Universidade Estadual de Campinas
Instituto de Computação
MC504 Sistemas Operacionais

T17 Thread Synchronization 1

Referência principal
Ch. 28 of *Operating Systems: Three Easy Pieces* by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 24 de setembro de 2018

Arthur João Catto, PhD

2º semestre de 2018

Background

Cooperating threads may affect and be affected by one another…

- Independent threads operate on disjoint subsets of memory.
  - So, reasoning about them differs little from reasoning about a set of independent single-threaded processes.
- Most multi-threaded programs have per-thread state (e.g., a thread's stack and registers) and shared state (shared variables on the heap).
- Cooperating threads read and write shared space and, thus, affect or can be affected by other simultaneously running threads.
- Concurrent access to shared data may result in data inconsistency.
  - Thus, maintaining data consistency requires mechanisms to ensure orderly execution of cooperating threads.

Writing correct programs involving multiple cooperating threads is not that easy...

- We are used to thinking "sequentially" when reasoning about programs.
- Unfortunately, this does not work for cooperating threads because...
  - Program execution depends on the possible interleavings of threads' access to shared state.
  - Program execution can be nondeterministic.
  - Compilers and processor hardware can reorder instructions.
- Let us have a closer look at each of these situations.

Program execution depends on the possible interleavings of threads' access to shared state.

- Assuming that x is a shared variable on the heap…

Thread A	Thread B
…	…
`x = 1;`	`x = 2;`
…	…

What will be the final value of x?

Program execution can be nondeterministic.

- Different runs of the same program may produce different results, because…
  - The scheduler may make different scheduling decisions.
  - The processor may run at a different frequency.
  - Another concurrent program may affect the cache hit rate.
  - Debugging techniques may affect the program's behavior.
- *Heisenbugs* (bugs that disappear or change when examined) are much more difficult to diagnose than *Bohrbugs* (deterministic bugs).

How can we debug programs whose behavior changes across runs?

Compilers and processor hardware can reorder instructions.

- Modern compilers and hardware reorder instructions to improve performance.
  - This is generally invisible to single-threaded programs but can become visible when multiple threads interact via shared variables.

Thread A	Thread B
…	…
`p = someComputation();`	`while (!pInitialized)`
`pInitialized = true;`	`;`
…	`q = anotherComputation(p)`
	…

Can we be sure that p has already been initialized when q is calculated?

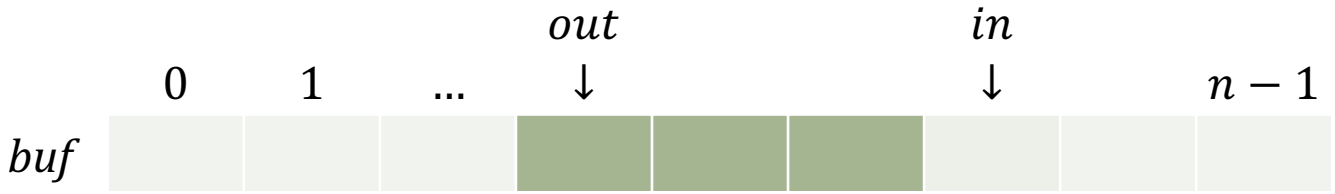
We will study a structured synchronization approach to sharing state in multi-threaded programs.

- By a *structured synchronization approach* we mean…
  - To structure the program to facilitate reasoning about concurrency.
  - To use a set of standard synchronization primitives to control access to shared state.
- Our goal is to be able to reason about a thread's behavior while avoiding the danger of having to consider all the possible interleavings that may occur.

Example

The Producer–Consumer problem

- A *producer* generates items that will be used by a *consumer*.
- The basic solution to this problem uses a bounded circular buffer mapped onto an array to enable threads to share memory.



- In this model, ***in*** and ***out*** are increased ***modulo n*** and point to the cell where the next item should be put in or taken from.

How do the producer and consumer check if the buffer is empty?

How do they check if the buffer is full?

Example

The Producer–Consumer problem

- The basic solution misses one place in the buffer to be able to distinguish between an empty and a full buffer.

Could we design a solution that fills the buffer completely?

  - Yes, we just need an integer $count$ that keeps track of the number of full slots.
    - Initially, $count$ is set to 0.
    - The producer increments $count$ after saving a new item from $nextProduced$.
    - The consumer decrements $count$ after releasing an item to $nextConsumed$.

