CS162: Operating Systems and Systems Programming

Michael Pham

Fall 2024

Contents

Week 1

An Introduction

1.1 Lecture – 8/29/2024

1.1.1 Course Information

! Note that the course is an **Early Drop Deadline** course; the deadline is on September 6th.

1.1.2 Basics

What is an operating system? Well, first, we have hardware; on the other hand, we also have the application which must run on the hardware.

How do these applications run on the hardware? This is where OS comes in, enabling applications to run on and share the hardware.

Definition 1.1 (Operating System)**.** It's a special layer of software that provides application software access to hardware resource.

- It's a convenient abstraction of hardware devices, which may be complex.
- Since multiple applications can be running at once, we must have protected access to shared resources so that they don't "step on each others' toes".
- Security and authentication.
- Communication amongst logical entities.

An OS does the following:

- Provides abstractions to applications.
  - File systems
  - Processes, threads
  - VM, containers
  - Naming systems
- Manage resources

  - Memory, CPU, storage, etc.

- Achieves the above by implementing specific algorithms and techniques
  - Scheduling
  - Concurrency
  - Transactions
  - Security

To elaborate, we note that an OS:

- Provides clean, easy-to-use abstractions of physical resources
  - Infinite memory, dedicated machine
  - Masking limitations
- Manage protection, isolation, and sharing of resources
  - Resource allocation and Communication
- Glue
  - Storage, Window system, Networking
  - Sharing, Authorization
  - Look and feel

Von Neumann Architecture

In this architecture, we have the CPU – comprised of the Control and Logic units – which takes in some input and puts out an output. At the same time, the program is saving states within the memory.

Processes

> **Definition 1.2** (Process)**.** A process is an execution environment with restrict rights provided by the OS.

A process consists of the following things:

- Address Space
- One or more threads of control executing within that address space

The Operating System provides each running program with its own process.

Now, note that a processor can only execute one process at a time; to combat this issue, we can do something called "context switching."

Also, we note that each program should not be able to access the memory of other programs (or even the OS' memory itself); this is where the OS comes into play. It isolates processes from each other, and itself from processes as well.

Week 2
Second Week Woes

2.1 Lecture – 9/3/2024

2.1.1 Review

Recall that virtualization provides the illusion where each process uses its own machine; it further provides the illusion of infinite memory and processor.

Furthermore, recall that the hypervisor creates virtual machines, which virtualizes hardware to OS. And in fact, virtual machines can be implemented on top of an existing OS too.

2.1.2 The Four Abstractions

We will be looking at the following four abstractions:

- Thread: Execution Context
  - Fully describes the program state
  - Program Counter, Registers, etc.
- Address Space
  - Set of memory addresses which is accessible to the program
- Process: an instance of a running program
  - Protected address space, and one or more threads.
- Dual Mode Operation/Protection

61C Review

Recall that for a program, we first write then compile them, then load the instruction and data segments of executable files into memory.

Then, we create stack (which goes from high to low; this includes things like function arguments, return addresses, etc.) and heap (which goes from low to high; this is when we use `malloc`).

We then transfer the control to program, and provide services to the program.

Recall that we fetch instruction from memory, decode it, then execute it.

2.1.3 Thread

Definition 2.1 (Thread)**.** A thread is a single unique execution.

- Program Counter, Registers, Execution Flags, Stack, Memory State

We note that a thread is executing on a processor (core) when it is *resident* in the processor registers.

Definition 2.2 (Resident)**.** Resident means: Registers hold the root state (context) of the thread.

- This includes the PC and currently executing instruction.

The Illusion of Multiple Processors

What we can do is multiplex in time; think of splitting the CPU in time-slices.

Threads are virtual codes. Contents include program counter, stack pointer, and registers.

The thread is on the real (physical) core, or is saved in chunk of memory ("TCB").

In order to trigger this switch, including: timer, I/O, voluntary yield, etc.

The Thread Control Block holds the contents of registers when thread isn't running, and is stored (for now) in the kernel.

! The kernel is part of the OS, representing the core functionalities. There are other services of the OS which doesn't run all the time; these are not part of the kernel.

2.1.4 Address Space

Definition 2.3. The address space is the set of accessible addresses and state associated with them.

Example 2.4. For example, on a 32-bit processor, we have 2^{32} addresses; on 64-bit processors, we have 2^{64}.

Recall that our operating system must protect itself from user programs, and must protect user programs from one another.

- Reliability
- Security
- Privacy
- Fairness

Simple Protection: Base and Bound

One way to implement this protection is to define some base address and some bound on it; we allow the application to read and write only within this area.

Relocation

One issue to consider is where we determine these bounds? In this case, we have to do it at runtime; the same program should be able to run on different machines, which may have different memory space...

So, what do we do?

One way is to ignore it; when we compile the program, we start at zero. And then we can translate it.

We can let the CPU do the translations on the fly at runtime nowadays.

Paged Virtual Address Space

We note that allocating the chunks isn't very flexible; what if the application is only using a fraction of the chunk?

Instead, what we can do is allocate smaller pieces of memory – pages – if the application needs it.

Hardware translates address using a page table.

- Each page has a separate base.

If a page isn't used in the memory, we can evict it.

2.1.5 Process

> **Definition 2.5.** A process is an execution environment with restricted rights.
>
> - (Protected) Address Space with One or More Threads
> - Owns memory (address space)
> - Owns file descriptors, file system context, etc.
> - Encapsulates one or more threads sharing process resources.

We note that processes are protected from each other by the OS.

We note that the overhead for threads to communicate between each other in the same process is a lot lower.

Single versus Multi-threaded

We note that in a multi-threaded environment, each thread has its own register and stack, but can share code/data/files within the same process.

Reasons to have multiple threads per address space includes:

- Parallelism: Takes advantage of actual hardware parallelism (such as multicore).
- Concurrency: ease of handling I/O and other simultaneous events.

2.1.6 Dual Mode Operation

Hardware provides at least two modes:

- Kernel Mode (“supervisor” mode)
- User Mode

Certain operations are prohibited when using User Mode:

- Changing the page table pointer
- Disabling interrupts
- Interacting directly with hardware
- Writing to kernel memory

We can have a mode bit to decide which mode to be in.

