Systemy operacyjne 2020-2021

Metody synchronizacji wątków - materiały pomocnicze

Mutex-y

Mutex (***MUT**ual **EX**clusion*, wzajemne wykluczanie) jest blokadą, którą może uzyskać **tylko jeden wątek**. Mutexy służą głównie do realizacji sekcji krytycznych, czyli bezpiecznego w sensie wielowątkowym dostępu do zasobów współdzielonych.

Schemat działania na mutexach jest następujący:

1. pozyskanie blokady
2. modyfikacja lub odczyt współdzielonego obiektu
3. zwolnienie blokady

Mutex w **pthreads** jest opisywany przez strukturę typu `pthread_mutex_t`, zaś jego atrybuty `pthread_mutexattr_t`.

Obiekt pthread_mutex_t (wzajemnego wykluczania) musi być przed użyciem zainicjalizowany, co odbywa się za pomocą funkcji pthread_mutex_init.

int pthread_mutex_init (**pthread_mutex_t** *mutex, **pthread_mutexattr_t** *attr);

- mutex - wcześniej stworzony przez nas mutex
- attr - atrybuty tworzonego mutexu (domyślne ustawienia: NULL)

Zablokowanie obiektu przez wątek może zostać wykonane poprzez jedną z następujących funkcji:

- **int** pthread_mutex_lock(**pthread_mutex_t** *mutex) - która jeśli obiektu mutex-a jest zablokowany przez inny wątek usypia obecny wątek, aż mutex zostanie odblokowany. Z kolei jeśli obiekt mutex-a jest już zablokowany przez przez obecny wątek to albo:
  - usypia wywołujący ją wątek (jeśli jest to mutex typu "fast")
  - zwraca natychmiast kod błędu EDEADLK (jeśli jest to mutex typu "error checking")
  - normalnie kontynuuje prace, zwiększając licznik blokad mutex-a przez dany wątek (jeśli mutex jest typu "recursive"); odpowiednia liczba razy odblokowań musi nastąpić aby mutex powrócił do stanu "unlocked"
- **int** pthread_mutex_trylock(**pthread_mutex_t** *mutex) - która zachowuje się podobnie jak powyższa, z tym że obecny wątek nie jest blokowany jeśli mutex jest już zablokowany przez inny wątek,a jedynie ustawia flagę EBUSY.
- pthread_mutex_timedlock - jest rozwinięciem funkcji pthread_mutex_lock - podawany jest maksymalny czas czekania wątku na odblokowanie (zablokowanego przez inny wątek) mutex-a.

Odblokowanie mutex-a wykonywane jest za pomocą funkcji pthread_mutex_unlock.

Ostrzeżenie: Należy zwrócić uwagę, aby nie używać mutex-ów w funkcjach obsługujących sygnały (signal handlers), gdyż może to doprowadzić do zablokowania się programu.

Jednym z najprostszych zastosowań mutex-ów jest ochrona zmiennej przed równoczesnym dostępem z wielu wątków. Pokazuje to następujący przykład:

Chronienie dostępu do zmiennej

```
int x;
pthread_mutex_t x_mutex = PTHREAD_MUTEX_INITIALIZER;

void my_thread_safe_function(...) {
  /* Każdy dostęp do zmiennej x powinien się odbywać w następujący sposób: */
  pthread_mutex_lock(&x_mutex);
  /* operacje na x... */
  pthread_mutex_unlock(&x_mutex);
}

...
```

Zakleszczenia

Zakleszczenia są najczęściej spowodowane nieprawidłowym używaniem mechanizmu mutexów przez programistę. Może to wystąpić np. w sytuacji gdy próbujemy zamknąć mutex w wątku, w którym już wcześniej to zrobiliśmy. Zachowanie programu jest wtedy uzależnione od typu używanego mutexu.

