CS 350
Personal Notes

Topic 1: Introduction to Operating Systems

An OS is "part cop, part facilitator".

An OS is a system that
1. manages resources;
   eg processor, memory, I/O
2. creates execution environments;
   eg interfaces → resources
3. loads programs; &
4. provides common services & utilities.

THREE VIEWS OF AN OS

3 views:
1. "Application view" — what services does it provide?
2. "System view" — what problems does it solve?
3. "Implementation view" — how is it built?

APPLICATION VIEW

The OS provides an "execution environment" for running programs.

More specifically:
1. Provides a program with resources that it needs to run;
2. Provides interfaces through which a program can use networks, storage, I/O devices and other system hardware components; &
3. Isolates running programs from one another and prevents undesirable interactions among them.

SYSTEM VIEW

The problems an OS solves:
1. It manages the hardware resources of a computer system;
2. Allocates resources among running programs; &
3. Controls the access to or sharing of resources among programs.

IMPLEMENTATION VIEW

The OS is a concurrent, real-time program.

CONCURRENT

"Concurrent" means multiple programs or sequences of instructions running or appearing to run at the same time.

Concurrency arises naturally in an OS when it supports concurrent applications, and because it must interact directly with the hardware.

REAL-TIME [PROGRAM]

A "real-time" program is a program that must respond to an event in a specific amount of time.

eg when you are watching a video, you do not want it to freeze up or skip.

KERNEL

The "kernel" of an OS is the part of the OS that responds to system calls, interrupts & exceptions.

Intuitively, it is the part of the OS that is always running from start-up to boot-down.

OPERATING SYSTEM

The OS on a whole includes the kernel, and may include other related programs that provide services for apps.

These include
1. Utility programs;
2. Command interpreters; &
3. Programming libraries.

MONOLITHIC KERNEL

A "monolithic kernel" has "everything" as part of the kernel, including device drivers, file system, virtual memory etc.

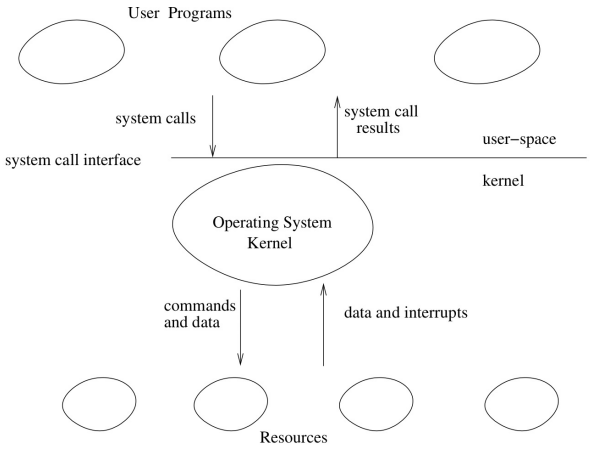
MICROKERNEL

A "microkernel" includes only absolutely necessary components; all other elements are user programs.

REAL-TIME OS

A "real-time OS" is an OS with a stringent event response time, guarantees, & pre-emptive scheduling.

SCHEMATIC VIEW OF AN OS



USER PROGRAMS

"User programs" are programs that the user interacts with directly.

eg desktop environments, web browsers, games, editors, compilers, etc

USER SPACE

The "user space" refers to all programs that run outside the kernel.

These programs do not interact with the hardware directly.

SYSTEM CALL

A "system call" is how a user process interacts with the OS.

KERNEL SPACE

The "kernel space" is the region of memory where the kernel runs.

OS ABSTRACTIONS

The execution environment provided by the OS includes a variety of abstract entities that can be manipulated by a running program.

