Wyścigi

Wątki są z definicji wykonywane współbieżnie i asynchronicznie, a także operują na współdzielonych globalnych strukturach danych. Może to prowadzić do korupcji tych struktur.

Rozważmy wielowątkowy program obsługujący operacje na dowolnych kontach bankowych, spływające z oddziałów i bankomatów. Operacja może być wpłatą lub wypłatą, w zależności od wartości zmiennej operacja (dodatnia lub ujemna).

Wątek 1:

```
// otrzymany numer konta
// oraz wartosc operacji
saldo[konto] += operacja;
```

Wątek 2:

```
// otrzymany numer konta
// oraz wartosc operacji
saldo[konto] += operacja;
```

Tak równoległe jak i quasi równoległe wykonanie tych wątków może spowodować takie pofragmentowanie tych operacji, że jeśli jednocześnie pojawią się dwie operacje na tym samym koncie, to zostanie obliczona niepoprawna wartość salda. Zjawisko takie jest nazywane **wyścigami**.

Sekcja krytyczna i muteksy

Tego typu problem wyścigów można rozwiązać identyfikując fragment programu, zwany **sekcją krytyczną**. Wyścigi mogą powstać jeśli sekcja krytyczna będzie wykonana równocześnie przez różne procesy/wątki.

Dla ochrony sekcji krytycznej można użyć **muteksu**. W danej chwili tylko jeden wątek może posiadać blokadę muteksu, a drugi będzie musiał na nią czekać. Założenie blokady muteksu powoduje, że żadna z operacji na saldzie nie będzie mogła być przerwana przez drugą.

Wątek 1:

```
// otrzymany numer konta
// oraz wartosc operacji
pthread_mutex_lock(transakcyjny);
saldo[konto] += operacja;
pthread_mutex_unlock(transakcyjny);
```

Wątek 2:

```
// otrzymany numer konta
// oraz wartosc operacji
pthread_mutex_lock(transakcyjny);
saldo[konto] += operacja;
pthread_mutex_unlock(transakcyjny);
```

Wprowadzenie muteksu powoduje **serializację** sekcji krytycznej. Nie będzie ona nigdy wykonywana równolegle, lecz zawsze szeregowo. Na komputerze posiadającym wiele procesorów lub rdzeni z reguły powoduje to spowolnienie programu przez ograniczenie jego współbieżności. Dlatego sekcja krytyczna zabezpieczana muteksem powinna być minimalna, obejmując wyłącznie instrukcje mogące spowodować wyścigi.

Muteksy — niepodzielność operacji

Mutex jest bardzo prostą koncepcją. W istocie jest jednobitową zmienną, normalnie o wartości 0, oznaczającej brak blokady. Założenie blokady muteksu nadaje mu wartość 1, być może uprzednio czekając na jego wyzerowanie, natomiast jej zwolnienie przywraca wartość 0.

Jednak operacje na muteksie muszą być zrealizowane niepodzielnie (*atomic*). Inaczej wątek, który sprawdziłby, że wartość muteksu wynosi 0 (wolny), i następnie ustawił go na 1 (zajęty), mógłby zostać wywłaszczony zaraz po sprawdzeniu wartości, i w wyniku wyścigów, po wznowieniu, zająć mutex w międzyczasie zajęty już przez inny wątek.

Jedynie system operacyjny może zapewnić nierozdzielne wykonanie sprawdzenia i zajęcia muteksu. Dlatego muteksy, podobnie jak inne mechanizmy synchronizacji, mają swój zestaw funkcji systemowych na nich operujących. Na przykład, operacje na muteksach biblioteki Pthread (z pominięciem operacji na atrybutach muteksów):

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Muteksy — inicjalizacja

Mutex, podobnie jak inne mechanizmy synchronizacji zdefiniowane przez standard POSIX, wykorzystują struktury pamięciowe tworzone jawnie w programach. Mutex jest zwykłą zmienną logiczną, ustawianą na 1 (blokada) lub 0 (brak blokady), plus informacja, który wątek założył blokadę muteksu, i zatem ma prawo ją zdjąć.

