CS 350
Operating Systems
Course Notes

Winter 2012

David R. Cheriton
School of Computer Science
University of Waterloo

What is an Operating System?

- Three views of an operating system

  Application View: what services does it provide?

  System View: what problems does it solve?

  Implementation View: how is it built?

An operating system is part cop, part facilitator.

Application View of an Operating System

- The OS provides an execution environment for running programs.
  - The execution environment provides a program with the processor time and memory space that it needs to run.
  - The execution environment provides interfaces through which a program can use networks, storage, I/O devices, and other system hardware components.
    - Interfaces provide a simplified, abstract view of hardware to application programs.
  - The execution environment isolates running programs from one another and prevents undesirable interactions among them.

Other Views of an Operating System

System View: The OS manages the hardware resources of a computer system.

- Resources include processors, memory, disks and other storage devices, network interfaces, I/O devices such as keyboards, mice and monitors, and so on.
- The operating system allocates resources among running programs. It controls the sharing of resources among programs.
- The OS itself also uses resources, which it must share with application programs.

Implementation View: The OS is a concurrent, real-time program.

- Concurrency arises naturally in an OS when it supports concurrent applications, and because it must interact directly with the hardware.
- Hardware interactions also impose timing constraints.

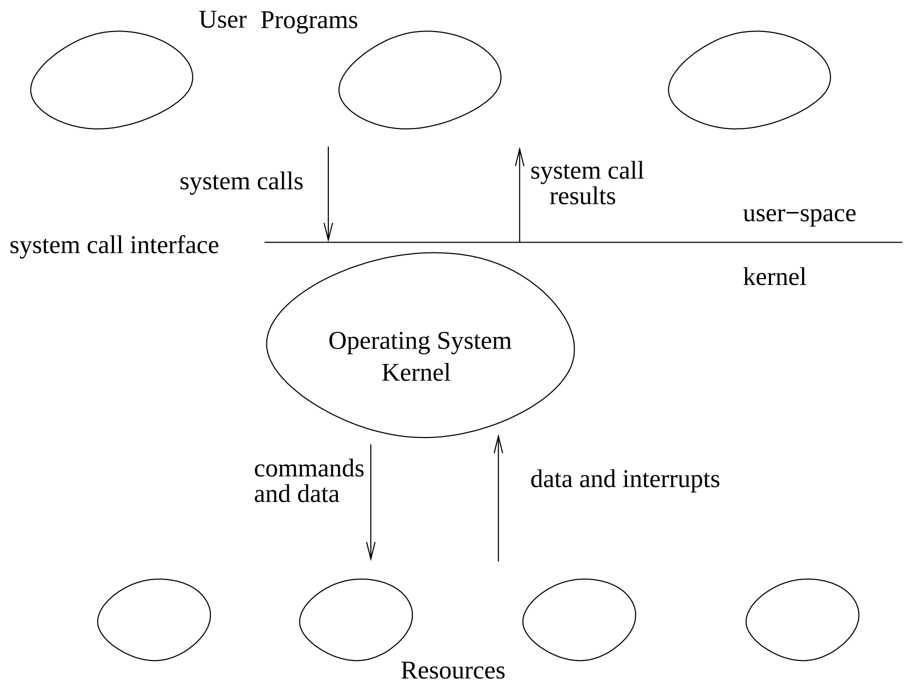
The Operating System and the Kernel

- Some terminology:

  kernel: The operating system kernel is the part of the operating system that responds to system calls, interrupts and exceptions.

  operating system: The operating system as a whole includes the kernel, and may include other related programs that provide services for applications. This may include things like:
  - utility programs
  - command interpreters
  - programming libraries

Schematic View of an Operating System



Operating System Abstractions

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

  files and file systems: abstract view of secondary storage

  address spaces: abstract view of primary memory

  processes, threads: abstract view of program execution

  sockets, pipes: abstract view of network or other message channels

- This course will cover
  - why these abstractions are designed the way they are
  - how these abstractions are manipulated by application programs
  - how these abstractions are implemented by the OS

Course Outline

- Introduction
- Threads and Concurrency
- Synchronization
- Processes and the Kernel
- Virtual Memory
- Scheduling
- Devices and Device Management
- File Systems
- Interprocess Communication and Networking (time permitting)

Review: Program Execution

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

Review: MIPS Register Usage

```
See also: kern/arch/mips/include/asmdefs.h
R0,  zero = ## zero (always returns 0)
R1,  at   = ## reserved for use by assembler
R2,  v0   = ## return value / system call number
R3,  v1   = ## return value
R4,  a0   = ## 1st argument (to subroutine)
R5,  a1   = ## 2nd argument
R6,  a2   = ## 3rd argument
R7,  a3   = ## 4th argument
```

Review: MIPS Register Usage

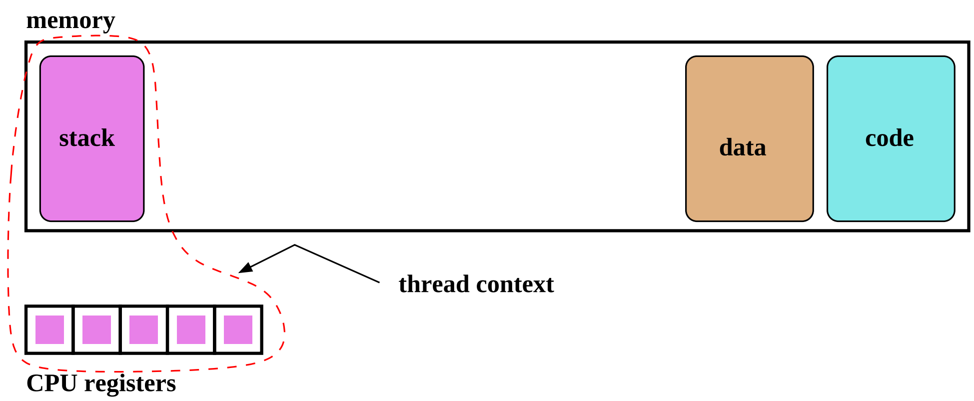
```
R08-R15,  t0-t7 = ## temps (not preserved by subroutines)
R24-R25,  t8-t9 = ## temps (not preserved by subroutines)
                  ##   can be used without saving
R16-R23,  s0-s7 = ## preserved by subroutines
                  ##   save before using,
                  ##   restore before return
R26-27,   k0-k1 = ## reserved for interrupt handler
R28,      gp    = ## global pointer
                  ## (for easy access to some variables)
R29,      sp    = ## stack pointer
R30,      s8/fp = ## 9th subroutine reg / frame pointer
R31,      ra    = ## return addr (used by jal)
```


What is a Thread?

- A thread represents the control state of an executing program.
- A thread has an associated *context* (or state), which consists of
  - the processor's CPU state, including the values of the program counter (PC), the stack pointer, other registers, and the execution mode (privileged/non-privileged)
  - a stack, which is located in the address space of the thread's process

Imagine that you would like to suspend the program execution, and resume it again later. Think of the thread context as the information you would need in order to restart program execution from where it left off when it was suspended.

Thread Context



Concurrent Threads

- more than one thread may exist simultaneously (why might this be a good idea?)
- each thread has its own context, though they may share access to program code and data
- on a uniprocessor (one CPU), at most one thread is actually executing at any time. The others are paused, waiting to resume execution.
- on a multiprocessor, multiple threads may execute at the same time, but if there are more threads than processors then some threads will be paused and waiting

Example: Concurrent Mouse Simulations

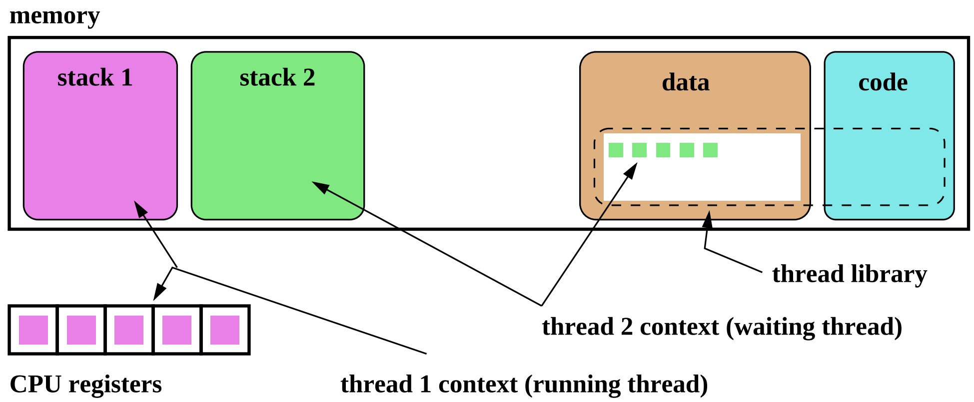
```
static void mouse_simulation(void * unusedpointer,
                             unsigned long mousenumber)
{
  int i; unsigned int bowl;

  for(i=0;i<NumLoops;i++) {
    /* for now, this mouse chooses a random bowl from
     * which to eat, and it is not synchronized with
     * other cats and mice
     */
    /* legal bowl numbers range from 1 to NumBowls */
    bowl = ((unsigned int)random() % NumBowls) + 1;
    mouse_eat(bowl,1);
  }

  /* indicate that this mouse is finished */
  V(CatMouseWait);
}
```


Implementing Threads

- a thread library is responsibile 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 a thread context is sometimes called a *thread control block*

In the OS/161 kernel's thread implementation, thread contexts are stored in `thread` structures.

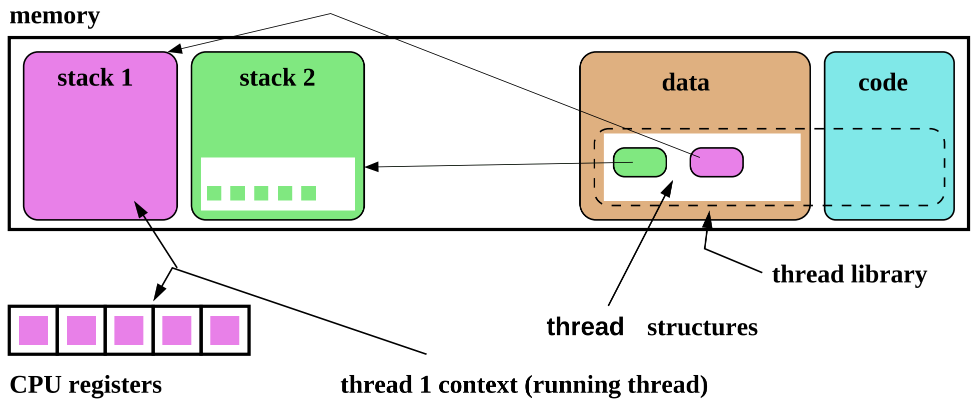
Thread Library and Two Threads



The OS/161 `thread` Structure

```
/* see kern/include/thread.h */

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 */
};
```

Thread Library and Two Threads (OS/161)



Context Switch, Scheduling, and Dispatching

- the act of pausing the execution of one thread and resuming the execution of another is called a *(thread) context switch*
- what happens during a context switch?
  1. save the context of the currently running thread
  2. decide which thread will run next
  3. restore the context of the thread that is to run next
- the act of saving the context of the current thread and installing 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?)

Dispatching on the MIPS (1 of 2)

```
/* 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 (2 of 2)

```
   /* 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 Interface

- the thread library interface allows program code to manipulate threads
- one key thread library interface function is *Yield()*
- Yield() causes the calling thread to stop and wait, and causes the thread library to choose some other waiting thread to run in its place. In other words, Yield() causes a context switch.
- in addition to `Yield()`, thread libraries typically provide other thread-related services:
  - create new thread
  - end (and destroy) a thread
  - cause a thread to *block* (to be discussed later)

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 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 . . .

Preemption

- `Yield()` allows programs to *voluntarily* pause their execution to allow another thread to run
- sometimes it is desirable to make a thread stop running even if it has not called `Yield()`
- this kind of *involuntary* context switch is called *preemption* of the running thread
- to implement preemption, the thread library must have a means of "getting control" (causing thread library code to be executed) even though the application has not called a thread library function
- this is normally accomplished using *interrupts*

Review: Interrupts

- an interrupt is an event that occurs during the execution of a program
- interrupts are caused by system devices (hardware), e.g., a timer, a disk controller, a network interface
- when an interrupt occurs, the hardware automatically transfers control to a fixed location in memory
- at that memory location, the thread library places a procedure called an *interrupt handler*
- the interrupt handler normally:
  1. saves the current thread context (in OS/161, this is saved in a *trap frame* on the current thread's stack)
  2. determines which device caused the interrupt and performs device-specific processing
  3. restores the saved thread context and resumes execution in that context where it left off at the time of the interrupt.

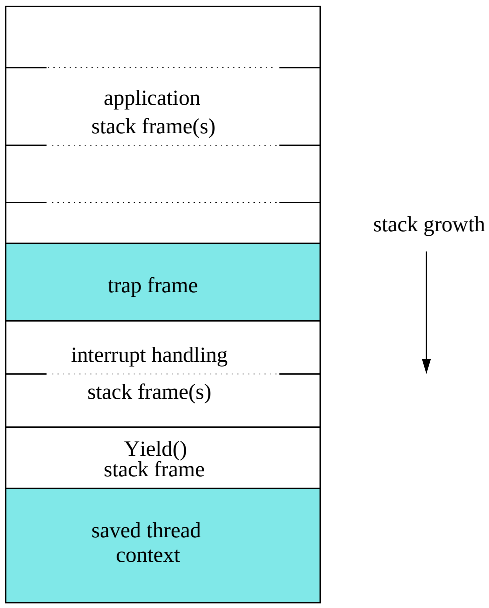
Round-Robin Scheduling

- *round-robin* is one example of a preemptive scheduling policy
- round-robin scheduling is similar to FIFO scheduling, except that it is preemptive
- as in FIFO scheduling, there is a ready queue and the thread at the front of the ready queue runs
- unlike FIFO, a limit is placed on the amount of time that a thread can run before it is preempted
- the amount of time that a thread is allocated is called the scheduling *quantum*
- when the running thread's quantum expires, it is preempted and moved to the back of the ready queue. The thread at the front of the ready queue is dispatched and allowed to run.

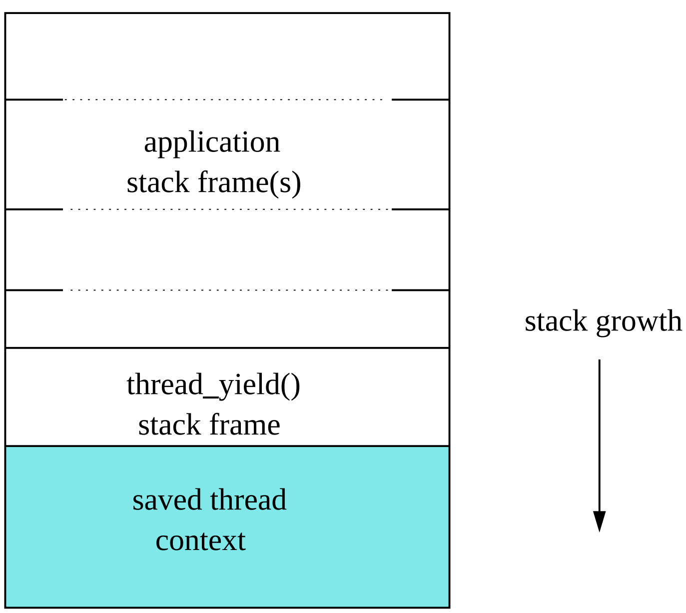
Implementing Preemptive Scheduling

- suppose that the system timer generates an interrupt every t time units, e.g., once every millisecond
- suppose that the thread library wants to use a scheduling quantum $q = 500t$, i.e., it will preempt a thread after half a second of execution
- to implement this, the thread library can maintain a variable called `running_time` to track how long the current thread has been running:
  - when a thread is intially dispatched, `running_time` is set to zero
  - when an interrupt occurs, the timer-specific part of the interrupt handler can increment `running_time` and then test its value
    - if `running_time` is less than q, the interrupt handler simply returns and the running thread resumes its execution
    - if `running_time` is equal to q, then the interrupt handler invokes `Yield()` to cause a context switch

OS/161 Stack after Preemption



OS/161 Stack after Voluntary Context Switch (`thread_yield()`)



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.

Thread Synchronization

- Concurrent threads can interact with each other in a variety of ways:
  - Threads share access, through the operating system, to system devices (more on this later . . .)
  - Threads may share access to program data, e.g., global variables.
- A common synchronization problem is to enforce *mutual exclusion*, which means making sure that only one thread at a time uses a shared object, e.g., a variable or a device.
- The part of a program in which the shared object is accessed is called a *critical section*.

Critical Section Example (Part 1)

```
int list_remove_front(list *lp) {
   int num;
   list_element *element;
   assert(!is_empty(lp));
   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;
}
```

The `list_remove_front` function is a critical section. It may not work properly if two threads call it at the same time on the same `list`. (Why?)

Critical Section Example (Part 2)

```
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++;
}
```

The `list_append` function is part of the same critical section as `list_remove_front`. It may not work properly if two threads call it at the same time, or if a thread calls it while another has called `list_remove_front`

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 `splhigh()` disables *all* interrupts, not just timer interrupts.) Keep critical sections *short* to minimize these problems.
  - will not enforce mutual exclusion on multiprocessors (why??)

Peterson's Mutual Exclusion Algorithm

```
/* shared variables */
/* note: one flag array and turn variable */
/* for each critical section */
boolean flag[2];  /* shared, initially false */
int turn;         /* shared */

flag[i] = true;   /* for one thread, i = 0 and j = 1 */
turn = j;         /* for the other, i = 1 and j = 0 */
while (flag[j] && turn == j) { }  /* busy wait */

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

flag[i] = false;
```

Ensures mutual exclusion and avoids starvation, but works only for two threads. (Why?)

