Outline

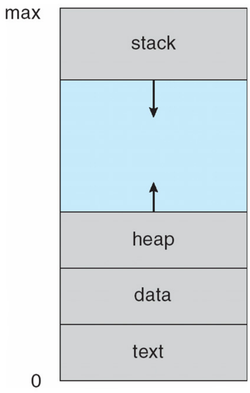
1. Processes and Threads
2. Synchronization
3. Memory Management

Processes

- **A *process* is an instance of a program running**
- **Modern OSes run multiple processes simultaneously**
- **Very early OSes only ran one process at a time**
- **Examples (can all run simultaneously):**
  - `emacs` – text editor
  - `firefox` – web browser
- **Non-examples (implemented as one process):**
  - Multiple firefox windows or emacs frames (still one process)
- **Why processes?**
  - Simplicity of programming
  - Speed: Higher throughput, lower latency

A process's view of the world

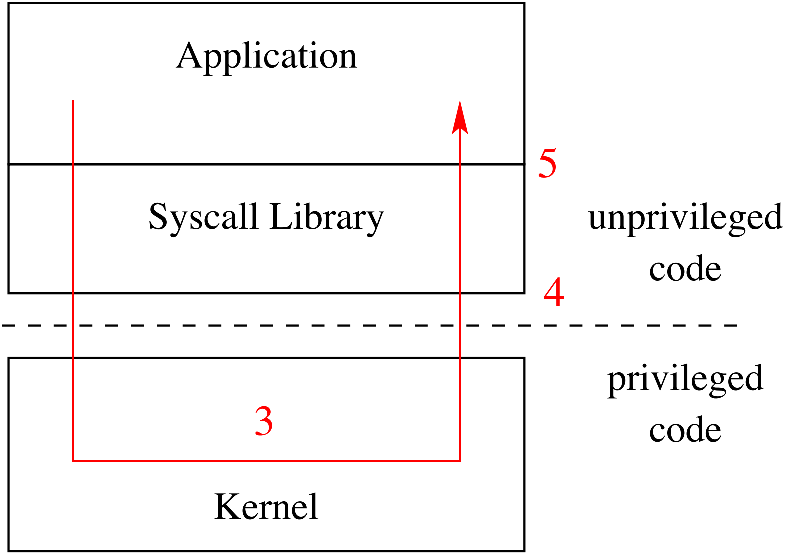
- **Each process has own view of machine**
  - Its own address space
  - Its own open files
  - Its own virtual CPU (through preemptive multitasking)
- `*(char *)0xc000` **different in P_1 & P_2**



System Calls

- **Systems calls are the interface between processes and the kernel**
- **A process invokes a system call to request operating system services**
- `fork()`, `waitpid()`, `open()`, `close()`
- **Note: Signals are another common mechanism to allow the kernel to notify the application of an important event (e.g., Ctrl-C)**
  - Signals are like interrupts/exceptions for application code

System Call Software Stack



Kernel Privilege

- **Hardware provides two or more privilege levels (or protection rings)**
- **Kernel code runs at a higher privilege level than applications**
- **Typically called *Kernel Mode* vs. *User Mode***
- **Code running in kernel mode gains access to certain CPU features**
  - Accessing restricted features (e.g. Co-processor 0)
  - Disabling interrupts, setup interrupt handlers
  - Modifying the TLB (for virtual memory management)
- **Allows the kernel to isolate processes from one another and from the kernel**
  - Processes cannot read/write kernel memory
  - Processes cannot directly call kernel functions

How System Calls Work

- **The kernel only runs through well defined entry points**
- **Interrupts**
  - Interrupts are generated by devices to signal needing attention
  - E.g. Keyboard input is ready
- **Exceptions**
  - Exceptions are caused by the processor executing code
  - E.g. Divide by zero, page fault, etc.