Types of Transfers

First, we can do Syscall:

- Process requests a system service (e.g. exit)
- Like a function call, but outside the process
- Doesn’t have the address of the system function to call
- Marshall the syscall id and args in registers and exec syscall

Next is an interrupt:

- An external asynchronous event triggers context switch (time, I/O device, etc.)
- It’s independent of user process

Finally is trap or exception:

- Internal synchronous event triggers context switch (segmentation fault, divide by zero, etc.)

Process Control Block

The kernel represents each process as a PCB:

- Status (running/read/blocked/…)
- Register state (when it isn’t ready)
- Process ID, User, Executable, etc.
- Execution Time
- Memory space, transition

Kernel Scheduler maintains a data structure which contains the PCB. Then, some scheduling algorithm selects the next one to run.

If no process is ready to run, we go into an idle state.

2.2 Discussion - 9/4/2024

2.2.1 C Review

Types

C is statically typed (types are known at compile time), and is weakly typed as well (can cast between any types). On the other hand, if we think of Python, it is both dynamically and strongly typed.

Primitive types are `char`, `short`, `int`, `long`, and `float`.

We work a lot with pointers; these are references that hold the address of an address of an object in memory.

- They're essentially unsigned integers.
- Use to get the value, and & to get the address.

Memory

Recall that our memory layout is roughly as follows:

- Text
- (Un)initialized Data
- Heap
- Stack
- Initialized strings/global constants which may be stored in read-only segments

C Concept Check

1. `sizeof(dbl_char)` is 4, since it's a pointer. In this case, this is because we're on a 32-bit system.
2. It shouldn't error; it'll also return 4.
3. In the case of `char a = "162"`, the string literal is stored in the read-only section of memory (and thus is immutable). On the other hand, `char b[] = "162"`, the string is placed onto the stack of the current function's stack frame and is mutable.
4. For this, one of the differences is that `struct point p` is a pointer to the struct `p` (and is uninitialized). In that case, we'll probably get a segfault on when we do `printf("%d", p->x = 1);`, as there's most likely garbage on where we're trying to access to when we do `p->x`.

> **!** Note that for 2., this is because of `sizeof()`; if we did something like `int x = dbl_char`, where `dbl_char == NULL`, then it would error.

Headers

1. The size should be something greater than or equal to 9 (for this problem specifically, it is 16).
2. The size should be something greater than or equal to 17 (for this problem specifically, it is 24).

! Although we have `char* string` and `char target`, which in total is 9 bytes on our 64-bit system, we note that GCC pads structs arbitrarily.

When we do something like:

Code: gcc Compiling

```
> gcc -DABC -c app.c -o app.o
> gcc -c lib.c -o lib.o
> gcc app.o lib.o -o app
```

In this case, `app.c` is compiled with ABC defined, but not `lib.c`; this leads to some undefined behaviour.

2.2.2 x86

Registers

Registers are small storage spaces directly on the processor, allowing for fast memory access.

General Purpose Registers store both data and addresses; we have 8 in x86. Started as 16-bits, but we can extend to 32-bit using e prefix.

AT&T Syntax

Prefix registers with %, constants with $.

The general structure is `inst src, dest`. Address memory with `offset(base, index, scale)`.

- `base, index` are registers. `offset` is any integer. `scale` is 1/2/4/8.

Calling Convention

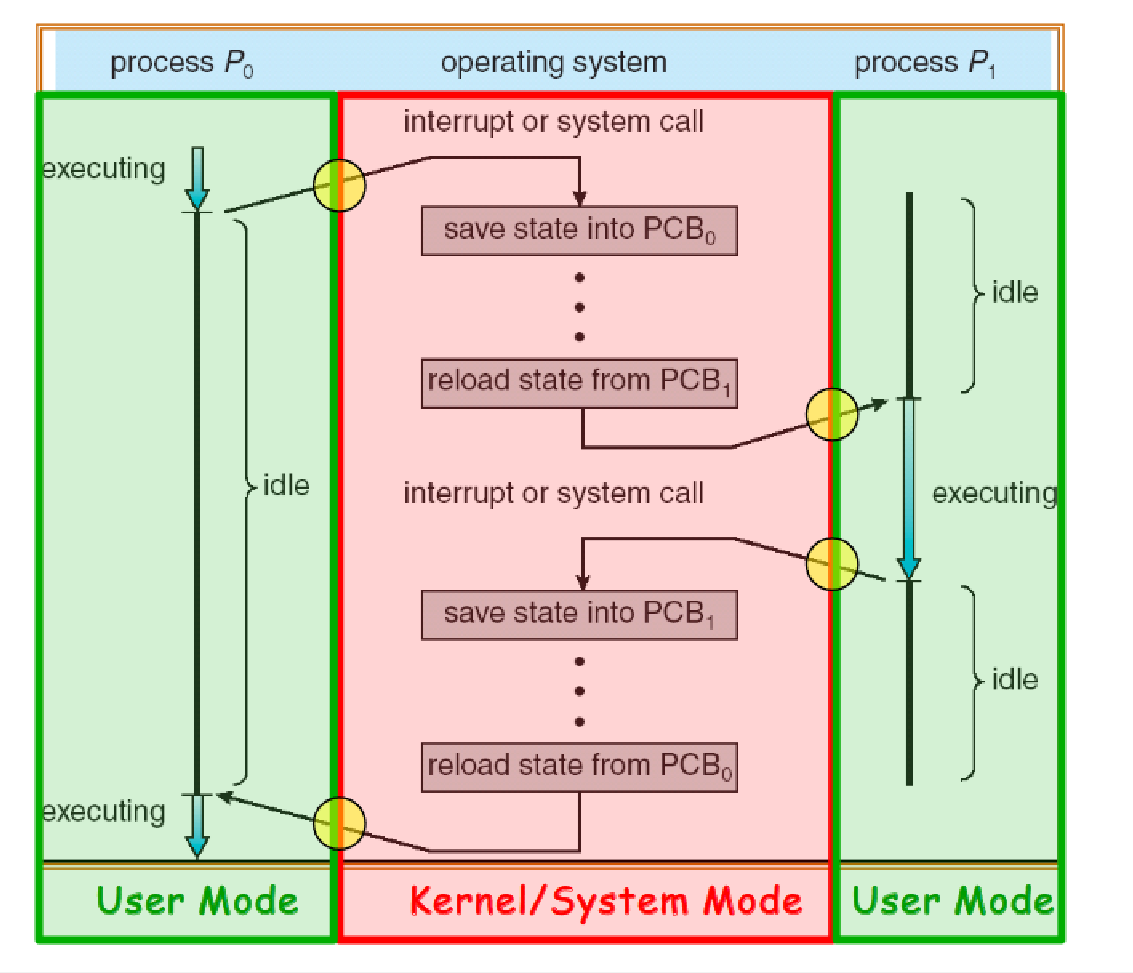
Recall the Calling Convention from CS161.