Hardware-Specific Synchronization Instructions

- a test-and-set instruction *atomically* sets the value of a specified memory location and either
  - places that memory location's *old* value into a register, or
  - checks a condition against the memory location's old value and records the result of the check in a register
- for presentation purposes, we will abstract such an instruction as a function `TestAndSet(address,value)`, which takes a memory location (`address`) and a value as parameters. It atomically stores `value` at the memory location specified by `address` and returns the previous value stored at that address

A Spin Lock Using Test-And-Set

- a test-and-set instruction can be used to enforce mutual exclusion
- for each critical section, define a `lock` variable

  ```
  boolean lock;  /* shared, initially false */
  ```

  We will use the lock variable to keep track of whether there is a thread in the critical section, in which case the value of `lock` will be `true`
- before a thread can enter the critical section, it does the following:

  ```
  while (TestAndSet(&lock,true)) { }  /* busy-wait */
  ```

- when the thread leaves the critical section, it does

  ```
  lock = false;
  ```

- this enforces mutual exclusion (why?), but starvation is a possibility

> This construct is sometimes known as a *spin lock*, since a thread "spins" in the while loop until the critical section is free. Spin locks are widely used on multiprocessors.

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*.

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`

Mutual Exclusion Using a Semaphore

```
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 */
```


Producer/Consumer Synchronization

- suppose we have threads that add items to a list (producers) and threads that remove items from the list (consumers)
- suppose we want to ensure that consumers do not consume if the list is empty - instead they must wait until the list has something in it
- this requires synchronization between consumers and producers
- semaphores can provide the necessary synchronization, as shown on the next slide

Producer/Consumer Synchronization using Semaphores

```
struct semaphore *s;
s = sem_create("Items", 0); /* initial value is 0 */

Producer's Pseudo-code:
  add item to the list (call list_append())
  V(s);

Consumer's Pseudo-code:
  P(s);
  remove item from the list (call list_remove_front())
```

The Items semaphore does not enforce mutual exclusion on the list. If we want mutual exclusion, we can also use semaphores to enforce it. (How?)

Bounded Buffer Producer/Consumer Synchronization

- suppose we add one more requirement: the number of items in the list should not exceed `N`
- producers that try to add items when the list is full should be made to wait until the list is no longer full
- We can use an additional semaphore to enforce this new constraint:
  - semaphore `Full` is used to count the number of full (occupied) entries in the list (to ensure nothing is produced if the list is full)
  - semaphore `Full` is used to count the number of empty (unoccupied) entries in the list (to ensure nothing is consumed if the list is empty)

```
struct semaphore *full;
struct semaphore *empty;
full = sem_create("Full", 0);   /* initial value = 0 */
empty = sem_create("Empty", N); /* initial value = N */
```

Bounded Buffer Producer/Consumer Synchronization with Semaphores

```
Producer's Pseudo-code:
  P(empty);
  add item to the list (call list_append())
  V(full);

Consumer's Pseudo-code:
  P(full);
  remove item from the list (call list_remove_front())
  V(empty);
```


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);
}
```

Thread Blocking

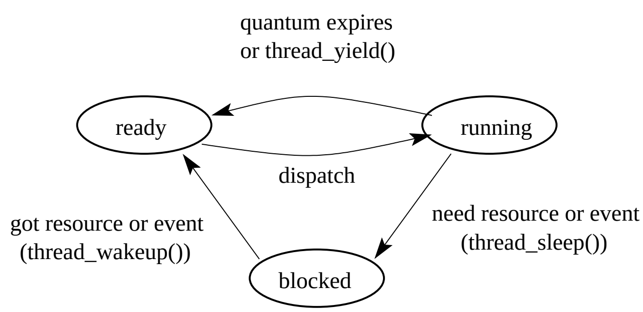
- Sometimes a thread will need to wait for an event. One example is on the previous slide: a thread that attempts a P() operation on a zero-valued semaphore must wait until the semaphore's value becomes positive.
- other examples that we will see later on:
  - wait for data from a (relatively) slow device
  - 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 all 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);
}
```

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_aquire(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.

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 `notempty`)
  - $count < N$ (condition variable `notfull`)
- 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 it 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, 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.

Deadlocks

- Suppose there are two threads and two locks, `lockA` and `lockB`, both initially unlocked.
- Suppose the following sequence of events occurs
  1. Thread 1 does `lock_acquire(lockA)`.
  2. Thread 2 does `lock_acquire(lockB)`.
  3. Thread 1 does `lock_acquire(lockB)` and blocks, because `lockB` is held by thread 2.
  4. Thread 2 does `lock_acquire(lockA)` and blocks, because `lockA` is held by thread 1.

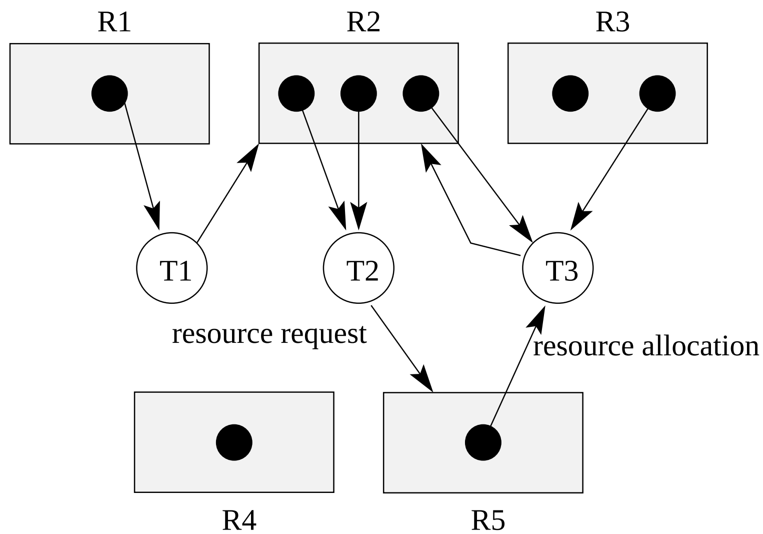
These two threads are *deadlocked* - neither thread can make progress. Waiting will not resolve the deadlock. The threads are permanently stuck.

Deadlocks (Another Simple Example)

- Suppose a machine has 64 MB of memory. The following sequence of events occurs.
  1. Thread A starts, requests 30 MB of memory.
  2. Thread B starts, also requests 30 MB of memory.
  3. Thread A requests an additional 8 MB of memory. The kernel blocks thread A since there is only 4 MB of available memory.
  4. Thread B requests an additional 5 MB of memory. The kernel blocks thread B since there is not enough memory available.

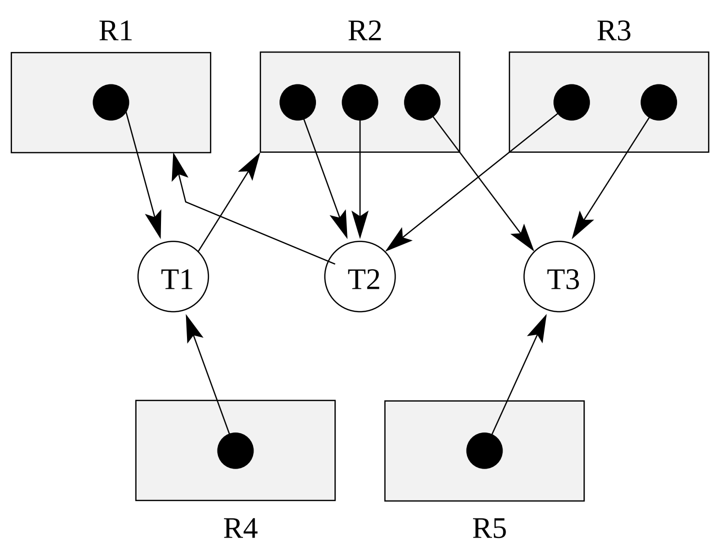
These two threads are deadlocked.

Resource Allocation Graph (Example)



Is there a deadlock in this system?

Resource Allocation Graph (Another Example)



Is there a deadlock in this system?

Deadlock Prevention

No Hold and Wait: prevent a thread from requesting resources if it currently has resources allocated to it. A thread may hold several resources, but to do so it must make a single request for all of them.

Preemption: take resources away from a thread and give them to another (usually not possible). Thread is restarted when it can acquire all the resources it needs.

Resource Ordering: Order (e.g., number) 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.

Deadlock Detection and Recovery

- main idea: the system maintains the resource allocation graph and tests it to determine whether there is a deadlock. If there is, the system must recover from the deadlock situation.
- deadlock recovery is usually accomplished by terminating one or more of the threads involved in the deadlock
- when to test for deadlocks? Can test on every blocked resource request, or can simply test periodically. Deadlocks persist, so periodic detection will not “miss” them.

Deadlock detection and deadlock recovery are both costly. This approach makes sense only if deadlocks are expected to be infrequent.

Detecting Deadlock in a Resource Allocation Graph

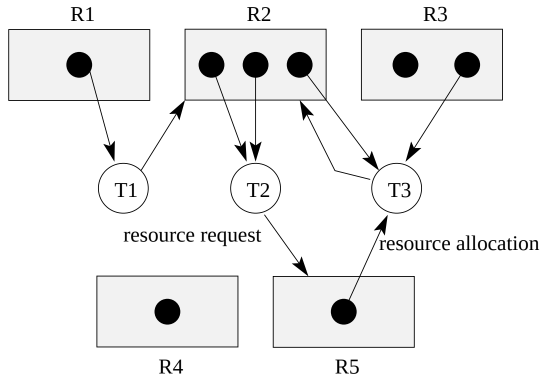
- System State Notation:
  - D_i: demand vector for thread T_i
  - A_i: current allocation vector for thread T_i
  - U: unallocated (available) resource vector
- Additional Algorithm Notation:
  - R: scratch resource vector
  - f_i: algorithm is finished with thread T_i? (boolean)

Detecting Deadlock (cont'd)

```
/* initialization */
R = U
for all i, f_i = false
/* can each thread finish? */
while ∃ i ( ¬ f_i ∧ (D_i ≤ R) ) {
      R = R + A_i
      f_i = true
}
/* if not, there is a deadlock */
if ∃ i ( ¬ f_i ) then report deadlock
else report no deadlock
```

Deadlock Detection, Positive Example

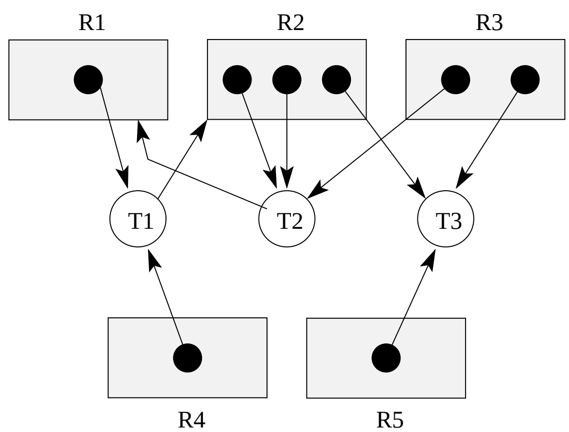
- $D_1 = (0, 1, 0, 0, 0)$
- $D_2 = (0, 0, 0, 0, 1)$
- $D_3 = (0, 1, 0, 0, 0)$
- $A_1 = (1, 0, 0, 0, 0)$
- $A_2 = (0, 2, 0, 0, 0)$
- $A_3 = (0, 1, 1, 0, 1)$
- $U = (0, 0, 1, 1, 0)$



The deadlock detection algorithm will terminate with $f_1 == f_2 == f_3 ==$ `false`, so this system is deadlocked.

Deadlock Detection, Negative Example

- $D_1 = (0, 1, 0, 0, 0)$
- $D_2 = (1, 0, 0, 0, 0)$
- $D_3 = (0, 0, 0, 0, 0)$
- $A_1 = (1, 0, 0, 1, 0)$
- $A_2 = (0, 2, 1, 0, 0)$
- $A_3 = (0, 1, 1, 0, 1)$
- $U = (0, 0, 0, 0, 0)$



This system is not in deadlock. It is possible that the threads will run to completion in the order T_3, T_1, T_2.

What is a Process?

Answer 1: a process is an abstraction of a program in execution

Answer 2: a process consists of

- an *address space*, which represents the memory that holds the program's code and data
- a *thread* of execution (possibly several threads)
- other resources associated with the running program. For example:
  - open files
  - sockets
  - attributes, such as a name (process identifier)
  - . . .

A process with one thread is a *sequential* process. A process with more than one thread is a *concurrent* process.

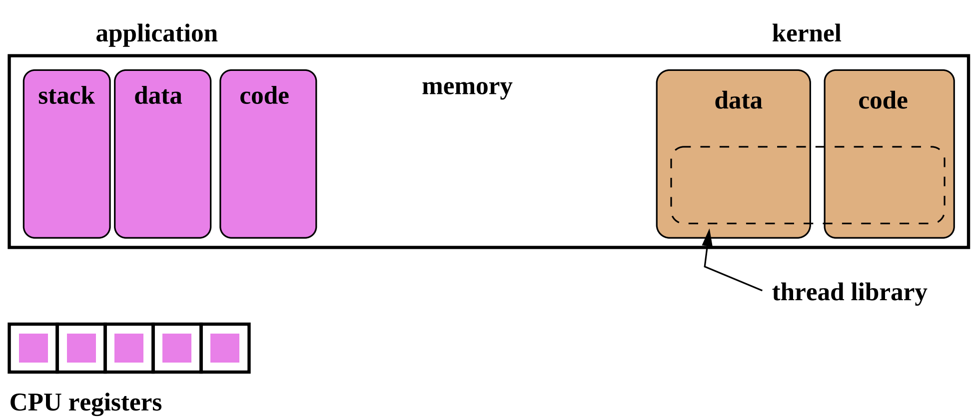
Multiprogramming

- multiprogramming means having multiple processes existing at the same time
- most modern, general purpose operating systems support multiprogramming
- all processes share the available hardware resources, with the sharing coordinated by the operating system:
  - Each process uses some of the available memory to hold its address space. The OS decides which memory and how much memory each process gets
  - OS can coordinate shared access to devices (keyboards, disks), since processes use these devices indirectly, by making system calls.
  - Processes *timeshare* the processor(s). Again, timesharing is controlled by the operating system.
- OS ensures that processes are isolated from one another. Interprocess communication should be possible, but only at the explicit request of the processes involved.

The OS Kernel

- The kernel is a program. It has code and data like any other program.
- Usually kernel code runs in a privileged execution mode, while other programs do not

An Application and the Kernel



Kernel Privilege, Kernel Protection

- What does it mean to run in privileged mode?
- Kernel uses privilege to
  - control hardware
  - protect and isolate itself from processes
- privileges vary from platform to platform, but may include:
  - ability to execute special instructions (like `halt`)
  - ability to manipulate processor state (like execution mode)
  - ability to access memory addresses that can't be accessed otherwise
- kernel ensures that it is *isolated* from processes. No process can execute or change kernel code, or read or write kernel data, except through controlled mechanisms like system calls.

System Calls

- System calls are an interface between processes and the kernel.
- A process uses system calls to request operating system services.
- From point of view of the process, these services are used to manipulate the abstractions that are part of its execution environment. For example, a process might use a system call to
  - open a file
  - send a message over a pipe
  - create another process
  - increase the size of its address space

How System Calls Work

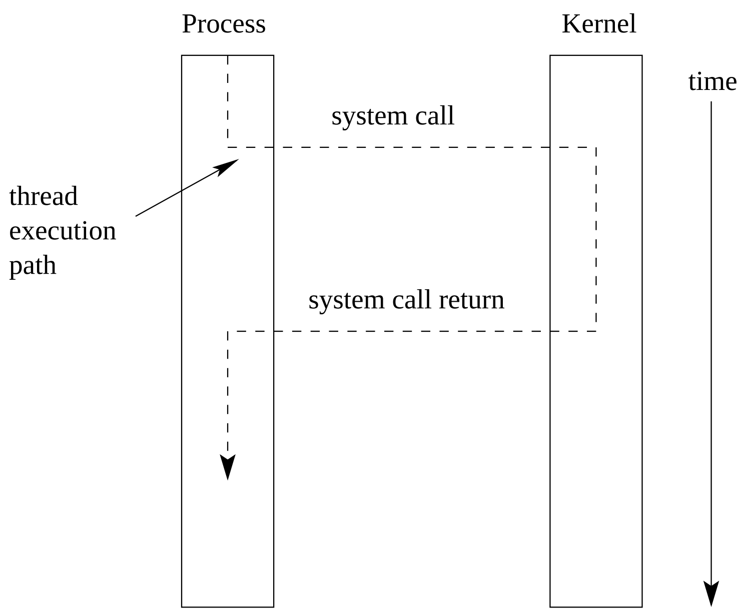
- The hardware provides a mechanism that a running program can use to cause a system call. Often, it is a special instruction, e.g., the MIPS `syscall` instruction.
- What happens on a system call:
  - the processor is switched to system (privileged) execution mode
  - key parts of the current thread context, such as the program counter, are saved
  - the program counter is set to a fixed (determined by the hardware) memory address, which is within the kernel's address space

System Call Execution and Return

- Once a system call occurs, the calling thread will be executing a system call handler, which is part of the kernel, in privileged mode.
- The kernel's handler determines which service the calling process wanted, and performs that service.
- When the kernel is finished, it returns from the system call. This means:
  - restore the key parts of the thread context that were saved when the system call was made
  - switch the processor back to unprivileged (user) execution mode
- Now the thread is executing the calling process' program again, picking up where it left off when it made the system call.

> A system call causes a thread to stop executing application code and to start executing kernel code in privileged mode. The system call return switches the thread back to executing application code in unprivileged mode.

System Call Diagram



OS/161 `close` System Call Description

Library: standard C library (libc)

Synopsis:

```
#include <unistd.h>
int
close(int fd);
```

Description: The file handle `fd` is closed. . . .

Return Values: On success, `close` returns 0. On error, -1 is returned and `errno` is set according to the error encountered.

Errors:

EBADF: `fd` is not a valid file handle

EIO: A hard I/O error occurred

An Example System Call: A Tiny OS/161 Application that Uses `close`

```
/* Program: uw-testbin/syscall.c */
#include <unistd.h>
#include <errno.h>

int
main()
{
  int x;
  x = close(999);
  if (x < 0) {
    return errno;
  }
  return x;
}
```


Disassembly listing of uw-testbin/syscall

```
00400050 <main>:
  400050: 27bdffe8  addiu sp,sp,-24
  400054: afbf0010  sw ra,16(sp)
  400058: 0c100047  jal 40011c <close>
  40005c: 240403e7  li a0,999
  400060: 04410003  bgez v0,400070 <main+0x20>
  400064: 00000000  nop
  400068: 3c021000  lui v0,0x1000
  40006c: 8c420000  lw v0,0(v0)
  400070: 8fbf0010  lw ra,16(sp)
  400074: 00000000  nop
  400078: 03e00008  jr ra
  40007c: 27bd0018  addiu sp,sp,24
```

The above can be obtained by disassembling the compiled `syscall` executable file with `cs350-objdump -d`

System Call Wrapper Functions from the Standard Library

```
...
00400114 <write>:
  400114: 08100030  j 4000c0 <__syscall>
  400118: 24020006  li v0,6
0040011c <close>:
  40011c: 08100030  j 4000c0 <__syscall>
  400120: 24020007  li v0,7
00400124 <reboot>:
  400124: 08100030  j 4000c0 <__syscall>
  400128: 24020008  li v0,8
...
```

The above is disassembled code from the standard C library (libc), which is linked with `uw-testbin/syscall.o`. See `lib/libc/syscalls.S` for more information about how the standard C library is implemented.

