CS 350 Spring 2023:
Lecture Notes

1 Operating Systems Introduction **2**

2 Threads **4**

3 Synchronization **7**

Back Matter **10**

Index of Defined Terms 10

Lecture notes taken, unless otherwise specified, by myself during section 001 of the Spring 2023 offering of CS 350, taught by Kevin Lanctot.

Lectures

Lecture 1 (05/09) 2
Lecture 2 (05/11) 3
Lecture 3 (05/16) 5
Lecture 4 (05/18) 5
Lecture 5 (05/25) 7
Lecture 6 (05/30) 7
Lecture 7 (06/01) 8

Chapter 1

Operating Systems Introduction

Lecture 1 (05/09)

Generally, an operating system acts partially as a cop (e.g., watching for unplugged USB drives) and as a facilitator (e.g., allowing you to interface with any storage device with `fopen`). It is responsible for:

- managing resources;
- creating execution environments;
- loading programs; and
- providing common services and utilities

We will consider an OS from three views:

1. Application: what does an OS provide? Provides an execution environment which provides resources, interfaces, and isolation.
2. System: what problems does an OS solve? Manages hardware resources, allocates them to programs, and controls access to shared resources between programs.
3. Implementation: how is an OS built? It must be concurrent (allow multiple things to run at once) and real-time (respond to events in a set time).

Definition (*kernel*)

The part of the operating system that responds to system calls, interrupts, and exceptions.

Definition (*operating system*)

Includes the kernel, but also other related programs that provide services for applications. For example, utility programs, command interpreters, and programming libraries.

The kernel protects from bad user programs by isolating them in user space (resp. kernel space) and allowing them only to interact with hardware using system calls.

Definition (*system call*)

A user interaction with the OS. For example, the C function `fopen` makes the Linux syscall `sys_open`. They are much slower than calling a user function.

Definition (*types of kernels*)

- Monolithic: when the entire OS is the kernel (e.g. Linux)
- Microkernel: when only absolutely necessary parts are in the kernel
- Hybrid: somewhere between monolithic kernels and microkernels (e.g. Windows, macOS)
- Real-time: with stringent event response time, guarantees, and scheduling

A monolithic kernel is faster, since we avoid slower system calls. However, they are less secure since third-party drivers must be trusted and included in the kernel.

Lecture 2 (05/11)

Provided by the execution environment are abstract entities that a program is able to manipulate:

- files and file systems (secondary storage; e.g. HDDs)
- address spaces (primary memory; RAM)
- processes, threads (program execution)
- sockets, pipes (message channels)

Chapter 2

Threads

Definition (*thread*)

Sequence of instructions

An ordinary sequential program has only a single thread. Analogous to how DFAs have a single state and NFAs can follow multiple paths, a program can have multiple threads of execution. The threads can be for the same role (e.g. one per server visitor) or for different roles (e.g. Chrome's JavaScript and graphics engines).

Threads allow for:

- concurrency: allow multiple tasks to occur at once
- parallelism: different threads on different processors to increase throughput
- responsiveness: allow blocking tasks to not block the whole system
- priority: do things that are more important first
- modularity: separate out tasks into threads that can't crash each other

A thread will pause execution when it is blocked.

Consider for example the traffic simulation for Assignment 1. Each thread represents a vehicle passing through an intersection, and we are trying to prevent collisions.

A thread can create a new thread using `thread_fork`. The original and new threads share global data and the heap. However, the new thread has a separate, private stack.

For example, in `/kern/synchprobs/traffic.c`:

```
for (i = 0; i < NumThreads; i++) {
  error = thread_fork("vehicle_simulation thread", NULL, vehicle_simulation,
                      NULL, i);
  if (error) {
    panic("traffic_simulation: thread_fork failed: %s\n", strerror(error));
  }
}
```

we start a thread running `vehicle_simulation(NULL, i)`.

In OS/161, we create a thread with

```
int thread_fork(
  const char *name,
```

```
    struct proc *proc,
    void (*func)(void *, unsigned long),
    void *data1,
    unsigned long data2
);
```

and can terminate with `void thread_exit(void)` and yield with `void thread_yield(void)`. However, we cannot control the order that threads run in.

Recall from CS 241 how to execute a single thread: fetch–execute cycle. In CS 241, we called all the registers $\$0, \ldots, \$31$. In real life, they have names like a0 and s8.

CS 241 passed all arguments via the stack. We will pass the first four as a0 to a3 and the rest on the stack.

Lecture 3 (05/16)

Recall: functions push arguments (not a0-a3), return address, local variables, and temporary-use registers onto the stack.

With multiple threads, we need multiple stacks. When swapping threads, save the value of registers to the stack and then load from the other stack.

The threads share the same code, global read-only data, global data, and heap. They have their own stacks and program counters. Since we might have lots of threads, each stack has a fixed size (e.g. 2 MB).

We can multithread a core by having multiple sets of registers but share an ALU, control unit, etc. by using the hardware when waiting for LW and SW instructions. Therefore, given P processors, each with C cores and M multithreads per core (almost always 2), we can execute PCM threads (truly) simultaneously.

Definition (*timeshare*)

> Switching rapidly from one thread to another. During a context switch, we schedule which thread runs next, save the register contents of the current thread, and load the register contents of the next thread. The saved/restored contents are also called the thread context.

The C function `thread_switch` saves and restores caller-save registers, and calls the assembly language subroutine `switchframe_switch` which saves and restores callee-save registers.

Lecture 4 (05/18)

There are four ways a context switch can be triggered:

1. voluntarily, via `thread_switch`;
2. by termination, via `thread_exit`;
3. when a thread is blocked, via `wchan_sleep`;[1] or
4. by preemption, when the thread schedule involuntarily stops it.