Wyróżniamy trzy rodzaje mutexów:

- **Szybki mutex** (fast mutex) – domyślny typ, próba zamknięcia takiego mutexu z wątku, w którym wcześniej już to zrobiliśmy spowoduje zakleszczenie.
- **Rekursywny mutex** – nie powoduje zakleszczenia. Zapamiętuje ile razy wątek zablokował danego mutexa i oczekuje na taką samą ilość odblokowań.
- **Mutex sprawdzający błędy** (error-checking mutex) – taka próba spowoduje błąd **EDEADLK** podczas wywoływania pthread_mutex_lock

By stworzyć niestandardowe mutexy używamy następujących typów i funkcji: **int** pthread_mutexattr_init(*attr) - inicjalizuje zmienną z atrybutami
int pthread_mutexattr_destroy(**pthread_mutexattr_t** *attr) - zwalnianie
int pthread_mutexattr_settype(**pthread_mutexattr_t** *attr, int type) - ustawia typ mutexu
int pthread_mutexattr_gettype(**const pthread_mutexattr_t** *attr, int *type) - pobiera typ mutexu

- **pthread_mutexattr_t attr** - zmienna przechowująca atrybuty mutexa
- **type** - typ mutexu - może przyjmować następujące wartości:

PTHREAD_MUTEX_NORMAL
PTHREAD_MUTEX_ERRORCHECK
PTHREAD_MUTEX_RECURSIVE

Współdzielenie mutexu z innymi procesami

Funkcje

- `int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared)`
- `int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr, int *pshared)`

Zmiana priorytetu wątku posiadającego blokadę

Dostępne, gdy istnieje rozszerzenie TPP (oraz TPI).

Wartość atrybutu decyduje o strategii wykonywania programu, gdy wiele wątków o różnych priorytetach stara się o uzyskanie blokady. Atrybut może mieć wartości:

1. `PTHREAD_PRIO_NONE`,
2. `PTHREAD_PRIO_PROTECT`,
3. `PTHREAD_PRIO_INHERIT` (opcja TPI).

W przypadku **`PTHREAD_PRIO_NONE`** priorytet wątku, który pozyskuje blokadę nie zmienia się.

W dwóch pozostałych przypadkach z mutexem powiązany zostaje pewien priorytet i gdy wątek uzyska blokadę, wówczas jego priorytet jest podbijany do wartość z mutexu (o ile oczywiście był wcześniej niższy). Innymi słowy w obrębie sekcji krytycznej wątek może działać z wyższym priorytetem.

Sposób ustalania priorytetu mutexu zależy od atrybutu:

- `PTHREAD_PRIO_INHERIT` - wybierany jest maksymalny priorytet spośród wątków oczekujących na uzyskanie danej blokady;
- `PTHREAD_PRIO_PROTECT` - priorytet jest ustalany przez programistę funkcją `pthread_mutexattr_setprioceiling` lub `pthread_mutex_setprioceiling` (opisane w następnej sekcji).

Dodatkowo jeśli wybrano wartość `PTHREAD_PRIO_PROTECT`, wówczas wszelkie próby założenia blokady funkcjami `pthread_mutex_XXXlock` z poziomu wątków o priorytecie niższym niż ustawiony dla mutexa nie powiodą się - zostanie zwrócona wartość błędu `EINVAL`.

Funkcje

- `int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol)` (*doc*)
- `int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr, int *protocol)` (*doc*)

Semafory nienazwane

- Nowy semafor jest tworzony przez funkcję sem_init
- Operacja Sygnalizuj jest wykonywana za pomocą sem_post, która zwiększa licznik semafora. Funkcja ta nigdy nie powoduje blokady wywołującego ją wątku.
- Operacja Czekaj jest wykonywana za pomocą jednej z następujących funkcji:

    - sem_wait - proces jest usypiany aż dany semafor ma nie-zerową wartość, po czym następuje atomowa operacja zmniejszenia licznika semafora
    - sem_trywait - jest nie blokującym wariantem sem_wait - jeśli semafor ma zerowy licznik, funkcja nie usypia wątku lecz zwraca -1 u stawia numer błędu na EAGAIN
- Semafor jest usuwany za pomocą sem_destroy

Warunki Sprawdzające (Condition Variables)

Czasami konieczne jest monitorowanie przez wątek pewnych warunków. Implementacja ich sprawdzania z wykorzystaniem konwencjonalnych mechanizmów (ciągłego sprawdzania czy warunek jest spełniony) byłaby mocno nieefektywna, gdyż powodowałaby że wątek byłby ciągle zajęty.