OS/161 MIPS System Call Conventions

- When the `syscall` instruction occurs:
  - An integer system call code should be located in register R2 (v0)
  - Any system call arguments should be located in registers R4 (a0), R5 (a1), R6 (a2), and R7 (a3), much like procedure call arguments.
- When the system call returns
  - register R7 (a3) will contain a 0 if the system call succeeded, or a 1 if the system call failed
  - register R2 (v0) will contain the system call return value if the system call succeeded, or an error number (errno) if the system call failed.

OS/161 System Call Code Definitions

```
...
#define SYS_read        5
#define SYS_write       6
#define SYS_close       7
#define SYS_reboot      8
#define SYS_sync        9
#define SYS_sbrk        10
...
```

This comes from `kern/include/kern/callno.h`. The files in `kern/include/kern` define things (like system call codes) that must be known by both the kernel and applications.

The OS/161 System Call and Return Processing

```
004000c0 <__syscall>:
  4000c0: 0000000c  syscall
  4000c4: 10e00005  beqz a3,4000dc <__syscall+0x1c>
  4000c8: 00000000  nop
  4000cc: 3c011000  lui at,0x1000
  4000d0: ac220000  sw v0,0(at)
  4000d4: 2403ffff  li v1,-1
  4000d8: 2402ffff  li v0,-1
  4000dc: 03e00008  jr ra
  4000e0: 00000000  nop
```

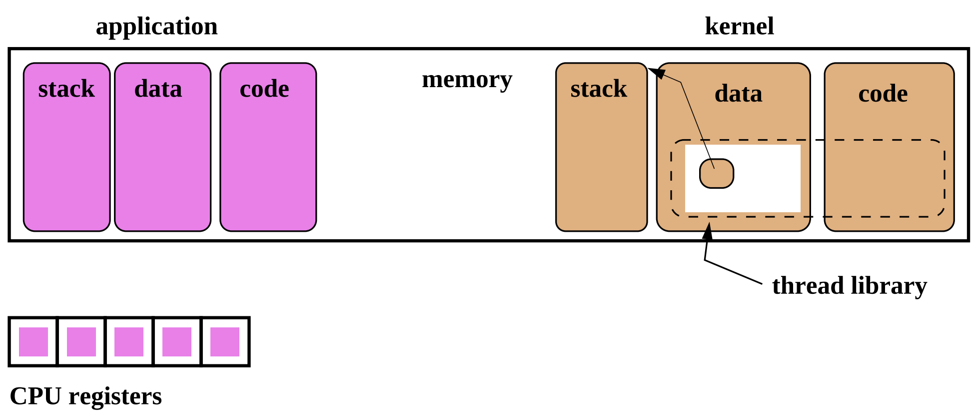
The system call and return processing, from the standard C library. Like the rest of the library, this is unprivileged, user-level code.

OS/161 MIPS Exception Handler

```
exception:
  move k1, sp       /* Save previous stack pointer in k1 */
  mfc0 k0, c0_status   /* Get status register */
  andi k0, k0, CST_KUp /* Check the we-were-in-user-mode bit */
  beq  k0, $0, 1f      /* If clear,from kernel,already have stack
  nop                  /* delay slot */
  /* Coming from user mode - load kernel stack into sp */
  la k0, curkstack     /* get address of "curkstack" */
  lw sp, 0(k0)         /* get its value */
  nop                  /* delay slot for the load */
1:
  mfc0 k0, c0_cause    /* Now, load the exception cause. */
  j common_exception   /* Skip to common code */
  nop                  /* delay slot */
```

When the `syscall` instruction occurs, the MIPS transfers control to address `0x80000080`. This kernel exception handler lives there. See `kern/arch/mips/mips/exception.S`

OS/161 User and Kernel Thread Stacks



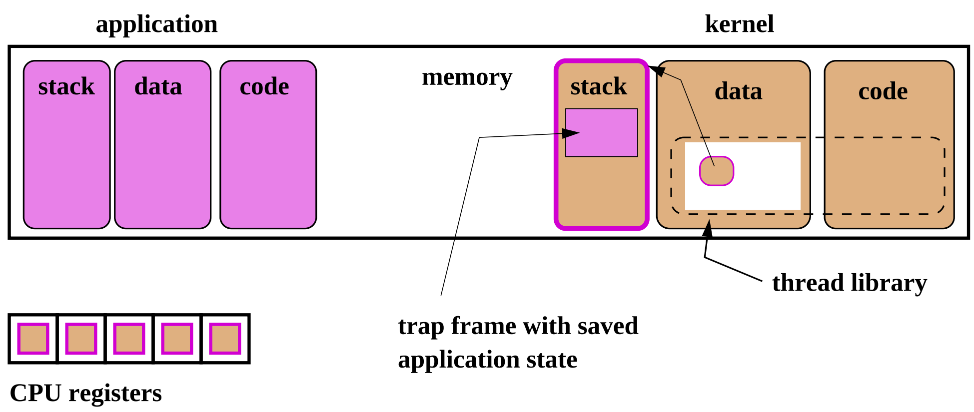
Each OS/161 thread has two stacks, one that is used while the thread is executing unprivileged application code, and another that is used while the thread is executing privileged kernel code.

OS/161 MIPS Exception Handler (cont'd)

The `common_exception` code does the following:

1. allocates a *trap frame* on the thread's kernel stack and saves the user-level application's complete processor state (all registers except k0 and k1) into the trap frame.
2. calls the `mips_trap` function to continue processing the exception.
3. when `mips_trap` returns, restores the application processor state from the trap frame to the registers
4. issues MIPS `jr` and `rfe` (restore from exception) instructions to return control to the application code. The `jr` instruction takes control back to location specified by the application program counter when the `syscall` occurred, and the `rfe` (which happens in the delay slot of the `jr`) restores the processor to unprivileged mode

OS/161 Trap Frame



While the kernel handles the system call, the application's CPU state is saved in a trap frame on the thread's kernel stack, and the CPU registers are available to hold kernel execution state.

mips_trap: Handling System Calls, Exceptions, and Interrupts

- On the MIPS, the same exception handler is invoked to handle system calls, exceptions and interrupts
- The hardware sets a code to indicate the reason (system call, exception, or interrupt) that the exception handler has been invoked
- OS/161 has a handler function corresponding to each of these reasons. The `mips_trap` function tests the reason code and calls the appropriate function: the system call handler (`mips_syscall`) in the case of a system call.
- `mips_trap` can be found in `kern/arch/mips/mips/trap.c`.

Interrupts and exceptions will be presented shortly

OS/161 MIPS System Call Handler

```
mips_syscall(struct trapframe *tf)  {
  assert(curspl==0);
  callno = tf->tf_v0; retval = 0;
  switch (callno) {
    case SYS_reboot:
      err = sys_reboot(tf->tf_a0); /* in kern/main/main.c */
      break;

    /* Add stuff here */

    default:
      kprintf("Unknown syscall %d\n", callno);
      err = ENOSYS;
      break;
  }
```

`mips_syscall` checks the system call code and invokes a handler for the indicated system call. See `kern/arch/mips/mips/syscall.c`

OS/161 MIPS System Call Return Handling

```
  if (err) {
    tf->tf_v0 = err;
    tf->tf_a3 = 1;      /* signal an error */
  } else {
    /* Success. */
    tf->tf_v0 = retval;
    tf->tf_a3 = 0;      /* signal no error */
  }

   /* Advance the PC, to avoid the syscall again. */
  tf->tf_epc += 4;

  /* Make sure the syscall code didn't forget to lower spl *
  assert(curspl==0);
}
```

`mips_syscall` must ensure that the kernel adheres to the system call return convention.

Exceptions

- Exceptions are another way that control is transferred from a process to the kernel.
- Exceptions are conditions that occur during the execution of an instruction by a process. For example, arithmetic overflows, illegal instructions, or page faults (to be discussed later).
- Exceptions are detected by the hardware.
- When an exception is detected, the hardware transfers control to a specific address.
- Normally, a kernel exception handler is located at that address.

Exception handling is similar to, but not identical to, system call handling. (What is different?)