Suggested Producer and Consumer Routines

Producer

```
while (true) {
   /* receive an item from nextProduced */
   while (count == BUFFER_SIZE)
      /* do nothing */ ;
   buffer[in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
   count++;
}
```

Consumer

```
while (true) {
   while (count == 0)
      /* do nothing */ ;
   nextConsumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   count--;
   /* release an item to nextConsumed */
}
```

What may happen if the two threads run simultaneously?

- A compiler could have translated `count++` into
  1. `register1 = count`
  2. `register1 = register1 + 1`
  3. `count = register1`
- ... and `count--` into
  1. `register2 = count`
  2. `register2 = register2 - 1`
  3. `count = register2`

Producer and consumer may now update `count` concurrently, and...

. . . get involved in a race condition over count.

- Consider `count == 5` and the following instruction interleaving

Step	Process	Instruction	count	reg1	reg2
0	producer	register1 = count	5	5	
1	producer	register1 = register1 + 1	5	6	
2	consumer	register2 = count	5	6	5
3	consumer	register2 = register2 - 1	5	6	4
4	producer	count = register1	6	6	4
5	consumer	count = register2	4	6	4

- Could the final value of `count` be different?

We say that there is a race condition when . . .

- Two or more threads share a variable

  and

  their execution is interleaved in an arbitrary manner

  so that

  they may leave the shared variable with a final value

  which

  could not be produced if they were run sequentially, in any possible order.

The Critical-Section Problem

Critical sections are code segments in two or more threads which may get involved in a race condition.

- A **solution to the critical section problem** involves the design of a protocol that such threads could use in order to avoid the creation of race conditions between them.
- A proper solution to the critical section problem must
  - Make no assumption about the threads' relative speed
  - Satisfy three additional requirements
    - **Safety** (*aka* mutual exclusion)
    - **Liveness** (*aka* progress)
    - **Bounded waiting** (*aka* starvation-free)

What's the meaning of those extra requirements?

1. **Safety** (*aka* **mutual exclusion**)
   - if a thread is executing in its critical section
     then no other thread can be executing in its critical section
2. . . .
3. . . .

Solving the Critical-Section Problem requires...

1. . . .
2. **Liveness** (*aka* **progress**)
   - if no thread is executing in its critical section
     and there are threads willing to enter their critical sections
     then one of these threads will be allowed to enter its critical section within a finite amount of time
3. . . .

Solving the Critical-Section Problem requires...

1. . . .
2. . . .
3. **Bounded Waiting** (*aka* **starvation-free**)
   - if a thread has made a request to enter its critical section
     and such request has not been granted yet
     then this thread must be allowed to enter its critical section
     within a finite amount of time

On designing a solution, we will assume that…

- The threads will be arbitrarily interleaved.
- After entering its critical section, a thread only stays there for a finite amount of time.
- Memory *fetch* and *store* are *atomic* (i.e. non-interruptible).

Also, a proper solution should be…

- **Efficient**
  - It should not consume an unreasonable amount of resources while waiting.
- **Fair**
  - It should not make some thread wait longer than others without reason.
- **Simple**
  - It should be easy to understand and use.

Solution attempt #1:

The threads use one shared flag to avoid entering the critical section at the same time.

	Thread 0	Thread 1
1.	`while (true) {`	`while (true) {`
2.	`  if (!flag) {`	`  if (!flag) {`
3.	`  flag = true;`	`  flag = true;`
4.	`  /* Critical Section */`	`  /* Critical Section */`
5.	`  flag = false;`	`  flag = false;`
6.	`  }`	`  }`
7.	`  …`	`  …`
8.	`}`	`}`

- Does this attempt satisfy the requirements that were given?

Safety (mutual exclusion) ✗

Liveness (progress)

Bounded waiting (starvation-free)

Solution attempt #2:

The threads use two shared flags to avoid entering the critical section at the same time.

	Thread 0	Thread 1
1.	`while (true) {`	`while (true) {`
2.	`  flag0 = true;`	`  flag1 = true;`
3.	`  if (!flag1) {`	`  if (!flag0) {`
4.	`  /* Critical Section */`	`  /* Critical Section */`
5.	`  flag0 = false;`	`  flag1 = false;`
6.	`  }`	`  }`
7.	`  …`	`  …`
8.	`}`	`}`

- Does this attempt satisfy the requirements that were given?

Safety (mutual exclusion) ✓

Liveness (progress) ✗

Bounded waiting (starvation-free)

Peterson's algorithm:

The threads use two shared flags and a turn indicator to avoid entering the critical section at the same time.