Przy deklaracji zmiennych typu mutex należy pamiętać, żeby zainicjalizować je na zero, co oznacza mutex odblokowany. Możliwe sposoby deklaracji muteksu:

```
static pthread_mutex_t mutex1;

pthread_mutex_t *mutex2;
mutex2 = (pthread_mutex_t *)
         calloc(1, sizeof(pthread_mutex_t));

pthread_mutex_t mutex3 = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t mutex4;
  pthread_mutexattr_t attr_ob;
  pthread_mutexattr_init(&attr_ob)
  pthread_mutex_init(&mutex4, &attr_ob)
```

Blokady zapisu i odczytu

Rozważmy teraz system obsługi haseł, lub PINów. Najczęstszą operacją jest pobranie PINu w celu jego sprawdzenia. Ponieważ równoczesne operacje odczytu nie mogą spowodować wyścigów, zatem na pozór nie ma potrzeby blokowania dostępu różnych wątków. Jednak od czasu do czasu pojawia się operacja zmiany PINu, która wymaga chwilowego zablokowania odczytów, jak również ewentualnych innych zmian.

Ten problem można rozwiązać efektywnie za pomocą **blokad zapisu i odczytu**, które są innym mechanizmem synchronizacyjnym. Umożliwiają one zakładanie **współdzielonych** blokad odczytu, wykorzystywanych przy pobraniach PINu, oraz **wyłącznych** blokad zapisu, wykorzystywanych przy zmianach.

```
// otrzymany numer konta
// oraz nowa wartosc PINu
if (oper=SPRAWDZENIE) {
  pthread_rwlock_rdlock(LOCK_PIN);
  aktualny_pin = pin[konto];
} else { // zmiana PINu
  pthread_rwlock_wrlock(LOCK_PIN);
  pin[konto] = nowy_pin
}
pthread_rwlock_unlock(LOCK_PIN);
```

Blokady zapisu i odczytu są pewnym uogólnieniem pojęcia muteksów. Zauważmy, że zajęcie muteksu jest równoważne zajęciu blokady zapisu i odczytu do zapisu. Natomiast zajęcie blokady zapisu i odczytu do odczytu jest nową możliwością, której muteksy nie mają. Korzyść z użycia rozdzielonych blokad zapisu i odczytu, zamiast muteksów, będzie znacząca tylko wtedy gdy operacji wymagających blokad odczytu będzie znacznie więcej niż tych wymagających blokad zapisu.

Operacje Pthread na blokadach zapisu i odczytu:

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Semafory — komentarz

Biblioteka Pthread nie wprowadza semaforów, ponieważ istnieje szereg innych standardów definiujących semafory, dobrze implementowanych we wszystkich systemach. Wcześniej poznaliśmy semafory standardu POSIX Realtime.

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. W jakiej sytuacji mogą powstać wyścigi w programach z wątkami?

2. Co to jest sekcja krytyczna?

3. W jaki sposób można ochronić sekcję krytyczną muteksem?

4. Czy dodanie synchronizacji do programu wielowątkowego może przyspieszyć, spowolnić, czy nie powinno wpłynąć na czas jego wykonania?

5. Wyjaśnij dlaczego operacje na muteksie, który jest zwykłą zmienną logiczną, muszą być realizowane przez system operacyjny, a nie przez program użytkownika.

6. Porównaj ochronę sekcji krytycznej muteksem i blokadą zapisu i odczytu.

7. Porównaj operacje dostępne na muteksach i wcześniej poznane operacje na semaforach.

Synchronizacja wątków: zmienne warunkowe

Zmienne warunkowe służą do czekania na, i sygnalizowania, spełnienia jakichś warunków. Ogólnie schemat użycia zmiennej warunkowej ze stowarzyszonym muteksem jest następujący:

```
struct {
  pthread_mutex_t mutex;
  pthread_cond_t  cond;
  { ... inne zmienne przechowujace warunek }
} global = { PTHREAD_MUTEX_INITIALIZER,
             PTHREAD_COND_INITIALIZER, {...} };
```

schemat sygnalizacji zmiennej warunkowej:

```
pthread_mutex_lock(&global.mutex);
// ustawienie oczekiwanego warunku
pthread_cond_signal(&global.cond);
pthread_mutex_unlock(&global.mutex);
```

schemat sprawdzania warunku i oczekiwania:

```
pthread_mutex_lock(&global.mutex);
while ( { warunek niespelniony } )
  pthread_cond_wait(&global.cond,
                    &global.mutex);
//wykorzystanie oczekiwanego warunku
pthread_mutex_unlock(&global.mutex);
```

Synchronizacja wątków: zmienne warunkowe (cd.)

Algorytm działania funkcji `pthread_cond_wait`:

1. odblokowanie muteksu podanego jako argument wywołania
2. uśpienie wątku do czasu aż jakiś inny wątek wywoła funkcję `pthread_cond_signal` na danej zmiennej warunkowej
3. ponowne zablokowanie muteksu

Pełny interfejs funkcjonalny zmiennych warunkowych Pthread:

```
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);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Przykład: producenci i konsumenci

Przykładowy program producentów i konsumenta (jednego), zaczerpnięty z książki Stevensa (nowe wydanie, tom 2). Pierwsza wersja nie zawiera żadnej synchronizacji operacji na zmiennych globalnych i może generować błędy.