MIPS Exceptions

```
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 OS/161, `mips_trap` uses these codes to decide whether it has been invoked because of an interrupt, a system call, or an exception.

Interrupts (Revisited)

- Interrupts are a third mechanism by which control may be transferred to the kernel
- Interrupts are similar to exceptions. However, they are caused by hardware devices, not by the execution of a program. For example:
  - a network interface may generate an interrupt when a network packet arrives
  - a disk controller may generate an interrupt to indicate that it has finished writing data to the disk
  - a timer may generate an interrupt to indicate that time has passed
- Interrupt handling is similar to exception handling - current execution context is saved, and control is transferred to a kernel interrupt handler at a fixed address.

Interrupts, Exceptions, and System Calls: Summary

- interrupts, exceptions and system calls are three mechanisms by which control is transferred from an application program to the kernel
- when these events occur, the hardware switches the CPU into privileged mode and transfers control to a predefined location, at which a kernel *handler* should be located
- the handler saves the application thread context so that the kernel code can be executed on the CPU, and restores the application thread context just before control is returned to the application

Implementation of Processes

- The kernel maintains information about all of the processes in the system in a data structure often called the process table.
- Per-process information may include:
  - process identifier and owner
  - current process state and other scheduling information
  - lists of resources allocated to the process, such as open files
  - accounting information

In OS/161, some process information (e.g., an address space pointer) is kept in the `thread` structure. This works only because each OS/161 process has a single thread.

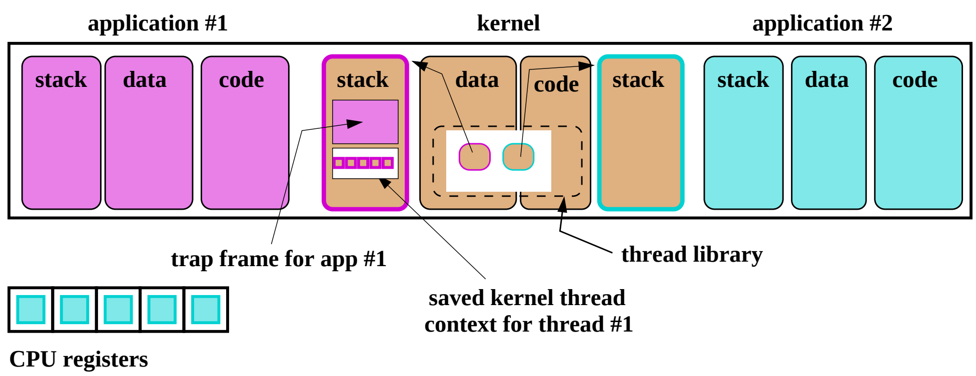
52

Implementing Timesharing

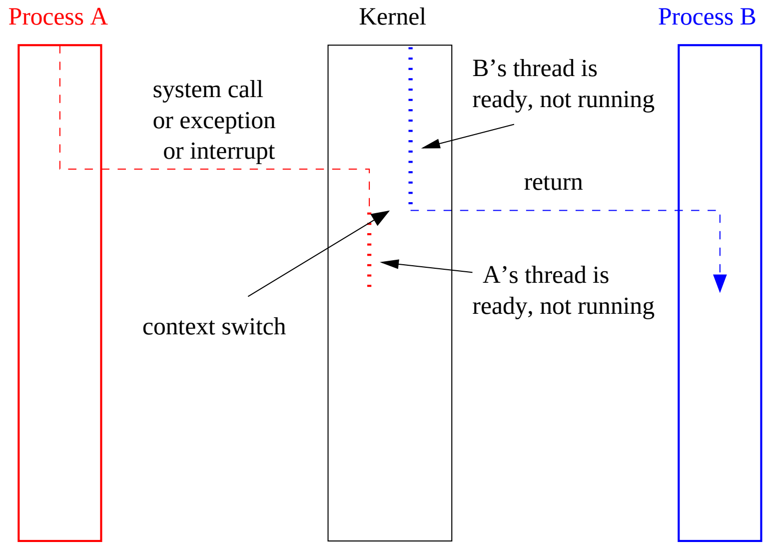
- whenever a system call, exception, or interrupt occurs, control is transferred from the running program to the kernel
- at these points, the kernel has the ability to cause a context switch from the running process' thread to another process' thread
- notice that these context switches always occur while a process' thread is executing kernel code

> By switching from one process's thread to another process's thread, the kernel timeshares the processor among multiple processes.

Two Processes in OS/161

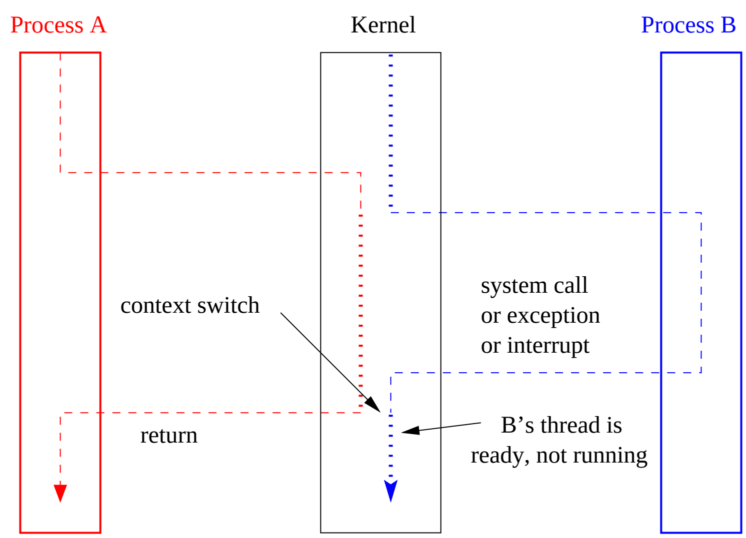


Timesharing Example (Part 1)



Kernel switches execution context to Process B.

Timesharing Example (Part 2)

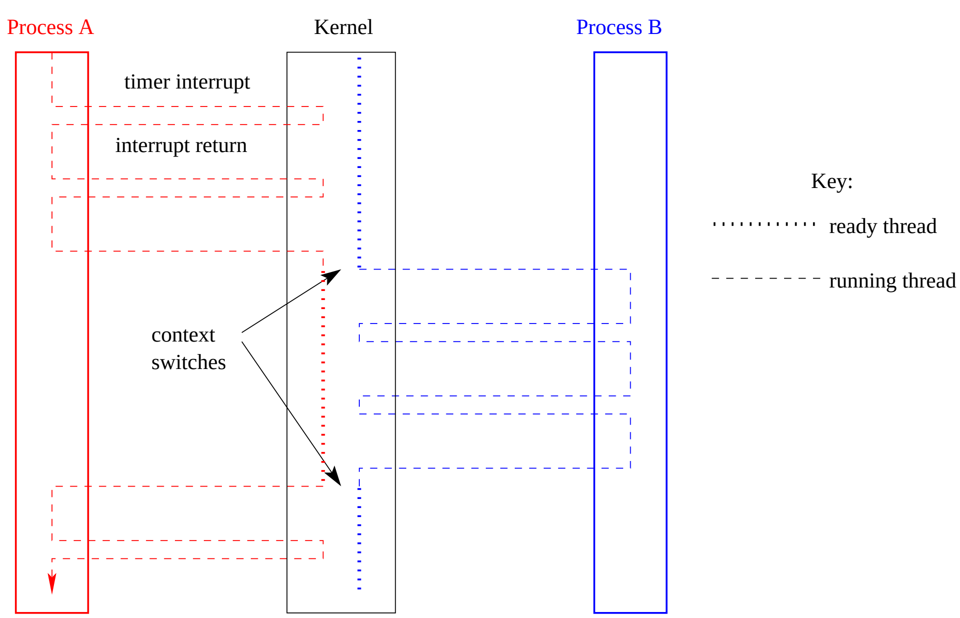


Kernel switches execution context back to process A.

Implementing Preemption

- the kernel uses interrupts from the system timer to measure the passage of time and to determine whether the running process's quantum has expired.
- a timer interrupt (like any other interrupt) transfers control from the running program to the kernel.
- this gives the kernel the opportunity to preempt the running thread and dispatch a new one.

Preemptive Multiprogramming Example



System Calls for Process Management

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

The Process Model

- Although the general operations supported by the process interface are straightforward, there are some less obvious aspects of process behaviour that must be defined by an operating system.

  Process Initialization: When a new process is created, how is it initialized? What is in the address space? What is the initial thread context? Does it have any other resources?

  Multithreading: Are concurrent processes supported, or is each process limited to a single thread?

  Inter-Process Relationships: Are there relationships among processes, e.g, parent/child? If so, what do these relationships mean?

Virtual and Physical Addresses

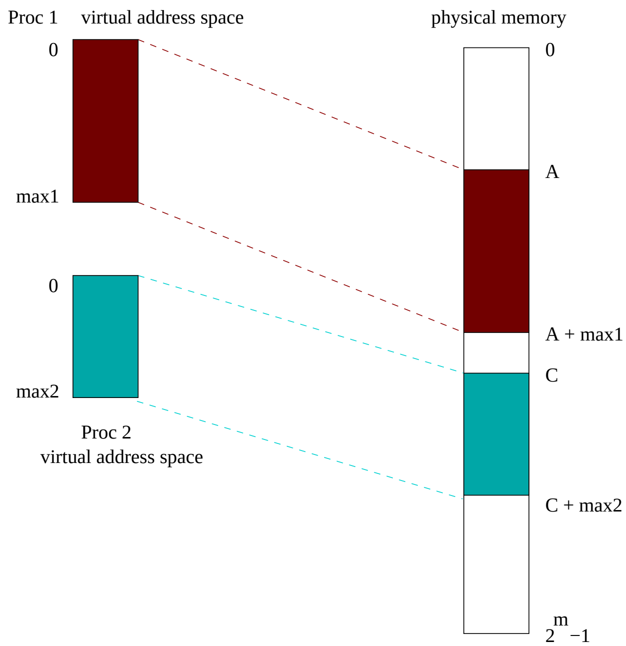
- Physical addresses are provided directly by the machine.
  - one physical address space per machine
  - the size of a physical address determines the maximum amount of addressable physical memory
- Virtual addresses (or logical addresses) are addresses provided by the OS to processes.
  - one virtual address space *per process*
- Programs use virtual addresses. As a program runs, the hardware (with help from the operating system) converts each virtual address to a physical address.
- The conversion of a virtual address to a physical address is called *address translation*.

On the MIPS, virtual addresses and physical addresses are 32 bits long. This limits the size of virtual and physical address spaces.

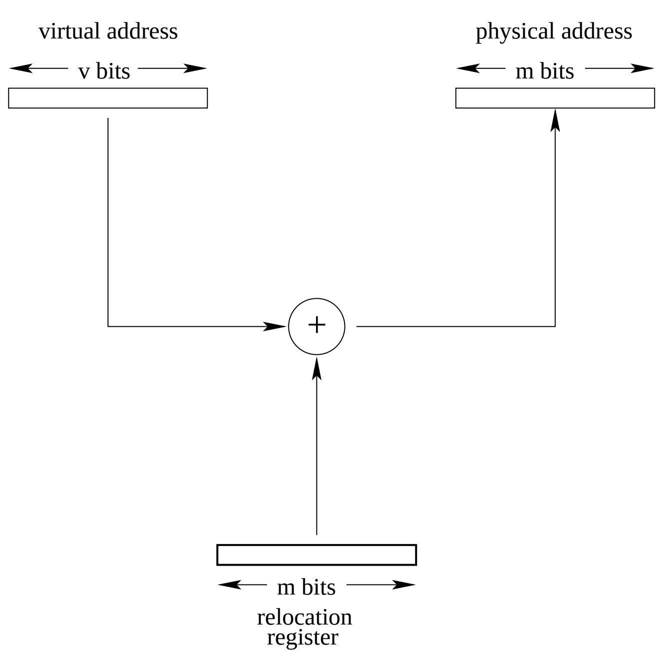
Simple Address Translation: Dynamic Relocation

- hardware provides a *memory management unit* which includes a *relocation register*
- at run-time, the contents of the relocation register are added to each virtual address to determine the corresponding physical address
- the OS maintains a separate relocation register value for each process, and ensures that relocation register is reset on each context switch
- Properties
  - each virtual address space corresponds to a contiguous range of physical addresses
  - OS must allocate/deallocate variable-sized chunks of physical memory
  - potential for *external fragmentation* of physical memory: wasted, unallocated space

Dynamic Relocation: Address Space Diagram



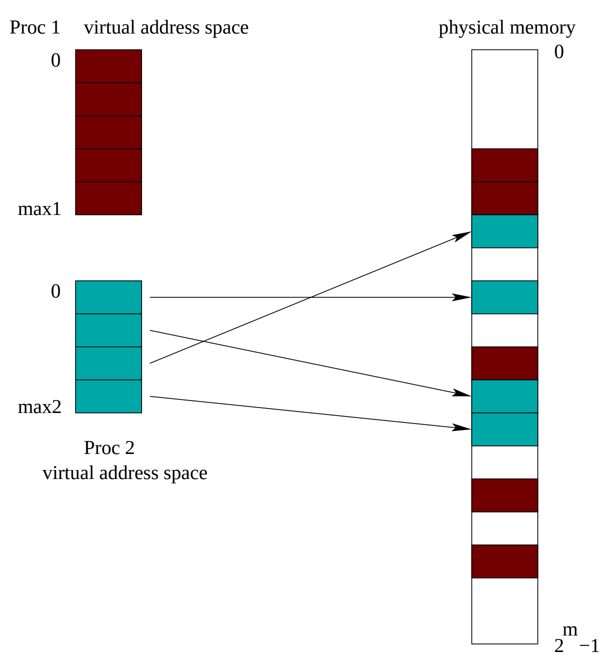
Dynamic Relocation Mechanism



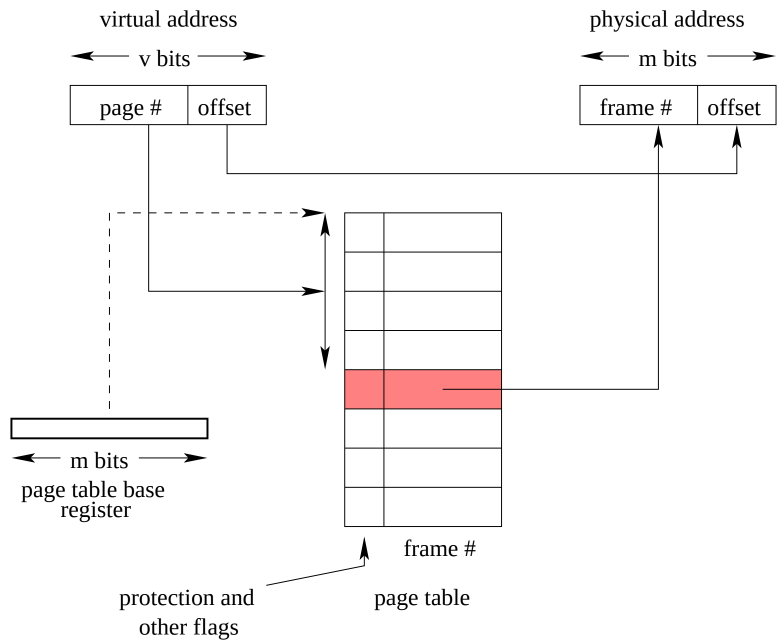
Address Translation: Paging

- Each virtual address space is divided into fixed-size chunks called *pages*
- The physical address space is divided into *frames*. Frame size matches page size.
- OS maintains a *page table* for each process. Page table specifies the frame in which each of the process's pages is located.
- At run time, MMU translates virtual addresses to physical using the page table of the running process.
- Properties
  - simple physical memory management
  - potential for *internal fragmentation* of physical memory: wasted, allocated space
  - virtual address space need not be physically contiguous in physical space after translation.

Address Space Diagram for Paging



Paging Mechanism



Memory Protection

- during address translation, the MMU checks to ensure that the process uses only *valid* virtual addresses
  - typically, each PTE contains a *valid bit* which indicates whether that PTE contains a valid page mapping
  - the MMU may also check that the virtual page number does not index a PTE beyond the end of the page table
- the MMU may also enforce other protection rules
  - typically, each PTE contains a *read-only* bit that indicates whether the corresponding page may be modified by the process
- if a process attempts to violated these protection rules, the MMU raises an exception, which is handled by the kernel

The kernel controls which pages are valid and which are protected by setting the contents of PTEs and/or MMU registers.

Roles of the Kernel and the MMU (Summary)

- Kernel:
  - save/restore MMU state on context switches
  - create and manage page tables
  - manage (allocate/deallocate) physical memory
  - handle exceptions raised by the MMU
- MMU (hardware):
  - translate virtual addresses to physical addresses
  - check for and raise exceptions when necessary

Remaining Issues

translation speed: Address translation happens very frequently. (How frequently?) It must be fast.

sparseness: Many programs will only need a small part of the available space for their code and data.

the kernel: Each process has a virtual address space in which to run. What about the kernel? In which address space does it run?

Speed of Address Translation

- Execution of each machine instruction may involve one, two or more memory operations
  - one to fetch instruction
  - one or more for instruction operands
- Address translation through a page table adds one extra memory operation (for page table entry lookup) for each memory operation performed during instruction execution
  - Simple address translation through a page table can cut instruction execution rate in half.
  - More complex translation schemes (e.g., multi-level paging) are even more expensive.
- Solution: include a Translation Lookaside Buffer (TLB) in the MMU
  - TLB is a fast, fully associative address translation cache
  - TLB hit avoids page table lookup

TLB

- Each entry in the TLB contains a (page number, frame number) pair.
- If address translation can be accomplished using a TLB entry, access to the page table is avoided.
- Otherwise, translate through the page table, and add the resulting translation to the TLB, replacing an existing entry if necessary. In a *hardware controlled* TLB, this is done by the MMU. In a *software controlled* TLB, it is done by the kernel.
- TLB lookup is much faster than a memory access. TLB is an associative memory - page numbers of all entries are checked simultaneously for a match. However, the TLB is typically small (typically hundreds, e.g. 128, or 256 entries).
- If the MMU cannot distinguish TLB entries from different address spaces, then the kernel must clear or invalidate the TLB. (Why?)

The MIPS R3000 TLB

- The MIPS has a software-controlled TLB that can hold 64 entries.
- Each TLB entry includes a virtual page number, a physical frame number, an address space identifier (not used by OS/161), and several flags (valid, read-only).
- OS/161 provides low-level functions for managing the TLB:

  TLB_Write: modify a specified TLB entry

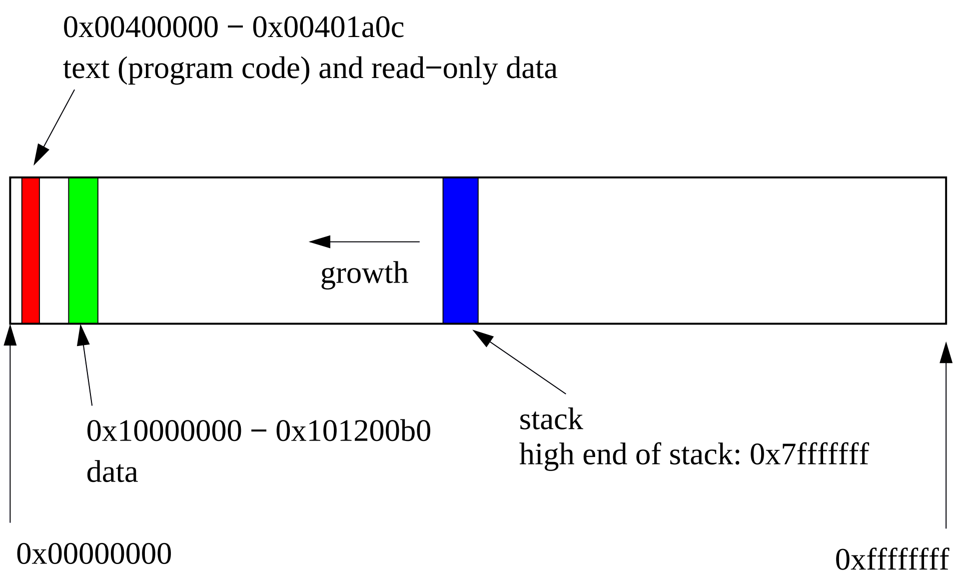
  TLB_Random: modify a random TLB entry

  TLB_Read: read a specified TLB entry

  TLB_Probe: look for a page number in the TLB
- If the MMU cannot translate a virtual address using the TLB it raises an exception, which must be handled by OS/161.

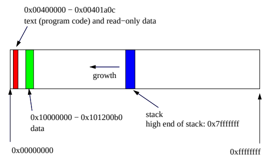
See `kern/arch/mips/include/tlb.h`

What is in a Virtual Address Space?



This diagram illustrates the layout of the virtual address space for the OS/161 test application `testbin/sort`

Handling Sparse Address Spaces: Sparse Page Tables



- Consider the page table for `testbin/sort`, assuming a 4 Kbyte page size:
  - need 2^{19} page table entries (PTEs) to cover the bottom half of the virtual address space.
  - the text segment occupies 2 pages, the data segment occupies 289 pages, and OS/161 sets the initial stack size to 12 pages
- The kernel will mark a PTE as invalid if its page is not mapped.
- In the page table for `testbin/sort`, only 303 of 2^{19} PTEs will be valid.

An attempt by a process to access an invalid page causes the MMU to generate an exception (known as a *page fault*) which must be handled by the operating system.

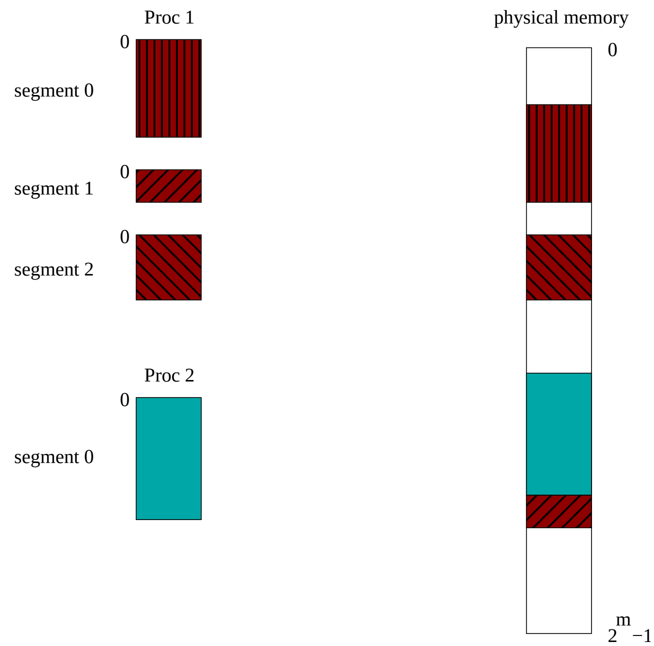
Segmentation

- Often, programs (like `sort`) need several virtual address segments, e.g, for code, data, and stack.
- One way to support this is to turn *segments* into first-class citizens, understood by the application and directly supported by the OS and the MMU.
- Instead of providing a single virtual address space to each process, the OS provides multiple virtual segments. Each segment is like a separate virtual address space, with addresses that start at zero.
- With segmentation, a virtual address can be thought of as having two parts:

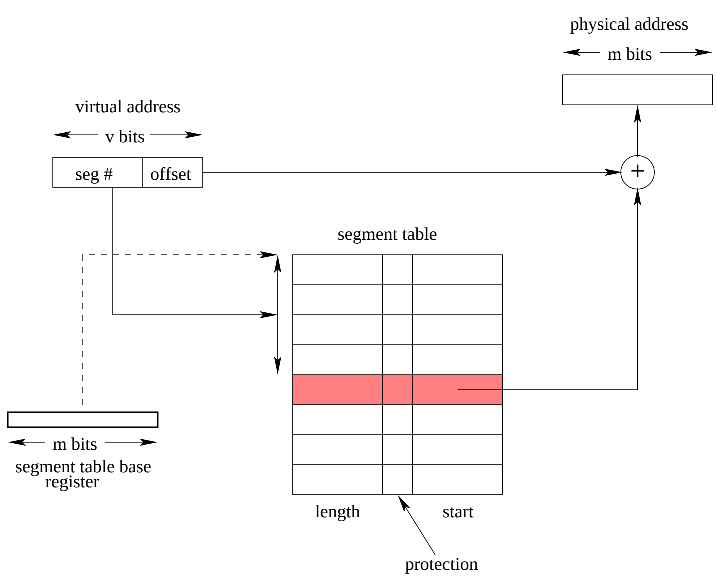
  (segment ID, address within segment)

- Each segment:
  - can grow (or shrink) independently of the other segments, up to some maximum size
  - has its own attributes, e.g, read-only protection

Segmented Address Space Diagram

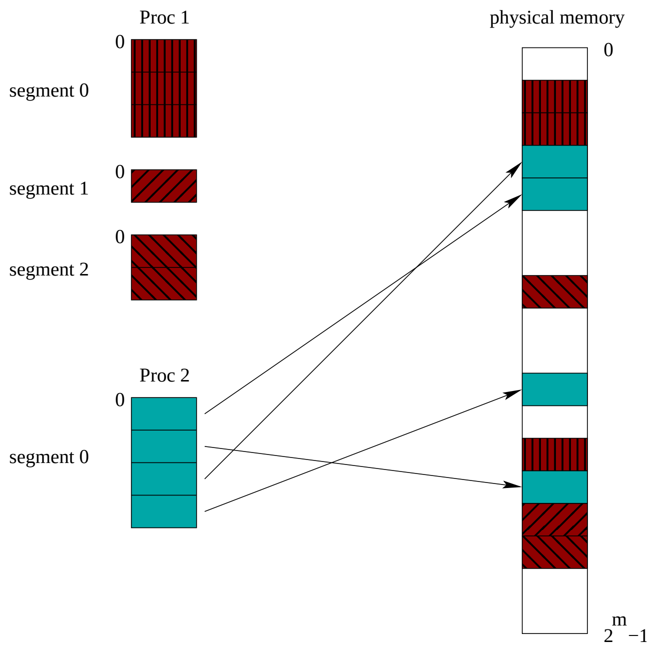


Mechanism for Translating Segmented Addresses

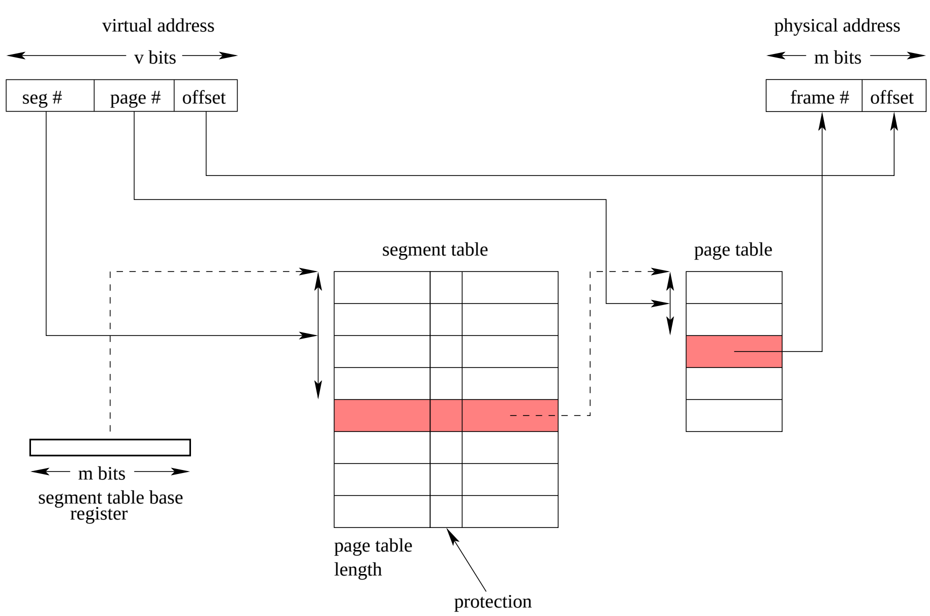


This translation mechanism requires physically contiguous allocation of segments.

Combining Segmentation and Paging



Combining Segmentation and Paging: Translation Mechanism



OS/161 Address Spaces: dumbvm

- OS/161 starts with a very simple virtual memory implementation
- virtual address spaces are described by `addrspace` objects, which record the mappings from virtual to physical addresses