2.3 Lecture – 9/5/2024

Today, we will take a deeper dive into threads and how we program with them.

2.3.1 Recap

Figure 2.1: Switching Between User and Kernel Mode



Running Multiple Programs

We have the basic mechanism to:

- Switch between user processes and the kernel,
- The kernel can switch among user processes,
- Protect OS from user processes and processes from each other.

But now, there are some questions to consider:

- How we represent user processes int he OS?
- How we decide which user process to run?
- How we pack up the process and set it aside?
- How we get a stack and heap for the kernel?

Multiplexing Processes: The PCB

Recall that the kernel represents each process with a PCB

- Status

The Kernel Scheduler maintains a data structure containing the PCBs, and it gives out CPU to different processes – this is a policy decision.

It also give out non-CPU resources, such as memory/IO. This is another policy decision.

Scheduler

> **Definition 2.6** (Scheduling)**.** Scheduling is the mechanism for deciding which processes/threads receive hardware CPU time, when, and for how long.

Lots of different scheduling policies provide fairness, real-time guarantees, latency optimization, etc.

Hyperthreading

Simultaneous Multi-threading (or Hyperthreading) is a hardware scheduling technique:

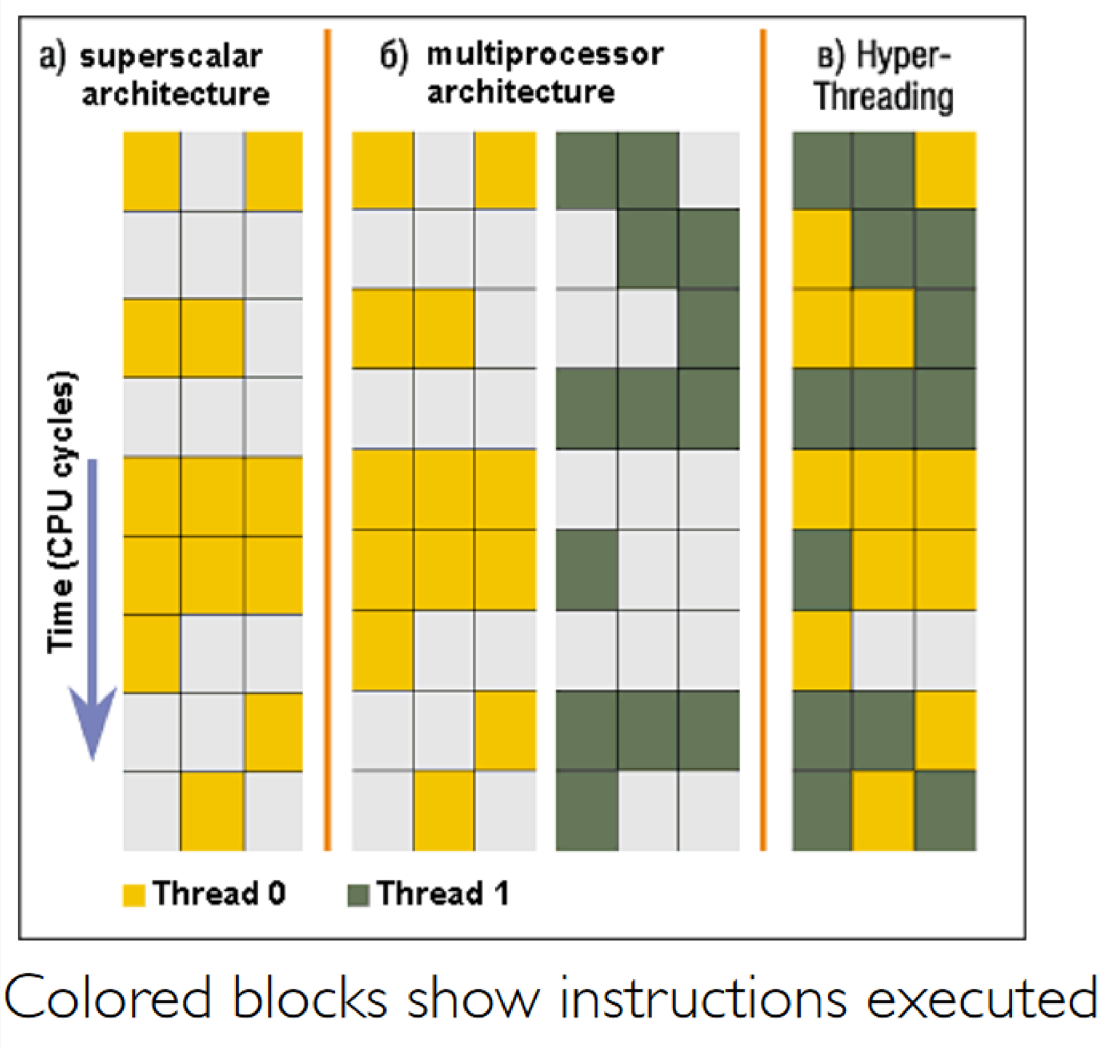
- Avoids software overhead of multiplexing

- Superscalar processors can execute multiple instructions that are independent

- Duplicates register state to make a second "thread" whcih allows more instructions to run

It can schedule each thread as if it were a separate CPU – however, it has sub-linear speedup.

Figure 2.2: Comparisons of Different Architecture


a) superscalar architecture
б) multiprocessor architecture
в) Hyper-Threading
Time (CPU cycles)
Thread 0
Thread 1
Colored blocks show instructions executed

We can think of hyperthreading as being similar to pipelining, but with threads instead.

2.3.2 Threads

Recall the definition of threads: this is a single unique execution context. It provides the abstraction of a signle execution sequence that represents a separately schedulable task.

