CS350: Operating Systems

Felix Zhou*

Spring 2020, University of Waterloo

*from Professor Leslie Istead's Lectures

Contents

1 Introduction

1.1 What is an Operating System?

Normally, an operating system is a program which

- manages resources
- creates execution environments
- loads programs
- provides common services and utilities

In a sense, the OS is both a facilitator and an enforcer.

1.2 Views of an Operating System

1.2.1 Application View

What services does it provide?

The OS's main job is to provide an execution environment for running programs.

Such an environment is responsible for

(1) providing a program with the necessary resources to run
(2) providing a program interfaces, an abstract view of hardware components such as networks, storage, and I/O devices
(3) isolating running programs and preventing them from interfering with each other

1.2.2 System View

What problems does it solve?

The OS

(1) manages the hardware resources of a computer system. Resources include
    - processors
    - memory,
    - disks & storage devices

- network interfaces
- I/O devices such as keyboards, mice, and monitors

(2) allocates resources among running programs.

(3) controls the sharing of resources among programs.

Remark that the OS itself utilizes resources which it must share with other programs.

1.2.3 Implementation View

How is it built?

Definition 1.2.1 (Concurrent)
Multiple programs/instructions running or appearing to run at the same time

Concurrency arises naturally in an OS when it supports concurrent applications.

Definition 1.2.2 (Real-Time)
Programs that must respond to events within specific timing constraints.

For example, hardware interactions impose timing constraints.

The OS is a concurrent, real-time program!

1.3 Implementing an OS

Definition 1.3.1 (Kernel)
The operating system kernel is the part which responds to system calls, interrupts and exceptions

As a whole, an operating system includes the kernel and so other related programs that provide services for applications.

1.3.1 Utility Programs

- task managers
- disk deframenting tools

1.3.2 Command Interpreters

- cmd.exe
- bash

1.3.3 Programming Libraries

- POSIX
- OpenGL

1.4 Types of Kernels

Definition 1.4.1 (Monolithic Kernel)
Almost everything, whether needed or not, is part of the Kernel.
This includes device drivers, file systems, virtual memory, IPC, etc

Definition 1.4.2 (Microkernel)
Only the absolutely necessary components are part of the kernel.
All others are user programs

Definition 1.4.3 (Real-Time OS)
An OS with stringent event response times, guarantees, and preemptive scheduling

Remark that Windows, Linux, Mac OSX, Android, and iOS are all monolithic operating systems. In addition, NONE of them are real-time.

QNX is a real-time, micro-kernel which originated here!

1.5 OS Abstractions

The execution environment provided by the OS includes a variety of abstract entities manipulable by running programs. For example

- files / file systems correspond to secondary storage (disk)
- address spaces correspond to primary memory (RAM)

- processes and threads correspond to program execution
- sockets and pipes correspond to network or other message channels

2 Threads & Concurrency

2.1 Introduction to Threads

2.1.1 What is a Thread?

Definition 2.1.1 (Thread)
A sequence of instructions

A normal sequential program consists of a single thread of execution.

Threads provide a way for programmers to express concurrency in a program.

In threaded concurrent programs, there are multiple threads of execution, all occuring at the same time.

A collections of threads may perform the same task but they may just as well perform different tasks.

2.1.2 Why Threads?

Definition 2.1.2 (Blocking)
Threads may block, ceasing execution for a period of time, or until some condition has been met.

When a thread blocks, it is not executing instructions (ie the CPU is idle). Concurrency allows the CPU to execute a different thread during this time. Time - CPU Time - is money!

Resource Utilization Blocked/Waiting threads give up resources, ie the CPU to others

Parallelism Multiple threads executing simultaneously improves performance

Responsiveness We can dedicate threads to UI and others to loading/long tasks

Modularization Programmers can organize the execution of tasks/responsibilities

2.2 OS/161: Thread Interface

```
// kern/include/thread.h

int thread_fork(
```

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

void thread_exit(void);

void thread_yield(void);
```

2.2.1 Other Thread Libraries and Functions

Definition 2.2.1 (Join)
A common thread function to force on ethread to block until another finishes

Joining is NOT offered by OS/161.

POSIX Threads (pthreads) A well-supported, popular, and sophisticated thread API. The C++ threading library is a wrapper aroudn this library.

OpenMP A cross-platform simple multi-processing and thread API.

GPGPU Programming General-purpose GPU programming APIs.

For example, NVidia's CUDA create/run threads on GPU instead of CPU.

2.3 Review

2.3.1 Sequential Program Execution

The fetch-execute cycle consists of

1) fetch instruction PC (Program Counter) points
2) decode and execute instruction
3) increment the PC

2.3.2 MIPS Registers

Definition 2.3.1 (Caller-Save)
The calling function is responsible to save/restore values in these registers.

Definition 2.3.2 (Callee-Save)
The called function is responsible to save/restore values in these registers.

Below are some conventions enforced in the compiler and used in the OS.

Number	Name	Use
0	z0	always 0
1	at	assembler reserved
2	v0	return value or syscall number
3	v1	return value
4-7	a0-a3	subroutine arguments
8-15	t0-t7	temporary values (caller-save)
16-23	s0-s7	saved values (callee-save)
24-25	t8-t9	temporary values (caller-save)
26-27	k0-k1	kernel temps
28	gp	global pointer
29	sp	stack pointer
30	s8/fp	frame pointer (callee-save)
31	ra	return address (for jal)

2.3.3 The Stack

Functions push arguments (a0-a3), return address, local variables, and temporary use registers onto the stack (backwards!).

2.4 Concurrent Program Execution (Two Threads)

Conceptually, each thread executes sequentially using its private register contents and stack. However, the threads share the address space (ie code, data). Although we can think of each thread having its own thread stack, what really happens is that threads are allocated a portion of the address space and MAY overflow onto other spaces if too much data is pushed onto the stack.

2.5 Implementing Concurrent Threads

2.5.1 Hardware Support

Suppose we have P processors, C cores, with each core supporting M simultaneous threads. Then PCM threads can execute simultaneously.

2.5.2 Timesharing

Threads share the CPU.

Multiple threads take turns on the same hardware. The processor rapidly switches between threads to create the illusion that all are making progress simultaneously.

2.5.3 Hardware Suppose & Timesharing

PCM threads running simultaneously with timesharing.

Note that while cores of a single processor share caches (L2, L3), threads execute separately.

2.6 Timesharing & Context Switches

Definition 2.6.1 (Context Switch)
When timesharing, the switch from one thread to another is called a context switch

2.6.1 High Level View

1) decide which thread will run next (scheduling)
2) save register contents of current thread

3) load register contents of next thread

Thread context must be saved/restored carefully, since thread execution continuously changes the context.

2.7 MIPS: Context Switching

