Synchronization

DM510 Operating Systems

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

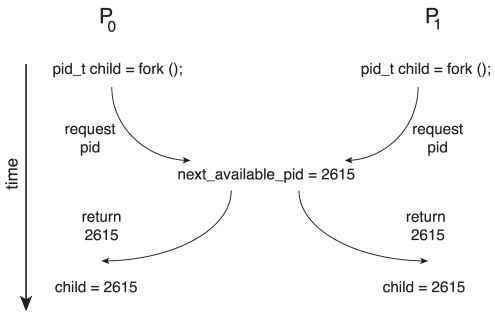
Today's lecture

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

The Problem

Race condition

- Concurrency and 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)

Synchronization Mechanisms

Critical section

- Each program defines “critical sections” and only during these it accesses shared resources

```
...
enter_critical_section()

perform some operation on shared resources

exit_critical_section()
...
```

Semantics

- **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

Can be used to avoid race conditions, but may be overprotective: also when processes work on **different** shared resources, they cannot work in parallel

Mutex

A mutex (for “mutual exclusion”) is an object that has two operations:

- `acquire()`
- `release()`

Semantics

- When several processes call `acquire()`, only one returns from the call; all others are waiting inside the call
- When `release()` is called, another process can return from `acquire()`
- Often bounded waiting similar to critical sections is guaranteed

Using one global mutex, we can implement critical sections by calling `acquire()` on entering it and `release()` on exit. But we can also use different mutexes per shared resource.

Semaphore

A semaphore has a counter variable and two operations:

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

Semantics

- when a process calls `wait()`, it waits until the counter is larger than zero, then decreases it by one and returns
- `signal()` increases the counter of the semaphore by one
- Often bounded waiting is guaranteed

Binary semaphores where the counter is always zero or one behave exactly like mutexes

Semaphore example: bounded buffer

We revisit the example from the lecture on processes

Given: buffer holding BUFLEN items, 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

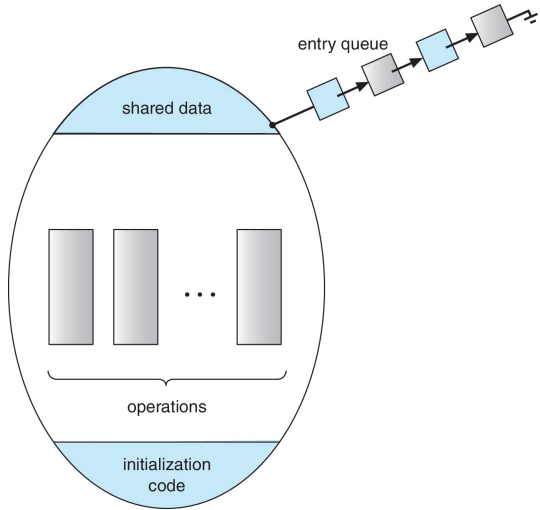
```
while (true) {
  item next_produced = produce();
  wait(&empty);
  wait(&mutex);
  buffer[in] = next_produced;
  in = (in + 1) % BUFLEN;
  signal(&mutex);
  signal(&full);
}
```

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
- Integrated for example in Java with the synchronized keyword



Implementation

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, system clock does not get updated (relies on frequent timer interrupts), ...
- 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
- More sophisticated atomic instructions are often available (architecture dependent)
- `<stdatomic.h>` for portable C code
- Slower than non-atomic variant: requires locking memory bus, which means other cores cannot execute memory operations in parallel

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 checkin status while it is waiting

To implement mutex, use binary variable B that indicates status

Naive implementation of acquire(mutex* mu)

```
while (mu->B)
        ; /* wait */
mu->B = 1;
```

Incorrect because of race condition!

Spinlocks

Busy waiting: thread keeps checkin status while it is waiting

To implement mutex, use binary variable B that indicates status

Naive implementation of acquire(mutex* mu)

```
while (mu->B)
        ; /* wait */
mu->B = 1;
```

Incorrect because of race condition!

Correct implementation of acquire(mutex* mu)

```
while (!compare_and_swap(&mu->B, 0, 1))
        ; /* wait */
```

Spinlocks

Busy waiting: thread keeps checkin status while it is waiting

To implement mutex, use binary variable `B` that indicates status

Naive implementation of acquire(mutex* mu)

```
while (mu->B)
        ; /* wait */
mu->B = 1;
```

Incorrect because of race condition!

Correct implementation of acquire(mutex* mu)

```
while (!compare_and_swap(&mu->B, 0, 1))
        ; /* wait */
```

`release()` can be safely implemented as `mu->B = 0`

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

Bounded waiting with spinlocks

```
int waiting[n] = {0}; /* Initialized as false */
int B = 0;

void thread(int i) {  /* i = 1,2,...,n is thread id */
        while ( 1 ) {
           waiting[i] = 1;
           while (waiting[i] && !compare_and_swap(&B,0,1);)
                ;
           waiting[i] = 0;
           /* critical section */
           j = (i + 1) % n;
           while ((j != i) && !waiting[j])
                j = (j + 1) % n;
           if (j == i)
                B = 0;
           else
                waiting[j] = 0;
           /* remainder section */
        }
}
```

Memory barriers

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

Code

```
int flag = 0;
int x = 0;

void thread1() {
        while (!flag)
                ;
        /* expected output: 100 */
        printf("%d", x);
}

void thread2() {
        x = 100;
        flag = 1;
}
```

Memory barriers

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

> **Problem 1:** modern compilers may reorder instructions if variable dependence within current function allows it

Code

```
int flag = 0;
int x = 0;

void thread1() {
        while (!flag)
                ;
        /* expected output: 100 */
        printf("%d", x);
}

void thread2() {
        x = 100;
        flag = 1;
}
```

Memory barriers

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

- **Problem 1:** modern compilers may reorder instructions if variable dependence within current function allows it
- **Problem 2:** depending on CPU architecture and its handling of caches, some cached data might be outdated

Code

```
int flag = 0;
int x = 0;

void thread1() {
        while (!flag)
                ;
        /* expected output: 100 */
        printf("%d", x);
}

void thread2() {
        x = 100;
        flag = 1;
}
```

Memory barriers

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

- **Problem 1:** modern compilers may reorder instructions if variable dependence within current function allows it
- **Problem 2:** depending on CPU architecture and its handling of caches, some cached data might be outdated
- Output may be 0!

Code

```
int flag = 0;
int x = 0;

void thread1() {
        while (!flag)
                ;
        /* expected output: 100 */
        printf("%d", x);
}

void thread2() {
        x = 100;
        flag = 1;
}
```

Memory barriers

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

- **Problem 1:** modern compilers may reorder instructions if variable dependence within current function allows it
- **Problem 2:** depending on CPU architecture and its handling of caches, some cached data might be outdated
- Output may be 0!
- **Solution:** architectures provide memory barrier instruction that guarantees that all memory operations before it are executed before continuing

Code

```
int flag = 0;
int x = 0;

void thread1() {
        while (!flag)
                ;
        /* expected output: 100 */
        printf("%d", x);
}

void thread2() {
        x = 100;
        flag = 1;
}
```

Synchronization without busy waiting

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

Alternative: use **system calls** to make requests to CPU 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

Next Lecture

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) {
        acquire(&chopstick[i]);
        acquire(&chopstick[(i+1)%5]);
        eat();
        release(&chopstick[i]);
        release(&chopstick[(i+1)%5]);
        think();
}
```

There is a problem with this code. Can you spot it?