Examples:

1. Files & file systems
   - ↳ secondary storage (HDD/SSD)
2. Address spaces
   - ↳ primary memory (RAM)
3. Processes & threads
   - ↳ program execution (processor cores)
4. Sockets & pipes
   - ↳ network & other message channels

Topic 2: Threads and Concurrency

THREADS

1. A "thread" is a sequence of instructions.
2. A "normal" sequential program consists of a single thread of execution.
3. A program can have multiple threads or a single thread of execution.
   - ↳ similar to NFA (set of current states) vs DFA (one current state)
4. In particular, a single program can have
   1. different threads responsible for different roles;
   2. multiple threads responsible for the same roles.
      - eg web-server might have 1 thread for each person using the site

CONCURRENCY

1. "Concurrency" occurs when multiple programs / sequences of instructions make progress; ie they run / appear to run at the same time.
2. Threads provide a way to express concurrency in a program.

PARALLELISM

1. "Parallelism" is when multiple programs or sequences of instructions can run at the same time.
   - ie there are multiple processors / cores.

WHY THREADS?

1. Advantages:
   1. Efficient use of resources
      - one thread can run whilst the other waits
   2. Parallelism
      - diff threads can run on diff cores
   3. Responsiveness
      - one thread can be responsible for a long task
   4. Priority
   5. Modular code
2. A thread blocks for a period of time or until some condition has been met; concurrency lets the processor execute a different thread during this time.

thread_fork

1. A thread can create new threads using `thread_fork`.
2. New threads start execution in a function specified as a parameter to `thread_fork`.
3. The original thread & new thread now proceed concurrently as two simultaneous sequential threads of execution.
4. Note:
   1. Each thread has its own private stack;
   2. All threads share access to the program's global variables and heap; &
   3. In the OS, a thread is represented by a structure / object.

Examples: kern/test/threadtest.c; kern/test/traffic.c.

OS/161's THREAD INTERFACE

1. Create a new thread;

```
int thread_fork(
 const char *name,          // name of new thread
 struct proc *proc,         // thread's process
 void (*func)               // new thread's function
  (void *, unsigned long),
 void *data1,               // function's first param
 unsigned long data2        // function's second param
);
```

* see kern/include/thread.h for details.

2. Terminate the calling thread;

```
void thread_exit(void);
```

3. Voluntarily yield execution;

```
void thread_yield(void);
```

4. Note: we cannot control the order that the threads run in without using additional tools.
   - eg synchronization primitives.

SEQUENTIAL PROGRAM EXECUTION

1. A single threaded program uses the "fetch-execute" cycle.

MIPS REGISTERS FOR OS/161

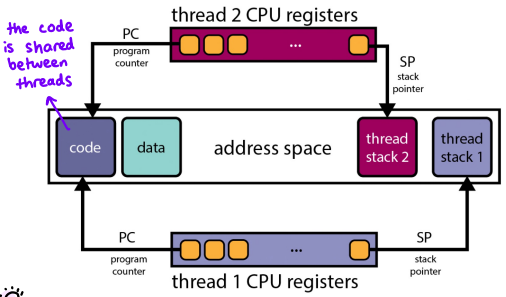
num	name	use	num	name	use
0	z0	always zero	24-25	t8-t9	temps (caller-save)
1	at	assembler reserved	26-27	k0-k1	kernel temps
2	v0	return val/syscall #	28	gp	global pointer
3	v1	return value	29	sp	stack pointer
4-7	a0-a3	subroutine args	30	s8/fp	frame ptr (callee-save)
8-15	t0-t7	temps (caller-save)	31	ra	return addr (for jal)
16-23	s0-s7	saved (callee-save)			

* you don't need to memorize this!

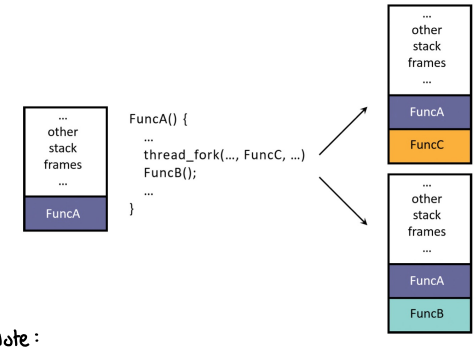
(see kern/arch/mips/include/kern/regdefs.h).

