DM510: Process Synchronization

Lars Rohwedder

Disclaimer

These slides contain (modified) content and media from the official Operating System Concepts slides: https://www.os-book.com/OS10/slide-dir/index.html

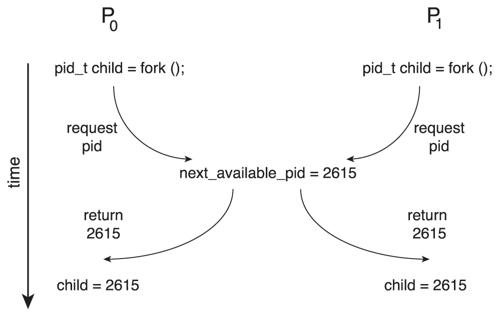
Today's lecture

- Chapter 6+7 of course book
- Due to overlap with **Concurrent Programming** course, we focus on system view

The Problem

Race condition

- Concurrency / parallelism can cause **race conditions**: different (unintended) behavior depending on timing of process execution and preemption
- Such errors are extremely difficult to reproduce and debug
- Can be issue in both kernel code and user code



Example

- Two processes might execute `fork()` at similar times
- Without proper mechanisms, they could obtain same pid for child
- Modern operating systems ensure system calls are thread-safe (no race conditions)

High-Level Mechanisms

Critical section

- Each program defines "critical sections" during which it works on shared memory

```
...
enter_critical_section()

perform some operation on shared memory

exit_critical_section()
...
```

Properties

- **Mutual exclusion:** only one process is in a critical section at any time
- **Progress:** processes do not wait indefinitely while there is no process in a critical section
- **Bounded waiting:** for a process waiting to enter critical section, number of other processes that can enter before it is bounded

Mutex

A mutex ("mutual exclusion") is an object that has two operations:

- `aquire()`
- `release()`

A mutex has a binary value B (initialized with `true`). The operations behave as follows

aquire()

```
while (!B)
        ; /* wait */
B = false;
```

release()

```
B = true;
```

Semaphores

More powerful than mutex. Again two operations:

- `wait()`
- `signal()`

A semaphore has a counter S. The operations behave as follows

wait()

```
while (S <= 0)
        ; /* wait */
S--;
```

signal()

```
S++;
```

Semaphore example: bounded buffer

(From previous lecture on process communication) given: buffer holding `BUFLEN` elements, producers that add elements and consumers that remove elements.

```
item buffer[BUFLEN];
int in = 0;
int out = 0;
```

- semaphore `mutex` initially 1
- semaphore `full` initially 0
- semaphore `empty` initially `BUFLEN`

Producer

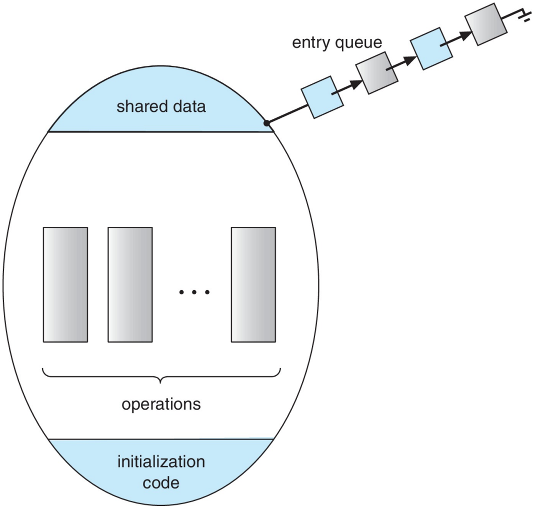
```
while (true) {
  item next_produced = produce();
  wait(&empty);
  wait(&mutex);
  buffer[in] = next_produced;
  in = (in + 1) % BUFLEN;
  signal(&mutex);
  signal(&full);
}
```

Consumer

```
while (true) {
  wait(&full);
  wait(&mutex);
  item next_consumed = buffer[out];
  out = (out + 1) % BUFLEN;
  consume(next_consumed);
  signal(&mutex);
  signal(&empty);
}
```

Monitors

- An object together with data (variables) and operations
- Only once thread at a time can execute an operation



Implementation Details of Synchronization

Disabling interrupts

- Simple way to implement critical sections (on single-core machines): disable interrupts when entering critical section, enable when leaving
- Process will finish critical section before anything else is performed

Problems

- When disabling interrupts for a longer duration: computer system not responsive; frequent timer interrupts are used to update system clock, which no longer happens; …
- On multi-core systems another process may still run in parallel, making this approach insufficient

Atomic instructions

- An operation on shared memory is **atomic** if at any time from the perspective of another thread, it is either not performed at all or completely performed

- Processor architectures typically implement basic memory operations (load and store) atomically
- Special sophisticated atomic instructions are often available (architecture dependent)
- <stdatomic.h> for portable C code
- Slower than non-atomic variant

Typical atomic instructions

- `increment(int* arg)`: increase `*arg` by 1
- `exchange(int* obj, int newval)`: set `*obj = newval` and return previous value of `*obj`
- `compare_and_swap(int* obj, int oldval, int newval)`: if `*obj == oldval`, set `*obj = newval` and return `true`, otherwise return `false`
- `compare_and_exchange(int* obj, int* oldval, int newval)`: if `*obj == *oldval`, set `*obj = newval` and return `true`, otherwise `*oldval = *obj` and return `false`

Spinlocks

Busy waiting: thread keeps running (e.g. polling the status of a mutex) while it is blocked

Naive implementation of aquire()

```
while (!B)
        ; /* wait */
B = false;
```

Prone to race conditions!