```
// kern/arch/mips/thread/switch.S

switchframe_switch:
// a0: address of switchframe pointer of old thread
// a1: address of switchframe pinter of new thread

// Allocate stack space for saving 10 registers
addi sp, sp, -40

sw ra, 36(sp)
sw gp, 32(sp)
sw s8, 28(sp)
sw s6, 24(sp)
sw s5, 20(sp)
sw s4, 16(sp)
sw s3, 12(sp)
sw s2, 8(sp)
sw s1, 4(sp)
sw s0, 0(sp)

// Store old stack pointer in the old thread
sw sp, 0(a0)

// Get new stack pointer from new thread
lw sp, 0(a1)
nop // delay for load

// Restore registers
lw s0, 0(sp)
lw s1, 4(sp)
lw s2, 8(sp)
lw s3, 12(sp)
lw s4, 16(sp)
lw s5, 20(sp)
lw s6, 24(sp)
lw s8, 28(sp)
lw gp, 32(sp)
```

```
    lw ra, 36(sp)
    nop // delay for load

    // Return
    j ra
    addi sp, sp, 40 // in delay slot
    .end switchframe_switch
```

`switchframe_switch` is called by the C function `thread_switch`.

`thread_switch` is the caller. It saves/restores the caller-save registers.

On the other hand, `switchframe_switch` is the callee. It must save/restore the callee-save registers.

`switchframe_switch` saves callee-save registers to the old thread stack. it restores the callee-save registers from the new threads stack.

Definition 2.7.1 (Load-use Hazards)
Loaded values are used in the next instruction (before being actually loaded into the register).

Definition 2.7.2 (Control Hazards)
When it is not known which instruction to fetch next.

MIPS R3000 is pipelined. Delay-slots are used to protect against load-use hazards and control hazards.

2.8 Causes for Context Switching

- the running thread calls `thread_yield` and voluntariloy allows other threads to run
- the running thread calls `thread_exit` and is terminated
- the running thread blocks via a call to `wchan_sleep`, more later
- the running thread calls `preempted`, it involuntarily stops running

Notice that an OS should strive to maintain high CPU utilization. This means in addition to timesharing, context switches occur whenever a thread ceases to execute instructions.

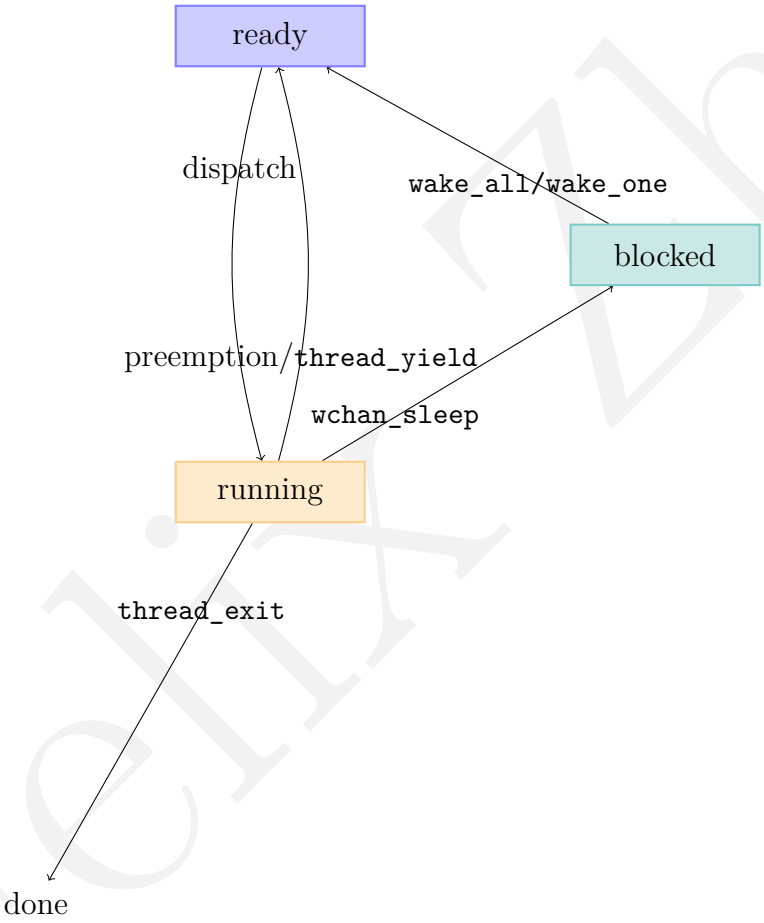
2.9 Thread States

There are three thread states

running currently executing

read ready to execute

blocked waiting for something, so not ready to execute



2.10 OS/161: Thread Stack after Voluntary Context Switch

It is important to keep in mind that the context switching API is still composed of function calls and therefore the same rules for regular function calls apply in MIPS.

1) current thread calls `thread_yield` to yield the CPU

2) `thread_yield` calls `thread_switch` to perform a context switch

3) `thread_switch` chooses a new thread, calls `switchframe_switch` to perform low-level context switch

2.11 Timesharing & Preemption

2.11.1 Timesharing

How rapidly should a processor switch between threads? We impose a limit on CPU time.

Definition 2.11.1 (Scheduling Quantum)
An upper bound on how long a thread can run before it MUST yield the CPU.

2.11.2 Preemption

However, how do we stop a running thread that never yields, blocks, or exits when the quantum expires?

Preemption forces a running thread to stop running, so that another thread can have a chance.

In order to implement this, the thread library MUST have a means of "getting control" (ie causing thread library code to execute) even though the running thread has not called a thread library function.

This is normally accomplished through interrupts.

2.11.3 Review

2.11.4 Interrupts

Definition 2.11.2 (interrupt)
An event that occurs during the execution of a program.

Interrupts are caused by system devices (hardware) such as a timer, a disk controller, or a network interface.

When an interrupt occurs, the hardware automatically transfers control to a fixed location in memory. At this location, the thread library places a procedure called an interrupt handler.

Definition 2.11.3 (Interrupt Handler)
Responsible for

(i) creating a trap frame to record the thread context at the time of the interrupt

(ii) determining which device caused the interrupt and performs device-specific processing

(iii) restoring the saved thread context from the trap frame and resumes execution of the thread

2.11.5 OS/161 Thread Stack after an Interrupt

At the momement of the interrupt, a trap frame is created. This emulates an immediate function call and saves the necessary registers before the interrupt handler then proceeds to do what it has to do.

2.12 Preemptive Scheduling

Definition 2.12.1 (Preemptive Scheduler)