```
struct addrspace {
#if OPT_DUMBVM
  vaddr_t as_vbase1; /* base virtual address of code segment */
  paddr_t as_pbase1; /* base physical address of code segment */
  size_t as_npages1; /* size (in pages) of code segment */
  vaddr_t as_vbase2; /* base virtual address of data segment */
  paddr_t as_pbase2; /* base physical address of data segment */
  size_t as_npages2; /* size (in pages) of data segment */
  paddr_t as_stackpbase; /* base physical address of stack */
#else
  /* Put stuff here for your VM system */
#endif
};
```

This amounts to a slightly generalized version of simple dynamic relocation, with three bases rather than one.

Address Translation Under `dumbvm`

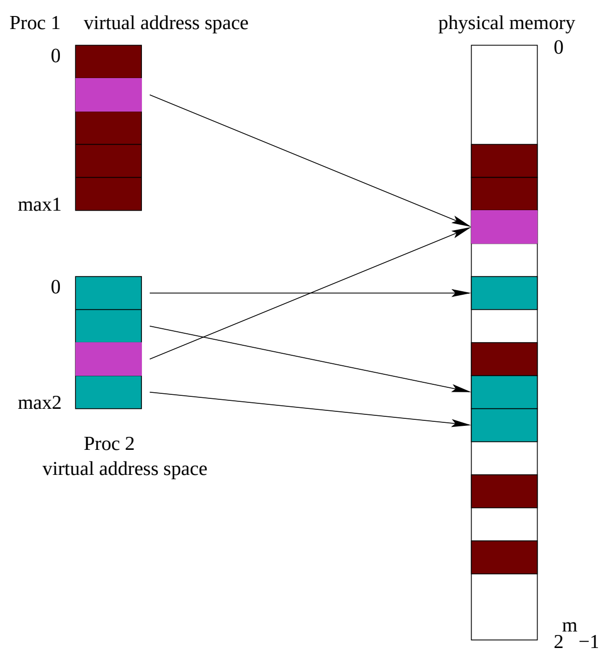
- the MIPS MMU tries to translate each virtual address using the entries in the TLB
- If there is no valid entry for the page the MMU is trying to translate, the MMU generates a TLB fault (called an *address exception*)
- The `vm_fault` function (see `kern/arch/mips/mips/dumbvm.c`) handles this exception for the OS/161 kernel. It uses information from the current process' `addrspace` to construct and load a TLB entry for the page.
- On return from exception, the MIPS retries the instruction that caused the page fault. This time, it may succeed.

`vm_fault` is not very sophisticated. If the TLB fills up, OS/161 will crash!

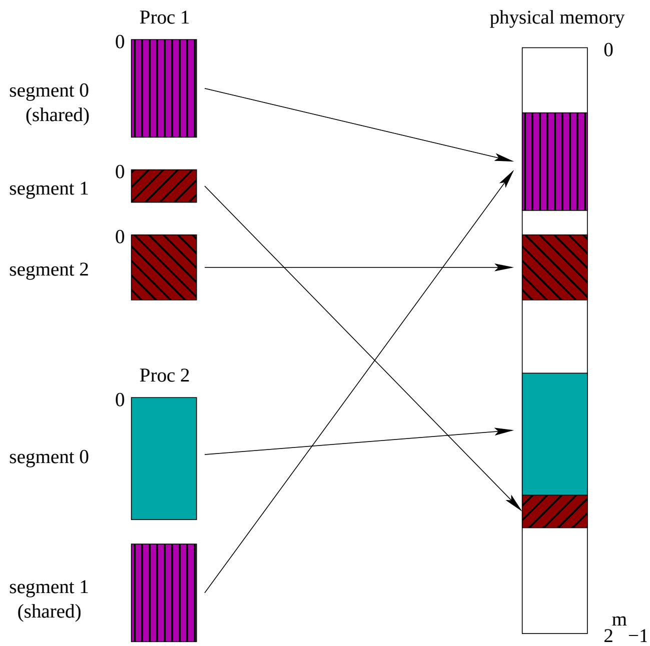
Shared Virtual Memory

- virtual memory sharing allows parts of two or more address spaces to overlap
- shared virtual memory is:
  - a way to use physical memory more efficiently, e.g., one copy of a program can be shared by several processes
  - a mechanism for interprocess communication
- sharing is accomplished by mapping virtual addresses from several processes to the same physical address
- unit of sharing can be a page or a segment

Shared Pages Diagram



Shared Segments Diagram



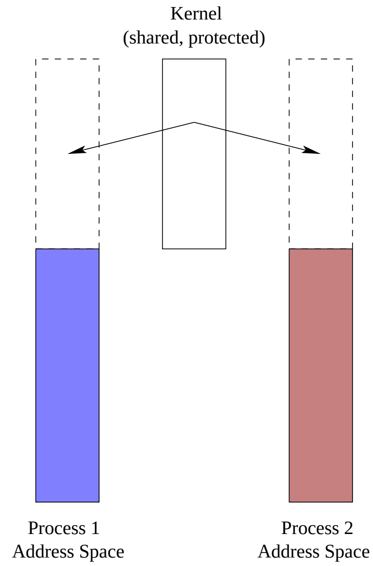
An Address Space for the Kernel

- Each process has its own address space. What about the kernel?
- Two possibilities

  Kernel in physical space: disable address translation in privileged system execution mode, enable it in unprivileged mode

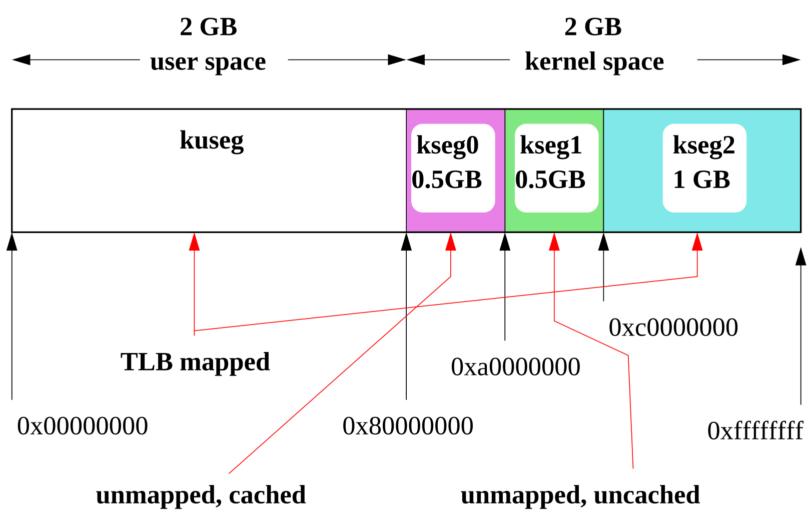
  Kernel in separate virtual address space: need a way to change address translation (e.g., switch page tables) when moving between privileged and unprivileged code
- OS/161, Linux, and other operating systems use a third approach: the kernel is mapped into a portion of the virtual address space of *every process*
- memory protection mechanism is used to isolate the kernel from applications
- one advantage of this approach: application virtual addresses (e.g., system call parameters) are easy for the kernel to use

The Kernel in Process' Address Spaces



Attempts to access kernel code/data in user mode result in memory protection exceptions, not invalid address exceptions.

Address Translation on the MIPS R3000



In OS/161, user programs live in kuseg, kernel code and data structures live in kseg0, devices are accessed through kseg1, and kseg2 is not used.

Loading a Program into an Address Space

- When the kernel creates a process to run a particular program, it must create an address space for the process, and load the program's code and data into that address space
- A program's code and data is described in an *executable file*, which is created when the program is compiled and linked
- OS/161 (and some other operating systems) expect executable files to be in ELF (**E**xecutable and **L**inking **F**ormat) format
- The OS/161 `execv` system call re-initializes the address space of a process

```
#include <unistd.h>
int
execv(const char *program, char **args)
```

- The `program` parameter of the `execv` system call should be the name of the ELF executable file for the program that is to be loaded into the address space.

ELF Files

- ELF files contain address space segment descriptions, which are useful to the kernel when it is loading a new address space
- the ELF file identifies the (virtual) address of the program's first instruction
- the ELF file also contains lots of other information (e.g., section descriptors, symbol tables) that is useful to compilers, linkers, debuggers, loaders and other tools used to build programs

Address Space Segments in ELF Files

- Each ELF segment describes a contiguous region of the virtual address space.
- For each segment, the ELF file includes a segment *image* and a header, which describes:
  - the virtual address of the start of the segment
  - the length of the segment in the virtual address space
  - the location of the start of the image in the ELF file
  - the length of the image in the ELF file
- the image is an exact copy of the binary data that should be loaded into the specified portion of the virtual address space
- the image may be smaller than the address space segment, in which case the rest of the address space segment is expected to be zero-filled

To initialize an address space, the kernel copies images from the ELF file to the specifed portions of the virtual address space

ELF Files and OS/161

- OS/161's `dumbvm` implementation assumes that an ELF file contains two segments:
  - a *text segment*, containing the program code and any read-only data
  - a *data segment*, containing any other global program data
- the ELF file does not describe the stack (why not?)
- `dumbvm` creates a *stack segment* for each process. It is 12 pages long, ending at virtual address `0x7fffffff`

Look at `kern/userprog/loadelf.c` to see how OS/161 loads segments from ELF files

ELF Sections and Segments

- In the ELF file, a program's code and data are grouped together into *sections*, based on their properties. Some sections:

  .text: program code

  .rodata: read-only global data

  .data: initialized global data

  .bss: uninitialized global data (Block Started by Symbol)

  .sbss: small uninitialized global data

- not all of these sections are present in every ELF file
- normally
  - the `.text` and `.rodata` sections together form the text segment
  - the `.data`, `.bss` and `.sbss` sections together form the data segement
- space for *local* program variables is allocated on the stack when the program runs

The `uw-testbin/segments.c` Example Program (1 of 2)

```
#include <unistd.h>

#define N   (200)

int x = 0xdeadbeef;
int t1;
int t2;
int t3;
int array[4096];
char const *str = "Hello World\n";
const int z = 0xabcddcba;

struct example {
  int ypos;
  int xpos;
};
```

The `uw-testbin/segments.c` Example Program (2 of 2)

```
int
main()
{
  int count = 0;
  const int value = 1;
  t1 = N;
  t2 = 2;
  count = x + t1;
  t2 = z + t2 + value;

  reboot(RB_POWEROFF);
  return 0; /* avoid compiler warnings */
}
```


ELF Sections for the Example Program

```
Section Headers:
[Nr] Name          Type             Addr     Off    Size   Flg
[ 0]               NULL             00000000 000000 000000
[ 1] .text         PROGBITS         00400000 010000 000200  AX
[ 2] .rodata       PROGBITS         00400200 010200 000020   A
[ 3] .reginfo      MIPS_REGINFO     00400220 010220 000018   A
[ 4] .data         PROGBITS         10000000 020000 000010  WA
[ 5] .sbss         NOBITS           10000010 020010 000014 WAp
[ 6] .bss          NOBITS           10000030 020010 004000  WA
...
Flags: W (write), A (alloc), X (execute), p (processor specific)

## Size = number of bytes (e.g., .text is 0x200 = 512 bytes
## Off  = offset into the ELF file
## Addr = virtual address
```

The `cs350-readelf` program can be used to inspect OS/161 MIPS ELF files: `cs350-readelf -a segments`

ELF Segments for the Example Program

```
Program Headers:
Type    Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
REGINFO 0x010220 0x00400220 0x00400220 0x00018 0x00018 R   0x4
LOAD    0x010000 0x00400000 0x00400000 0x00238 0x00238 R E 0x10000
LOAD    0x020000 0x10000000 0x10000000 0x00010 0x04030 RW  0x10000
```

- segment info, like section info, can be inspected using the `cs350-readelf` program
- the REGINFO section is not used
- the first LOAD segment includes the .text and .rodata sections
- the second LOAD segment includes .data, .sbss, and .bss

Contents of the Example Program's `.text` Section

```
Contents of section .text:
 400000 3c1c1001 279c8000 2408fff8 03a8e824  <...'...$......$
...

## Decoding 3c1c1001 to determine instruction
## 0x3c1c1001 = binary 111100000111000001000000000001
## 0011 1100 0001 1100 0001 0000 0000 0001
## instr  | rs    | rt    |   immediate
## 6 bits | 5 bits| 5 bits|   16 bits
## 001111 | 00000 | 11100 |  0001 0000 0000 0001
## LUI    | 0     | reg 28|   0x1001
## LUI    | unused| reg 28|   0x1001
## Load upper immediate into rt (register target)
## lui gp, 0x1001
```

The `cs350-objdump` program can be used to inspect OS/161 MIPS ELF file section contents: `cs350-objdump -s segments`

Contents of the Example Program's `.rodata` Section

```
Contents of section .rodata:
 400200 abcddcba 00000000 00000000 00000000  ................
 400210 48656c6c 6f20576f 726c640a 00000000  Hello World.....
...
## const int z = 0xabcddcba
## If compiler doesn't prevent z from being written,
##    then the hardware could.
## 0x48 = 'H' 0x65 = 'e' 0x0a = '\n' 0x00 = '\0'
```

The `.rodata` section contains the "Hello World" string literal and the constant integer variable `z`.

Contents of the Example Program's `.data` Section

```
Contents of section .data:
 10000000 deadbeef 00400210 00000000 00000000  ....@.........
...
## Size = 0x10 bytes = 16 bytes (padding for alignment)
## int x = deadbeef (4 bytes)
## char const *str = "Hello World\n"; (4 bytes)
## address of str = 0x10000004
## value stored in str = 0x00400210.
## NOTE: this is the address of the start
## of the string literal in the .rodata section
```

The `.data` section contains the initialized global variables `str` and `x`.

Contents of the Example Program's `.bss` and `.sbss` Sections