Threads are a mechanism for concurrency (overlapping execution); however, they can also run in parallel (simultaneous execution).

And recall that the protection is handled by the process.

Some Definitions

> Definition 2.7. **Multiprocessing** is where we have multiple CPUs (Core). **Multiprogramming** is where we have multiple jobs/processes. **Multithreading** is where we have multiple threads/processes.

Going back to concurrency, this is where the scheduler is free to run threads in any order and interleaving. Threads may run to completion or time-slice in big or small chunks.

> **!** Concurrency is **not** parallelism! Concurrency is about handling multiple things at once; parallelism is about doing multiple things *simultaneously*.

To elaborate, each thread handles or manages a separate task, but they are not being executed simultaneously!

2.3.3 Threads Mask I/O Latency

Recall that a thread is in one of the following three states:

- RUNNING – running
- READY – eligible to run, but not currently running
- BLOCKED – ineligible to run

If no thread performs I/O, the scheduler tries to be fair and give half of the time to one thread, and the other half to another.

However, if, say, T1 has an I/O operation, we can dedicate more time to T2.

2.3.4 Multi-threaded Programs

When we compile a C program and run the executable, this creates a process that is executing the program. Initially, the new process only has one thread in its own address space.

But, how do we make it multi-threaded? The solution is that the process issues system calls to create new threads – these new threads are part of the process and share its address space.

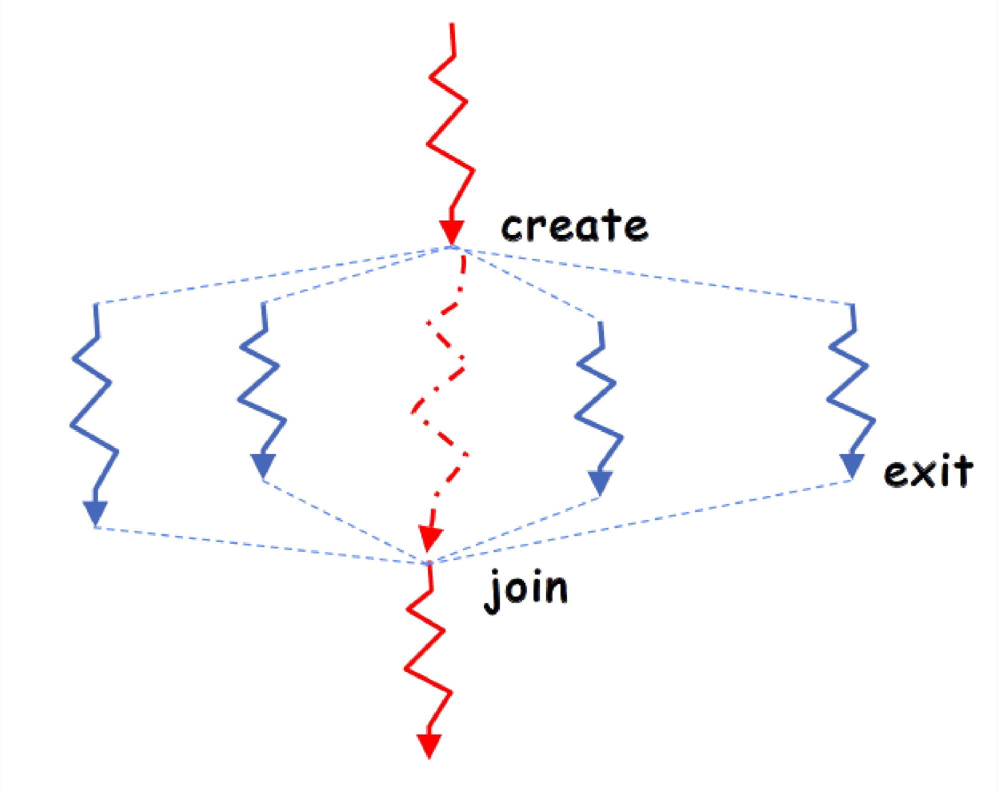
2.3.5 OS Library API for Threads: pthreads

Code: pthreads

```c
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);

// Thread is created, executing start_routine with arg as its sole argument.
// Return is implicit call to pthread_exit
// (attr contains info like stack size, scheduling policy, etc.)


void pthread_exit(void *value_ptr);
// Terminates the thread and makes value_ptr available to any successful join

int pthread_join(pthread_t thread, void **value_ptr);
// Suspends execution of the calling thread until the target thread terminates
```

Now we introduce a new idea: Fork-Join Pattern.

Figure 2.3: Fork-Join Pattern



The main thread creates forks, which is a collection of sub-threads, passing them args to work on. And at the end, it joins with them, collecting the result.

2.3.6 Thread State

State shared by all threads in process:

- Content of memory
- I/O state

State private to each thread:

- Kept in the TCB
- CPU registers
- Execution stack

Execution Stack:

- Parameters, temporary variables
- Return PCs are kept while called procedures are executing

2.3.7 Memory Layout with Two Threads

We note that we have two sets of memory registers and stacks. If threads violate this, we can get a stack overflow.

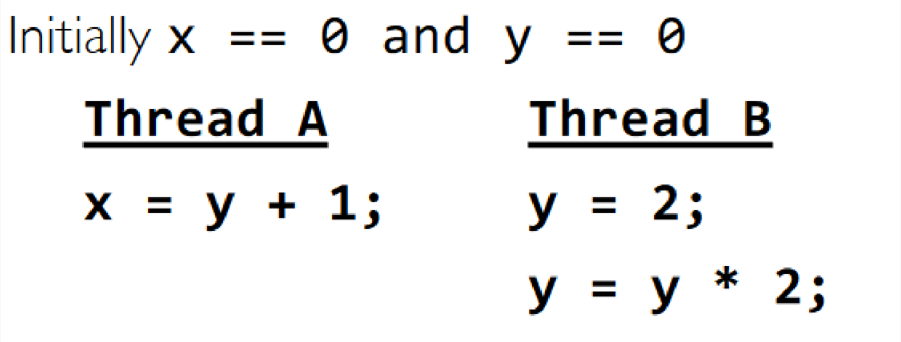
2.3.8 Interleaving and Non-Determinism

When we write a program, we see it sequentially. However, due to the nature of the scheduler, we have to take note of how it can interleave and make things non-deterministic.