A preemptive scheduler uses the scheduling quantum to impose a time limit on running threads

Threads may block or yield before their quantum has expired. However, periodic timer interrupts allows the running time to be tracked. If a thread has run too long, the timer interrupt handler preempts the thread by calling `thread_yield`.

The preempted thread changes state from running to read, and it is placed on the ready queue.

Each time a thread goes from ready to running, the runtime starts out at 0. Runtime does NOT accumulate.

Note that the OS/161 threads use preemptive round-robin scheduling.

2.13 OS/161: Thread Stack after Preemption

Assume the trap frame and interrupt handler stack frames have been correctly pushed onto our stack. Then, the interrupt handler calls `thread_yield`, this function in turn calls `thread_switch`, which finally calls `switchframe`.

Example 2.13.1
This is a possible high level sequence of events.

Suppose there are only two threads and Thread 2 called `thread_yield`, following which the frames for itself and `thread_switch` and `switchframe` are both succesfully pushed onto the Thread 2 stack. Then, suppose Thread 1 is dispatched and its runtime has exceeded the scheduling quantum.

1) Thread 1 Stack
   (a) timer interrupt
   (b) trap frame pushed
   (c) interrupt handler stack frame pushed
   (d) `thread_yield` frame pushed
   (e) `thread_switch` frame pushed
   (f) `switchframe` frame pushed
2) Thread 2 Stack
   (a) `switchframe` frame popped
   (b) `thread_switch` frame popped
   (c) `thread_yield` frame popped
   (d) Thread 2's context is restored and it runs its regular program
   (e) `thread_yield` frame pushed
   (f) `thread_switch` frame pushed
   (g) `switchframe` frame pushed
3) Thread 1 Stack
   (a) `switchframe` frame popped
   (b) `thread_switch` frame popped
   (c) `thread_yield` frame popped
   (d) Thread 1's context is restored and it runs its regular program
4) repeat

3 Synchronization

3.1 Thread Synchronization

All threads in a concurrent program share access to the same global variables and heap.

Definition 3.1.1 (Critical Section)
The part of a concurrent program in which a shared object is accessed

We can have issues when several threads attempt to access the same global variables or heap object at the same time.

3.2 Race Condition

```
int volatile total = 0;

void add() {
    int i;
    for(i=0; i<N; i++) total++;
}

void sub() {
    int i;
    for(i=0; i<N; i++) total--;
}
```

What happens when one thread executes add and another executes sub? What is the total value of total when they have finished?

In theory, `total` should be 0 at the very end. However, if `sub` mutates `total` right after the read in `aad` and before the write in `add`, the program will not behave as we wish.

Definition 3.2.1 (Race Condition)
When the program result depends on the order of execution

Race conditions occur when multiple threads are reading and writing the same memory at the same time.

For now, assume the only source of race conditions is implementation.

3.3 Tips for Identifying Race Conditions

```
int list_remove_front(list *lp) {
    int num;
    list_element *element;

    assert(!is_empty(ls));

    element = lp->first;
    num = lp->first->item;

    if(lp->first == lp->last) {
        lp->first = lp->last = NULL;
    }
    else {
        lp->first = element->next;
    }
    lp->num_in_list--;
    free(element);

    return num;
}

void list_append(list *lp, int new_item) {
    list_element *element = malloc(sizeof(list_element));
    element->item = new_item;

    assert(!is_in_list(lp. new_item));

    if(is_empty(lp)) {
        lp->first = element;
        lp->last = element;
    }
    else {
        lp->last->next = element;
        lp->last = element;
    }
    lp->num_in_list++;
}
```

To identity potential race conditions, we first find the critical sections.

Inspect each variable. Is it possible for multiple threads to read/write it at the same time?

Constants and memory that all threads only READ do not cause race conditions.

Assuming we found the critical section, how can we prevent race conditions?

3.4 Enforcing Mutual Exclusion with Locks

Acquire/Release ensures that only one thread at a time can hold the lock. Even if multiple threads attempt to acquire at the same time, only one will successfully do so. If a thread cannot acquire the lock right away, it must wait until the lock is available.

Definition 3.4.1 (Mutex)
Mutual Exclusion.
Provided by locks.