```
#define MAXNITEMS       1000000
#define MAXNTHREADS         100

int     nitems; /* read-only by producer and consumer */
struct {
  pthread_mutex_t   mutex;
  int   buff[MAXNITEMS];
  int   nput;
  int   nval;
} shared = { PTHREAD_MUTEX_INITIALIZER };

void    *produce(void *), *consume(void *);

int
main(int argc, char **argv)
{
  int         i, nthreads, count[MAXNTHREADS];
  pthread_t   tid_produce[MAXNTHREADS], tid_consume;
```

```
  if (argc != 3)
    err_quit("usage: prodcons2 <#items> <#threads>");
  nitems = min(atoi(argv[1]), MAXNITEMS);
  nthreads = min(atoi(argv[2]), MAXNTHREADS);

  Set_concurrency(nthreads);
  /* start all the producer threads */
  for (i = 0; i < nthreads; i++) {
    count[i] = 0;
    Pthread_create(&tid_produce[i], NULL, produce,
                   &count[i]);
  }

  /* wait for all the producer threads */
  for (i = 0; i < nthreads; i++) {
    Pthread_join(tid_produce[i], NULL);
    printf("count[%d] = %d\n", i, count[i]);
  }

  /* start, then wait for the consumer thread */
  Pthread_create(&tid_consume, NULL, consume, NULL);
  Pthread_join(tid_consume, NULL);

  exit(0);
}
```

```
void *
produce(void *arg) {
  for ( ; ; ) {
    Pthread_mutex_lock(&shared.mutex);
    if (shared.nput >= nitems) {
      Pthread_mutex_unlock(&shared.mutex);
      return(NULL); /* array is full, we're done */
    }
    shared.buff[shared.nput] = shared.nval;
    shared.nput++;
    shared.nval++;
    Pthread_mutex_unlock(&shared.mutex);
    *((int *) arg) += 1;
  }
}

void *
consume(void *arg) {
  int     i;
  for (i = 0; i < nitems; i++) {
    if (shared.buff[i] != i)
      printf("buff[%d] = %d\n", i, shared.buff[i]);
  }
  return(NULL);
}
```

Przykład: producenci i konsumenci (2)

Druga wersja programu z konsumentem pracującym równolegle i synchronizacją wszystkich operacji na zmiennych globalnych za pomocą muteksu. Wykorzystanie muteksu powoduje całkowicie poprawne działanie programu.

Ponieważ jednak konsument pracuje równolegle z producentami, konsument poza użyciem muteksu przy dostępie do obszaru wspólnego, musi upewnić się, że element produkowany jest gotowy, i czekać, gdy nie jest.

Fragment funkcji `main` uruchamiający wszystkie wątki równolegle:

```
Set_concurrency(nthreads + 1);
/* create all producers and one consumer */
for (i = 0; i < nthreads; i++) {
  count[i] = 0;
  Pthread_create(&tid_produce[i], NULL, produce,
                 &count[i]);
}
Pthread_create(&tid_consume, NULL, consume, NULL);
```

```
void *
produce(void *arg)
{
  for ( ; ; ) {
    Pthread_mutex_lock(&shared.mutex);
    if (shared.nput >= nitems) {
      Pthread_mutex_unlock(&shared.mutex);
      return(NULL); /* array is full, we're done */
    }
    shared.buff[shared.nput] = shared.nval;
    shared.nput++;
    shared.nval++;
    Pthread_mutex_unlock(&shared.mutex);
    *((int *) arg) += 1;
  }
}
```

```
void
consume_wait(int i)
{
  for ( ; ; ) {
    Pthread_mutex_lock(&shared.mutex);
    if (i < shared.nput) {
      Pthread_mutex_unlock(&shared.mutex);
      return;          /* an item is ready */
    }
    Pthread_mutex_unlock(&shared.mutex);
  }
}

void *
consume(void *arg)
{
  int     i;

  for (i = 0; i < nitems; i++) {
    consume_wait(i);
    if (shared.buff[i] != i)
      printf("buff[%d] = %d\n", i, shared.buff[i]);
  }
  return(NULL);
}
```

Przykład: producenci i konsumenci (3)

Kolejna wersja programu producentów i konsumentów uwalnia konsumenta od konieczności okresowego sprawdzania, czy jest już wyprodukowany element. Użycie zmiennej warunkowej pozwala konsumentowi czekać w zmiennej na wyprodukowanie elementu, co wymaga by producent zasygnalizował zmienną po wyprodukowaniu elementu.