Interrupts

- **An interrupt or exception causes the hardware to transfer control to a fixed location in memory, where the *interrupt handler* is located**
- **Interrupt handlers are part of the kernel**
- **When an interrupt occurs, the processor switches to kernel mode (or privileged mode) allowing the kernel to take over**
  - This is how the kernel gets run with privileges
  - Interrupts can still be delivered while running the kernel
  - Exception is that spinlocks disabled interrupts

Exceptions

- **Exceptions are conditions that occur during the execution of a program (or kernel) that require attention**
  - E.g. divide by zero, page faults, illegal instructions, etc.
- **Exceptions are detected by the CPU during execution**
- **CPU handles exceptions just like interrupts by transferring control to the kernel**
  - Control is transferred to a fixed location where the exception handler is located
  - Processor is switches into privileged mode

MIPS Exception Vectors

```
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 */
```

- Interrupts, exceptions, and system calls are handled through the same mechanism
- Some processors specially handle system calls for performance reasons

How System Calls Work Continued

- **System calls are performed by triggering an exception**
- **Applications execute the `syscall` instruction to trigger the `EX_SYS` exception**
  - Many processors include a similar instruction
  - For example, x86 contains the `syscall` and/or `sysenter` instructions, but with an optimized implementation

Hardware Handling

- **Exception handlers in the R3000 are at fixed locations**
- **The processor jumps to these addresses whenever an exception is encountered**
  - `0x8000_0000` User TLB Handler
  - `0x8000_0080` General Exception Handler
- **Remember that in MIPS `0x8000_0000-0x9FFF_FFFF` is mapped to the first 512 MBs of physical memory.**

System Call Operations

- **Application calls into C library (e.g. calls `write()`)**
- **Library executes the `syscall` instruction**
- **Kernel exception handler `0x8000_0080` runs**
  - Switch to kernel stack
  - Create a trap frame to save program state
  - Determine the type of system call
  - Determine which system call is being invoked
  - Process call
  - Restore application state from trap frame
  - Return from exception
- **Library wrapper function returns to application**