```
int volatile total = 0;
bool volatile total_lock

void add() {
    int i;
    for(i=0; i<N; i++) {
        Acquire(&total_lock);
        total++;
        Release(&total_lock);
    }
}

void sub() {
    int i;
    for(i=0; i<N; i++) {
        Acquire(&total_lock);
        total--;
        Release(&total_lock);
    }
}
```

3.4.1 Lock Acquire & Release

Consider the following implementation. Does it work as intended?

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

```
void Release(bool *lock) {
    *lock = false;
}
```

Unfortunately, the same problems as before occur as our test and set operation is NOT atomic. If multiple threads are spinning in wait for the lock to be free, it is possible that more than one thread "obtains" the lock.

3.4.2 Hardware-Specific Synchronization Instructions

We must provide a way to implement atomic test-and-set for synchronization primitives like locks.

An example in MIPS is the atomic x86 (and x64) xchg instruction:

```
xchg src, addr
```

where `src, addr` is a register and memory address, respectively. The instruction swaps the values stored in `src, addr`.

The logical behavior of `xchg` is an atomic function which basically performs the following

```
Xchg(value, addr) {
    old = *addr;
    *addr = value;
    return(old);
}
```

This allows the simple implemention of a spinlock.

3.4.3 Lock Acquire & Release with Xchg

```
Acquire(bool *lock) {
    while(Xchg(true, lock) == true);
}

Release(bool *lock) {
    *lock = false;
}
```

If `Xchg` returns `true`, the lock was already set and we must keep looping. Otherwise, the lock was free and we now acquired it.

This is known as a spin-lock since a thread busy-waits (loops) in Acquire until the lock is free.

3.4.4 ARM Synchronization Instructions

ARM offers exclusive load (`LDREX`) and store (`STREX`) operations.

The operations act as a barrier and MUST be used together.

`LDREX` loads a value from address `addr` while `STREX` will ATTEMPT to store a value to address `addr`. However, `STREX` will fail to store value at the address if it was touched between `LDREX` and `STREX`.

`STREX` may fail even if the distance between the two is small. However, it should succeed after a few attempts.

3.4.5 Lock Acquire with `LDREX, STREX`

```
ARMTestAndSet(addr, value) {
    tmp = LDREX addr
    result = STREX value, addr
    if (result == SUCCEED) return tmp
    return TRUE
}

void Acquire(bool *lock) {
    while(ARMTestAndSet(lock, true) == true);
}
```

Notice that the first function returns false only if we successfully set the result as well as the lock was false. So we only acquire the lock if it was free and we were the ones to "lock" it.

3.4.6 MIPS Synchronization Instructions

Similar to ARM instruction in the form of `ll, sc`.

load linked: loads value at address `addr`.

set conditional: conditionally stores a new value at `addr` if the value at `addr` has not changed since `ll`.

`sc` returns SUCCESS if the value stored at the address has not changed since `ll`. The value stored at the address can be any 32 bit value. The instruction does NOT check what the value at the address is, it only checks if it has changed!

3.4.7 Lock Acquire with `ll, sc`

```
MIPSTestAndSet(addr, value) {
    tmp = ll addr
    result = sc addr, value
    if (result == SUCCEED) return tmp
    return TRUE
}

void Acquire(bool *lock) {
    while(MIPSTestAndSet(lock, true) == true);
}
```

The logic behind this program is similar to the ARM spinlock.

3.5 OS/161: Spinlocks & Locks

3.5.1 Spinlocks

A spinlock is a lock that repeatedly tests lock availability in a loop until the lock is available. Threads actively use the CPU while they wait for the lock. In OS/161, spinlocks are already defined.

```
struct spinlock {
    volatile spinlock_data_t lk_lock;
    struct cpu *lk_holder;
};

void spinlock_init(struct spinlock *lk);
void spinlock_acquire(struct spinlock *lk);
void spinlock_release(struct spinlock *lk);
```

Remark that `spinlock_acquire` calls `spinlock_data_testandset` in a loop until the lock is acquired.

3.5.2 `spinlock_acquire`

```
// 0 indicates lock was acquired

spinlock_data_testandset(volatile spinlock_data_t *sd) {
    spinlock_data_t x, y;
    y = 1;

    __asm volatile(
        // assembly instructions
        // x=%0, y=%0, sd=%2
        ".set push;" // save assembler mode
        ".set mips32;" // allow MIPS32 instructions
        ".set volatile;" // avoid unwanted optimization
        "ll %0, 0(%2)" // x = *sd
        "sc %1, 0(%2)" // *sd = y; y = success?
        ".set pop" // restore assembler mode
        : "=r" (x), "+r" (y) : "r" (sd)
    );

    if(y == 0) return 1;
    return x;
}

"=r" // write only, stored in a register
"+r" // read and write, stored in a register
"r" // input, stored in a register
```

3.5.3 Locks

OS/161 also has locks in addition to spinlocks. The purpose of the mechanisms is also to enforce mutual exclusion.