```
...
10000000 D x
10000004 D str
10000010 S t3       ## S indicates sbss section
10000014 S t2
10000018 S t1
1000001c S errno
10000020 S __argv
10000030 B array    ## B indicates bss section
10004030 A _end
10008000 A _gp
```

The `t1`, `t2`, and `t3` variables are in the `.sbss` section. The `array` variable is in the `.bss` section. There are no values for these variables in the ELF file, as they are uninitialized. The `cs350-nm` program can be used to inspect symbols defined in ELF files: `cs350-nm -n <filename>`, in this case `cs350-nm -n segments`.

System Call Interface for Virtual Memory Management

- much memory allocation is implicit, e.g.:
  - allocation for address space of new process
  - implicit stack growth on overflow
- OS may support explicit requests to grow/shrink address space, e.g., Unix `brk` system call.
- shared virtual memory (simplified Solaris example):

  Create: `shmid = shmget(key,size)`

  Attach: `vaddr = shmat(shmid, vaddr)`

  Detach: `shmdt(vaddr)`

  Delete: `shmctl(shmid,IPC_RMID)`

Exploiting Secondary Storage

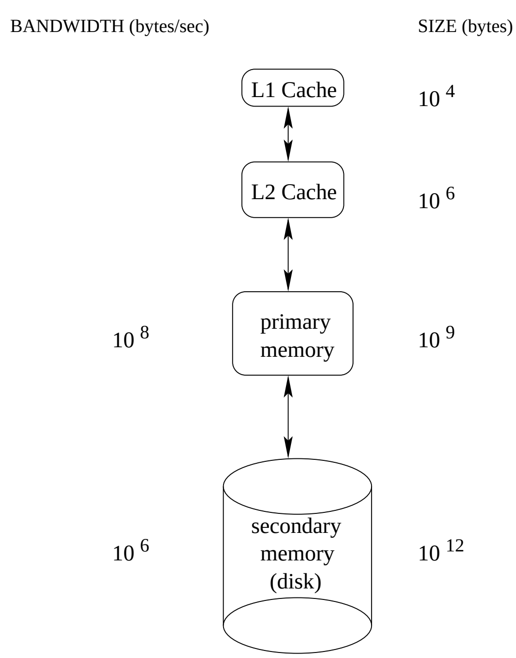
Goals:

- Allow virtual address spaces that are larger than the physical address space.
- Allow greater multiprogramming levels by using less of the available (primary) memory for each process.

Method:

- Allow pages (or segments) from the virtual address space to be stored in secondary memory, as well as primary memory.
- Move pages (or segments) between secondary and primary memory so that they are in primary memory when they are needed.

The Memory Hierarchy



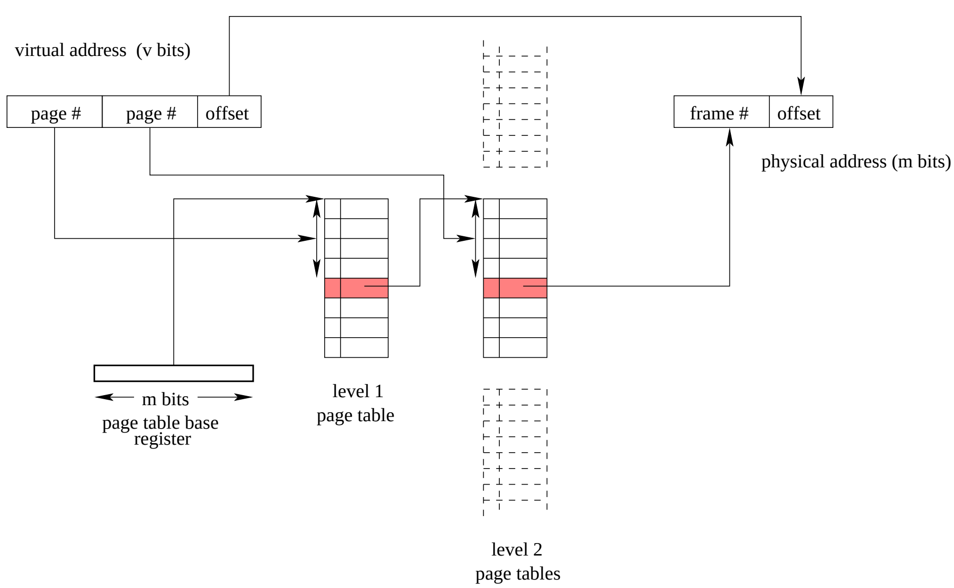
Large Virtual Address Spaces

- Virtual memory allows for very large virtual address spaces, and very large virtual address spaces require large page tables.
- example: 2^{48} byte virtual address space, 8 Kbyte (2^{13} byte) pages, 4 byte page table entries means

$$\frac{2^{48}}{2^{13}}2^2 = 2^{37} \text{ bytes per page table}$$

- page tables for large address spaces may be very large, and
  - they must be in memory, and
  - they must be physically contiguous
- some solutions:
  - multi-level page tables - page the page tables
  - inverted page tables

Two-Level Paging



Inverted Page Tables

- A normal page table maps virtual pages to physical frames. An inverted page table maps physical frames to virtual pages.
- Other key differences between normal and inverted page tables:
  - there is only one inverted page table, not one table per process
  - entries in an inverted page table must include a process identifier
- An inverted page table only specifies the location of virtual pages that are located in memory. Some other mechanism (e.g., regular page tables) must be used to locate pages that are not in memory.

Paging Policies

When to Page?:

Demand paging brings pages into memory when they are used. Alternatively, the OS can attempt to guess which pages will be used, and *prefetch* them.

What to Replace?:

Unless there are unused frames, one page must be replaced for each page that is loaded into memory. A *replacement policy* specifies how to determine which page to replace.

Similar issues arise if (pure) segmentation is used, only the unit of data transfer is segments rather than pages. Since segments may vary in size, segmentation also requires a *placement policy*, which specifies where, in memory, a newly-fetched segment should be placed.

Global vs. Local Page Replacement

- When the system's page reference string is generated by more than one process, should the replacement policy take this into account?

  Global Policy: A global policy is applied to all in-memory pages, regardless of the process to which each one "belongs". A page requested by process X may replace a page that belongs another process, Y.

  Local Policy: Under a local policy, the available frames are allocated to processes according to some memory allocation policy. A replacement policy is then applied separately to each process's allocated space. A page requested by process X replaces another page that "belongs" to process X.

Paging Mechanism

- A *valid* bit (V) in each page table entry is used to track which pages are in (primary) memory, and which are not.

  $V = 1$**:** valid entry which can be used for translation

  $V = 0$**:** invalid entry. If the MMU encounters an invalid page table entry, it raises a *page fault* exception.

- To handle a page fault exception, the operating system must:
  - Determine which page table entry caused the exception. (In SYS/161, and in real MIPS processors, MMU puts the offending virtual address into a register on the CP0 co-processor (register 8/c0_vaddr/BadVaddr). The kernel can read that register.
  - Ensure that that page is brought into memory.

  On return from the exception handler, the instruction that resulted in the page fault will be retried.

- If (pure) segmentation is being used, there will be a valid bit in each segment table entry to indicate whether the segment is in memory.

A Simple Replacement Policy: FIFO

- the FIFO policy: replace the page that has been in memory the longest
- a three-frame example:

Num	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	d	d	d	e	e	e	e	e	e
Frame 2		b	b	b	a	a	a	a	a	c	c	c
Frame 3			c	c	c	b	b	b	b	b	d	d
Fault ?	x	x	x	x	x	x	x			x	x	

Optimal Page Replacement

- There is an optimal page replacement policy for demand paging.
- The OPT policy: replace the page that will not be referenced for the longest time.

Num	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	a	a	a	a	a	a	c	c	c
Frame 2		b	b	b	b	b	b	b	b	b	d	d
Frame 3			c	d	d	d	e	e	e	e	e	e
Fault ?	x	x	x	x			x			x	x	

- OPT requires knowledge of the future.

Other Replacement Policies

- FIFO is simple, but it does not consider:

  Frequency of Use: how often a page has been used?

  Recency of Use: when was a page last used?

  Cleanliness: has the page been changed while it is in memory?

- The *principle of locality* suggests that usage ought to be considered in a replacement decision.
- Cleanliness may be worth considering for performance reasons.

Locality

- Locality is a property of the page reference string. In other words, it is a property of programs themselves.
- *Temporal locality* says that pages that have been used recently are likely to be used again.
- *Spatial locality* says that pages "close" to those that have been used are likely to be used next.

In practice, page reference strings exhibit strong locality. Why?

Frequency-based Page Replacement

- Counting references to pages can be used as the basis for page replacement decisions.
- Example: LFU (Least Frequently Used)
  Replace the page with the smallest reference count.
- Any frequency-based policy requires a reference counting mechanism, e.g., MMU increments a counter each time an in-memory page is referenced.
- Pure frequency-based policies have several potential drawbacks:
  - Old references are never forgotten. This can be addressed by periodically reducing the reference count of every in-memory page.
  - Freshly loaded pages have small reference counts and are likely victims - ignores temporal locality.

Least Recently Used (LRU) Page Replacement

- LRU is based on the principle of temporal locality: replace the page that has not been used for the longest time
- To implement LRU, it is necessary to track each page's recency of use. For example: maintain a list of in-memory pages, and move a page to the front of the list when it is used.
- Although LRU and variants have many applications, LRU is often considered to be impractical for use as a replacement policy in virtual memory systems. Why?

Least Recently Used: LRU

- the same three-frame example:

Num	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	d	d	d	e	e	e	c	c	c
Frame 2		b	b	b	a	a	a	a	a	a	d	d
Frame 3			c	c	c	b	b	b	b	b	b	e
Fault ?	x	x	x	x	x	x	x			x	x	x

The “Use” Bit

- A *use bit* (or *reference bit*) is a bit found in each TLB entry that:
  - is set by the MMU each time the page is used, i.e., each time the MMU translates a virtual address on that page
  - can be read and modified by the operating system
  - operating system copies use information into page table
- The use bit provides a small amount of efficiently-maintainable usage information that can be exploited by a page replacement algorithm.

Entries in the MIPS TLB do not include a use bit.

What if the MMU Does Not Provide a "Use" Bit?

- the kernel can emulate the "use" bit, at the cost of extra exceptions
  1. When a page is loaded into memory, mark it as *invalid* (even though it as been loaded) and set its simulated "use" bit to false.
  2. If a program attempts to access the page, an exception will occur.
  3. In its exception handler, the OS sets the page's simulated "use" bit to "true" and marks the page *valid* so that further accesses do not cause exceptions.
- This technique requires that the OS maintain extra bits of information for each page:
  1. the simulated "use" bit
  2. an "in memory" bit to indicate whether the page is in memory

The Clock Replacement Algorithm

- The clock algorithm (also known as "second chance") is one of the simplest algorithms that exploits the use bit.
- Clock is identical to FIFO, except that a page is "skipped" if its use bit is set.
- The clock algorithm can be visualized as a victim pointer that cycles through the page frames. The pointer moves whenever a replacement is necessary:

```
while use bit of victim is set
   clear use bit of victim
   victim = (victim + 1) % num_frames
choose victim for replacement
victim = (victim + 1) % num_frames
```

Page Cleanliness: the "Modified" Bit

- A page is *modified* (sometimes called dirty) if it has been changed since it was loaded into memory.
- A modified page is more costly to replace than a clean page. (Why?)
- The MMU identifies modified pages by setting a *modified bit* in the TLB entry when the contents of the page change.
- Operating system clears the modified bit when it cleans the page
- The modified bit potentially has two roles:
  - Indicates which pages need to be cleaned.
  - Can be used to influence the replacement policy.

MIPS TLB entries do not include a modified bit.

What if the MMU Does Not Provide a "Modified" Bit?

- Can emulate it in similar fashion to the "use" bit
  1. When a page is loaded into memory, mark it as *read-only* (even if it is actually writeable) and set its simulated "modified" bit to false.
  2. If a program attempts to modify the page, a protection exception will occur.
  3. In its exception handler, if the page is supposed to be writeable, the OS sets the page's simulated "modified" bit to "true" and marks the page as writeable.
- This technique requires that the OS maintain two extra bits of information for each page:
  1. the simulated "modified" bit
  2. a "writeable" bit to indicate whether the page is supposed to be writeable

Enhanced Second Chance Replacement Algorithm

- Classify pages according to their use and modified bits:

  (0,0): not recently used, clean.

  (0,1): not recently used, modified.

  (1,0): recently used, clean

  (1,1): recently used, modified

- Algorithm:
  1. Sweep once looking for (0,0) page. Don't clear use bits while looking.
  2. If none found, look for (0,1) page, this time clearing "use" bits for bypassed frames.
  3. If step 2 fails, all use bits will be zero, repeat from step 1 (guaranteed to find a page).

Page Cleaning

- A modified page must be cleaned before it can be replaced, otherwise changes on that page will be lost.
- *Cleaning* a page means copying the page to secondary storage.
- Cleaning is distinct from replacement.
- Page cleaning may be *synchronous* or *asynchronous*:

  synchronous cleaning: happens at the time the page is replaced, during page fault handling. Page is first cleaned by copying it to secondary storage. Then a new page is brought in to replace it.

  asynchronous cleaning: happens before a page is replaced, so that page fault handling can be faster.

  – asynchronous cleaning may be implemented by dedicated OS *page cleaning threads* that sweep through the in-memory pages cleaning modified pages that they encounter.

Belady's Anomaly

- FIFO replacement, 4 frames

Num	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	a	a	a	e	e	e	e	d	d
Frame 2		b	b	b	b	b	b	a	a	a	a	e
Frame 3			c	c	c	c	c	c	b	b	b	b
Frame 4				d	d	d	d	d	d	c	c	c
Fault?	x	x	x	x			x	x	x	x	x	x

- FIFO example on Slide 51 with same reference string had 3 frames and only 9 faults.

More memory does not necessarily mean fewer page faults.

Stack Policies

- Let $B(m,t)$ represent the set of pages in a memory of size m at time t under some given replacement policy, for some given reference string.
- A replacement policy is called a *stack policy* if, for all reference strings, all m and all t:

$$B(m,t) \subseteq B(m+1,t)$$

- If a replacement algorithm imposes a total order, independent of memory size, on the pages and it replaces the largest (or smallest) page according to that order, then it satisfies the definition of a stack policy.
- Examples: LRU is a stack algorithm. FIFO and CLOCK are not stack algorithms. (Why?)

Stack algorithms do not suffer from Belady's anomaly.

Prefetching

- Prefetching means moving virtual pages into memory before they are needed, i.e., before a page fault results.
- The goal of prefetching is *latency hiding*: do the work of bringing a page into memory in advance, not while a process is waiting.
- To prefetch, the operating system must guess which pages will be needed.
- Hazards of prefetching:
  - guessing wrong means the work that was done to prefetch the page was wasted
  - guessing wrong means that some other potentially useful page has been replaced by a page that is not used
- most common form of prefetching is simple sequential prefetching: if a process uses page x, prefetch page $x + 1$.
- sequential prefetching exploits spatial locality of reference

Page Size

- the virtual memory page size must be understood by both the kernel and the MMU
- some MMUs have support for a configurable page size
- advantages of larger pages
  - smaller page tables
  - larger *TLB footprint*
  - more efficient I/O
- disadvantages of larger pages
  - greater internal fragmentation
  - increased chance of paging in unnecessary data

OS/161 on the MIPS uses a 4KB virtual memory page size.

How Much Physical Memory Does a Process Need?

- Principle of locality suggests that some portions of the process's virtual address space are more likely to be referenced than others.
- A refinement of this principle is the *working set model* of process reference behaviour.
- According to the working set model, at any given time some portion of a program's address space will be heavily used and the remainder will not be. The heavily used portion of the address space is called the *working set* of the process.
- The working set of a process may change over time.
- The *resident set* of a process is the set of pages that are located in memory.

> According to the working set model, if a process's resident set includes its working set, it will rarely page fault.

Resident Set Sizes (Example)