Application Binary Interface/Calling Conventions

- **Each architecture and OS define calling conventions**
- **Describes how registers are used in function calls and system calls**
- **MIPS+OS/161 Calling Conventions**
  - System call number in `v0`
  - First four arguments in `a0`, `a1`, `a2`, `a3`
  - Remaining arguments passed on stack
  - Result success/fail in `a3` and return value/error code in `v0`
- **Number for each system call in 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
...
```

Creating processes

- `int fork (void);`
  - Create new process that is exact copy of current one
  - Returns *process ID* of new process in “parent”
  - Returns 0 in “child”
- `int waitpid (int pid, int *stat, int opt);`
  - `pid` – process to wait for, or -1 for any
  - `stat` – will contain exit value, or signal
  - `opt` – usually 0 or `WNOHANG`
  - Returns process ID or -1 on error

Deleting processes

- `void exit (int status);`
  - Current process ceases to exist
  - `status` shows up in `waitpid` (shifted)
  - By convention, `status` of 0 is success, non-zero error
- `int kill (int pid, int sig);`
  - Sends signal `sig` to process `pid`
  - `SIGTERM` most common value, kills process by default (but application can catch it for “cleanup”)
  - `SIGKILL` stronger, kills process always
- `pid_t getpid(void);`
  - Get the current process ID
- `pid_t getppid(void);`
  - Get the process ID of the parent process

Running programs

- `int execve (char *prog, char **argv, char **envp);`
  - `prog` – full pathname of program to run
  - `argv` – argument vector that gets passed to `main`
  - `envp` – environment variables, e.g., `PATH`, `HOME`
- **Generally called through a wrapper functions**
  - `int execvp (char *prog, char **argv);`
    Search `PATH` for prog, use current environment
  - `int execlp (char *prog, char *arg, ...);`
    List arguments one at a time, finish with `NULL`

Error returns

- **What if `open` fails? Returns -1 (invalid fd)**
- **Most system calls return -1 on failure**
  - Specific kind of error in global int `errno`
- `#include <sys/errno.h>` **for possible values**
  - 2 = `ENOENT` "No such file or directory"
  - 13 = `EACCES` "Permission Denied"
- `perror` **function prints human-readable message**
  - `perror ("initfile");`
    → "`initfile: No such file or directory`"
- **Details:**
  - Typically `errno` is a thread local variable
  - FreeBSD: C macro that calls `__errno()` to return the result

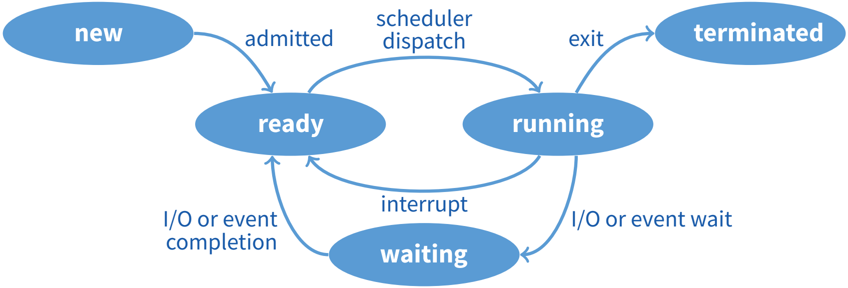
Implementing processes

- **Keep a data structure for each process**
  - Process Control Block (PCB)
  - Called `proc` in Unix, `task_struct` in Linux
- **Tracks *state* of the process**
  - Running, ready (runnable), waiting, etc.
- **Includes information necessary to run**
  - Registers, virtual memory mappings, etc.
  - Open files (including memory mapped files)
- **Various other data about the process**
  - Credentials (user/group ID), signal mask, controlling terminal, priority, accounting statistics, whether being debugged, which system call binary emulation in use, …

Process state
Process ID
User id, etc.
Program counter
Registers
Address space (VM data structs)
Open files

PCB

Process states

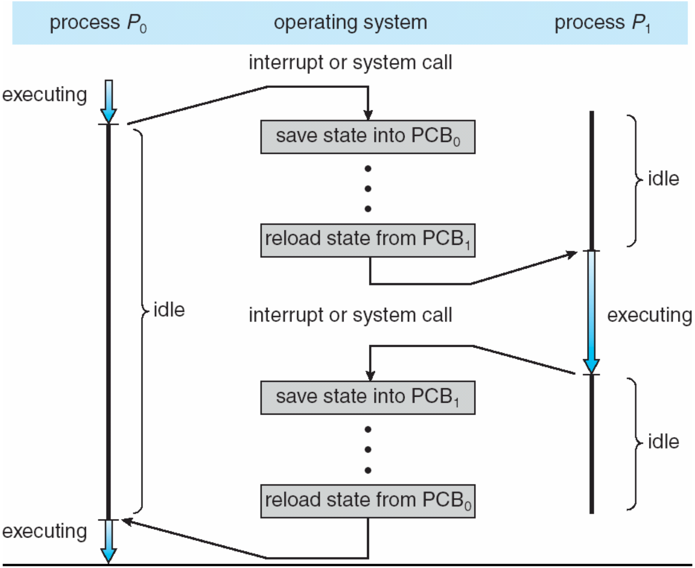


- **Process can be in one of several states**
  - *new* & *terminated* at beginning & end of life
  - *running* – currently executing (or will execute on kernel return)
  - *ready* – can run, but kernel has chosen different process to run
  - *waiting* – needs async event (e.g., disk operation) to proceed
- **Which process should kernel run?**
  - if 0 runnable, run idle loop (or halt CPU), if 1 runnable, run it
  - if >1 runnable, must make scheduling decision

Preemption

- **Can preempt a process when kernel gets control**
- **Running process can vector control to kernel**
  - System call, page fault, illegal instruction, etc.
  - May put current process to sleep—e.g., read from disk
  - May make other process runnable—e.g., fork, write to pipe
- **Periodic timer interrupt**
  - If running process used up quantum, schedule another
- **Device interrupt**
  - Disk request completed, or packet arrived on network
  - Previously waiting process becomes runnable
  - Schedule if higher priority than current running proc.
- **Changing running process is called a *context switch***

Context switch



Context switch details

- **Very machine dependent. Typical things include:**
  - Save program counter and integer registers (always)
  - Save floating point or other special registers
  - Save condition codes
  - Change virtual address translations
- **Non-negligible cost**
  - Save/restore floating point registers expensive
    - Optimization: only save if process used floating point
  - May require flushing TLB (memory translation hardware)
    - HW Optimization 1: don't flush kernel's own data from TLB
    - HW Optimization 2: use tag to avoid flushing any data
  - Usually causes more cache misses (switch working sets)

Outline

1. Processes and Threads
2. Synchronization
3. Memory Management

Critical Sections