```
struct lock *mylock = lock_create("LockName")

lock_acquire(mylock);
// critical section
lock_release(mylock)
```

Note that spinlocks spin and locks BLOCK. So a thread which calls `spinlock_acquire` spins until acquiring the lock while a thread which calls `lock_acquire` blocks until the lock can be acquired.

Locks are a type of mutex suitable for protecting large critical sections without being a

burden on the CPU. Remark that Locks have owners.

3.5.4 Additional Notes

Spinlocks and locks both have owners. This is to prevent them from being involutarily released.

A spinlock is owned by a CPU. A lock is owned by a thread.

Spinlocks disable interrupts on their CPU. This preemption is disabled on that CPU (hence owned by CPU but not others).

We should minimize spinning to maximize the utility of the CPU. In particular do NOT use spinlocks to protect large critical sections.

3.5.5 Spinlock & Lock API

```
void spinlock_init(struct spinlock *lk);
void spinlock_acquire(struct spinlock *lk);
void spinlock_release(struct spinlock *lk);
void spinlock_do_i_hold(struct spinlock *lk);
void spinlock_cleanup(struct spinlock *lk);

void lock *lock_create(const char *name);
void lock_acquire(struct lock *lk);
void lock_release(struct lock *lk);
void lock_do_i_hold(struct lock *lk);
void lock_destroy(struct lock *lk);
```

3.6 Thread Blocking

Sometimes a thread will need to wait for something. For example

- sdata to be released by another thread
- data from a (relatively) slow device
- input from a keyboard
- busy device to become idle

When a thread blocks, it stops running.

The scheduler then chooses a new thread to run. A context switch from the blocking thread to the new thread occurs. Then, the blocking thread is queued in a wait queue (NOT on the ready list).

Eventually, a blocked thread is signaled and awakened by another thread.

3.7 OS/161: Wait Channels

```
void wchan_sleep(struct wchan *wc); // blocks calling thread on wait channel
    wc, causes context switch
void wchan_wakeall(struct wchan *wc); // unblocks all threads sleeping on the
    wait channel
void wchan_wakeone(struct wchan *wc); // unblocks one (indeterminant) thread
    on the wait channel
void wchan_lock(struct wchan *wc); // prevent operations on wait channel,
    more on this later!
```

There can be many wait channels, holding threads that are blocked for completely different reasons. These channels are implemented with queues in OS/161.

3.8 Semaphores

Definition 3.8.1 (Semaphore)
An object that has an interger value supporting two atomic operations

P wait until the value is greater than 0, then decrement it.

V increment the value of the semaphore

A semaphore is a synchronization primitive that can be used to enforce mutual exclusion requirements. It can also be used to solve other kinds of synchronization problems.

3.8.1 Types of Semaphores

binary semaphore a semaphore with a single resource. It behaves like a lock, but does NOT keep track of ownership

counting semaphore a semaphore with an arbitrary number of resources

barrier semaphore a semaphore used to force one thread to wait for others to complete. The initial count is typically 0

3.8.2 Differences between Locks and Semaphores

- V does not follow P
- A semaphore can start with 0 resources. It calls V to increment the count. This forces a thread to wait until resources are produced before continuing
- Semaphores do NOT have owners

3.8.3 Mutual Exclusion Using a Semaphore

```
volatile int total = 0;
struct semaphore *total_sem;
total_sem = sem_create("total mutex", 1); // initial value of 1

void add() {
    for (int i=0; i<N; ++i) {
        P(sem);
        ++total;
        V(sem);
    }
}

void sub() {
    for (int i=0; i<N; ++i) {
        P(sem);
        --total;
        V(sem);
    }
}
```

3.8.4 Producer/Consumer Synchronization with Bounded Buffer

Suppose we have threads (producers) that add items to a buffer and threads (consumers) that remove items from the buffer.

We want consumers to wait if the buffer is empty and producers to wait if the buffer is full. This requires synchronization between consumers and buffers which semaphores can provide.

```
struct semaphore *items, *spaces;
items = sem_create("Buffer Items", 0)
spaces = sem_create("Buffer Spaces", N)
```

```
produce() {
    P(spaces);
    // add items to buffer
    V(items);
}

consume() {
    P(items);
    // remove item from buffer
    V(spaces);
}
```

Remark that we can still have race conditions as no mutual exclusion is enforced in the critical sections. To complete the pseudocode above, we should add a binary semaphore or more conventionally, a lock.

3.8.5 Semaphore Implementation

```
P(struct semaphore *sem) {
    spinlock_acquire(&sem->sem_lock);
    while(sem->sem_count == 0) {
        wchan_lock(sem->sem_wchan);
        spinlock_release(&sem->semlock);
        wchan_sleep(sem->sem_wchan);
        spinlock_acquire(&sem->sem_lock);
    }

    --sem->sem_count;
    spinlock_release(&sem->sem_lock);
}

V(struct semaphore *sem) {
    spinlock_acquire(&sem->sem_lock);
    ++sem->sem_count;
    wchan_wakeone(sem->sem_wchan);
    spinlock_release(&sem->sem_lock);
}
```

Remark that we must call `wchan_lock` manually before `wchan_sleep`. This allows us to release the spinlock but still maintain mutual exclusion as the internal lock of the wait channel is held. Indeed, before `wchan_sleep` is called, `wchan_wakeone` spins/blocks as it internally tries to obtain the wait channel lock. Only when the `P` thread sleepes will `V` release the semaphore spin lock and finish execution.

All other wait channel functions automatically lock and unlock internally but when we call `wchan_sleep`, it is normally with another (spin)lock held. Sleeping with a (spin)lock held leads to poor performance or even deadlocks as no thread is able to release the (spin)lock until the sleeping thread is woken up.

In this particular case, it would be impossible for `V` to increment the counter as it requires acquisition of the semaphore's internal spinlock, which would have been held in `P` (and not released before sleeping).

3.9 Condition Variables

OS/161 supports another common synchronization primitive: condition variables.

Each CV is intended to work together with a lock. CVs are only used from within the critical section that is protected by the lock.

Three operations are possible on a CV

wait causes the calling thread to block and releases the lock associated with the CV. Once the thread is unblocked, it reacquires the lock

signal if threads are blocked on the signaled CV, one of those threads is unblocked

broadcast similar to signal, but unblocks all threads that are blocked on the CV

Condition variables get their name as they allow threads to wait for arbitrary conditions to become true inside of a critical section. Normally, each CV corresponds to a particular condition that is of interest to an application.

For example, in the bounded buffer producer/consumer scenario, the two conditions are

(i) `count > 0`

(ii) `count < N`

When a condition is NOT true, a thread can wait on the corresponding CV until it becomes true. When a thread detects that a conditions is true, it uses signal or broadcast to notify any threads that may be waiting.

Remark that signals/broadcasts do not accumulate. This means that signalling/broadcasting a CV without waiters has NO effect.

Example 3.9.1