```
int     buff[MAXNITEMS];
struct {
  pthread_mutex_t   mutex;
  pthread_cond_t    cond;
  int               nput;
  int               nval;
  int               nready;
} nready = { PTHREAD_MUTEX_INITIALIZER,
             PTHREAD_COND_INITIALIZER };
```

```
void *
produce(void *arg) {
  for ( ; ; ) {
    Pthread_mutex_lock(&nready.mutex);
    if (nready.nput >= nitems) {
      Pthread_mutex_unlock(&nready.mutex);
      return(NULL);       /* array is full, we're done */
    }
    buff[nready.nput] = nready.nval;
    nready.nput++;
    nready.nval++;
    nready.nready++;
    Pthread_cond_signal(&nready.cond);
    Pthread_mutex_unlock(&nready.mutex);
    *((int *) arg) += 1;
  }
}
```

```
void *
consume(void *arg) {
  int     i;

  for (i = 0; i < nitems; i++) {
    Pthread_mutex_lock(&nready.mutex);
    while (nready.nready == 0)
      Pthread_cond_wait(&nready.cond, &nready.mutex);
    nready.nready--;
    Pthread_mutex_unlock(&nready.mutex);

    if (buff[i] != i)
      printf("buff[%d] = %d\n", i, buff[i]);
  }
  return(NULL);
}
```

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Opisz współpracę wątków synchronizowaną za pomocą zmiennej warunkowej.

2. Dlaczego funkcja `pthread_cond_wait` odblokowuje mutex zadany jako jej parametr? I dlaczego ponownie go potem blokuje?

Bariery — inny rodzaj synchronizacji

Bariera jest mechanizmem synchronizacji innego rodzaju. Zamiast zapewniać, że wątki nie będą wykonywały jednocześnie sekcji krytycznej programu, bariera służy do zapewnienia, że wszystkie wątki przejdą do kolejnej sekcji programu równocześnie (albo quasi-równocześnie).

```
#include <pthread.h>

int pthread_barrier_init(pthread_barrier_t *barrier,
                         const pthread_barrierattr_t *restrict attr,
                         unsigned count);
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr, int pshared);
int pthread_barrierattr_getpshared(const pthread_barrierattr_t *restrict attr,
                                   int *restrict pshared);
int pthread_barrier_wait(pthread_barrier_t *barrier);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
int  pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
```

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Przeczytaj opisy funkcji `pthread_barrier_init` oraz `pthread_barrier_wait`.
   Wyjaśnij jak działa bariera.
   Jakie jest znaczenie parametru `count`?

Programowanie współbieżne: drobnoziarnistość

Przy tworzeniu aplikacji współbieżnych istotną kwestią jest **drobnoziarnistość równoległości** (*fine-grained parallelism*). Aplikacja może być współbieżna na poziomie programów, procedur, lub instrukcji. Przy większej drobnoziarnistości ogólnie osiąga się większą rzeczywistą równoległość pracy, ponieważ jest większa szansa, że w danym momencie jakiś wątek będzie mógł się wykonywać, zamiast czekać na inne wątki. Jednak większa drobnoziarnistość pociąga za sobą większe narzuty na tworzenie i obsługę tych równoległych wątków.

Rozważmy transformację obrazu rastrowego, np. o rozmiarze jednego miliona pikseli. Można ją wykonać za pomocą jednego wątku, przechodzącego sekwencyjnie od piksela do piksela, lub za pomocą pewnej liczby wątków (maksymalnie miliona), z których każdy będzie przetwarzał pewną liczbę przypisanych mu pikseli. Jednowątkowe rozwiązanie nie ponosi żadnych narzutów na zrównoleglanie, jednak nie będzie w stanie wykorzystać wielu rdzeni procesora. Drugie ekstremum równoległości, czyli stworzenie miliona wątków, z których każdy przetworzy tylko jeden piksel, zapewne też nie będzie efektywne (o ile nie mamy do dyspozycji miliona rdzeni), ponieważ narzuty na stworzenie pojedynczego wątku na pewno przewyższą zysk z jego równoległej pracy nad pojedynczym pikselem. Dla pewnego stopnia równoległości osiągnięte będzie minimum czasu przetwarzania całego obrazu, zależne od rodzaju procesora i wielu innych czynników.

Drobnoziarnistość mechanizmów synchronizacji

Zauważyliśmy wcześniej, że mechanizmy synchronizacji, jakkolwiek potrzebne, ograniczają współbieżność działania wielowątkowych programów. Wiele wątków może zostać wstrzymanych na muteksie lub blokadzie założonej przez pojedynczy wątek.

Przy wprowadzaniu elementów synchronizacji pojawia się kwestia **drobnoziarnistości synchronizacji** (*fine-grained synchronization*). Aplikacja może założyć jedną blokadę na potrzeby wprowadzania jakichkolwiek zmian, lub może wprowadzić wiele blokad, z których każda dotyczy innego rodzaju zmian. Zwiększanie drobnoziarnistości synchronizacji pozwala na zwiększenie równoległości programu. Jednocześnie, może generować większe narzuty. Jednak w odróżnieniu od zwiększania drobnoziarnistej równoległości programu, które zawsze generuje większe narzuty, zwiększanie drobnoziarnistości synchronizacji nie zawsze je generuje.

Na przykład, aplikacja bankowa może założyć pojedynczą blokadę na wykonanie transakcji, a następnie sprawdzić numer PINu, i jeśli się zgadza, wykonać transakcję. Alternatywnie, może założyć blokadę na operację sprawdzenia PINu, zdjąć blokadę, i jeśli PIN się zgadza, wykonać transakcję, po uprzednim założeniu osobnej blokady.

Drobnoziarnistość mechanizmów synchronizacji (2)

Sposoby kontrolowania drobnoziarnistości synchronizacji:

wielkość sekcji krytycznej — sekcja krytyczna powinna być jak najmniejsza; możemy to osiągnąć albo przenosząc poza sekcję instrukcje niegenerujące wyścigów, albo przez rozbicie większej sekcji na kilka mniejszych; pierwsze rozwiązanie nie wnosi dodatkowych narzutów, ale drugie tak, ponieważ wymaga dodatkowych blokad,

rozdzielenie mechanizmów synchronizacji — stworzenie oddzielnych mechanizmów synchronizacji dla ochrony obiektów, które nie są modyfikowane razem; zwykle nie wnosi dodatkowych narzutów,

użycie właściwych mechanizmów synchronizacji — na przykład użycie współdzielonych blokad zapisu i odczytu zamiast muteksu, gdzie to jest właściwe; zwiększa drobnoziarnistość synchronizacji, bez dodatkowych narzutów,

brak synchronizacji — niekiedy w programie można całkowicie wyeliminować synchronizację, np. przenosząc zmienne globalne do wnętrza procedury, dzięki czemu nie będą one dostępne dla innych wątków, albo stosując inny algorytm działania, niewymagający synchronizacji; w oczywisty sposób zmniejsza to narzuty bez zmniejszenia równoległości.

Przykład: producenci i konsumenci (4)

