

# CS 471

OS/161 Threads/Synchronization

# Tips and Tricks

- Get familiar with code, specifically the following directories in kern
  - thread, userprog, main, vm – You might end up modifying files in these
  - Inside /mips: spl.c, syscall.c, trap.c, exception.s, threadstart.s
- Get familiar with the following commands in zeus:
  - Screen –creates new windows (Will be helpful in debugging)
  - Find (search for files)
  - Grep is your friend (grep –rw ‘path’ –e ‘pattern’ search for pattern)
  - If you are using two separate sessions for debugging use the same node in both sessions (zeus runs on two nodes use zeus1.vse.gmu.edu instead of just zeus.vse.gmu.edu while logging in)
  - Familiarize with some gdb commands: where, frame, info
  - [https://ccrma.stanford.edu/~jos/stkintro/Useful\\_commands\\_gdb.html](https://ccrma.stanford.edu/~jos/stkintro/Useful_commands_gdb.html)
  - <https://sourceware.org/gdb/onlinedocs/gdb/Registers.html>

# Review: Program Execution

- Registers
  - program counter, stack pointer, . . .
- Memory
  - program code
  - program data
  - program stack containing procedure activation records
- CPU
  - fetches and executes instructions

# Implementing Threads

- A thread library is responsible for implementing threads
- The thread library stores threads' contexts (or pointers to the threads' contexts) when they are not running
- The data structure used by the thread library to store the misc. hardware-specific thread context is called a *thread control block* (in os161 `t_pcb`)
- In the OS/161 kernel's thread implementation, thread contexts are stored in thread structures

# The OS/161 Thread Structure

```
struct pcb {
    u_int32_t pcb_switchstack; // stack saved during context switch
    u_int32_t pcb_kstack;      // stack to load on entry to kernel
    u_int32_t pcb_ininterrupt; // are we in an interrupt handler?

    pcb_faultfunc pcb_badfaultfunc; // recovery for fatal kernel traps
    jmp_buf pcb_copyjmp;           // jump area used by copyin/out etc.
};
```

```
struct thread {

    /* Private thread members - internal to the thread system */

    struct pcb t_pcb;           /* misc. hardware-specific stuff */
    char *t_name;              /* thread name */
    const void *t_sleepaddr;   /* used for synchronization */
    char *t_stack;             /* pointer to the thread's stack */

    /* Public thread members - can be used by other code */

    struct addrspace *t_vmspace; /* address space structure */
    struct vnode *t_cwd;         /* current working directory */
};
```

# How Does it Start?

- `thread_bootstrap()`: this is where it starts, the first thread is created here and this is made as the current thread (Where is this function called?)
- `curthread` points to the thread that is currently running (look for `curthread.h`)
- Thread specific information is stored in `struct thread`

# Context Switch, Scheduling, and Dispatching

- What is context switch? Why do we need one?
- What is the context of a thread, that will help us to resume if we stopped now?
- The act of saving the context of the current thread and resuming the context of the next thread to run is called *dispatching* (the next thread)
- Sounds simple, but . . .
  - architecture-specific implementation
  - thread must save/restore its context carefully, since thread execution continuously changes the context
  - can be tricky to understand (at what point does a thread actually stop? What is it executing when it resumes?)
  - To answer the above questions in os161 look into `hardclock.c`

# Dispatching on the MIPS

```
/* see kern/arch/mips/mips/switch.S */
mips_switch:
    /* a0/a1 points to old/new thread's control block */

    /* Allocate stack space for saving 11 registers. 11*4 = 44 */
    addi sp, sp, -44

    /* Save the registers */
    sw ra, 40(sp)
    sw gp, 36(sp)
    sw s8, 32(sp)
    sw s7, 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 the old stack pointer in the old control block */
    sw sp, 0(a0)
```