```
// solution with lock
int volatile numberOfGeese = 100;
lock geeseMutex;
```

```
int SafeToWalk() {
    lock_acquire(geeseMutex);
    while(numberOfGeese > 0) {
        lock_release(geeseMutex);
        lock_acquire(geeseMutex);
    }
}

// solution with CV
int volatile numberOfGeese = 100;
lock geeseMutex;
cv zeroGeese;

int SafeToWalk() {
    lock_acquire(geeseMutex);
    while(numberOfGeese > 0) {
        cv_wait(zeroGeese, geeseMutex);
    }
}
```

`cv_wait` will handle releasing and re-acquiring the lock passed in. It also puts the calling thread onto the conditions wait channel to block. `cv_signal` and `cv_broadcast` are used to wake threads waiting on the cv.

When a blocked thread is unblocked, it reacquires the lock before returning from the wait call.

It follows that a thread in the critical section when it calls `wait` will still be in the critical section upon return. However, between the call and the return, while the caller is blocked, the caller is out of the critical system, and other threads may enter.

This describes Mesa-style CVs, which are used in OS/161. There are alternative CV semantics (Hoare semantics), which differ from the semantics described here.

3.9.1 Bounded Buffer Producer/Consumer with Locks & CVs

```
int volatile count = 0;
struct lock *mutex;
struct cv *notfull, *notempty;

// initialize locks and cvs

Produce(itemType item) {
```

```
    lock_acquire(mutex);
    while (count == N) {
        cv_wait(notfull, mutex);
    }

    // add item to buffer
    count += 1;
    cv_signal(notempty, mutex);
    lock_release(mutex);
}

itemType Consume() {
    lock_acquire(mutex);
    while(count == 0) {
        cv_wait(notempty, mutex);
    }

    // remove item from buffer
    count -= 1;
    cv_signal(notfull, mutex);
    lock_release(mutex);
    return item;
}
```

3.10 Volatile & Other Sources of Race Conditions

Notice that throughout these slides, shared variables were declared `volatile`.

Race conditions can occur for reasons OTHER than the programmer's control. Specifically that of the compiler or the CPU.

In both cases, the compiler and CPU introduces race conditions due to optimizations.

Definition 3.10.1 (Memory Models)
Description of how thread access to memory in shared regions behave

A memory model tells the compiler and CPU which optimizations can be performed.

For example, it is faster to access values from a register than from memory. Compilers optimize for this by storing values in registers for as long as possible.

`volatile` disables this optimization, forcing a value to be loaded/stored to memory with each use. It also prevents the compiler from re-ordering loads and stores for that variable.

This is precisely what we want for shared variables and thus shared variables should be declared `volatile` in your code.

3.11 Other Language & Instruction level Instructions

Many languages supprot multi-threading with memory models and language-level synchronization functions (ie locks).

The compiler is aware of critical sections via language-level synchronization functions and does NOT perform optimizations which cause race conditions. The version of C used by OS/161 does NOT support this!

The CPU also has a memory model as it also re-orders loads and stores to improve performance.

Modern architectures provide barrier or fence instructions to disable and reenable these CPU-level optimizations to prevent race conditions at this level. The MIPS R3000 CPU used in this course does not have or require these instructions.

3.12 Deadlocks

Deadlocks happen when two threads holding locks attempt to acquire the locks of the other thread. Neither thread can make progress. Waiting will not resolve the deadlock and the threads are permanently stuck.

3.12.1 Techniques for Deadlock Prevention

No Hold and Wait A thread is NOT allowed to request for resources it if currently has resources allocated to it. A thread may hold several resources but must make a single request for all of them

Resource Ordering Order the resource types and require that each thread acquire resources in increasing resource type order. That is, a thread may make no requests for resources of type less than or equal to i if it is holding resources of type i.

4 Processes and the Kernel

4.1 What is a Process?

Definition 4.1.1 (Process)
An environment in which an application program runs.

A process includes virtualized resources that its program can use, including

1+ threads

virtual memeory for the program's code and data

as well as others such as file and socket descriptors.

Processes are created and managed by the kernel. Each program's process isolates it from other programs in other processes.

4.2 Process Management Calls

Processes can be created, managed, and destroyed. Each OS supports a variety of functions to perform these tasks.

Usage	Linux	OS/161
Creation	fork, execv	fork, execv
Destruction	_exit, kill	_exit
Synchronization	wait, waitpid, pause, …	waitpid
Attribute Management	getpid, getuid, nice, getrusage, …	getpid

4.2.1 `fork, _exit, & waitpid`

fork creates a new process (the child) that is a clone of the original (the parent). After `fork`, both parent and child are executing copies of the same program. Virtual memories of parent and child are identical at the time of the fork, but may diverge afterwards. Lastly, `fork` is called by the parent, but returns in BOTH the parent and the child. The parent and child see DIFFERENT return values from `fork`

_exit terminates the process that calls it. Processes can supply an exit status code when it exits. The kernel records the exit status code in case another process asks for it (via `waitpid`)

waitpid allows a process to wait for another to terminate. The exit status code is then retrieved.

Example 4.2.1