```
PID   VSZ  RSS COMMAND
805 13940 5956 /usr/bin/gnome-session
831  2620  848 /usr/bin/ssh-agent
834  7936 5832 /usr/lib/gconf2/gconfd-2 11
838  6964 2292 gnome-smproxy
840 14720 5008 gnome-settings-daemon
848  8412 3888 sawfish
851 34980 7544 nautilus
853 19804 14208 gnome-panel
857  9656 2672 gpilotd
867  4608 1252 gnome-name-service
```

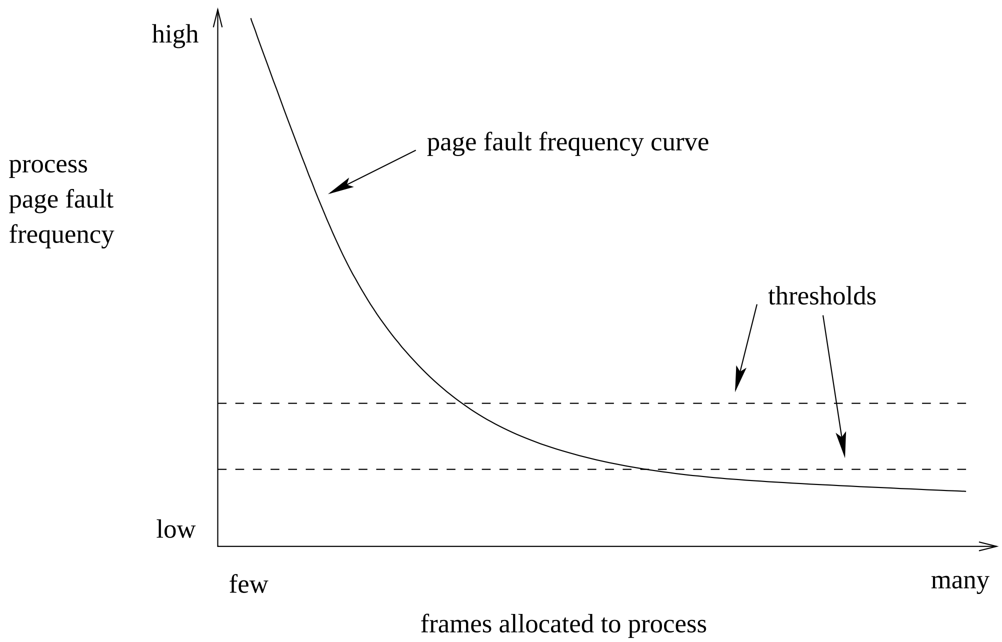
Refining the Working Set Model

- Define $WS(t, \Delta)$ to be the set of pages referenced by a given process during the time interval $(t - \Delta, t)$. $WS(t, \Delta)$ is the working set of the process at time t.
- Define $|WS(t, \Delta)|$ to be the size of $WS(t, \Delta)$, i.e., the number of *distinct* pages referenced by the process.
- If the operating system could track $WS(t, \Delta)$, it could:
  - use $|WS(t, \Delta)|$ to determine the number of frames to allocate to the process under a local page replacement policy
  - use $WS(t, \Delta)$ directly to implement a working-set based page replacement policy: any page that is no longer in the working set is a candidate for replacement

Page Fault Frequency

- A more direct way to allocate memory to processes is to measure their *page fault frequencies* - the number of page faults they generate per unit time.
- If a process's page fault frequency is too high, it needs more memory. If it is low, it may be able to surrender memory.
- The working set model suggests that a page fault frequency plot should have a sharp "knee".

A Page Fault Frequency Plot



Thrashing and Load Control

- What is a good multiprogramming level?
  - If too low: resources are idle
  - If too high: too few resources per process
- A system that is spending too much time paging is said to be *thrashing*. Thrashing occurs when there are too many processes competing for the available memory.
- Thrashing can be cured by load shedding, e.g.,
  - Killing processes (not nice)
  - Suspending and *swapping out* processes (nicer)

Swapping Out Processes

- Swapping a process out means removing all of its pages from memory, or marking them so that they will be removed by the normal page replacement process. Suspending a process ensures that it is not runnable while it is swapped out.
- Which process(es) to suspend?
  - low priority processes
  - blocked processes
  - large processes (lots of space freed) or small processes (easier to reload)
- There must also be a policy for making suspended processes ready when system load has decreased.

The Nature of Program Executions

- A running thread can be modeled as alternating series of *CPU bursts* and *I/O bursts*
  - during a CPU burst, a thread is executing instructions
  - during an I/O burst, a thread is waiting for an I/O operation to be performed and is not executing instructions

Preemptive vs. Non-Preemptive

- A *non-preemptive* scheduler runs only when the running thread gives up the processor through its own actions, e.g.,
  - the thread terminates
  - the thread blocks because of an I/O or synchronization operation
  - the thread performs a Yield system call (if one is provided by the operating system)
- A *preemptive* scheduler may, in addition, force a running thread to stop running
  - typically, a preemptive scheduler will be invoked periodically by a timer interrupt handler, as well as in the circumstances listed above
  - a running thread that is preempted is moved to the ready state

FCFS and Round-Robin Scheduling

First-Come, First-Served (FCFS):

- non-preemptive - each thread runs until it blocks or terminates
- FIFO ready queue

Round-Robin:

- preemptive version of FCFS
- running thread is preempted after a fixed time quantum, if it has not already blocked
- preempted thread goes to the end of the FIFO ready queue

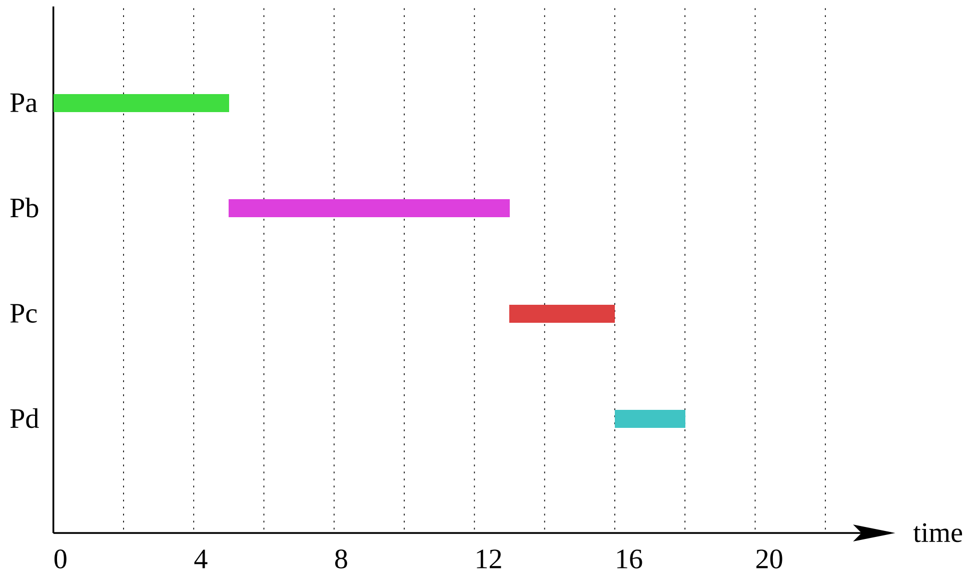
Shortest Job First (SJF) Scheduling

- non-preemptive
- ready threads are scheduled according to the length of their next CPU burst - thread with the shortest burst goes first
- SJF minimizes average waiting time, but can lead to starvation
- SJF requires knowledge of CPU burst lengths
  - Simplest approach is to estimate next burst length of each thread based on previous burst length(s). For example, exponential average considers all previous burst lengths, but weights recent ones most heavily:

    $$B_{i+1} = \alpha b_i + (1 - \alpha) B_i$$

    where B_i is the predicted length of the ith CPU burst, and b_i is its actual length, and $0 \leq \alpha \leq 1$.
- Shortest Remaining Time First is a preemptive variant of SJF. Preemption may occur when a new thread enters the ready queue.

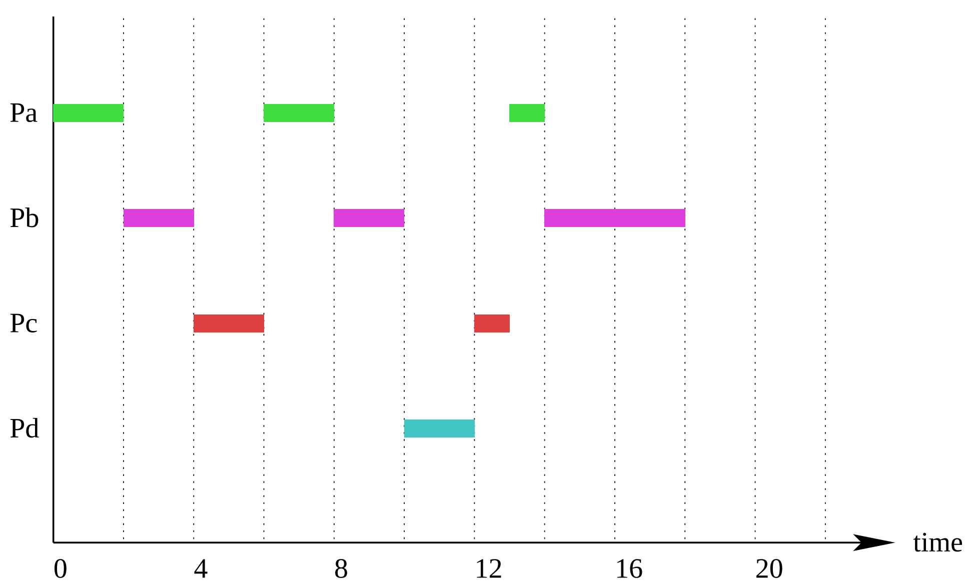
FCFS Gantt Chart Example



Initial ready queue: Pa = 5 Pb = 8 Pc = 3

Thread Pd (=2) "arrives" at time 5

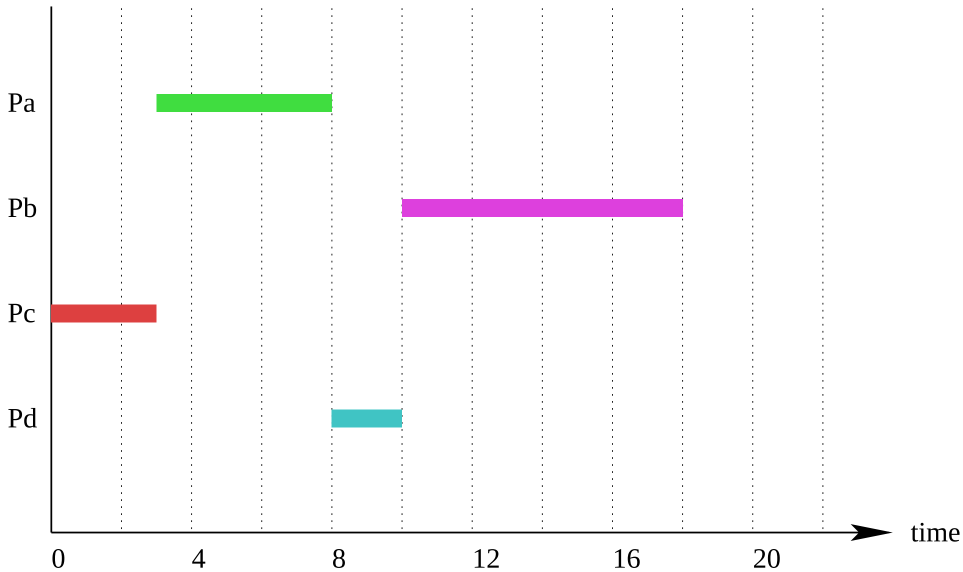
Round Robin Example



Initial ready queue: Pa = 5 Pb = 8 Pc = 3

Thread Pd (=2) "arrives" at time 5 Quantum = 2

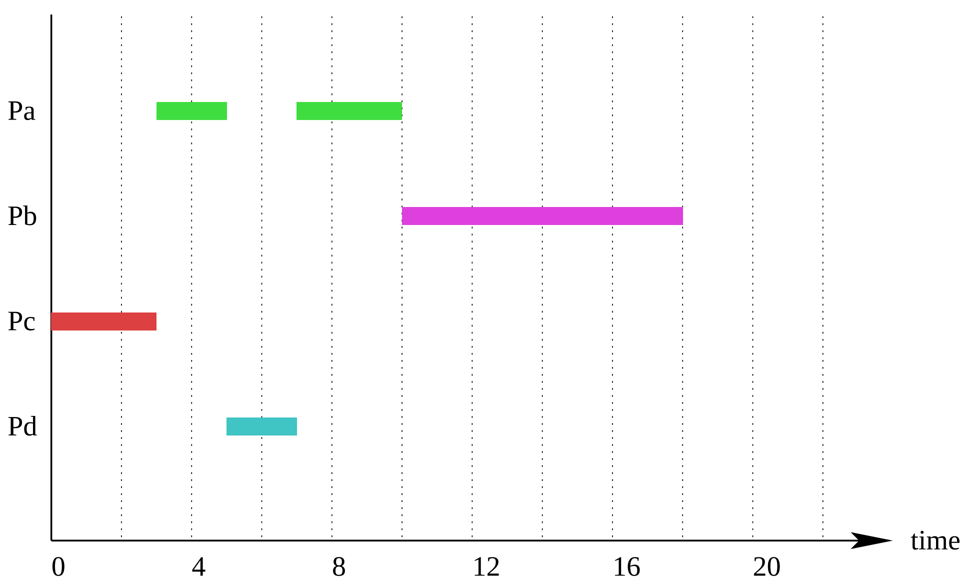
SJF Example



Initial ready queue: Pa = 5 Pb = 8 Pc = 3

Thread Pd (=2) "arrives" at time 5

SRTF Example



Initial ready queue: Pa = 5 Pb = 8 Pc = 3

Thread Pd (=2) "arrives" at time 5

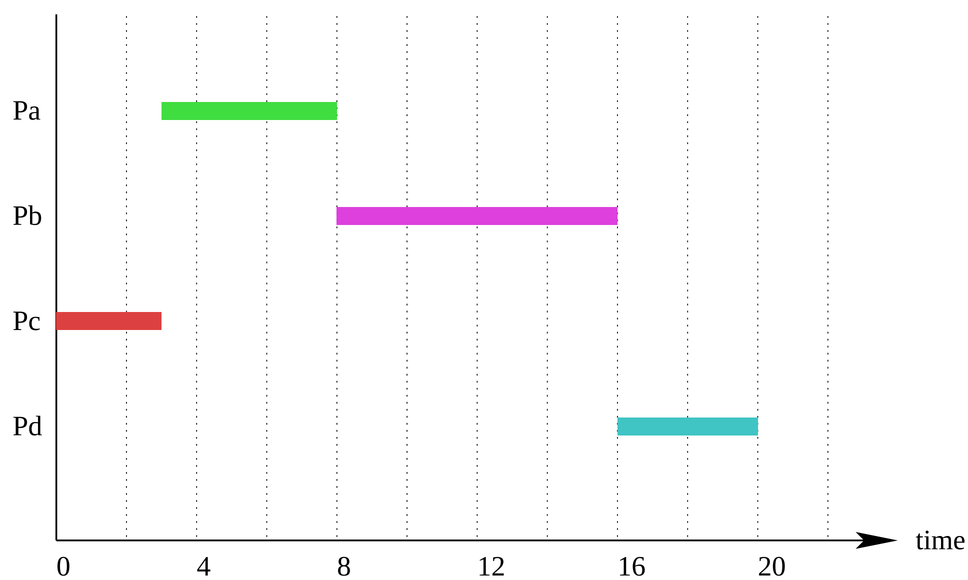
Highest Response Ratio Next

- non-preemptive
- response ratio is defined for each ready thread as:

$$\frac{w + b}{b}$$

  where b is the estimated CPU burst time and w is the actual waiting time
- scheduler chooses the thread with the highest response ratio (choose smallest b in case of a tie)
- HRRN is an example of a heuristic that blends SJF and FCFS

HRRN Example



Initial ready queue: Pa = 5 Pb = 8 Pc = 3
Thread Pd (=4) "arrives" at time 5

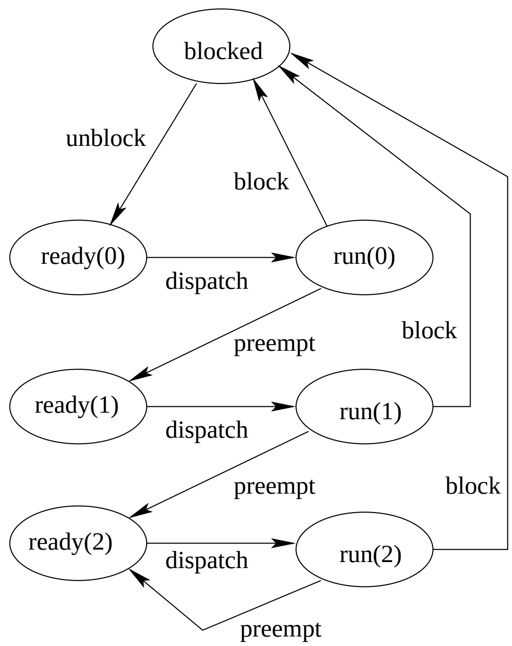
Prioritization

- a scheduler may be asked to take process or thread priorities into account
- for example, priorities could be based on
  - user classification
  - application classification
  - application specification
    (e.g., Linux `setpriority/sched_setscheduler`)
- scheduler can:
  - always choose higher priority threads over lower priority threads
  - use any scheduling heuristic to schedule threads of equal priority
- low priority threads risk starvation. If this is not desired, scheduler must have a mechanism for elevating the priority of low priority threads that have waited a long time

Multilevel Feedback Queues

- gives priority to interactive threads (those with short CPU bursts)
- scheduler maintains several ready queues
- scheduler never chooses a thread in queue i if there are threads in any queue $j < i$.
- threads in queue i use quantum q_i, and $q_i < q_j$ if $i < j$
- newly ready threads go into queue 0
- a level i thread that is preempted goes into the level $i + 1$ ready queue

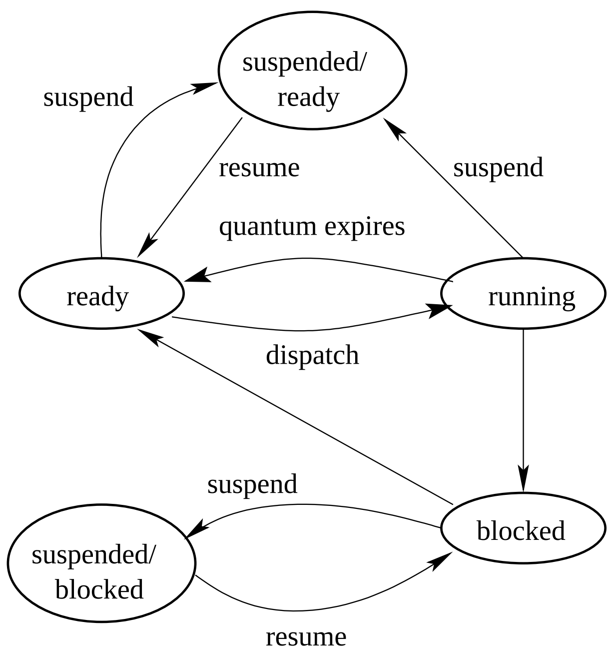
3 Level Feedback Queue State Diagram



Suspending Processes

- suspension prevents a process from running for an extended period of time, until the kernel decides to *resume* it.
- usually because a resource, especially memory, is overloaded
- kernel releases suspended process's resources (e.g., memory)
- operating system may also provide mechanisms for applications or users to request suspension/resumption of processes

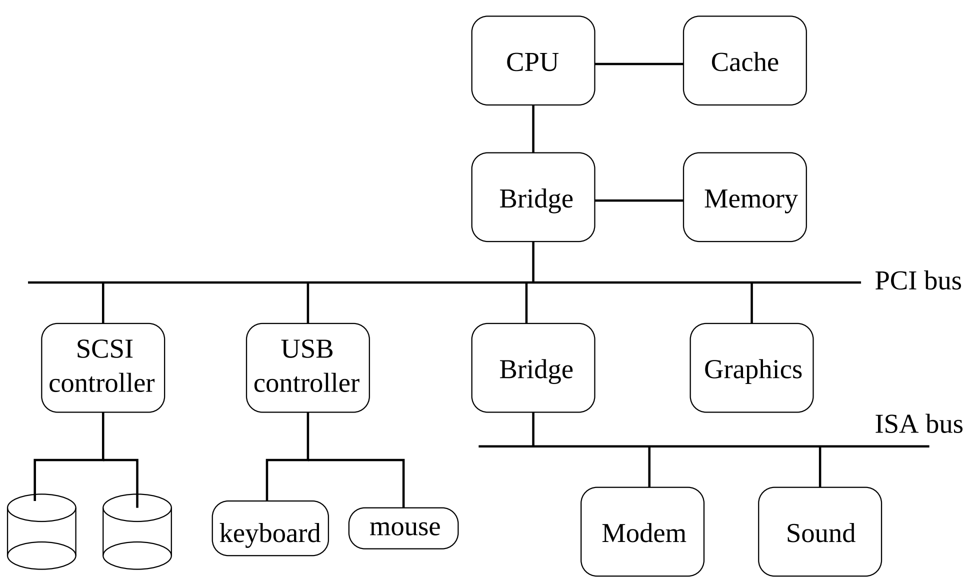
Scheduling States Including Suspend/Resume



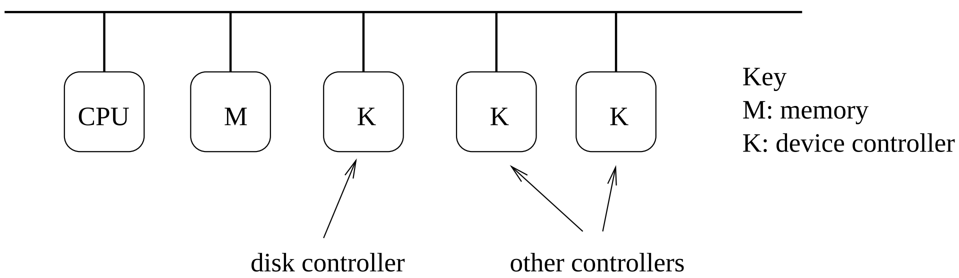
Devices and Device Controllers

- network interface
- graphics adapter
- secondary storage (disks, tape) and storage controllers
- serial (e.g., mouse, keyboard)
- sound
- co-processors
- . . .

Bus Architecture Example



Simplified Bus Architecture



Sys/161 LAMEbus Devices

- LAMEbus controller
- timer/clock - current time, timer, beep
- disk drive - persistent storage
- serial console - character input/output
- text screen - character-oriented graphics
- network interface - packet input/output
- emulator file system - simulation-specific
- hardware trace control - simulation-specific
- random number generator

Device Interactions

- device registers
  - command, status, and data registers
  - CPU accesses register via:
    - special I/O instructions
    - *memory mapping*
- interrupts
  - used by device for asynchronous notification (e.g., of request completion)
  - handled by interrupt handlers in the operating system

Example: LAMEbus timer device registers

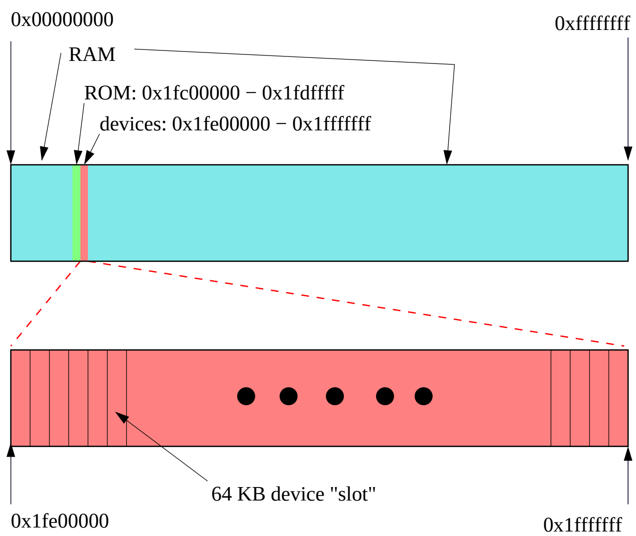
Offset	Size	Type	Description
0	4	status	current time (seconds)
4	4	status	current time (nanoseconds)
8	4	command	restart-on-expiry (auto-restart countdown?)
12	4	status and command	interrupt (reading clears)
16	4	status and command	countdown time (microseconds)
20	4	command	speaker (causes beeps)

Sys/161 uses memory-mapping. Each device's registers are mapped into the *physical address space* of the MIPS processor.

Example: LAMEbus disk controller

Offset	Size	Type	Description
0	4	status	number of sectors
4	4	status and command	status
8	4	command	sector number
12	4	status	rotational speed (RPM)
32768	512	data	transfer buffer

MIPS/OS161 Physical Address Space



Each device is assigned to one of 32 64KB device "slots". A device's registers and data buffers are memory-mapped into its assigned slot.

Device Control Example: Controlling the Timer

