

Synchronization primitives for threads

Nekriach Vladyslav, TK-41



Постановка проблеми

Розпаралелювання програми дозволяє виконувати обчислення значно швидше, або дозволяє зробити процес використання програми більш зручним для користувача.

На жаль, коли програмісти додають паралелізм до і так складного програмування, знаходити помилки іноді буває дуже складно, особливо коли кодова база росте в розмірах.

Дебаггінг подібних помилок є доволі складним через комбінаторний вибух можливих послідовностей виконання тих чи інших операцій, та через можливі “перегони потоків” (race condition).



Race condition

Нехай два паралельних примітиви (також відомі як **потоки**) намагаються записати значення в загальну змінну X. Який можливий результат?

1. X = "thread_1_hello"
2. X = "thread_2_world" (як зробити так, щоб ми точно могли визначити порядок між 2 потоками?)
3. X = "thread_12_world" (як зробити так, щоб подібної ситуації не було?)
4. (можливих комбінацій більш ніж вдосталь)



Рішення: примітиви синхронізації

Для спрощення роботи програмістам науковці в 20 столітті працювали над різними примітивами синхронізації, які можна використовувати для вирішення щоденних задач.

Сьогодні ми розглянемо три примітиви:

1. Mutex
2. Semaphore
3. Conditional variable



Mutex

Mutex - примітив, який дозволяє створити шматок коду, який може виконуватися максимум 1 потоком в будь-який момент часу.

Всі інші потоки в цей час чекають, допоки цей потік не закінчить виконувати цей код.

Цей шматок коду має назву **критична секція**, а імплементація mutex складається з протоколу **входу та виходу**. Звичайний розробник ці протоколи не програмує, бо зазвичай mutex вже імплементований в мові програмування. В той же час, критичну секцію програміст задає вручну.

```
wait (mutex);  
...  
Critical Section  
...  
signal (mutex);
```



Властивості mutex більш формально

- Безпека: максимум один потік в критичній секції в будь-який момент часу
- Відсутність “дедлоків”: якщо потік знаходиться в протоколі входу нескінченно довго, то це означає, що є інший потік, який знаходиться в критичній секції нескінченно довго (читай: якийсь потік має виконувати критичну секцію якщо туди не можна зайти)
- Відсутність “лайвлоків”: жоден потік не знаходиться в протоколі входу нескінченно довго (читай: кожному потоку дають зайти в критичну секцію і він завжди рано чи пізно отримає можливість зайти в критичну секцію)



Наукова історія Mutex

- Едгар Дейкстра, 1965: Mutex з припущенням про атомічні зчитування/записи в пам'ять
 - Проблема: немає відсутності "лайвлоків"
- Леслі Лемпорт, 1974: Mutex з припущенням про single-writer, multi-reader реєстри
 - Проблема: неефективний при збільшенні кількості процесів
- Леслі Лемпорт, Джеймс Бернс, 1981: Token Mutex
- Алгоритм Деккера
- Алгоритм Петерсона



Проблеми Mutex

В Mutex є декілька проблем:

- Потоки, які очікують в протоколі входу, простоюють.
- Якщо потік випадково “помирає” в критичній секції, то вся програма опиняється в дедлоку



Mutex example

```
// lock mutex on job queue
pthread_mutex_lock(&jobQueueMutex);

// queue not null, get next job and remove it from list
nextJob = jobQueue;
jobQueue = jobQueue->next;

// unlock mutex
pthread_mutex_unlock(&jobQueueMutex);
```



Semaphore


Semaphore - примітив, який синхронізує потоки в залежності від значення спільного лічильника.

В семафора є дві операції: `wait()` та `post()`.

`post()` збільшує лічильник на 1.

`wait()` зменшує лічильник на 1, якщо він більше за 0. Якщо лічильник дорівнює нулю, то потік чекає, поки він не стане знову додатним, щоб відняти 1 і пройти далі.

Semaphore дуже гарно підходить в consumer-producer сценаріях, де є обмежена кількість спільних ресурсів, за які борються потоки.




```
// wait on jobQueue semaphore
// if value >= 1 decrement count
// if queue is empty, block until new job is added to queue
sem_wait(&jobQueueCount);

// lock mutex on job queue
pthread_mutex_lock(&jobQueueMutex);

// queue not null, get next job and remove it from list
nextJob = jobQueue;
jobQueue = jobQueue->next;

// unlock mutex
pthread_mutex_unlock(&jobQueueMutex);
```



```
// place job at head of queue
newJob->next = jobQueue;
jobQueue = newJob;

// post semaphore - another job is available
// if threads are blocked, waiting on semaphore
// one is unblocked to process the job
sem_post(&jobQueueCount);
```



Conditional variable

Conditional variable - примітив, який виступає в ролі “черги очікування”: якщо умова виконується, то потік продовжує роботу, інакше потік очікує сигналу про те, що умова має бути правдивою.

Важливі деталі:

- Conditional variable використовуються разом з mutex
- Потік не витрачає цикли CPU під час очікування в нікуди

Conditional Wait Variables

```
int is_done;  
mutex_t done_lock;  
cond_t done_cond;
```

// Thread A

```
// do the work ....  
mutex_lock(&done_lock);  
is_done = 1;  
cond_signal(&done_cond);  
mutex_unlock(&done_lock);
```

// Thread B

```
// stop here until work done  
mutex_lock(&done_lock);  
if(!is_done)  
    cond_wait(&done_cond, &done_lock);  
mutex_unlock(&done_lock);
```



Дякую за увагу!