```
main() {
    rc = fork();

    if(rc == 0) {
        my_pid = getpid();
        x = child_code()
        _exit(x);
    }
    else {
        child_pid = rc;
        parent_pid = getpid();
        parent_code();
        p = waitpid(child_pid, &child_exit, 0);
        if(WIFEEXITED(child_exit)) {
            printf("child exit status was %d\n", WEXITSTATUS(child_exit));
        }
    }
}
```

4.2.2 `execv`

`execv` chanves the program that a process is running. The calling process's cirrent virtual memory is destroyed.

The process gets a new virtual memory, initiated with the code and data of the new program to run.

After `execv`, the new program starts executing.

Remark that the process ID stays the SAME. In addition, `execv` can pass arguments to the new program, if required.

Example 4.2.2

```
int main() {
    int rc = 0;
```

```
    char *args[4];

    args[0] = (char *) "/testbin/argtest";
    args[1] = (char *) "first";
    args[2] = (char *) "second";
    args[3] = (char *) "0";

    rc = esecv("/testbin/argtest", args);
    printf("If you see this execv failed\n");
    printf("rc = %d errno = %d\n", rc, errno);
    _exit(0);
}
```

4.2.3 Combining `fork`, `execv`

We can use `fork` to duplicate our current process and `execv` to change program the child (or parent if desired) is running.

```
main() {
    char *args[4];
    // set arguments

    rc = fork();
    if (rc == 0) {
        status = execv("/testbin/argtest", args);
        printf("If you see this execv failed\n");
        printf("rc = %d errno = %d\n", rc, errno);
        _exit(0);
    }
    else {
        child_pid = rc;
        parent_code();
        p = waitpid(child_pid, &child_exit, 0);
    }
}
```

4.3 System Calls

Definition 4.3.1 (System Call)
Process management calls by user programs.

These are the interface between processes and the kernel.

Servie	OS/161
create, destroy, and manage processes	fork, execv, waitpid, getpid
create, destroy, and manage files	open, close, remove, read, write
manage file system and directories	mkdir, rmdir, link, Synchronization
interprocess communication	pipe, read, write
manage virtual memory	sbrk
query, manage system	reboot, ___time

The user should not be able to execute provileged code. The system call library allows user programs to safely REQUEST for certain things which the kernel will ensure does not crash the OS.

4.4 Kernel Privilege

The CPU implements different levels (rings) of execution privilege as a security and isolation mechanism. Kernel code runs at the highest privilege level while application code runs at the lower privilege level as user programs should NOT be permitted to perform certain tasks such as halting the CPU or modifying the page tables that the kernel uses to implement process virtual memories (address spaces).

Programs cannot execute code or instructions belonging to a high-level of privilege. These restrictions allow the kernel to keep processes isolated from one another and from the kernel.

Application programs cannot directory call ernel functions or access kernel data structures.

The Meltdown vulnerability found on Intel chips let user applications bypass execution privilege and access any address in physical memory.

4.5 How System Calls Work

There are only two things which make kernel code run

Interrupts generated by devices when they require attention

Exceptions caused by instruction execution when a running program needs attention

4.5.1 Interrupts

Recall that an interrupt causes the hardware to transfer control to a fixed location in memeory where an interrupt handler is located. These handlers are part of the kernel.

When an interrupt occurs while an application program is running, control jumps from the application to the kernel's interrupt handler.

The processsor switches to privileged execution mode when it transfers control to the interrupt handler. This is how the kernel gets its execution provilege.

4.5.2 Exceptions

Definition 4.5.1 (Exception)
Conditions which occur during the execution of a program instruction.
ie arithmetic overflows, illegal instructions, page faults, etc.

Exceptions are detected by the CPU during instruction executions. The CPU handles exceptions like it handles interrupts. Control is trasferred to a fixed location where the exeception handler is located. Also, the processor is switched to privileged execution mode.

Note that the exeption handler is part of the kernel.

4.5.3 MIPS Exception Types

```
EX_IRQ 0 /* Interrupt */
EX_MOD 1 /* TLB Modify (write to read-only page) */
EX_TLBL 2 /* TLB miss on load */
EX_TLBS 3 /* TLB miss on store */
EX_ADEL 4 /* Address error on load */
EX_ADES 5 /* Address error on store */
EX_IBE 6 /* Bus error on instruction fetch */
EX_DBE 7 /* Bus error on data load *or* store */
EX_SYS 8 /* Syscall */
EX_BP 9 /* Breakpoint */
EX_RI 10 /* Reserved (illegal) instruction */
EX_CPU 11 /* Coprocessor unusable */
EX_OVF 12 /* Arithmetic overflow */
```

In MIPS, the SAME mechanism handles exceptions and interrupts, using the codes above to determine what trggered it to run.

To perform a system call, the application program needs to cause an exception to make the kernel execute. On MIPS, `EX_SYS` is the system call exception.

To cause the desired exception to run, the applicaiton executes a special instruction `syscall`. Other processors also include similar instructions.

The kernel's exception handler checks the exception code (set by the CPU when the exception is generated) to distinguish system call exceptions from other types of exceptions.

4.6 How are System Calls Implemented

4.6.1 System Call Codes

`fork, getpid` are two different system calls but there is only one `syscall` exeception. How does the kernel know which call the application procress is requesting?

The kernel defines a code for each system call it understands. Application processes are expected to place the application code at a specified location before executing the `syscall` instruction. For MIPS, the code does in register `v0`.

Such codes and code locations are defined as part of the kernel ABI (Application Binary Interface).

`/kern/include/kern/syscall.h`

