Operating System: Process Synchronization

Alex Chi

Update: March 30, 2020

Contents

1 Background

- concurrent access to shared data may result in data consistency
- maintaining data consistency requires mechanisms
- a personal note: I highly recommend reading *OSTEP* for its explanation in these synchronizing mechanisms.

2 Producer-Consumer Problem

2.1 Race Condition

- several process access and manipulate the same data concurrently
- outcome is non-deterministic
- e.g. you write `counter++`, and called this once in two threads, but you won't know finally counter would be 1 or 2. If counter is not a integer, but a structure implemented by yourself, the situation could be even worse.

2.2 Critical Section

- only one process can enter critical section
- this can be ensured by using mutual exclusion, progress, bonded waiting

    - sequential access: request acquired sequence is the same as access sequence

2.3 Mechanisms for Process Synchronization

2.3.1 Synchronize Hardware

- atomic instructions (amo.swap on RISC-V)
- memory barrier (fence, sync instruction)
- `TestAndSet` or `CompareAndSwap` (CAS) in high-level languages will be translated into the above two mechanisms.
- `TestAndSet` fetch from memory, set memory to target, and return previous memory value. This process is atomic.
- `CompareAndSwap` if `*value == expected` then set it to target. It always returns original value.
- using test-and-set
    - set `lock` to 0 initially
    - lock `while (__sync_lock_test_and_set(&lock, 1) == 1) {}`
    - unlock `__sync_lock_release(&lock)`
    - why not directly set lock to 0? Compiler and processor will do instruction reorder. Using `sync_lock_release` will add a memory barrier.
- using memory fence
    - acquire ordering when locking
    - release ordering when unlocking
- bounded-waiting Mutual Exclusion
    - refer to textbook for code
    - by selecting next running process sequentially
    - every thread will wait at most $n - 1$ entries before going into critical section
    - sequential access is not guaranteed

2.3.2 Peterson's Solution

- software solution
- won't be correct on all machine, as this requires strong memory order
- `turn` indicates whose turn
- `flag` array indicates if a process is ready to enter
- only consider exactly 2 process (not 1, not more)

```
flag[i] = true;
turn = j; // j is id of another process
while (flag[j] && turn == j);
// critical section
flag[i] = false;
```

2.3.3 Semaphore

- S A integer variable
- wait and signal
- wait: `while (S <= 0) S--;`
- signal: `S++`
- originally: P() to test, V() to increment
- implementation in OS
  - wait: ~ s->Val -= 1; if s->Val < 0 { add to list; sleep; }~
  - signal: `s->Val += 1; if s->Val <= 0 { wake up P in list }`
- in the textbook, `wait` and `signal` are considered as atomic operations.
- from my experience, semaphore is built upon spin-lock and process sleep/wake-up (kernel space)
- or using mutex and conditional$_{\text{variable}}$ (user space)
- A kernel-space semaphore example (xv6)

```
wait(sem* S) {
  acquire(S->spinlock);
  while (S->val <= 0) {
    sleep(S, S->spinlock);
  }
  // Only one process will reach this point
  S->val -= 1;
  release(S->spinlock);
  // After critical section, spinlock is released,
  // and other process being waken up will find that val is <= 0, then goes back to s
}

signal(sem* S) {
  acquire(S->spinlock);
```

```
    S->val += 1;
    wakeup(S); // Wake up all process bounded to semaphore S
    release(S->spinlock);
}
```

2.4 Semaphore as General Synchronization

- binary semaphore: initialized to 1, known as mutex locks
- counting semaphore: integer value can range over unrestricted value

2.5 Q&A

1. How can we assign `i` in bounded-waiting example program?
   We previously know what n is. In practice, there should be more implementations.
2. Requirements for Peterson's Algorithm
   - no memory reorder
   - for reordering, we can add memory barrier after `turn=j`, and before `flag[i] = FALSE`.
   - more processes?
3. Why `S->list != NULL`?
   It is possible that semaphore is initialized with negative value. In this case, there will be fewer processes in list than abs(value).
4. All-time sleep?
   This is a dead-lock. Which means that a thread goes into sleep, but no one will wait it up. Its possible if you doesnt pair lock with unlock.
5. What will happen after signal?
   Only one process will go into critical section.