Thus, we have to note the following:

- Non-Determinism
  - Scheduler can run threads in any order, and can switch threads at any time.
  - This results in testing being very difficult.
- Independent Threads
  - No state shared with other threads; this is deterministic, and reproducible conditions.
- Cooperating Threads
  - Shared state between multiple threads.

Figure 2.4: Non-Determinism Example



```
Initially x == 0 and y == 0
    Thread A          Thread B
    x = y + 1;        y = 2;
                      y = y * 2;
```

We see that in this example, `x` can be either 1, 3, or 5 depending on which order the scheduler decides to execute our threads, and when it switches.

2.3.9 Solution

Now, we introduce the following terms:

Definition 2.8 (Synchronization)**.** Synchronization is the coordination among threads, usually regarding shared data.

Definition 2.9 (Mutual Exclusion)**.** Mutual Exclusion is ensuring only one thread does a particular thing at a time (one thread excludes the others). This is at ype of synchronization.

> **Definition 2.10** (Critical Section)**.** Critical Section is code which exactly one thread can execute at once. This is a result of mutual exclusion.

> **Definition 2.11** (Lock)**.** Lock is an object only one thread can hold at a time. This is a mechanism for mutual exclusion.

Lock

Locks provide two atomic operations:

- `Lock.acquire()` – wait until lock is free; then mark it as busy.
  - After this returns, we say that the calling thread "holds" the lock.
- `Lock.release()` – this marks the lock as free.
  - This should only be called by a thread that currently holds the lock.
  - Once this returns, the calling thread no longer holds the lock.

Issues with Lock

Note that if we run into an infinite loop when a thread acquires a lock, then we can't release it at all.

Furthermore, we note that if everything requires a lock, then it is no better than just not having multiple threads.

pthreads Lock

Code: pthreads Lock Functions

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

2.3.10 Conclusion

- Threads are the OS unit of concurrency and execution.
  - We can use `pthread_create` to manage threads within a process.
  - They share data; we need synchronization to avoid data races.

WEEK 3

THIRD WEEK

3.1 Lecture – 9/10/2024

3.1.1 Handling Things Safely

Instead of calling a function, we have an identifier for each system function call.

- Vector Through well-defined syscall entry points
  - Table which maps system call number to handler.
  - Set to kernel mode at the same time as jumpt o system call code in kernel.
  - Separate kernel stack in kernel memory during syscall execution
- On entry, we copy arguments from the user memory/registers/stack into the kenerl memory. Furthermore, we validate the arguments.
- On exit, we copy results back.

Interrupts

- Interrupt processing not visible to the user process
- Interrupt vector
- Kernel interrupt stack
- Interrupt masking
- Atomic transfer of control

Interrupt Controller

The interrupt controller chooses which interrupt request to honor. Interrupt identity is specific with ID line. Mask enables/disables interrupts. It picks the one with highest priority.

3.1.2 Separate Stacks

We note that the kernel needs space to work, but we can't put anything on the user stack. Thus, a solution is to consider a two-stack model.

The OS thread has an interrupt stack (located in the kernel memory) plus user stack (located in user memory).

Syscall handler copies user args to kernel space before invoking specific function.

3.1.3 Managing Processes

First, we recall that everything outside of the kernel is running in a process.

- Even the shell!

A key point to take note of is that processes are started by other processes.

Bootstrapping

If processes are started by other processes, the question arises on how the first process start.

Note then that the first process is started by the kernel.

Process Management API

We have the following:

- `exit`: terminate a process
- `fork`: copy the current process
- `exec`: change the program being run by the current process
- `wait`
- `kill`
- `sigaction`

Creating Processes

Each process has a unique identifier.

When we do `pid_t fork()`, we copy the current process. The new process has a different pid and contains a single thread.

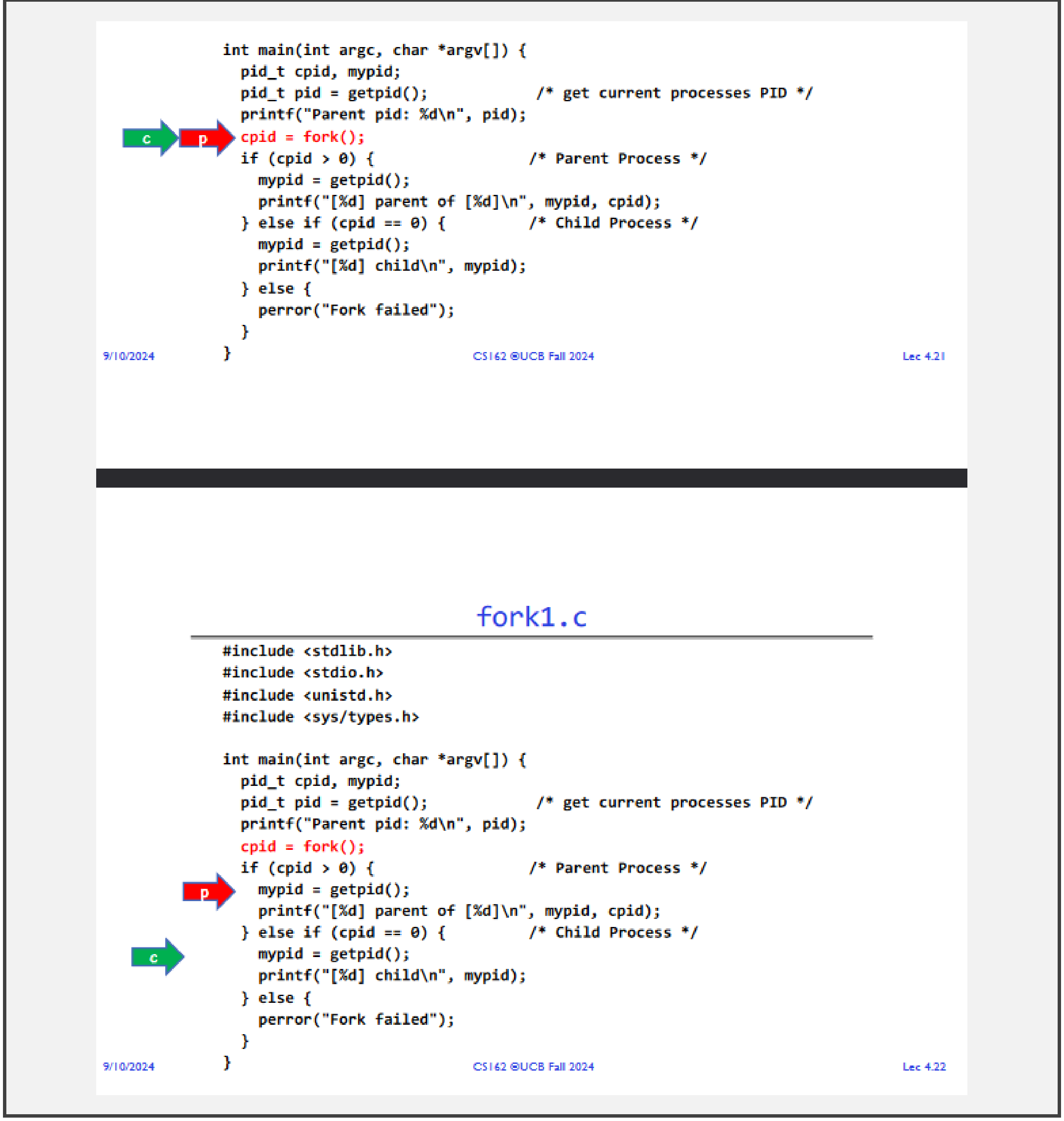
The return value is a pid.