CONCURRENT PROGRAM EXECUTION (2 THREADS)

1. Each thread executes sequentially using its private register contents and data.



2. The stack gets duplicated with thread_fork; when it is executed, a copy of the stack & register values are copied over into the new thread.



3. Note:
   1. Both threads are executing the same program.
   2. Both threads share the same code, read-only data, global variables & heap, but each thread may be executing different functions within the code.
   3. Each thread has its own stack and program counter (PC).
   4. Each thread has a fixed size.

IMPLEMENTING CONCURRENT THREADS: TIMESHARING

1. Options:
   1. Hardware support
      - P processors, C cores, M multithreading per core
      - PxCxM can execute simultaneously
   2. "Timesharing" — multiple threads take turns on the same hardware, rapidly switching between threads so they all make progress.
   3. Both 1 & 2!

MULTICORE PROCESSOR

1. A "multi-core processor" has the property that / a single die / computer chip can contain more than one processor unit.
2. These units can share some components, eg L3 cache, but execute code separately.

MULTITHREADING

1. "Multi-threading" refers to the property of having multiple threads run on the same core.
2. We need specialized hardware to do this.

CONTEXT SWITCH (IN TIMESHARING)

1. A "context switch" is the switch from one thread to another in time-sharing.
2. Process:
   1. "Scheduling" — CPU decides which thread to run next
   2. Save register contents of current thread
   3. Load register contents of next thread
3. The "thread context" (register values) must be saved/restored carefully, since thread execution continuously changes the context.

CONTEXT SWITCHING ON MIPS

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

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

   /* Allocate stack space for saving 10 registers. 10*4 = 40 */
   addi sp, sp, -40

   sw   ra, 36(sp)     /* Save the registers */
   sw   gp, 32(sp)
   sw   s8, 28(sp)     /* a.k.a. frame pointer */
   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 the old stack pointer in the old thread */
   sw   sp, 0(a0)


   /* Get the new stack pointer from the new thread */
   lw   sp, 0(a1)
   nop             /* delay slot for load */   --> to avoid load-use hazards.

   /* Now, restore the 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)       /* a.k.a. frame pointer */
   lw   gp, 32(sp)
   lw   ra, 36(sp)
   nop                         /* delay slot for load */