Correct implementation of aquire()

```
while (!compare_and_swap(B, true, false))
        ; /* wait */
```

- Although busy waiting seems wasteful at first, it is perfectly suitable in many situations, especially when it is unlikely that other process holds lock
- Implementation above might not guarantee bounded waiting

Bounded waiting with spinlocks

```
bool waiting[n];
bool B;

/* thread i */
while (true) {
   waiting[i] = true;
   while (waiting[i] && !compare_and_swap(&B,false,true);)
         ;
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      B = false;
   else
      waiting[j] = false;
   /* remainder section */
}
```

Memory barriers

Busy waiting also without sophisticated instructions (e.g. `compare_and_swap()`) is possible, see e.g. Dekker's algorithm in exercises. But this is dangerous:

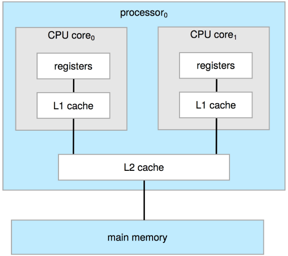
Code

```
bool flag = false;
int x = 0;

/* Thread 1 */
while (!flag)
        ;
/* expected output: 100 */
print x;

/* Thread 2 */
x = 100;
flag = true;
```

Memory barriers

Busy waiting also without sophisticated instructions (e.g. `compare_and_swap()`) is possible, see e.g. Dekker's algorithm in exercises. But this is dangerous:

- **Problem:** modern architectures and compilers may reorder instructions or not all updates visible yet due to caching ...



Code

```
bool flag = false;
int x = 0;

/* Thread 1 */
while (!flag)
        ;
/* expected output: 100 */
print x;

/* Thread 2 */
x = 100;
flag = true;
```

Memory barriers

Busy waiting also without sophisticated instructions (e.g. `compare_and_swap()`) is possible, see e.g. Dekker's algorithm in exercises. But this is dangerous:

- **Problem:** modern architectures and compilers may reorder instructions or not all updates visible yet due to caching ...
- possible output: 0!

Code

```
bool flag = false;
int x = 0;

/* Thread 1 */
while (!flag)
        ;
/* expected output: 100 */
print x;

/* Thread 2 */
x = 100;
flag = true;
```

Memory barriers

Busy waiting also without sophisticated instructions (e.g. `compare_and_swap()`) is possible, see e.g. Dekker's algorithm in exercises. But this is dangerous:

- **Problem:** modern architectures and compilers may reorder instructions or not all updates visible yet due to caching ...
- possible output: 0!
- **Solution:** architectures provide memory barrier instruction that guarantees that all memory operations before it are executed before continuing

Code

```
bool flag = false;
int x = 0;

/* Thread 1 */
while (!flag)
        ;
/* expected output: 100 */
print x;

/* Thread 2 */
x = 100;
flag = true;
```

Synchronization without busy waiting

In some situations (e.g. locks are kept for a long time or single-core processor) busy waiting is not a sensible option.

Alternative: use **system calls** to scheduler

- **block():** ask kernel scheduler to not execute process anymore by putting it into a waiting queue
- **wakeup():** move process from waiting queue to ready queue

Disadvantages

- High overhead (user-kernel mode switch, context switches, etc.)

Summary

Disabling interrupts

Bad for: multi-core systems, long critical sections

Spinlocks

Bad for: single-core systems, long critical sections

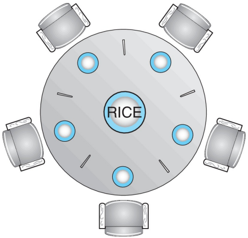
Scheduler requests (block/wakeup)

Bad for: short critical sections

Deadlocks

Dining philosophers

- 5 philosophers alternatingly think and eat
- To eat they need to pick up their left and right chopstick (one at a time)
- Chopsticks (implemented as mutexes) are shared with the neighbors




```
/* Algorithm for philosopher i */
while (true) {
        aquire(&chopstick[i]);
        aquire(&chopstick[(i+1)%5]);
        eat();
        release(&chopstick[i]);
        release(&chopstick[(i+1)%5]);
        think();
}
```

This code has a problem. What is it? (more next lecture)