A thread can be either:

- running, currently executing on the processor;
- ready, waiting in a ready pool; or
- blocked, waiting for something to happen and not ready to execute

Running to blocked via `wchan_sleep`, blocked to ready via `wake_one` or `wake_all`, and ready to running via `dispatch`. A running thread can become ready by preemption or

[1] where `wchan` stands for "wait channel"

via `thread_yield`.

Chapter 3

Synchronization

Lecture 5 (05/25)
Lecture 6 (05/30)

todo: slides 90 to 99

Definition (*race condition*)

A program where the order of execution affects the program result. The pieces of code that can create a race condition by accessing a shared variable are called critical sections.

We can use a lock to provide mutual exclusion (when exactly one of the code chunks runs in its entirety):

```
int volatile total = 0;
bool volatile total_lock = false; // false means unlocked
                                  // true means locked
void add() {                    void sub() {
  int i;                          int i;
  for (i=0; i<N; i++) {           for (i=0; i<N; i++) {
    Acquire(&total_lock);           Acquire(&total_lock);
    total++;                        total--;
    Release(&total_lock);           Release(&total_lock);
  }                               }
}                               }
```

We can imagine implementing `Acquire` and `Release` by spinning until we can proceed (this is a spinlock):

```
Acquire(bool *lock) {
  while (*lock == true) {} // spin until the lock is free
  *lock = true;            // grab the lock
}

Release(bool *lock) {
  *lock = false;           // give up the lock
}
```

This does not actually work because `Acquire` could be preempted before grabbing the lock. There are special assembly language instructions which allow us to make this work and create a spinlock.

Since spinlocks use the processor while they wait, they should not be used for long waiting times. While a spinlock is spinning, we must also disable interrupts.

In OS/161, we can create and manipulate `spinlock` structs using the methods `spinlock_init`, `spinlock_acquire`, and `spinlock_release`.

We can instead block the thread instead of eating up CPU time. The OS/161 methods `lock_create`, `lock_acquire`, and `lock_release` are analogous to the `spinlock` methods. A lock is owned by a thread (since it only blocks a single thread) and not a CPU (since a spinlock takes up an entire CPU while spinning).

While a thread is blocked by a lock, it goes on a wait channel for that lock, i.e., a queue of threads that will be awoken when the lock is released. In OS/161, we can use `wchan_lock`, `wchan_sleep`, `wchan_wakeall`, and `wchan_wakeone`.

If we need something more complex than a boolean lock, we can use a semaphore, which holds an integer. We can either call `P`(roberen) (lit. "try", waits until able to decrement) or `V`(erhogen) (lit. "increase", increments).

There are three kinds of semaphore:

- binary: 0 or 1, behaves like a lock
- counting: an arbitrary number of resources
- barrier: force one thread to wait for others to complete, start count at 0

We do not need to call `V` after `P`. We can also start at whatever initial value we want. Semaphores also do not have owners.

Lecture 7 (06/01)

In summary:

Spinlocks	Locks	Sempahores
Consumes a CPU	Owned by a thread	No ownership
Spins (no interrupts)	Blocks	Blocks
Binary (held/not held)	Binary	Non-negative integer

We implement semaphores and locks using spinlocks and wait channels because we do not want a race condition on `sem->sem_lock`. Simplified:

```
void P(struct semaphore *sem) {
  spinlock_acquire(&sem->sem_lock);     // lock the semaphore
  while (sem->sem_count == 0) {         // check if resources available
    wchan_lock(sem->sem_wchan);         // lock wait channel's queue
    spinlock_release(&sem->sem_lock);   // unlock the semaphore
    wchan_sleep(sem->sem_wchan);        // context switch (unlocks queue)
    spinlock_acquire(&sem->sem_lock);   // relock the semaphore
  }
  sem->sem_count--;                     // use a resource
  spinlock_release(&sem->sem_lock);     // unlock the semaphore
}
```

and

```
void V(struct semaphore *sem) {
  spinlock_acquire(&sem->sem_lock);     // lock the semaphore
  sem->sem_count++;                     // add a resource
  wchan_wakeone(sem->sem_wchan);        // unblock a thread on the queue
  spinlock_release(&sem->sem_lock);     // unlock the semaphore
}
```

We can abstract away from integers entirely and have a condition variable. When a condition is true, the thread can run; when it is not true, the thread waits until it becomes true. If a thread sets the condition to true, it can `signal` one or `broadcast` all blocked threads.

Two threads can deadlock if they are trying to acquire locks held by each other. To avoid this, either have a standard order of acquisition or do retries (No Hold and Wait):

```
lock_acquire(lock1);
while (!try_acquire(lock2)) {
  lock_release(lock1);
  lock_acquire(lock1);
}
```

but OS/161 does not have `bool try_acquire(struct lock *)`.

Index of Defined Terms

blocking, 4

concurrency, 2, 4
condition variable, 9
context switch, 5
critical section, 7

deadlock, 9

execution environment, 2

kernel, 2
    hybrid kernel, 3
    microkernel, 3
    monolithic kernel, 3
kernel space, 2

lock, 7

modularity, 4
multithread, 5
mutual exclusion, 7

operating system, 2

parallelism, 4
preemption, 5
priority, 4

race condition, 7
real-time, 2
real-time operating system, 3
responsiveness, 4

semaphore, 8
    barrier, 8
    binary, 8
    counting, 8
sequential program, 4
spinlock, 7
system call, 2

thread, 4
    blocked, 5
    ready, 5
    running, 5
thread context, 5
timeshare, 5

user programs, 2
user space, 2

wait channel, 8