   /* and return. */
   j ra
   addi sp, sp, 40      /* in delay slot */
   .end switchframe_switch
```

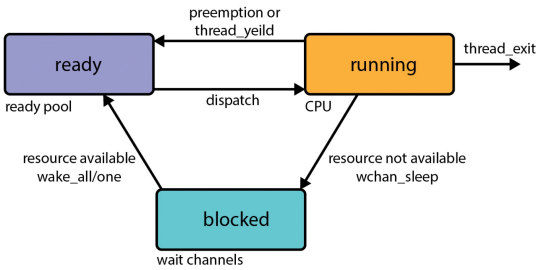
Switchframe_switch

1. The C function `thread_switch` calls the subroutine `switchframe_switch`.
2. `thread_switch` is the caller; it saves & restores the caller-save registers, including the return address (ra)
3. `switchframe_switch` is the callee; it must save & restore the callee-save registers, including the FP in OS/161.

* high-level functions are in C; low-level functions are in assembly.

THREAD STATES

1. States:
   1. "Running" — currently executing
   2. "Ready" — ready to execute
   3. "Blocked" — waiting for something, so not ready to execute.



TIMESHARING

"Timesharing" is concurrency achieved by rapidly switching between threads.

SCHEDULING QUANTUM

The "scheduling quantum" is a limit on processor time that is an upper bound on how long a thread can run before it needs to yield to the processor.

PREEMPTION

"Preemption" forces a running thread to stop running, so another thread can run.

- to implement this, the thread library must have a means of "getting control".

This is usually accomplished using "interrupts".

INTERRUPTS

An "interrupt" is an event that occurs during a program's execution, and are caused by system devices (hardware).

When an interrupt occurs, the hardware automatically transfers control to a fixed location in memory.

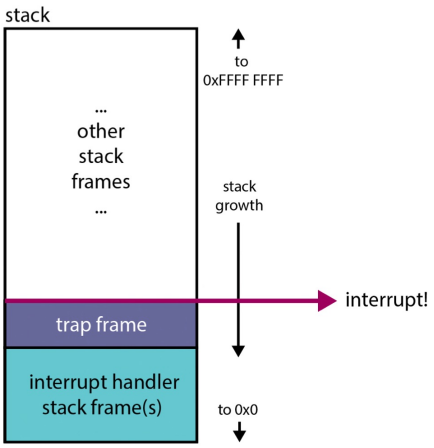
Here, the thread library must place the "interrupt handler".

INTERRUPT HANDLER

The "interrupt handler" is a procedure that:

1. creates a "trap frame" to record the thread context at the time of interrupt;
2. determines which device caused the interrupt & performs device-specific processing;
3. restores the saved thread context from the trap frame & resumes execution of the thread.

OS/161 THREAD STACK AFTER AN INTERRUPT



SCHEDULING

"Scheduling" means deciding which thread should run next.

This is implemented by a "scheduler", which is a part of the thread library.

PREEMPTIVE ROUND-ROBIN SCHEDULING

"Pre-emptive round-robin scheduling" involves:

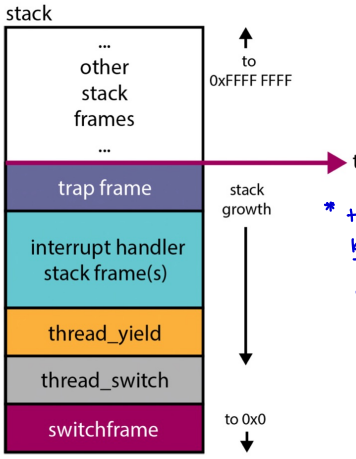
1. scheduler maintains the "ready queue" of threads;
2. on a context switch, the running thread is moved to the end of the ready queue, and the first thread on the queue is allowed to run.
3. Newly created threads are placed at the end of the ready queue.

* OS/161 uses this type of scheduling.

The order threads run is non-deterministic since threads

1. can be migrated to other cores; or
2. interrupted by a device.

OS/161 THREAD STACK AFTER PRE-EMPTION



* this is the kernel space stack — used to call kernel functions for threads (eg `thread_yield`)

VOLUNTARY CONTEXT SWITCH

In a "voluntary context switch",

1. The thread calls `thread_yield`, which calls `thread_switch`.
2. This picks another runnable thread & calls `switchframe_switch` to store a switchframe on the stack.
3. We then do the context switch.

INVOLUNTARY CONTEXT SWITCH

In an "involuntary context switch":

1. The thread is interrupted, resulting in a trap frame, followed by a stack frame for an interrupt handler.
2. This calls thread_yield, and then we follow the steps for a voluntary context switch.

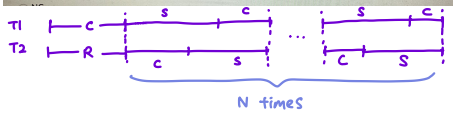
EXAMPLE

There are two threads in an OS that uses preemptive round-robin scheduling with a scheduling quantum of Q milliseconds. The system has a single processor with a single core. Each thread runs a function which behaves as follows:

```
for i from 1 to N do
  compute for C milliseconds
  sleep for S milliseconds
end
```

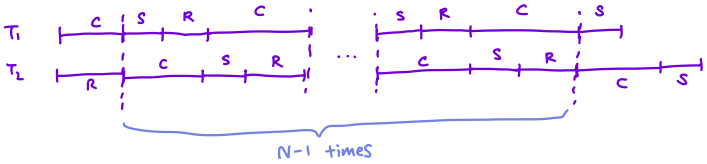
At the end of its for loop, a thread is finished and it exits. During the compute part of each iteration, a thread is runnable (running or ready to run). During the sleep part of each of its iterations, a thread is blocked. For both parts of this question C < Q, i.e. the compute time is not interrupted.

Both of the threads are created at time t = 0 and C < S. At what time will both of the threads be finished?



⇒ time = C + N(C+S)

If C>S:



⇒ time = C + (N-1)(2C) + C + S = 2NC + S

Topic 3: Synchronization

MOTIVATION

Consider

```
int volatile total = 0;

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

If one thread executes add & one executes sub, what is `total` when they have finished?