```
int total = 0;

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

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

Critical Sections: Assembly Pseudocode

```
int total = 0;

void add() {
    int i;
    /* r8 := &total */
    for (i=0; i<N; i++) {
        lw r9, 0(r8)
        add r9, 1
        sw r9, 0(r8)
    }
}

void sub() {
    int i;
    for (i=0; i<N; i++) {
        lw r9, 0(r8)
        sub r9, 1
        sw r9, 0(r8)
    }
}
```

Memory Model

- **_Sequential Consistency_: statements execute in program order**
- **Compilers/HW reorder loads/stores for performance**
- **Language-level Memory Model**
  - C/Java: sequential consistency for race free programs
  - Compiler must be aware of synchronization
  - Language provides barriers and atomics
- **Processor-level Memory Model**
  - TSO: Total Store Order - X86, SPARC (default)
  - PSO: Partial Store Order - SPARC PSO
  - RMO: Relaxed Memory Order - Alpha, POWER, ARM, PA-RISC, SPARC RMO, x86 OOS
  - Even more nuanced variations between architectures!

Mutexes

- **Thread packages typically provide *mutexes*:**

```
void mutex_init (mutex_t *m, ...);
void mutex_lock (mutex_t *m);
int mutex_trylock (mutex_t *m);
void mutex_unlock (mutex_t *m);
```

  - Only one thread acuires `m` at a time, others wait

Simple Spinlock in C11

```
typedef struct Spinlock {
    alignas(CACHELINE) _Atomic(uint64_t) lck;
} Mutex;

void Spinlock_Init(Spinlock *m) {
    atomic_store(&m->lck, 0);
}

void Spinlock_Lock(Spinlock *m) {
    while (atomic_exchange(&m->lck, 1) == 1)
        ;
}

void Spinlock_Unlock(Spinlock *m) {
    atomic_store(&m->lck, 0);
}
```

Atomics in C11

Where's the barriers?

```
// Implicit Sequential Consistency
C atomic_load(const volatile A* obj);
// Explicit Consistency
C atomic_load_explicit(const volatile A* obj,
                       memory_order order);
// Barrier or Fence
void atomic_thread_fence(memory_order order);

enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
```

Pre-C11 Compilers (including OS/161)

- **Use assembly routines for compiler barriers:**
  - `asm("" :::  "memory");`
  - Compiler will not reorder loads/stores nor cache values
- **Use `volatile` keyword**
  - `volatile` originally meant for accessing device memory
  - loads/stores to `volatile` variables will not be reordered with respect to other `volatile` operations
  - Use of `volatile` is deprecated on modern compilers
  - `volatile` operations are not atomics!
  - Use `volatile` with inline assembly to use atomics

Spinlocks in OS/161

```
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);
```

- **Spinlocks based on using** `spinlock_data_testandset`
- **Spinlocks don’t yield CPU, i.e., they spin**
- **Raise the interupt level to prevent preemption**

MIPS Atomics

- **Load Linked `ll`: Loads a value and monitors memory for changes**
- **Store Conditional `sc`: Stores if memory didn't change**
- **`sc` can fail for multiple reasons**
  - Value from `ll` was modified by another processor
  - An interrupt preempted the thread between `ll` and `sc`
- **Otherwise `sc` will succeed returning 1**
- **On failure we can retry the operation**
- **Powerful primitives**
  - Can implement any read-modify-write operation
  - For example, atomic add or increment
  - Some architectures are implemented this way internally

Mutex Locks

- **Provide mutual exclusion like spinlocks**
- **Yield the CPU when waiting on the lock**
- **Mutex locks deal with priority inversion**
  - Problem: Low priority thread sleeps while holding lock then a high priority thread wants the lock
  - Mutex locks typically boost the priority of the lower thread to unblock the higher thread