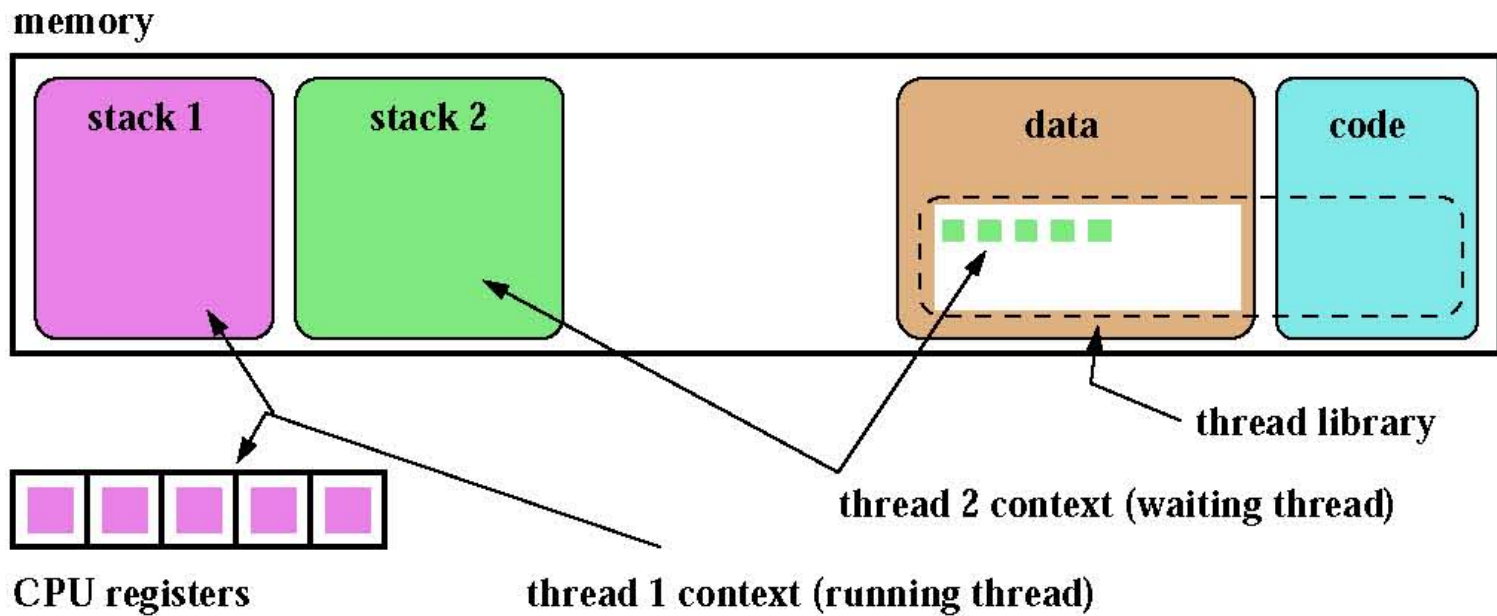
# Dispatching on the MIPS (cont.)

```
/* Get the new stack pointer from the new control block */
lw sp, 0(a1)
nop /* delay slot for load */

/* 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 s7, 28(sp)
lw s8, 32(sp)
lw gp, 36(sp)
lw ra, 40(sp)
nop /* delay slot for load */

j ra /* and return. */
addi sp, sp, 44 /* in delay slot */
.end mips_switch
```

# Thread Library and Two Threads



# The OS/161 Thread Interface (incomplete)

```
/* see kern/include/thread.h */
/* create a new thread */
int thread_fork(const char *name,
                void *data1, unsigned long data2,
                void (*func)(void *, unsigned long),
                struct thread **ret);

/* destroy the calling thread */
void thread_exit(void);

/* let another thread run */
void thread_yield(void);

/* block the calling thread */
void thread_sleep(const void *addr);

/* unblock blocked threads */
void thread_wakeup(const void *addr);
```

# Creating a New Thread

- Heard of fork? What does fork do?
- `thread_fork` is the like fork, but they are not quite the same
- It creates a new thread, allocates its stack space and inherits the directory from current thread. The new thread starts in the provided function pointer, and takes two arguments
- How is it different from fork?
- What if you want to pass more than 2 arguments?

