywanie wątku można blokować funkcją:

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

Wątek może „odłączyć” się od procesu i kontynuować pracę niezależnie od wątku głównego. Służy do tego funkcja:

```
int pthread_detach(pthread_t th);
```

Uniezależnienie wątku oznacza, że jego zasoby będą zwolnione po jego zakończeniu. Z drugiej jednak strony nie będzie możliwe zsynchronizowanie z innym wątkiem poprzez wywołanie `pthread_join`.

Zamki

Zamki są binarnymi semaforami, a więc mogą znajdować się w jednym z dwóch stanów: podniesiony lub opuszczony. Do inicjowania zamka wykorzystywana jest funkcja:

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

Poniższy fragment kodu tworzy zamek z domyślnymi atrybutami:

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

Do wykonywania operacji na zamkach służą następujące funkcje:

```
pthread_mutex_lock()
```

Zajęcie zamka. Funkcja jest blokująca do czasu aż operacja może zostać wykonana.

```
pthread_mutex_trylock()
```

Zajęcie wolnego zamka. Próba zajęcia już zajętego zamka kończy się zasygnalizowaniem błędu (EBUSY).

Zamki – cd.

`pthread_mutex_unlock()`

Zwolnienie zamka. Zwolnienia powinien dokonać wątek, który zajął zamek.

`pthread_mutex_destroy()`

Usunięcie zamka.

Wszystkie wymienione funkcje pobierają wskaźnik do struktury `pthread_mutex_t` jako jedyny argument i zwracają wartość typu `int`.

Distributed Systems
RESEARCH GROUP

Zmienne warunkowe

Zmienne warunkowe pozwalają na kontrolowane zasypianie i budzenie procesów w momencie zajścia określonego zdarzenia.

Poniższy fragment kodu tworzy zmienną warunkową z domyślnymi atrybutami:

```
pthread_cond_t c;
...
pthread_cond_init(&c, NULL);
```

Distributed Systems
RESEARCH GROUP

Zmienne warunkowe – cd.

Budzenie wątków realizowane jest z wykorzystaniem jednej z dwóch poniższych funkcji:

```
int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

Funkcja pthread_cond_signal() budzi co najwyżej jeden wątek oczekujący na wskazanej zmiennej warunkowej. Funkcja pthread_cond_broadcast() budzi wszystkie wątki uśpione na wskazanej zmiennej warunkowej.

Distributed Systems
RESEARCH GROUP

Zmienne warunkowe – cd.

Kolejne dwie funkcje służą do usypiania wątku na zmiennej warunkowej:

```
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);
```

Funkcja `pthread_cond_wait()` oczekuje bezwarunkowo do czasu odebrania sygnału budzącego, podczas gdy funkcja `pthread_cond_timedwait()` ogranicza maksymalny czas oczekiwania. Zaśnięcie wątku wymaga użycia jednocześnie zmiennej warunkowej i zamka. Przed zaśnięciem zamek musi być już zajęty. Zaśnięcie oznacza atomowe zwolnienie zamka i rozpoczęcie oczekiwania na sygnał budzący. Obudzenie wątku powoduje ponowne zajęcie zamka

Distributed Systems
RESEARCH GROUP

Zmienne warunkowe – cd.

Poniższy przykład pokazuje zastosowanie zmiennej warunkowej do synchronizacji wątków:

wątek oczekujący:

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

wątek budzący:

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

Zmienne warunkowe – cd.

Pomimo, że nie jest to wymagane, często do poprawnej synchronizacji wywołuje się funkcję budzącą podczas przetrzymywania zamka:

```
pthread_mutex_lock(&m);

pthread_cond_signal(&c);

pthread_mutex_unlock(&m);
```

Semafory nienazwane

Do synchronizacji wątków można wykorzystać semafory standardu POSIX. Jest to rozwiązanie całkowicie niezależne od semaforów Systemu V będących częścią zestawu mechanizmów komunikacji międzyprocesowej IPC. Semafor POSIX jest licznikiem przyjmującym wartości od zera wzwyż. Do obsługi semaforów POSIX przewidziano następujące funkcje:

`sem_init()`
Funkcja tworząca i inicjująca nowy semafor. Semafor może być strukturąwewnętrzną procesu lub może służyć do synchronizacji niezależnych procesów, podobnie jak semafory IPC1 . Argumentem funkcji `sem_init()` jest początkowa wartość semafora.

`sem_wait()`
Oczekiwanie na wartość semafora większą od zera i obniżenie jej o 1.

Semafory – cd.

`sem_trywait()`
Nieblokująca próba zmniejszenia wartości semafora o 1.

`sem_post()`
Zwiększenie wartości semafora o 1. Operacja zawsze wykonuje się poprawnie i nie jest blokująca.

`sem_getvalue()`
Pobranie aktualnej wartości semafora.

`sem_destroy()`
Usunięcie semafora.

Distributed Systems
RESEARCH GROUP

Zadania

zadanie 1:
Napisz program synchronizujący dwa wątki (niech każdy z nich wypisze określony komunikat) za pomocą zamków i przekazywania zamka poprzez parametr.

zadanie 2:
Napisz program synchronizujący trzy wątki. Pierwsze dwa starają się dostać do sekcji krytycznej, a trzeci co 2 sekundy wpuszcza jednego z nich.

zadanie 3:
Napisz program z wykorzystaniem semaforów, który uruchomi dwa wątki. Każdy z nich ma w pętli 10x odwiedzić sekcję krytyczną (`sleep(2);`).

Distributed Systems
RESEARCH GROUP

Zadania – cd.

zadanie 4:
Przećwicz przekazywanie argumentów do wątku i odbiór wartości zwrotnych. Uruchom w tym celu 3 nowe wątki, które będą zwracały podwojoną wartość typu `int`.

zadanie 2:
Utwórz 2 wątki i wstrzymaj ich wykonanie zmienną warunkową. Wątek główny powinien wznowić ich pracę sygnalizując to odpowiedniej zmiennej warunkowej. Sprawdź różnicę pomiędzy funkcją p`thread_cond_signal()` a `pthread_cond_broadcast()`.

zadanie 3:
Zaproponuj implementację operacji bariera(), której działanie polegałoby na zsynchronizowaniu wskazanej liczby wątków. Operacja kończy się w momencie jej wywołania przez wszystkie wątki.

zadanie 4:
Zaproponuj synchronizację 2 wątków w problemie producenta-konsumenta. Przeanalizuj czy rozwiązanie to działa poprawnie również w przypadku większej liczby producentów i konsumentów.