3 Classic Problems of Synchronization

3.1 Producer-Consumer problem

- wrap counter++ / – with mutex.
- use semaphore `full` and `empty`.
  - initialize: empty = N, full = 0
  - producer: wait empty, signal full

    - consumer: wait full, signal empty

3.2 Dining-Philosophers Problem

- refer to figure in slides
- naive solution: if everyone gets their left-hand chopstick, therell be deadlock.
- improved solution
    - get two chopsticks at same time
    - a guarding process ask someone to release lock (introduce a waiter)
    - give up chopstick after certain time
    - limit number of diners
    - some grab left first, some grab right first

3.3 Readers-Writers Problem

- allow multiple readers to read at the same time
- only one single writer can access shared data
- shared data
    - semaphore wrt initialized to 1
    - integer readcount initialized to 0
    - mutex initialized to 1 (protect readcount)

3.3.1 Solution

- writer: `wait wrt; write; signal wrt;`
- reader: `with mutex locked { if first reader, wait wrt }; read; { if last reader, signal wrt; }`
- writer will get starved
- as long as there is one reader, reader can always get into critical section
- reader-preferred solution
- another solution: RCU (read-copy-update), or RwLock

3.3.2 Writer-Preferred Solution

- semaphore: mutexrc=1, mutexwc=1, wrt=1, rd=1
- readcount=0, writecount=0

- writer `with mutexwc locked { if first writer, wait rd } with wrt locked { critical section } with mutexwc locked { if last writer, signal rd }`
- reader `with rd locker { with mutexrc locked { if first reader, wait wrt }}`

3.3.3 Without Starvation

- semaphore mutex=1, wrt=1, rd=1
- readcount=0
- writer: `with wrt locked { with rd locked { write }}`
- reader: `with wrt locked { with mutex locked { get readcount } if first reader, wait rd} read with mutex locked { get readcount } if last reader { signal rd }`
- basic idea: make reader and writer both wait for wrt mutex, thus forming a queue.
- basic principle: make critical section shorter.

3.4 Q&A

1. Why not protect counter when reading?
   Even if theres a racing condition (not the latest data is read), it doesnt matter to both producer and consumer.
2. Why should we use read mutex in writer-preferred solution?
   Theres possibility that two readers complete read at the same time.
3. Why should we use write mutex in writer-preferred solution?
   One writer completes write, and another writer spends much less time after the former one. In this case, the previous writer and the new one may change writecount at the same time.

3.4.1 A side note

- To solve this problems, you should select your way of abstraction. For example, in diner's problem, we consider each philosopher as a thread, and solve the synchronization problem.
- And we may implicitly abstract some problems from the question. For example, we implicitly implemented sleep and barber chair in barber's problem.
- Therefore, I don't give solution for all these problems, as how much you consider the problem in detail is up to you. There's no standard answer.

4 Deadlock and Starvation

- P0: wait(S); wait(Q);
- P1: wait(Q); wait(S);
- they will wait for each other forever!

5 Monitors

- a high-level abstraction for process synchronization
- only one process may be active in monitor
- refer to *OSTEP*
- `monitor name { variables; procedure; init; }`

6 Condition Variables

- yet another synchronization scheme provided by OS
- OS synchronization mechanisms: semaphore, mutex, conditional variable
- kernel synchronization mechanisms: spinlock
- `x.wait()` suspend current process
- `x.signal()` resume suspended process
- a side note: typically condition variable is used with mutex
  - why? refer to *OSTEP*
    - prevent lost wakeup (if a process is in process of being slept, signal cannot be called with mutex held)
    - when broadcasting, allow only one process to get into critical section (use cond variable with loop)
  - e.g. `cond_wait(&cond, &mutex)`
- options
  - signal and wait (signal process waits)
  - signal and continue (signal process continues)

6.1 Solution to Dining Philosophers' Problem

- refer to slides
- putdown wake up neighbors

- pickup waits if philosopher cannot eat for now
- only one function runs at a time
- very similar to async programming (`.wait` like await, `.signal` like future.done)
- suffers from starvation, there might be a philosopher whose neighbor is always eating
- basic idea: don't directly use mutex to represent chopstick. Instead, use mutex (or monitor) to protect chopstick status.

6.1.1 Q&A

1. Will all philosophers execute state[i]=eating?
   Nope. Monitor only executes one function.
2. Can I remove wait and signal?
   Wait and signal help suspend current execution until a philosopher is possible to pick up. This is for synchronization