# Creating Threads using thread\_fork()

```
/* from catmouse() in kern/asst1/catmouse.c */
/* start NumMice mouse_simulation() threads */
for (index = 0; index < NumMice; index++) {
    error = thread_fork("mouse_simulation thread", NULL, index,
                       mouse_simulation, NULL);

    if (error) {
        panic("mouse_simulation: thread_fork failed: %s\n",
             strerror(error));
    }
}

/* wait for all of the cats and mice to finish before
   terminating */
for(i=0; i < (NumCats+NumMice); i++) {
    P(CatMouseWait);
}
```

# Scheduling

- scheduling means deciding which thread should run next
- scheduling is implemented by a *scheduler*, which is part of the thread library
- simple FIFO scheduling:
  - scheduler maintains a queue of threads, often called the *ready queue*
  - the first thread in the ready queue is the running thread
  - on a context switch the running thread is moved to the end of the ready queue, and new first thread is allowed to run
  - newly created threads are placed at the end of the ready queue
- more on scheduling later . . .

# Concurrency

- On multiprocessors, several threads can execute simultaneously, one on each processor.
- On uniprocessors, only one thread executes at a time. However, because of preemption and timesharing, threads appear to run concurrently.

---

---

Concurrency and synchronization are important even on uniprocessors.

---

---

# What is Going on Here?





## Enforcing Mutual Exclusion

- mutual exclusion algorithms ensure that only one thread at a time executes the code in a critical section
- several techniques for enforcing mutual exclusion
  - exploit special hardware-specific machine instructions, e.g., *test-and-set* or *compare-and-swap*, that are intended for this purpose
  - use mutual exclusion algorithms, e.g., *Peterson's algorithm*, that rely only on atomic loads and stores
  - control interrupts to ensure that threads are not preempted while they are executing a critical section

## Disabling Interrupts

- On a uniprocessor, only one thread at a time is actually running.
- If the running thread is executing a critical section, mutual exclusion may be violated if
  1. the running thread is preempted (or voluntarily yields) while it is in the critical section, and
  2. the scheduler chooses a different thread to run, and this new thread enters the same critical section that the preempted thread was in
- Since preemption is caused by timer interrupts, mutual exclusion can be enforced by disabling timer interrupts before a thread enters the critical section, and re-enabling them when the thread leaves the critical section.

---

---

This is the way that the OS/161 kernel enforces mutual exclusion. There is a simple interface (`splhigh()`, `spl0()`, `splx()`) for disabling and enabling interrupts. See `kern/arch/mips/include/spl.h`.

---

---

## Pros and Cons of Disabling Interrupts

- advantages:
  - does not require any hardware-specific synchronization instructions
  - works for any number of concurrent threads
- disadvantages:
  - indiscriminate: prevents all preemption, not just preemption that would threaten the critical section
  - ignoring timer interrupts has side effects, e.g., kernel unaware of passage of time. (Worse, OS/161's `sp1high()` disables *all* interrupts, not just timer interrupts.) Keep critical sections *short* to minimize these problems.
  - will not enforce mutual exclusion on multiprocessors (why??)

# Other Ways

- Use synchronization primitives
  - Locks
  - Semaphores
  - Condition Variables
- Use hardware-based synchronization:
  - Test\_and\_set()
  - Compare\_and\_swap()

# Semaphores

- 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.
- A semaphore is an object that has an integer value, and that supports two operations:
  - P:** if the semaphore value is greater than 0, decrement the value. Otherwise, wait until the value is greater than 0 and then decrement it.
  - V:** increment the value of the semaphore
- Two kinds of semaphores:
  - counting semaphores:** can take on any non-negative value
  - binary semaphores:** take on only the values 0 and 1. (V on a binary semaphore with value 1 has no effect.)

---

---

By definition, the P and V operations of a semaphore are *atomic*.