The value of `total` can not be 0!

To understand this, consider the code at the (pseudo) assembly language level.

TRACE 1: SEQUENTIAL EXECUTION

```
;; Thread 1                 Thread 2
loadaddr R8 total
lw R9 0(R8)   ; R9=0
add R9 1      ; R9=1
sw R9 0(R8)   ; total=1
        ----- Preemption ----->
                            loadaddr R10 total
                            lw R11 0(R10)   ; R11= 1
                            sub R11 1       ; R11= 0
                            sw R11 0(R10)   ; total= 0
```

→ total = 0.

TRACE 2: INTERLEAVED EXECUTION

```
;; Thread 1                 Thread 2
loadaddr R8 total
lw R9 0(R8)   ; R9= 0
add R9 1      ; R9= 1
          ----- Preemption ----->
                            loadaddr R10 total
                            lw R11 0(R10)   ; R11= 0
                            sub R11 1       ; R11= -1
                            sw R11 0(R10)   ; total= -1
         <----- Preemption -----
sw R9 0(R8) ; total= 1
```

→ total = 1

TRACE 3: MULTI-CORE EXECUTION

```
;; Thread 1                 Thread 2
loadaddr R8 total
lw R9 0(R8)   ; R9=0        loadaddr R10 total
add R9 1      ; R9=1        lw R11 0(R10)   ; R11=0
sw R9 0(R8)   ; total=1     sub R11 1       ; R11=-1
                            sw R11 0(R10)   ; total=-1
```

→ total = -1.

RACE CONDITION

A "race condition" is when the program result depends on the order of execution of the threads.

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

Sources:

- implementation of code.

To identify the critical sections for race conditions:

1. inspect each variable and ask if it is possible for multiple threads to read & write it concurrently.
2. note constants & memory that are read-only by all threads do not cause race conditions.

ENFORCING MUTUAL EXCLUSION WITH LOCKS

```
int volatile total = 0;

void add() {                    void sub() {
   int i;                          int i;
   for (i=0; i<N; i++) {           for (i=0; i<N; i++) {
   ------------ start mutual exclusion ------------
      total++;                        total--;
   ------------ stop mutual exclusion  ------------
   }                               }
}                               }
```

MUTEX / MUTUAL EXCLUSION

"Mutex" is a method to ensure only one thread can access this region at a time.

LOCK

A lock is a mechanism to provide mutual exclusion.

```
int volatile total = 0;
bool volatile total_lock = false;  // false means unlocked
                                   // true means locked

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

- acquire/release must ensure that only 1 thread can hold the lock at a time
- if a thread cannot acquire the lock immediately, it must wait until the lock becomes available

One possible implementation of acquire & release:

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

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

- "spinning" - a thread continually checks for a variable to change in a while loop.

But this does not work.

Why? → in the time a thread breaks out of the while loop (ie *) it could be preempted & another thread could then set *lock to true.

→ so there are 2 threads in the critical section.

TEST-AND-SET

Approach:
create a function that given the address of a lock, `lock`, between when the `*lock` is tested & set, no other thread can change its value.

MIPS SYNCHRONIZATION INSTRUCTIONS

We use
① "ll" (load linked): load value at address addr
② "sc" (store conditional): store new value at addr if the value at addr has not changed since ll was executed.

sc returns "success" if the value stored at addr has not changed since ll was executed.

Implementation of locks:

```
MIPSTestAndSet(addr, new_val) {
   old_val = ll addr           // test
   status = sc addr, new_val   // set
   if ( status == SUCCEED ) return old_val
   else return true            // lock is being held
}