Wait Channels in OS/161

- **Wait channels are used to implement thread blocking in OS/161**
- **Many different wait channels holding threads sleeping for different reasons**
- **Similar primitives exist in most operating systems**
- `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);`
  - Unblocks all threads sleeping on the wait channel
- `void wchan_wakeone(struct wchan *wc);`
  - Unblocks one threads sleeping on the wait channel
- `void wchan_lock(struct wchan *wc);`
  - Prevent operations on the wait channel
  - More on this later

Producer

```
mutex_t mutex = MUTEX_INITIALIZER;

void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();

        mutex_lock (&mutex);
        while (count == BUFFER_SIZE) {
          mutex_unlock (&mutex); /* <--- Why? */
          thread_yield ();
          mutex_lock (&mutex);
        }

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        mutex_unlock (&mutex);
    }
}
```

Consumer

```
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        while (count == 0) {
          mutex_unlock (&mutex);
          thread_yield ();
          mutex_lock (&mutex);
        }

        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        mutex_unlock (&mutex);

        consume_item (nextConsumed);
    }
}
```

Condition variables

- **Busy-waiting in application is a bad idea**
  - Consumes CPU even when a thread can't make progress
  - Unnecessarily slows other threads/processes or wastes power
- **Better to inform scheduler of which threads can run**
- **Typically done with *condition variables***
- `struct cond_t;` (`pthread_cond_t` **or** `cv` **in OS/161)**
- `void cond_init (cond_t *, ...);`
- `void cond_wait (cond_t *c, mutex_t *m);`
  - Atomically unlock `m` and sleep until `c` signaled
  - Then re-acquire `m` and resume executing
- `void cond_signal (cond_t *c);`
  `void cond_broadcast (cond_t *c);`
  - Wake one/all threads waiting on `c`

Improved producer

```
mutex_t mutex = MUTEX_INITIALIZER;
cond_t nonempty = COND_INITIALIZER;
cond_t nonfull = COND_INITIALIZER;

void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();

        mutex_lock (&mutex);
        while (count == BUFFER_SIZE)
          cond_wait (&nonfull, &mutex);

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        cond_signal (&nonempty);
        mutex_unlock (&mutex);
    }
}
```

Improved consumer

```
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        while (count == 0)
          cond_wait (&nonempty, &mutex);

        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        cond_signal (&nonfull);
        mutex_unlock (&mutex);

        consume_item (nextConsumed);
    }
}
```

Semaphores [Dijkstra]

- **A *Semaphore* is initialized with an integer N**
  - `sem_create(N)`
- **Provides two functions:**
  - `sem_wait (S)` (originally called P)
  - `sem_signal (S)` (originally called V)
- **Guarantees `sem_wait` will return only N more times than `sem_signal` called**
  - Example: If $N == 1$, then semaphore acts as a mutex with `sem_wait` as lock and `sem_signal` as unlock

Using a Semaphore as a Mutex

- **We can use a semaphore as a mutex**

```
semaphore *s = sem_create(1);

/* Acquire the lock */
sem_wait(s); /* Semaphore count is now 0 */
/* critical section */
/* Release the lock */
sem_signal(s); /* Seamphore count is now 1 */
```

Using a Semaphore as a Mutex

- **We can use a semaphore as a mutex**

```
semaphore *s = sem_create(1);

/* Acquire the lock */
sem_wait(s); /* Semaphore count is now 0 */
/* critical section */
/* Release the lock */
sem_signal(s); /* Seamphore count is now 1 */
```

- **Couple important differences:**
  - Mutex requires the same thread to acquire/relase the lock
  - Allows mutexes to implement priority inversion

Semaphore producer/consumer

- **Initialize `full` to 0 (block consumer when buffer empty)**
- **Initialize `empty` to *N* (block producer when queue full)**

```
void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();
        sem_wait (&empty);
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        sem_signal (&full);
    }
}
void consumer (void *ignored) {
    for (;;) {
        sem_wait (&full);
        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        sem_signal (&empty);
        consume_item (nextConsumed);
    }
}
```

Implementation of P and V

- **See os161/kern/thread/synch.c**

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

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

Outline