---

---

Using SpinLocks or disabling interrupts to do this inside the kernel

# Why are We Looking at This?

- For Project #1 you are going to complete the code for the synchronization primitive locks
- Although they are different from semaphores looking at semaphores implementation can give some insight on how to design the locks
- Specifically you will get to know what precautions to take in making sure your locks meet the required conditions for synchronization primitives

## OS/161 Semaphores

```
struct semaphore {
    char *name;
    volatile int count;
};

struct semaphore *sem_create(const char *name,
    int initial_count);
void P(struct semaphore *);
void V(struct semaphore *);
void sem_destroy(struct semaphore *);
```

---

---

see

- kern/include/synch.h
  - kern/thread/synch.c
- 
-

## OS/161 Locks

- OS/161 also uses a synchronization primitive called a *lock*. Locks are intended to be used to enforce mutual exclusion.

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

```
lock_acquire(mylock);
```

```
    critical section /* e.g., call to list_remove_front */
```

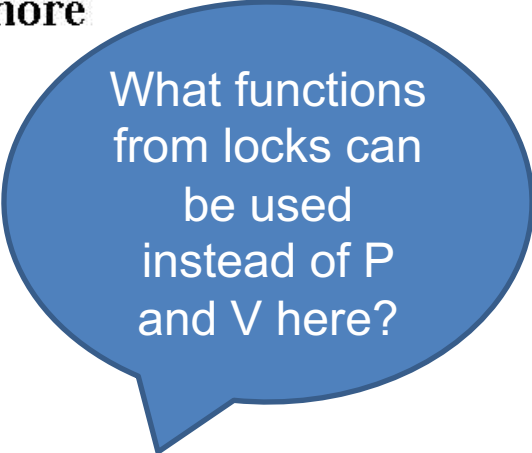
```
lock_release(mylock);
```

- A lock is similar to a binary semaphore with an initial value of 1. However, locks also enforce an additional constraint: the thread that releases a lock must be the same thread that most recently acquired it.
- The system enforces this additional constraint to help ensure that locks are used as intended.

Not fully implemented. This is what you need to code.



## Mutual Exclusion Using a Semaphore



What functions from locks can be used instead of P and V here?

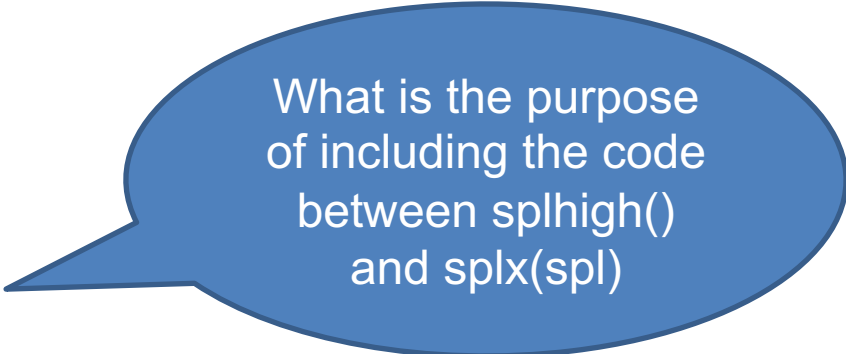
```
struct semaphore *s;  
s = sem_create("MySem1", 1); /* initial value is 1 */  
  
P(s); /* do this before entering critical section */  
  
    critical section /* e.g., call to list_remove_front */  
  
V(s); /* do this after leaving critical section */
```

## OS/161 Semaphores: P()

```
void
P(struct semaphore *sem)
{
    int spl;
    assert(sem != NULL);

    /*
     * May not block in an interrupt handler.
     * For robustness, always check, even if we can actually
     * complete the P without blocking.
     */
    assert(in_interrupt==0);

    spl = splhigh();
    while (sem->count==0) {
        thread_sleep(sem);
    }
    assert(sem->count>0);
    sem->count--;
    splx(spl);
}
```



What is the purpose of including the code between splhigh() and splx(spl)