```
#define SYS_fork 0
#define SYS_vfork 1
#define SYS_execv 2
#define SYS__exit 3
#define SYS_waitpid 4
#define SYS_getpid 5
```

4.6.2 System Call Parameters

System calls take parameters and return values just like functions. But how is this implemented in the context of exceptions?

The application is expected to place parameter values in a kernel-specified location before the syscall. The kernel will also place return values in the predetermined locations after the exception handler returns.

Again, this is part of the kernel ABI.

In MIPS, parameters go in registers `a0, a1, a2, a3`.

4.6.3 Kernel Exception Handler

1) create trap frame to save application program context
2) determine that this is a system call
3) determine which system call is being requested
4) does the actual work
5) restore the application program state from the trap frame
6) return from exception

4.7 User and Kernel Stack

Every OS/161 process thread has two stacks. However, it only uses one at a time.

4.7.1 User (Application) Stack

In use while application code is executing.

This stack is located in the application virtual memory. It holds activation records for application functions. The kernel creates this stack when it sets up the virtual address memory for the process.

4.7.2 Kernel Stack

In use while the thread is executing kernel code after an exception or interrupt.

This stack is a kernel structure. In OS/161, the `t_stack` field of the `thread` structure points to this stack. This stack holds activation records for kernel functions. It also holds trap frames and switch frames as the kernel creates trap frames and switch frames.

4.8 Exception Handling in OS/161

`kern/arch/mips/locore/exception-mips1.S`

1) assembly code
   (a) saves the application stack pointer
   (b) switches the stack pointer to the kernel stack

(c) saves appliction state and the address of the instruction that was interrupted in a trap frae on kernel stack

(d) calls `mips_trap`, passing a pointer to the trap frame as a parameter

2) the handler then restores the application state (application stack pointer) using the trap frame from the kernel stack

3) jump back to the application instruction that was interrupted

4.9 Multiprocessing

Definition 4.9.1 (Multiprocessing)
Having multiple processes existing at the same time.

All processes share the available hardware resources, with the sharing being coordinated by the operating system.

Each process' virtual memory is implemented using some of the available physical memory. The OS decides how much memory each process gets.

Each process' threads are scheduled onto the available CPUs (or CPU cores) by the OS.

In addition, processes share access to other resources (ie disks, network devices, I/O devices) by making system calls. The OS controls this sharing.

It is the responsibility of the OS ensure that processes are isolated from one another. Inter-process communication should be possible, but only at the explicit request of the processes involved.

Note that Processes can have many threads, but MUST have at least one to execute. OS/161 only supports a single thread per process.

4.10 Inter-Process Communication (IPC)

Processes are isolated from each other. But, what if they want to communicate (share data) with each other?

IPC is a family of methods used to send data between processes.

File data to be shared is written to a file and accessed by both processes

Socket data is sent via network interface between processes

Shared Memory data is sent via blocks of shared memory visible to both processes

Message Passing/Queue a queue/data stream provided by the OS to send data between processes

5 Virtual Memory

5.1 Motivation

If physical addresses are P bits, then the maximum amount of addressable memory is 2^P bytes.

Sys/161 MIPS uses $P = 32$. In the following notes, we will use $P = 18$ for the sake of simplicity.

Modern kernels provide a separate private virtual memory for each process which holds code, data, and stack of the process as before. If virtual addresses are V bits, then the maximum amount of addressable virtual memory is 2^V bytes.

Again, $V = 32$ for MIPS but we will use $V = 16$ for these notes.

The following all are virtual addresses

- program counter
- stack counter
- variable pointers
- destinations of jump and branch instructions

The important thing to note is that each process is isolated in its virtual memory space and CANNOT access the virtual memory of other processes.

VM not only provides separation between memory spaces but also gives rise to the possibility of VM larger than physical memory. Also, the total size of all VMs can be larger than physical memory.

Each access to VM requires a translation to physical memory. The load or store is then applied to the physical memory. This address translation is done in hardward, in the Memory Management Unit (MMU).

Definition 5.1.1 (Memory Management Unit)

5.2 Segmentation

Realistically, instead of mapping each individual VM address to a physical addres, we map each segment of virtual memory to a chunk of physical memory.

We can think of the V bits of VM addresses as having K bits to indicate a segment ID and $V - k$ bits to indicate offset from a segment.

5.2.1 Translation Using Registers

One way of translating virtual segments is for the MMU to have a relocation and limit register for each segment.o

As for dynamic relocation, the kernel maintains a separate set of relocation offsets and limits for each process, and changes the values in the MMU's registers when there is a context switch between processes.

5.2.2 Segment Table

Another way to go about the translation is to use a segment table.

Given a valid segment number, use the segment table to lookup the limit and relocation values from the segment table.

5.3 Paging

VM is divided into pages which has the same size as frames. Each page is mapped to one frame and each frame can map to any frame. Pages a mapped with a Page Table.

Definition 5.3.1 (Page Table)

Each Page Table Entry contains the frame the page is mapped to then a valid bit.o

5.3.1 MMU

The MMU gets the page number and offset using the VM address.. Then, it determines the physical frame number using the PT (if valid). Finally, the appropriate physical address is computed.

5.3.2 Page Table Entries

Each entry may contains additional information.

The write protection bit determines if a page is read-only.

The reference bit (used by caches) tells us if the page has been recently used.

The dirty bit determines if the page has been written to and thus needs to be written back to physical memory.

5.3.3 Multi-Level Paging

Page Tables can get very very large.

Instead of only having one table. We can use a trie of tables where leaves give actual physical pages and each fix sized segment of the VM address up to a predetermined bit is a PT pointing to the next trie node (PT).

The advantage of this is that if a PT contains no valid PTE, it does NOT have to be created!

We want each table to fit within a page, if it does not, create more levels until the desired result is attained.

5.4 Kernel & MMU

Kernel responsible for

- Managing MMU registers on address space switches
- Creating and managing PT
- Managing physical memory
- Handling exceptions raised by the MMU

MMU responsible for

- Translating VM addresses to physical addresses
- Checking for and raising exceptions when necessary