Acquire(bool *lock) {          // spin until hold lock
   while( MIPSTestAndSet(lock, true) == true ) {};
}
```

	lock value at ll	lock value before sc	lock value after sc	status	Lock State
1.	false	false	true	succeed	lock acquired
2.	true	true	true	succeed	keep spinning, no lock
3.	false	true	true	fail	keep spinning, no lock
4.	true	false	false	fail	keep spinning, no lock

SPINLOCKS IN OS/161

A "spin-lock" is a lock that spins; ie it repeatedly tests the lock's availability in a loop until the lock is returned.

When threads use a spin-lock, they "busy-wait" — they use the processor whilst they wait for the lock.

Spinlocks are efficient for short waiting times. (a few instructions).

OS/161 uses spinlocks to
① update shared variables / data structures
② implement other synchronization primitives (eg locks, semaphores, condition variables)

OS/161 uses ll & sc to implement test-and-set.

LOCKS IN OS/161

Locks also enforce mutex, but they block (instead of spinning, like spinlocks).

Locks can be used to protect larger critical sections without being a burden on the processor.

SPINLOCKS VS LOCKS

Similarities:
① The thread would `acquire` it (access the critical section) & `release` it (when it is done)
② Both need to be initialized / created before using them.
③ Both have `do_i_hold`.
④ Both have owners & can only be released by their owner. But:
- a spinlock is owned by a CPU;
- a lock is owned by a thread.

Why? → in OS/161, interrupts are disabled on the CPU that holds the spinlock but not other CPUs
→ to minimize spinning.

THREAD BLOCKING

When a thread blocks, it stops running; ie it stops using the processor.
① the scheduler chooses a new thread to run;
② a context switch from the blocking thread to the new thread occurs; &
③ the blocking thread is queued in a wait channel (not the ready queue).
④ eventually, a blocked thread is signaled & awakened by another thread.

WAIT CHANNELS IN OS/161

"Wait channels" are used to implement thread blocking in OS/161.

- `void wchan_sleep(struct wchan *wc);`
  - blocks calling thread on wait channel `wc`
  - causes a context switch, like `thread_yield`
- `void wchan_wakeall(struct wchan *wc);`
  - unblock all threads sleeping on wait channel `wc`
- `void wchan_wakeone(struct wchan *wc);`
  - unblock one thread sleeping on wait channel `wc`
- `void wchan_lock(struct wchan *wc);`
  - prevent operations on wait channel `wc`
  - more on this later!

SEMAPHORES

1. "Semaphores" are synchronization primitives that can be used to enforce mutual exclusion requirements, & can be used to solve other kinds of synchronization problems.
2. It is an object with an integer value & supports 2 operations:
   ① "P" (procure): if the semaphore value > 0, decrement it; otherwise, wait until the value is > 0 & then decrement it.
   ② "V" (vacate): increment the value of the semaphore.

TYPES OF SEMAPHORES

1. "Binary semaphore" — a semaphore with a single resource; behaves like a lock but does not keep track of ownership.
2. "Counting semaphore": a semaphore with an arbitrary # of resources.
3. "Barrier semaphore": a semaphore used to force 1 thread to wait for others to complete; initial count is typically zero.

LOCK VS SEMAPHORE

Differences:
① V does not have to follow P;
② A semaphore can start with a count of 0 (ie no resources available).
③ Calling V increments the semaphore by 1;
④ This sequence of operations forces a thread to wait until resources are produced before continuing;
⑤ Semaphores do not have owners; locks do.

IMPLEMENTATION OF SEMAPHORES