When we create a copy of the current process, they no longer have access to each other. Remember that processes are protected from each other!

We note that based on the return value of `fork()`, we have:

- When the return value is greater than zero, we are running in the original parent process.
- When the return value is zero, we are running in the child process.
- When the return value is less than zero, there's an issue...

Figure 3.1: Illustration of `fork()`



```
int main(int argc, char *argv[]) {
  pid_t cpid, mypid;
  pid_t pid = getpid();            /* get current processes PID */
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                  /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  } else if (cpid == 0) {          /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
  }
}
```

9/10/2024 CS162 ©UCB Fall 2024 Lec 4.21

fork1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
  pid_t cpid, mypid;
  pid_t pid = getpid();            /* get current processes PID */
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                  /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  } else if (cpid == 0) {          /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
  }
}
```

9/10/2024 CS162 ©UCB Fall 2024 Lec 4.22

Variants of `exec`

The shell forks a process, and that process calls `exec`.

Waiting

We see that `wait` can be thought of as being similar to `join` with threads.

Signals

3.1.4 Files

A Unix/POSIX idea is that "Everything is a File." We have identical interface for:

- Files on disk
- Devices (terminals, printers, etc.)
- Regular files on disk
- Networking (sockets)
- Local interprocess communication (pipes, sockets)

It's based ont he system calls `open()`, `read()`, `write()`, and `close()`.

It also includes `ioctl()` for custom configuration that doesn't quite fit in.

The Abstraction

Files are a named collection of data in a file system. The data is a sequence of bytes (could be text, binary, etc.). Each file also has metadata such as its size, modification time, owner, etc.

We have a directory, which is a folder containing files (and other directories).

From here, we have a hierarchical naming system (the path).

The Connection between Processes, File Systems, and Users

Every process has a ***current working directory*** (CWD).

- We note that this can be set with system call.

We have absolute and relative paths (the former ignores CWD, the latter being relative to it).

Streams

3.2 Discussion – 9/11/2024

3.2.1 Fundamentals

Operating System

The OS's job is to provide hardware abstractions to software applications and manage hardware resources.

> **!** OS is not really a well-defined term! We can sorta think of it like a referee, illusionist, and glue.

Address Space

If we don't have the illusion of infinite resources, each process has to fight over the resources on our device.

Base and bound splits up our memory up so that each specific area is for some process; if the process accesses things outside of their allowed segment, it causes some sort of error.

There are other ways such as segmentation, and page-tables.

An address space is like a concept, and not like the physical hardware.

Dual Mode Operation

We assign privileges to processes. We have kernel mode, which has access to everything; OS will mostly be acting in this mode.

User mode has less privileges.

> **Example 3.1.** To switch between the modes, we think of Spotify wanting to play audio.
>
> The transfer between user mode to kernel mode is call the `syscall`. Alternatively, there's interrupts which transfers from user to kernel mode; think of a keyboard.
>
> We also have exceptions; we want the kernel to handle it in a "privileged" way.

We also have the IVT (Interrupt Virtual Table).

3.2.2 Concept Check

> **Problem 3.1.** What is the importance of address translation?

Solution. It allows us to map virtual memory to physical memory in order to give us the illusion of having the entire address space.

Furthermore, it provides isolation/protection between different processes' address space. ■

> **Problem 3.2.** Similar to what's done in the prologue at calling convention, what needs to happen before a mode transfer occurs?

Solution. We need to copy all of our registers/data first before we do a mode transfer occurs.

Namely, we need to save the processor state in the TCB (since the kernel may overwrite it). ■

> **Problem 3.3.** How does the syscall handler protect the kernel from corrupt or malicious user code?

Solution. User program specifies an index instead of direct address of the handler. Arguments are validated and copied over to the kernel stack to prevent TOCTTOU attacks.

After the syscall finishes, the results are copied back into the user memory. The user process isn't allowed to access the results stored in kernel memory.

We note then that the user process never accesses the kernel memory directly. ■

> **Problem 3.4.** Trivia: In Linux, the `/dev/kmem` file contains the entirety of kernel virutal memory and it can be read. Why do we let a user program read kernel memory?

Solution. Since it's restricted to only root users, this means that we are assuming that the user has the privileges of a supervisor to begin with in order to read the `/dev/kmem` file. ■

3.2.3 Processes

PCB

The OS needs to run many programs, which requires being able to switch between user processes and kernel; switching among user processes through the kernel; protecting the OS from processes, and processes from each other.

Thus, the kernel represents each process with a Process Control Block (PCB), which can be thought of as a metadata storage block.

> **Remark 3.2.** For the first project, we can just think of a one-to-one mapping between threads and processes, since we aren't having multi-threaded processes.

Syscall

`exec` changes the program being run by the current process. Unlike `fork`, it doesn't create a new process.

> **Remark 3.3.** Each thread has its own stack, but since they share a process, they can access the other threads' stack.
>
> They also share a heap.
>
> Whereas if we have two processes, if we share address on one stack with another, the other process can't access it.

Signal Handling

> **Problem 3.5.** Why is SIGSTOP and SIGKILL overriding disabled?

Solution. We can think that if we accidentally ran a malicious code, if they could override it, then we can't terminate the program. ■

3.2.4 Pintos Lists

...

3.3 Lecture – 9/12/2024

3.3.1 Low-Level vs High-Level File API

First, low-level:

- Low-level direct use of syscall interface: `open()`, `read()`, etc.
- Opening of file returns descriptor: `int myfile = open(...);`
- File descriptor only meaning to kernel.
  - We index into the PCB which holds pointers to kernel-level structure describing file.

- Every `read()` or `write()` causes syscall no matter how small.

On the other hand, high-level:

- High-level buffered access: `fopen()`, `fread()`, etc.
- Opening of file returns ptr to FILE: `FILE *myfile = fopen(...);`
- File structure is user space contained

Consider a low-level operation: we require to do some syscalls, then save all of our arguments onto registers, and so on.

But high-level is more sophisticated, not needing to necessarily go through syscalls.

The high-level API has the issue of not doing what we imagine it would do. For example, looking at the `sleep(10)` example: we see that using high-level API, everything before and after the sleep gets printed at once.

Another example, if we are doing a write then read, the information may not be up to date; we would need to flush.

3.3.2 Files

Definition 3.4. A file descriptor number is an int, which we can think of a index/pointer which directs us to where the file is located at.

Definition 3.5. A file description is ...

The file description is created in the kernel. The two most important things to take note of are:

- Where to find the file data on disk, `inode`.
- The current position within the file.

Remark 3.6. We note that `unlikely()` hints to the branch prediction that the chances of this condition occurring is unlikely.

3.3.3 Device Driver

Definition 3.7 (Device Driver)**.** This is device-specific code in the kernel that interacts directly with device hardware.

Device driver is usually divided into two pieces:

- Top half: accessed in call path from system calls.
- Bottom half: run as interrupt routine.

Handler functions for each of the file operations.

Week 4

Week For Suffering

4.1 Discussion – 9/17/2024

Administrivia

Homework 1 is due this Sunday, and Project 1 Design Doc is due next Thursday.

4.1.1 Project Timeline

We want to roughly follow this timeline for projects:

1. Work on design document.
2. Submit design document.
   - Note that the design document has a hard ceiling of 15 pages.
3. TA reads through design document.
4. Meet with TA for a 30-minute design review.
5. Code.
6. Submit code, report, and peer evaluations.

4.1.2 Threads

Recall that threads as single unique execution contextes with their own set of registers and stack. They are sometimes referred to as "lightweight processes."

Components that are shared between threads in the same process don't need to be persisted by each individual thread.

A thread still needs to persist registers, and its stack in the TCB.

4.1.3 Syscall

Recall that POSIX has a `pthread` library for syscalls, similar to the process syscalls.

Problem 4.1. What are the possible outputs of the following program? How would we ensure that "HELPER" is printed before "MAIN"...?

Code: pthread_order.c