	Thread 0	Thread 1
1.	`while (true) {`	`while (true) {`
2.	`  flag[0] = true;`	`  flag[1] = true;`
3.	`  turn = 1;`	`  turn = 0;`
4.	`  while (flag[1] && turn == 1)`	`  while (flag[0] && turn == 0)`
5.	`  /* do nothing */ ;`	`  /* do nothing */ ;`
6.	`  /* Critical Section */`	`  /* Critical Section */`
7.	`  flag[0] = false;`	`  flag[1] = false;`
8.	`  …`	`  …`
9.	`}`	`}`

- Does this attempt satisfy the requirements that were given?

Safety (mutual exclusion) ✓

Liveness (progress) ✓

Bounded waiting (starvation-free) ✓

Peterson's algorithm: what do you say?

Peterson's algorithm works, but it is unsatisfactory.

- Solution is complicated.
- Proving correctness is tricky, even for a simple case.
- While thread is waiting, it is consuming processor time.

Can we do better? How?

- Define higher-level programming abstractions to simplify concurrent programming.
- Use hardware features to eliminate busy-waiting or at least reduce it to an absolute minimum.

Locks

Lessons from Producer-Consumer

- Solution is complicated…
  - … so "obvious code" often has bugs.
- Modern compilers/architectures reorder instructions…
  - … thus making reasoning even more difficult.
- Generalizing to many threads/processors…
  - … is even more complex: see Peterson's algorithm.

The Basic Idea

- We want any **critical section** to execute as if it were a single atomic instruction.
  - For example, consider the canonical update of a shared variable

```
4    balance = balance + 1;
```

  - Let us add a *lock* – i.e. some protective code – around the critical section

```
1    lock_t mutex; // some globally-allocated lock 'mutex'
2    …
3    lock(&mutex);
4    balance = balance + 1;
5    unlock(&mutex);
```

  - Now assume that the lock variable (e.g. `mutex`) holds *the state of the lock*.
    - `available` (or **unlocked** or **free**) → no thread holds the lock.
    - `acquired` (or **locked** or **held**) → exactly one thread holds the lock and (presumably) is in the critical section protected by that lock.

The semantics of `lock()` and `unlock()`

- `lock()`
  - The thread **tries to** acquire the lock.
  - If no other thread holds the lock, the thread will **acquire** the lock and may enter the critical section.
  - Otherwise, `lock()` will not return as long as another thread is holding the lock, thus *preventing the thread from entering the critical section*.
  - The thread that holds the lock is said to be *the owner of the lock*.
- `unlock()`
  - When the owner of the lock performs this operation, the lock becomes available again.
  - If there are other threads waiting on `lock()`, one of them will succeed and acquire the lock. Otherwise, the lock will remain available (or free).

Why only `lock` and `unlock`?

- Suppose we add another method to ask if a lock is free.
- Suppose it returns true.
- In this case, would the lock be…
  - Free?
  - Busy?
  - Don't know?

Locks exhibit three main properties

1. At most one lock holder at a time (*safety*).
2. If no one holding, acquirer gets lock (*progress*).
3. If all lock holders finish and there are no higher priority waiters, waiter eventually gets lock (*progress*).

What about starvation?

What are `mutex` variables?

- The POSIX library implements `mutex` locks to provide **mutual exclusion** between threads.

```
1    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3    Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
4    balance = balance + 1;
5    Pthread_mutex_unlock(&lock);
```

  - As before, we will add wrapper functions to our `mythreads.h` header file to deal with the potential failure of any call.

- We may be using *different locks* to protect *different critical sections*, thus adopting a more **fine-grained** approach, to increase the potential for concurrent execution.

Mutual exclusion between threads in POSIX

- The POSIX pthreads package uses **`mutex`** objects to implement mutual exclusion between threads and prevent race conditions.
  - **`mutex`** variables are one of the primary means for implementing thread synchronization and for protecting shared data when multiple writes occur.
  - A **`mutex`** variable acts like a "**`lock`**" protecting access to a shared data resource.
- The basic concept of a **`mutex`** as used in pthreads is that only one thread can lock (or own) a **`mutex`** variable at any given time.
  - Thus, even if several threads try to lock a **`mutex`** only one thread will succeed.
  - No other thread can own that **`mutex`** until the owning thread unlocks it.
  - Threads must "*take turns*" in order to access protected data.
- **`mutex`** semantics is similar to **`lock`** semantics and both work essentially in the same way.

Can we build an efficient lock?

Is there any required hardware support?

Any required OS support?