## Thread Blocking

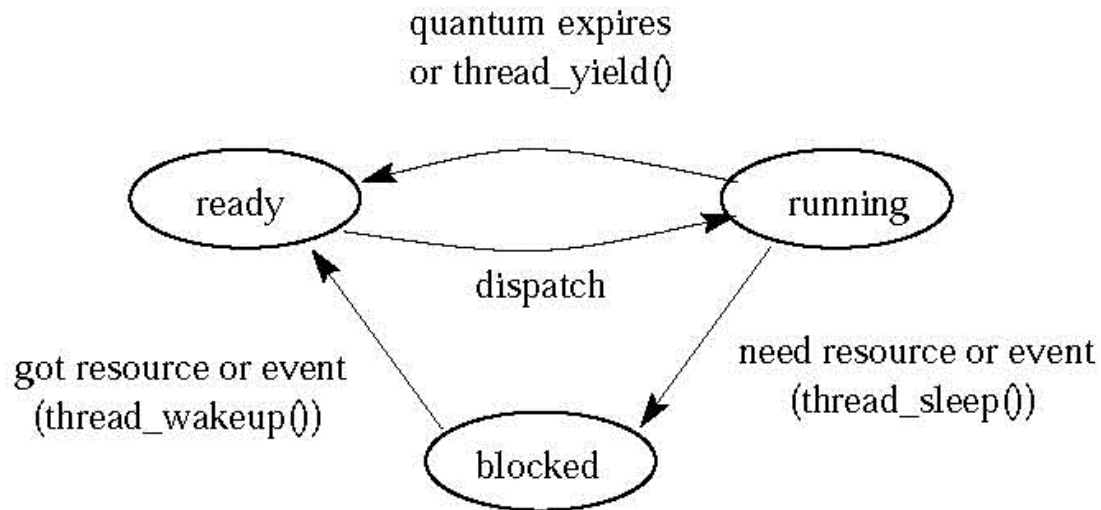
- Sometimes a thread will need to wait for an event. One example is on the previous slide: a thread that attempts a P() operation on a zero-valued semaphore must wait until the semaphore's value becomes positive.
- other examples
  - wait for data from a (relatively) slow device
  - wait for input from a keyboard
  - wait for busy device to become idle
- In these circumstances, we do not want the thread to run, since it cannot do anything useful.
- To handle this, the thread scheduler can *block* threads.

## Thread Blocking in OS/161

- OS/161 thread library functions:
  - `void thread_sleep(const void *addr)`
    - \* blocks the calling thread on address `addr`
  - `void thread_wakeup(const void *addr)`
    - \* unblock threads that are sleeping on address `addr`
- `thread_sleep()` is much like `thread_yield()`. The calling thread voluntarily gives up the CPU, the scheduler chooses a new thread to run, and dispatches the new thread. However
  - after a `thread_yield()`, the calling thread is *ready* to run again as soon as it is chosen by the scheduler
  - after a `thread_sleep()`, the calling thread is blocked, and should not be scheduled to run again until after it has been explicitly unblocked by a call to `thread_wakeup()`.

# Thread States

- a very simple thread state transition diagram



- the states:

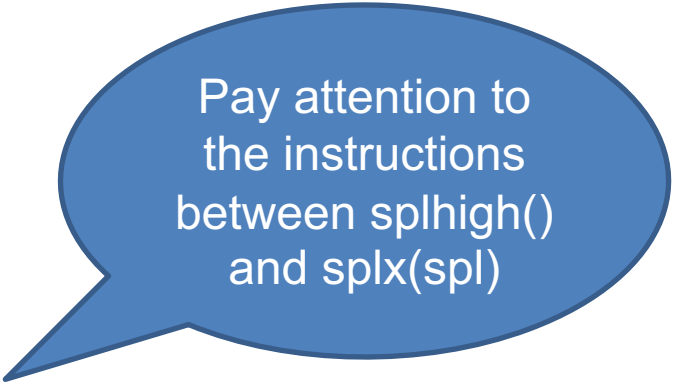
**running:** currently executing

**ready:** ready to execute

**blocked:** waiting for something, so not ready to execute.

## OS/161 Semaphores: V() kern/thread/synch.c

```
void
V(struct semaphore *sem)
{
    int spl;
    assert(sem != NULL);
    spl = splhigh();
    sem->count++;
    assert(sem->count > 0);
    thread_wakeup(sem);
    splx(spl);
}
```



Pay attention to  
the instructions  
between splhigh()  
and splx(spl)