Rozwiązaniem tego problemu jest mechanizm warunków sprawdzających (Condition Variables). Pozwala on na uśpienie wątku aż do momentu gdy pewne warunki na dzielonych danych zostaną spełnione.

Za pomocą zmiennych warunkowych możemy wstrzymywać wykonywanie wątku, aż do momentu gdy zajdzie określony przez nas warunek. Zmiennej warunkowej towarzyszy mutex, który zapewnia wyłączność w trakcie odczytu/zmiany wartości flagi.

Gdy wątek dochodzi do sekcji zależnej od pewnego warunku (np. flagi), wykonywana jest sekwencja:

- Wątek zajmuje mutexa następnie sprawdza warunek.
- Jeżeli warunek jest spełniony – wtedy wątek wykonuje kolejne instrukcje.
- Jeśli warunek nie jest spełniony – wtedy wątek jednocześnie odblokowuje mutex i wstrzymuje działanie aż do spełnienia warunku (poinformowania o zmianie warunku przez inny watek).

Przy zmianie warunku muszą być podjęte następujące kroki:

- Zajęcie mutexa towarzyszącego zmiennej warunkowej
- Podjęcie akcji, która może zmienić warunek
- Zasygnalizowanie oczekującym wątkom zmiany warunku.
- Odblokowanie mutexa.

Zmienne warunkowe przechowywane są w typie pthread_cond_t.
Zmienne warunkowe obsługują następujące funkcje:

int pthread_cond_init(**pthread_cond_t** *cond, pthread_condattr_t *attr) - inicjalizacja zmiennej
int pthread_cond_destroy(**pthread_cond_t** *cond) - usunięcie zmiennej
int pthread_cond_wait(**pthread_cond_t** *cond, pthread_mutex_t *mutex) - ustawia wątek w tryb oczekiwania w czasie, którego Mutex jest odblokowany
int pthread_cond_timedwait(**pthread_cond_t** *cond, pthread_mutex_t *mutex, const struct timespec *timeout) - usypia wątek na określoną ilość czasu
int pthread_cond_broadcast(**pthread_cond_t** *cond) - powiadamia wszystkie oczekujące wątki
int pthread_cond_signal(**pthread_cond_t** *cond) - powiadamia tylko jeden wątek

Zmienna warunkowa może być również inicjalizowana makrem **PTHREAD_COND_INITIALIZER**, które zainicjalizuje ją standardowymi atrybutami.

Do obsługi atrybutów służa funkcje: **int** pthread_condattr_init(**pthread_condattr_t** *attr) - inicjalizacja
int pthread_condattr_destroy(**pthread_condattr_t** *attr) - zwolnianie
int pthread_condattr_getpshared(**const pthread_condattr_t** *attr, **int** *pshared) - pobiera atrybut process-shared
int pthread_condattr_setpshared(**pthread_condattr_t** *attr, **int** pshared) - ustawia atrybut process-shared

Poniżej znajduje się "urywek" kodu wykorzystujący warunki sprawdzające. Dwie zmienne dzielone (x i y) są sprawdzana pod kątem większości x od y. Każda zmiana dowolnej ze zmiennych powoduje obudzenie "zainteresowanych" wątków.

Przykład użycia warunków sprawdzających

```
int x,y;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond   = PTHREAD_COND_INITIALIZER;

...


// (Watek 1)
// Czekanie aż x jest większe od y jest
// przeprowadzane następująco:

pthread_mutex_lock(&mutex);
while (x <= y) {
    pthread_cond_wait(&cond, &mutex);
}

...

pthread_mutex_unlock(&mutex);

...


// (Watek 2)
// Kazda modyfikacja x lub y może
// powodować zmianę warunków. Należy
// obudzić pozostałe wątki, które korzystają
// z tego warunku sprawdzającego.

pthread_mutex_lock(&mutex);
/* zmiana x oraz y */
if (x > y)
    pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mutex);
```

Ostatnia modyfikacja: środa, 17 maja 2017, 21:28

◄ Zadania - Zestaw 9

Przejdź do...

Zadania - Zestaw 10 ►