```
1.  P(struct semaphore * sem) {
2.    spinlock_acquire(&sem->sem_lock);
3.    while (sem->sem_count == 0) {       // test
4.        wchan_lock(sem->sem_wchan);     // prepare to sleep
5.        spinlock_release(&sem->sem_lock);
6.        wchan_sleep(sem->sem_wchan);    // sleep
7.        spinlock_acquire(&sem->sem_lock);
8.    }
9.    sem->sem_count--;                   // set
10.   spinlock_release(&sem->sem_lock);
11. }
```

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

DESIGNING MULTITHREADED CODE

When designing/debugging multi-threaded code, consider what could go wrong if our code was preempted at each step at the procedure.

CONDITION VARIABLES

1. "Condition variables" have 3 operations:
   ① "Wait": this causes the calling thread to block, & releases the lock associated with the condition variable. Once the thread is unblocked it reacquires the lock.
   ② "Signal": if threads are blocked on the signaled condition variable, then one of those threads are unblocked.
   ③ "Broadcast": like signal, but unblocks all threads that are blocked on the condition variable.
2. Condition variables are intended to work together with locks.
3. These are also only used from within the critical section that is protected by the lock.
4. Condition variables allow threads to wait for arbitrary conditions to become true inside a critical section.
   ① If the condition is false, a thread can `wait` on the corresponding condition variable until it becomes true.
   ② If it is true, it uses `signal` or `broadcast` to notify any blocked threads.

EXAMPLE: GEESE

```
int volatile numberOfGeese = 100;
lock geeseMutex;
cv zeroGeese;          → condition variable

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

- `cv_wait` handles releasing/reacquiring the lock passed in.
- it will put the calling thread in the cv's wait channel to block
- cv_signal & cv_broadcast wake threads waiting on the cv.

VOLATILE

Race conditions can occur for reasons beyond the programmer's control; in particular they can be caused by

① the compiler; or

② the CPU.

These are due to optimizations to make the code run faster.

Compilers optimize for this difference, storing values in registers for as long as possible.

But for shared variables, this might not work.

eg
```
int sharedTotal = 0;
int FuncA() { ... code that modifies sharedTotal ... }
int FuncB() { ... code that modifies sharedTotal ... }
```
- if funcA stores sharedTotal in register S1
- & funcB stores it in register S2
- which register has the correct value?

"Volatile" disables this, forcing the value to be loaded from & stored to memory with each use.

- prevents compiler from re-ordering loads & stores for that variable.

Shared variables should be declared `volatile`!

DEADLOCKS

We say threads are "deadlocked" if neither can make progress, as they are blocking each other.

eg
```
lock lock1, lock2;
int FuncA() {                  int FuncB() {
   lock_acquire(lock1)            lock_acquire(lock2)
   lock_acquire(lock2)            lock_acquire(lock1)
   ...                            ...
   lock_release(lock1)            lock_release(lock2)
   lock_release(lock2)            lock_release(lock1)
}                               }
```

What happens if the instructions are interleaved as follows:

- Thread 1 executes `lock_acquire(lock1)`
- Thread 2 executes `lock_acquire(lock2)`
- Thread 1 executes `lock_acquire(lock2)`
- Thread 2 executes `lock_acquire(lock1)`

This is what happens...

- Thread 1 executes `lock_acquire(lock1)` and holds it.
- Thread 2 executes `lock_acquire(lock2)` and holds it.
- Thread 1 executes `lock_acquire(lock2)` and blocks because `lock2` is being held by Thread 2.
- Thread 2 executes `lock_acquire(lock1)` and blocks because `lock1` is being held by Thread 1.

How to prevent deadlocks?

① "No Hold & Wait": prevent a thread from requesting resources if it currently has resources allocated to it.

- a thread may hold several resources, but must obtain them all at once.

eg
```
lock_acquire(lock1);     // try get both locks
while(!try_acquire(lock2)) {
   lock_release(lock1); // didn't get lock2 so try again
   lock_acquire(lock1);
}                        // can now obtain both resources
```

② "Resource Ordering": order the resource types.

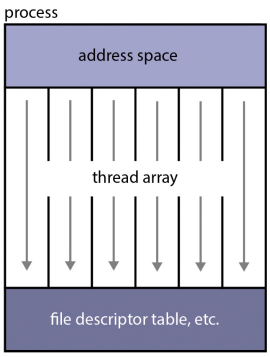
- each thread must acquire resources in increasing resource type order.

Topic 4: Processes and the Kernel

PROCESS

A "process" is an environment in which an application program runs.
ie a "program that is running".