## Condition Variables

- OS/161 supports another common synchronization primitive: *condition variables*
- each condition variable is intended to work together with a lock: condition variables are only used *from within the critical section that is protected by the lock*
- three operations are possible on a condition variable:
  - wait:** this causes the calling thread to block, and it releases the lock associated with the condition variable
  - signal:** if threads are blocked on the signaled condition variable, then one of those threads is unblocked
  - broadcast:** like signal, but unblocks all threads that are blocked on the condition variable

## Using Condition Variables

- Condition variables get their name because they allow threads to wait for arbitrary conditions to become true inside of a critical section.
- Normally, each condition variable corresponds to a particular condition that is of interest to an application. For example, in the bounded buffer producer/consumer example on the following slides, the two conditions are:
  - $count > 0$  (condition variable not empty)
  - $count < N$  (condition variable not full)
- when a condition is not true, a thread can wait on the corresponding condition variable until it becomes true
- when a thread detects that a condition is true, it uses `signal` or `broadcast` to notify any threads that may be waiting

---

---

Note that signalling (or broadcasting to) a condition variable that has no waiters has *no effect*. Signals do not accumulate.

---

---



## Waiting on Condition Variables

- when a blocked thread is unblocked (by `signal` or `broadcast`), it reacquires the lock before returning from the `wait` call
- a thread is in the critical section when it calls `wait`, and it will be in the critical section when `wait` returns. However, in between the call and the return, while the caller is blocked, the caller is out of the critical section, and other threads may enter.
- In particular, the thread that calls `signal` (or `broadcast`) to wake up the waiting thread will itself be in the critical section when it signals. The waiting thread will have to wait (at least) until the signaller releases the lock before it can unblock and return from the `wait` call.

---

---

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

---

---

## Bounded Buffer Producer Using Condition Variables

```
int count = 0; /* must initially be 0 */
struct lock *mutex; /* for mutual exclusion */
struct cv *notfull, *notempty; /* condition variables */

/* Initialization Note: the lock and cv's must be created
 * using lock_create() and cv_create() before Produce()
 * and Consume() are called */

Produce(item) {
    lock_acquire(mutex);
    while (count == N) {
        cv_wait(notfull, mutex);
    }
    add item to buffer (call list_append())
    count = count + 1;
    cv_signal(notempty, mutex);
    lock_release(mutex);
}
```

## Bounded Buffer Consumer Using Condition Variables

```
Consume () {  
    lock_acquire(mutex);  
    while (count == 0) {  
        cv_wait(notempty, mutex);  
    }  
    remove item from buffer (call list_remove_front())  
    count = count - 1;  
    cv_signal(notfull, mutex);  
    lock_release(mutex);  
}
```

---

---

Both Produce() and Consume() call cv\_wait() inside of a while loop. Why?

---

---

# Monitors

- Condition variables are derived from *monitors*. A monitor is a programming language construct that provides synchronized access to shared data. Monitors have appeared in many languages, e.g., Ada, Mesa, Java
- a monitor is essentially an object with special concurrency semantics
- it is an object, meaning
  - it has data elements
  - the data elements are encapsulated by a set of methods, which are the only functions that directly access the object's data elements
- only *one* monitor method may be active at a time, i.e., the monitor methods (together) form a critical section
  - if two threads attempt to execute methods at the same time, one will be blocked until the other finishes
- inside a monitor, so called *condition variables* can be declared and used

## Monitors in OS/161

- The C language, in which OS/161 is written, does not support monitors.
- However, programming convention and OS/161 locks and condition variables can be used to provide monitor-like behavior for shared kernel data structures:
  - define a C structure to implement the object's data elements
  - define a set of C functions to manipulate that structure (these are the object "methods")
  - ensure that only those functions directly manipulate the structure
  - create an OS/161 lock to enforce mutual exclusion
  - ensure that each access method acquires the lock when it starts and releases the lock when it finishes
  - if desired, define one or more condition variables and use them within the methods.

# Testing your work

- Use the existing tests in os161 to test your work
- ?t – will list the available tests in os161
- Once you are done with your locks, use sy2 to test them. You will get to know if you are locks are working or not.
- Don't start with synchronization problem before your locks are working correctly.