```
int     buff[MAXNITEMS];
struct {
  pthread_mutex_t   mutex;
  int               nput;   /* next index to store */
  int               nval;   /* next value to store */
} put = { PTHREAD_MUTEX_INITIALIZER };

struct {
  pthread_mutex_t   mutex;
  pthread_cond_t    cond;
  int               nready; /* number ready for consumer */
} nready = { PTHREAD_MUTEX_INITIALIZER,
             PTHREAD_COND_INITIALIZER };
```

```
void * produce(void *arg) {
  for ( ; ; ) {
    Pthread_mutex_lock(&put.mutex);
    if (put.nput >= nitems) {
      Pthread_mutex_unlock(&put.mutex);
      return(NULL);       /* array is full, we're done */
    }
    buff[put.nput] = put.nval;
    put.nput++;
    put.nval++;
    Pthread_mutex_unlock(&put.mutex);

    Pthread_mutex_lock(&nready.mutex);
    if (nready.nready == 0)
      Pthread_cond_signal(&nready.cond);
    nready.nready++;
    Pthread_mutex_unlock(&nready.mutex);
    *((int *) arg) += 1;
  }
}
```

```
void * consume(void *arg) {
  int     i;

  for (i = 0; i < nitems; i++) {
    Pthread_mutex_lock(&nready.mutex);
    while (nready.nready == 0)
      Pthread_cond_wait(&nready.cond, &nready.mutex);
    nready.nready--;
    Pthread_mutex_unlock(&nready.mutex);
    if (buff[i] != i)
      printf("buff[%d] = %d\n", i, buff[i]);
  }
  return(NULL);
}
```

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Wyjaśnij pojęcie drobnoziarnistości synchronizacji.

2. Wyjaśnij dlaczego większa drobnoziarnistość synchronizacji zwiększa współbieżność programów.

3. Wyjaśnij dlaczego większa drobnoziarnistość synchronizacji może zwiększyć czas wykonania programów. Oraz dlaczego może go zmniejszyć.

4. Wymień sposoby zwiększania współbieżności programów za pomocą mechanizmów synchronizacji.