The process's environment includes virtualized resources that its program can use:
① Threads to execute code on processors;
② An "address space" for code & data;
③ A file system;
④ I/O.
eg keyboard, display



Processes are created & managed by the kernel, & each program's process isolates it from other programs running in other processes.
- when a program is executed, it seems like it has exclusive access to the processor, RAM & I/O even though they are actually shared.

VIRTUAL MEMORY

"Virtual memory" is an address space that employs virtualization with RAM & secondary storage.

It appears to have a large amount of primary memory (RAM) that the process has exclusive access to for its code, data & stacks.

VIRTUALIZED RESOURCE

A "virtualized resource" is to take a physical resource (eg hard drive) & create a more abstract version of it (eg file system).

PROCESS MANAGEMENT CALLS

Processes can be created, managed & destroyed, & each OS supports a variety of functions to support these tasks.

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

fork

The system call `fork` creates a new process, the "child" (C), that is a clone of the parent (P), the original process.
- it copies P's address space (code, globals, heap, stack) into C's.
- it makes a new thread for C.
- it copies P's trap frame to C's (kernel) stack

The address space of P & C are identical at the time of the fork, but may diverge afterwards.

`fork` is called by the parent, but the function returns in both the parent & child.
- it returns 0 to the child process
- it returns the child's pid to the parent process
- this allows us to distinguish the child & parent.

_exit

`_exit` terminates the process that calls it.
- a process can supply an exit status code when it exits
- the kernel records the exit status code in case another process needs it, via waitpid.

waitpid

`waitpid` lets a process wait for another to terminate/block, and then retrieves its exit status code.
* pid = process identifier

EXAMPLE 1 : fork, _exit, getpid, waitpid

```
1.  main() {
2.     int rc = fork();    // returns 0 to child, pid to parent
3.     if (rc == 0) {      // child executes this code
4.        my_pid = getpid();
5.        x = child_code();
6.        _exit(x);  <-- ends process w/ return value x
7.     } else {           // parent executes this code
8.       child_pid = rc;
9.       parent_pid = getpid();
10.      parent_code();
11.      p = waitpid(child_pid,&child_exit,0);  <-- wait for child to exit. get its exit status
12.      if (WIFEXITED(child_exit))  <-- returns true if child called _exit()
13.        printf("child exit status was %d",
             WEXITSTATUS(child_exit))  <-- returns exit status code
14.    }
15. }
```

getpid

`getpid` returns the process identifier (ie pid) of the current process.
- each existing process has its own unique pid to identify it

execv

`execv` changes the program that a process is running.
- the calling process's current virtual memory is destroyed
- the process gets a new virtual memory (code, heap, stack etc) initialized with the code & data of the new program to run.
- the pid remains the same.

After `execv`, the new program starts executing.

`execv` can pass arguments to the new program if required.

EXAMPLE 2 : execv

```
//    sample code that uses execv to execute the command:
//    /testbin/argtest first second

      int main()  {
 1.     int status = 0;  // status of execv function call
 2.     char *args[4];   // argument vector

                         // prepare the arguments
 3.     args[0] = (char *) "/testbin/argtest";
 4.     args[1] = (char *) "first";
 5.     args[2] = (char *) "second";
 6.     args[3] = 0;     // end of args

 7.     status = execv("/testbin/argtest", args);
 8.     printf("If you see this output then execv failed");
 9.     printf("status = %d errno = %d", status, errno);
10.     exit(0);
      }
```

EXAMPLE 3 : execv & fork

```
// the child executes execv while the parent waits

      main() {
 1.     char * args[4];     // args for argtest
 2.     // set args here
 3.     int rc = fork();    // returns 0 to child, pid to parent

 4.     if (rc == 0) {  // child's code
 5.       status = execv("/testbin/argtest",args);
 6.       printf("If you see this output, then execv failed");
 7.       printf("status = %d errno = %d", status, errno);
 8.       exit(0);

 9.     } else {        // parents's code
10.       child_pid = rc;
11.       parent_code();
12.       p = waitpid(child_pid,&child_exit,0);
        }
      }
```

SYSTEM CALLS

"System calls" are the interface between user processes & the kernel.

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

SOFTWARE STACK