# CS 350 Spring 2023:
## Lecture Notes



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

### Lectures

# 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`.

# Index of Defined Terms