```c
void *helper(void *arg) {
        printf("HELPER");
        return NULL;
}
int main() {
        pthread_t thread;
        pthread_create(&thread, NULL, &helper, NULL);
        sched_yield();
        printf("MAIN");
        return 0;
}
```

Solution. We observe that the program will have the following possible outputs:

- MAINHELPER
- HELPERMAIN
- MAIN
  - This happens either when `pthread_create` fails, or when the main thread exits out first.

To ensure that HELPER gets printed before MAIN, we would need to use `pthread_join(thread, NULL)`. ■

Problem 4.2. What does the following program print?

Code: `pthread_stack.c`

```c
void *helper(void *arg) {
        int *num = (int*) arg;
        *num = 2;
        return NULL;
}

int main() {
        int i = 0;
        pthread_t thread;
        pthread_create(&thread, NULL, &helper, &i);
        pthread_join(thread, NULL);
        printf("i is %d", i);
        return 0;
}
```

Solution. We see that helper takes in a pointer which is stored on the main thread's stack, and then setting the region the pointer is pointing to.

The program will print 2. This is because both threads share the same address space, since they are both from the same process.

Note that even though each thread have their own stack, they can access each other's stack as well. ■

Problem 4.3. What does the following program print?

Code: `pthread_stack.c`

```c
void* helper(void *arg) {
        int* num = (int*) arg;
        printf("%d", *num);
        return NULL;
}

void spawn_thread(void) {
        int i = 162;
        pthread_t thread;
        pthread_create(&thread, NULL, &helper, &i);
        return;
}

int main() {
        spawn_thread();
        return 0;
}
```

Solution. We observe that since we aren't using `pthread_join()`, it is possible that the main thread will exit before the helper thread; thus, we could be printing out nothing.

Alternatively, the main thread could return from `spawn_thread()` early, dereferencing everything, and leading to garbage being printed out.

Or we could also have it work correctly, and we print out 162 as expected. ■

4.1.4 I/O

Recall the following design philosophy for UNIX:

- Uniformity
- Open before Use
- Byte Oriented
- Kernel Buffered Reads/Writes

Low-Level API

When forking a process, both process A and B will have mirrored FDT's.

Say Child B writes to some file it has opened up; if A tries to read that data, it can read/write more data into that same file.

High-Level API

These operate on streams of data. Note that low-level read isn't a guarantee on whether it can actually get the bytes or not, so it can be painful to work with.

Problem 4.4. What is the difference between `fopen` and `open`?

Solution. `fopen` is high-level API, while `open` is low-level API. ■

Problem 4.5. What will the `test.txt` file look like after this program is run?

Code: `test.txt`

```
int main() {
        char buffer[200];
        memset(buffer, 'a', 200);
        int fd = open("test.txt", O_CREAT|O_RDWR);
        write(fd, buffer, 200);
        lseek(fd, 0, SEEK_SET);
        read(fd, buffer, 100);
        lseek(fd, 500, SEEK_CUR);
        write(fd, buffer, 100);
}
```

Solution. The file will look like:

- The first 200 bytes will be "a."
- Then, the next 400 bytes will be null bytes.
- Then the next 100 bytes will be the character "a."

■

4.1.5 dup and dup2

...

4.2 Lecture – 9/19/2024

Recall the dispatch loop: we run a thread, then choose the next thread. Then, we save the state of our current TCB, and load the state of the next TCB.

This is an infinite loop, and one could argue that this is all that the OS does. One question then is how the thread gives control back to the OS? Furthermore, when should we – if ever – exit the loop?

For the first question, remember I/O operations: we need to give OS control to do something, since they access the files on the storage device, not the application itself.

For the second, the loop exits if we shut down the OS, and also in the case of interrupts.

4.2.1 Running a Thread

Recall that to run a thread, we load its state into the CPU, load its environment, then jump to the PC.

Remark 4.1. We give control of processor/core to the user code; the OS isn't running because the user is running. Furthermore, we can give control back to the OS with both internal and external events.

4.2.2 Internal Events

- Block on I/O: the act of requesting I/O implicitly yields the CPU.
- Waiting on a signle from other threads: thread asks to wait, and thus yields the CPU.
- `yield()`: the thread volunteers to give up on CPU.

> **Remark 4.2.** Cooperative Multitasking

4.2.3 Stack for Yielding Thread

Yield will perform a syscall, moving us to the kernel. Now, we are on the OS stack. We then call `run_new_thread`. Once we run the new thread, we can switch.

- How does the dispatcher switch to a new thread?
  - They save anything next thread may trash: PC, registers, stack pointer.
  - Maintains isolation for each thread.

4.2.4 Context Switching

> **Remark 4.3.** It is very hard to test switch code; there are too many combinations and interleaving.

> **!** Topaz kernel saved one instruction for optimization. What ended up happening was that, since it works as long as the kernel is under a megabyte, people were fine for a while. But eventually, they forgot, and weird issues began popping up.

Cost of Context Switching

We note that it's around 30 to 40 times cheaper to switch between threads within the same process.

> **Remark 4.4.** There are thread libraries in the user-level. These threads are not visible to the operating system, and must use yield. The operating system would only give control to that thread without yield.
>
> Switching between threads using yield is even cheaper in user-space.

4.2.5 External Events

If a thread never does any I/O, never waits, and never yield control, we must find a way that the dispatcher can regain control.

The solution to this is to use external events, such as interrupts and timer.

Interrupt Controller

Recall that interrupts are invoked with interrupt lines from devices. The interrupt controller chooses which interrupt request to honor. CPU can disable all interrupts with internal flag.

Note that an interrupt is a hardware-invoked context switch.

4.2.6 Threads

Initialization

We initialize the registers of TCB, and then for the stack, we setup the return to go back to ThreadRoot. We can think of the stack as just before the body of ThreadRoot really gets started.

4.2.7 Concurrency and Parallelism

We have multiple threads in the same process which share the same data (so they can access the same data). As a result, unexpected behaviors can occur.

> **!** If a scheduler can make the worst possible interleaving, assume that it will!

Bank Example

Note that we can try using multiple threads to make things faster. However, this can lead to data corruption.

Atomic Operations

Atomic operations always run to completion, or not at all. It is indivisible; it can't be stopped in the middle, and the state cannot be modified by someone else in the middle. This is a fundamental building block – if there are no atomic operations, then we have no way for threads to work together.

However, many instructions aren't atomic; double-precision floating point store is often not atomic.

4.2.8 Locks

Recall the following definitions:

> **Definition 4.5** (Synchronization)**.** Synchronization is using atomic operations to ensure cooperation between threads.

> **Definition 4.6** (Mutual Exclusion)**.** Ensures that only one thread does a particular thing at a time.

> **Definition 4.7** (Critical Section)**.** Piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code.

Now, for our lock, we have two main instructions: `acquire` and `release`. We assume that these operations are atomic. And when acquiring, we should wait; whoever has a lock may be in a critical section, and we can't enter this.

Going back to our banking example, we can simply use locks to create a critical section.

> **!** Threaded programs must work for all interleaving of thread instruction sequences! Cooperating threads are non-deterministic and non-reproducible, making it a nightmare to debug.

4.2.9 Conclusion

As a recap, we discussed threads, context switching, and the creation of threads. And in the context of context switching, we looked at how they can occur (via internal and external events).

We also looked at concurrency.

WEEK 5

WEEK FIVE

5.1 Discussion – 9/24/2024

5.1.1 Mutual Exclusion

A challenge is trying to keep certain section of codes executed by only one thread at a time, or executing things in the order we want.

Synchronization is possible through atomic operations, which always run to completion or not at all. Atomic operations are indivisible. Typically, loads/stores are atomic.

Code that runs as part of mutual exclusion is a critical section.

Locks

Locks allows you to create critical sections pretty easily.

Code: Locks and Critical Sections

```
lock_acquire(&lock);

/* CRITICAL SECTION */

lock_release(&lock);
```

Semaphore

Semaphores are synchronization variables with a non-negative integers. We have two (atomic) operations:

- Down – it waits for a semaphore's value to become strictly positive, then decrements it by 1.
- Up – it increments the value of the semaphoer by 1.

We have two workflows:

- Mutual Exclusive: we use semaphore as a lock
  1. Initialize semaphore to 1.
  2. We down the semaphore when entering a critical section.

3. We up the semaphore when exiting the critical section.

- Scheduling: one thread waits for another thread to do something.

  1. We initialize the semaphore to 0.
  2. We down the semaphore in the waiting thread.
  3. Once the active thread is done, it ups the semaphore and allows the waiting thread to work.

Example 5.1 (Semaphores as Locks). Suppose that we have threads A and B.

Thread A enters a critical section and downs the semaphore. Thread B tries to enter the critical section, we see that the semaphore is zero, so it can't down it.

Once A exits the critical section, it can up the semaphore so B can enter the critical section.

Example 5.2. We can use a semaphore to make Thread A to do something before Thread B. That is, we want Thread B to run after Thread A.

So, Thread B is our waiting thread. We down the semaphore in Thread B. Once Thread A is done with its work, we can then up the semaphore, allowing for B to do its work.

Problem 5.1. Given the following code, describe how a malicious user might exploit some unintended behavior. What changes can be made to defend against the exploit?

Solution. We observe that it's possible for us to do `transfer(a, b, 5)` and `transfer(b, a, 5)`, and there is an interleaving which results in one of them gaining an extra five dollars.

To protect against this, we can simply add a lock around the transferring process. ■