```
/* Registers (offsets within the device slot) */
#define LT_REG_SEC   0  /* time of day: seconds */
#define LT_REG_NSEC  4  /* time of day: nanoseconds */
#define LT_REG_ROE   8  /* Restart On countdown-timer Expiry flag
#define LT_REG_IRQ   12 /* Interrupt status register */
#define LT_REG_COUNT 16 /* Time for countdown timer (usec) */
#define LT_REG_SPKR  20 /* Beep control */

/* Get the number of seconds from the lamebus timer */
/* lt->lt_buspos is the slot number of the target device */
secs = bus_read_register(lt->lt_bus, lt->lt_buspos,
    LT_REG_SEC);

/* Get the timer to beep. Doesn't matter what value is sent */
bus_write_register(lt->lt_bus, lt->lt_buspos,
    LT_REG_SPKR, 440);
```


Device Control Example: Address Calculations

```
/* LAMEbus mapping size per slot */
#define LB_SLOT_SIZE         65536
#define MIPS_KSEG1  0xa0000000
#define LB_BASEADDR  (MIPS_KSEG1 + 0x1fe00000)

/* Compute the virtual address of the specified offset */
/* into the specified device slot */
void *
lamebus_map_area(struct lamebus_softc *bus, int slot,
                 u_int32_t offset)
{
    u_int32_t address;
    (void)bus;   // not needed

    assert(slot>=0 && slot<LB_NSLOTS);
    address = LB_BASEADDR + slot*LB_SLOT_SIZE + offset;
    return (void *)address;
}
```

Device Control Example: Commanding the Device

```
/* FROM: kern/arch/mips/mips/lamebus_mips.c */
/* Read 32-bit register from a LAMEbus device. */
u_int32_t
lamebus_read_register(struct lamebus_softc *bus,
    int slot, u_int32_t offset)
{
    u_int32_t *ptr = lamebus_map_area(bus, slot, offset);
    return *ptr;
}

/* Write a 32-bit register of a LAMEbus device. */
void
lamebus_write_register(struct lamebus_softc *bus,
    int slot, u_int32_t offset, u_int32_t val)
{
    u_int32_t *ptr = lamebus_map_area(bus, slot, offset);
    *ptr = val;
}
```

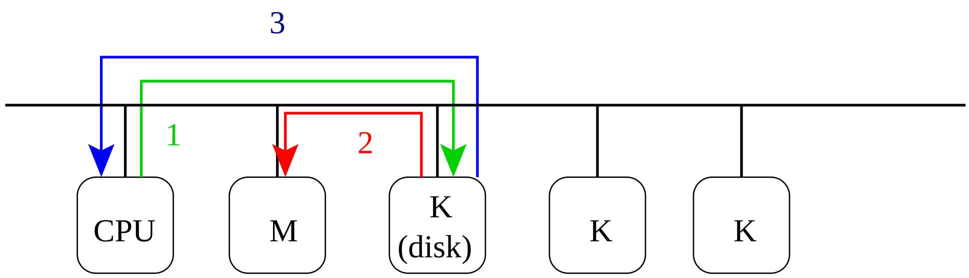

Device Data Transfer

- Sometimes, a device operation will involve a large chunk of data - much larger than can be moved with a single instruction. Example: reading a block of data from a disk.
- Devices may have data buffers for such data - but how to get the data between the device and memory?
- If the data buffer is memory-mapped, the kernel can move the data iteratively, one word at a time. This is called *program-controlled I/O*.
- Program controlled I/O is simple, but it means that the CPU is *busy executing kernel code* while the data is being transferred.
- The alternative is called Direct Memory Access (DMA). During a DMA data transfer, the CPU is *not busy* and is free to do something else, e.g., run an application.

Sys/161 LAMEbus devices do program-controlled I/O.

Direct Memory Access (DMA)

- DMA is used for block data transfers between devices (e.g., a disk controller) and memory
- Under DMA, the CPU initiates the data transfer and is notified when the transfer is finished. However, the device (not the CPU) controls the transfer itself.



1. CPU issues DMA request to controller
2. controller directs data transfer
3. controller interrupts CPU

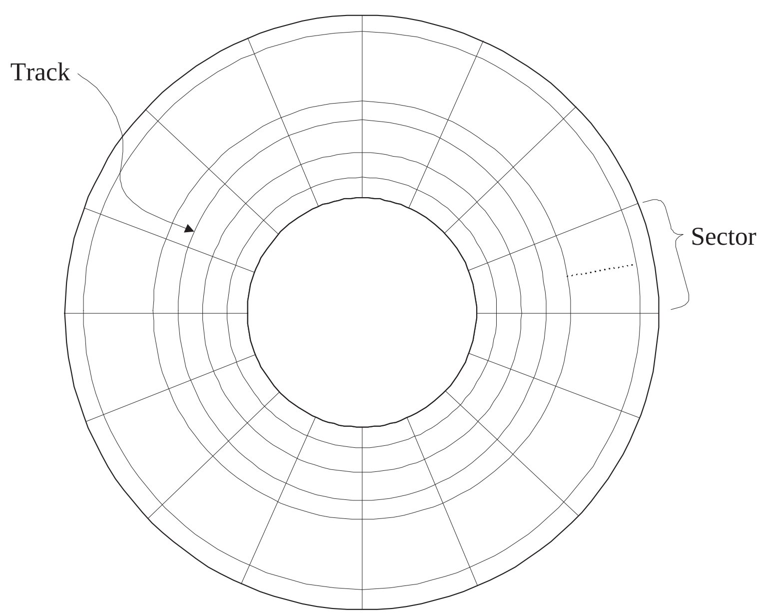
Applications and Devices

- interaction with devices is normally accomplished by device drivers in the OS, so that the OS can control how the devices are used
- applications see a simplified view of devices through a system call interface (e.g., block vs. character devices in Unix)
  - the OS may provide a system call interface that permits low level interaction between application programs and a device
- operating system often *buffers* data that is moving between devices and application programs' address spaces
  - benefits: solve timing, size mismatch problems
  - drawback: performance

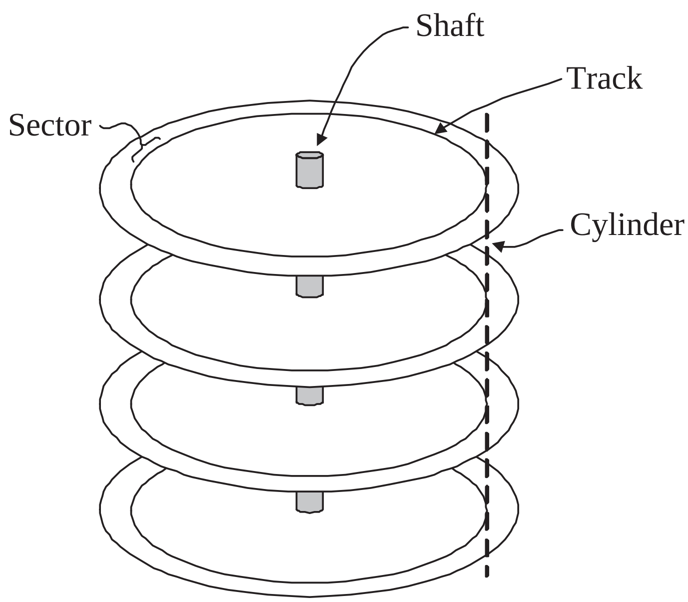
Logical View of a Disk Drive

- disk is an array of numbered blocks (or sectors)
- each block is the same size (e.g., 512 bytes)
- blocks are the unit of transfer between the disk and memory
  - typically, one or more contiguous blocks can be transferred in a single operation
- storage is *non-volatile*, i.e., data persists even when the device is without power

A Disk Platter's Surface



Physical Structure of a Disk Drive



Simplified Cost Model for Disk Block Transfer

- moving data to/from a disk involves:

  seek time: move the read/write heads to the appropriate cylinder

  rotational latency: wait until the desired sectors spin to the read/write heads

  transfer time: wait while the desired sectors spin past the read/write heads

- request service time is the sum of seek time, rotational latency, and transfer time

$$t_{service} = t_{seek} + t_{rot} + t_{transfer}$$

- note that there are other overheads but they are typically small relative to these three

Rotational Latency and Transfer Time

- rotational latency depends on the rotational speed of the disk
- if the disk spins at ω rotations per second:

$$0 \leq t_{rot} \leq \frac{1}{\omega}$$

- expected rotational latency:

$$\bar{t}_{rot} = \frac{1}{2\omega}$$

- transfer time depends on the rotational speed and on the amount of data transferred
- if k sectors are to be transferred and there are T sectors per track:

$$t_{transfer} = \frac{k}{T\omega}$$

Seek Time

- seek time depends on the speed of the arm on which the read/write heads are mounted.
- a simple linear seek time model:
  - $t_{maxseek}$ is the time required to move the read/write heads from the innermost cylinder to the outermost cylinder
  - C is the total number of cylinders
- if k is the required *seek distance* ($k > 0$):

$$t_{seek}(k) = \frac{k}{C} t_{maxseek}$$

Performance Implications of Disk Characteristics

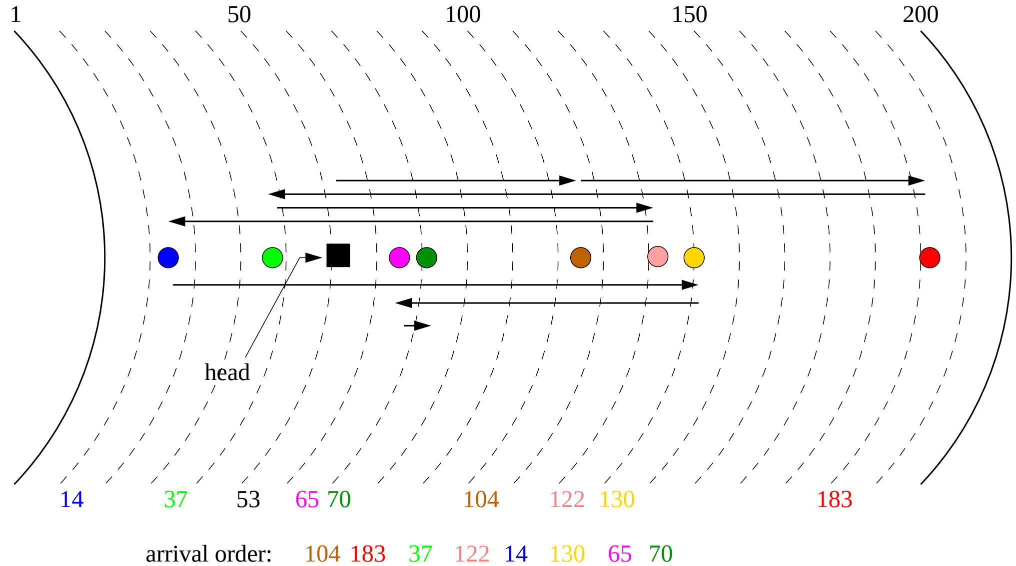
- larger transfers to/from a disk device are *more efficient* than smaller ones. That is, the cost (time) per byte is smaller for larger transfers. (Why?)
- sequential I/O is faster than non-sequential I/O
  - sequential I/O operations eliminate the need for (most) seeks
  - disks use other techniques, like *track buffering*, to reduce the cost of sequential I/O even more

Disk Head Scheduling

- goal: reduce seek times by controlling the order in which requests are serviced
- disk head scheduling may be performed by the controller, by the operating system, or both
- for disk head scheduling to be effective, there must be a queue of outstanding disk requests (otherwise there is nothing to reorder)
- an on-line approach is required: the disk request queue is not static

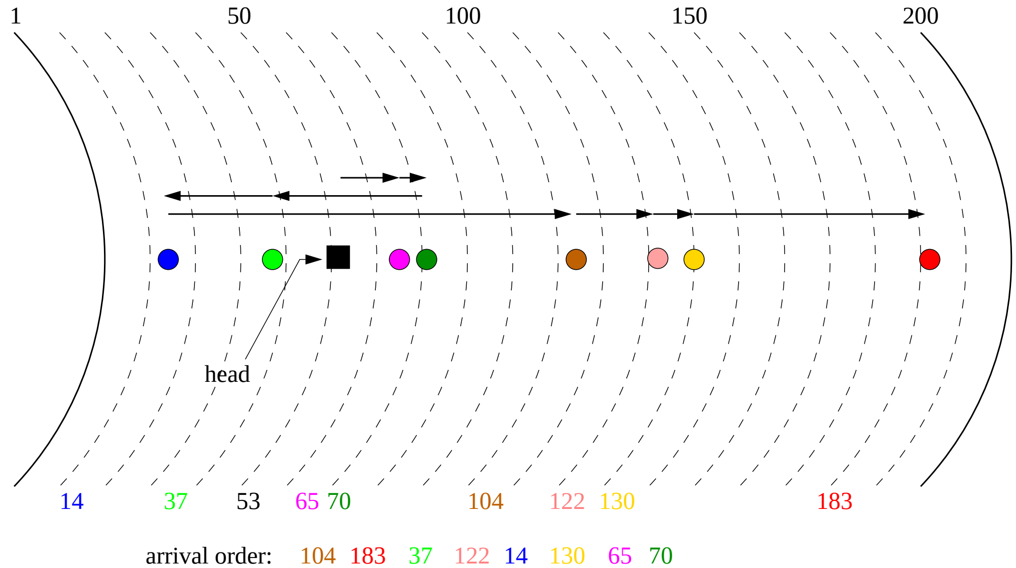
FCFS Disk Head Scheduling

- handle requests in the order in which they arrive
- fair and simple, but no optimization of seek times



Shortest Seek Time First (SSTF)

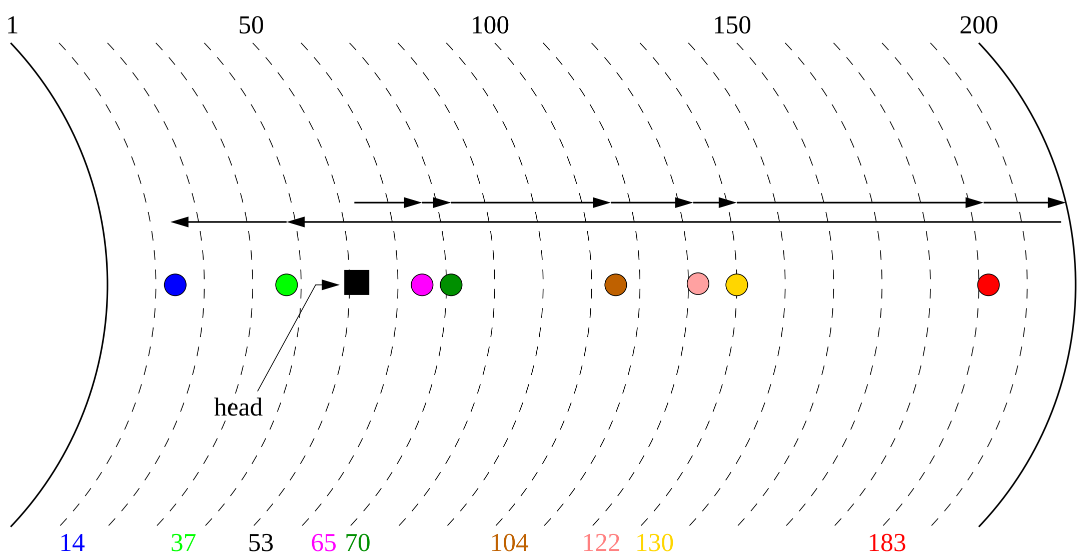
- choose closest request (a greedy approach)
- seek times are reduced, but requests may starve



SCAN and LOOK

- LOOK is the commonly-implemented variant of SCAN. Also known as the “elevator” algorithm.
- Under LOOK, the disk head moves in one direction until there are no more requests in front of it, then reverses direction.
- seek time reduction without starvation
- SCAN is like LOOK, except the read/write heads always move all the way to the edge of the disk in each direction.

SCAN Example

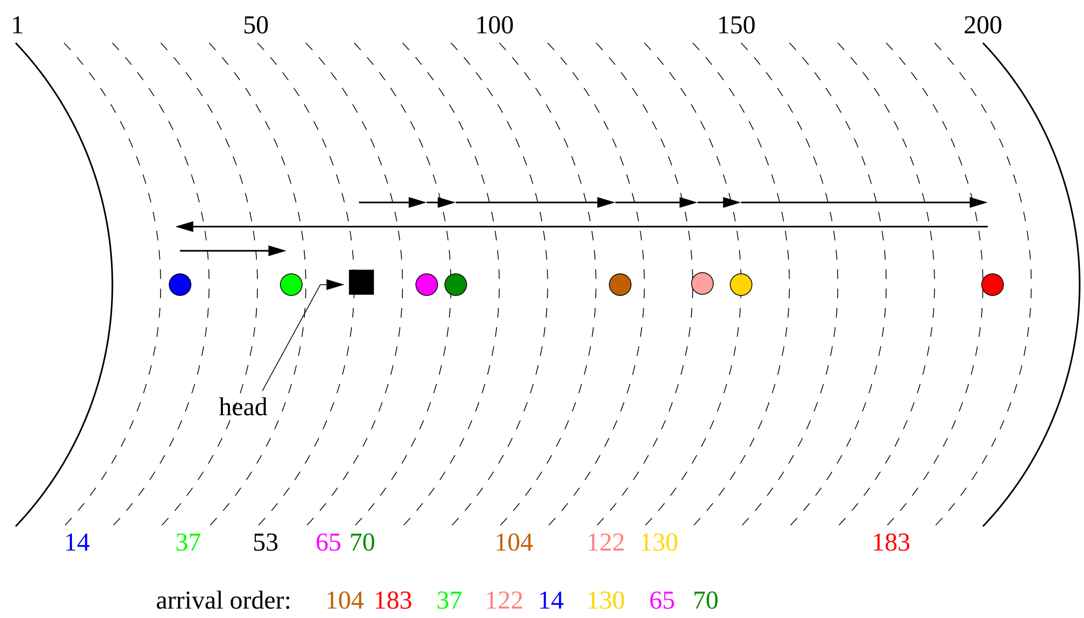


arrival order: 104 183 37 122 14 130 65 70

Circular SCAN (C-SCAN) and Circular LOOK (C-LOOK)

- C-LOOK and C-SCAN are variants of LOOK and SCAN
- Under C-LOOK, the disk head moves in one direction until there are no more requests in front of it, then it jumps back and begins another scan in the same direction as the first.
- C-LOOK avoids bias against "edge" cylinders

C-LOOK Example



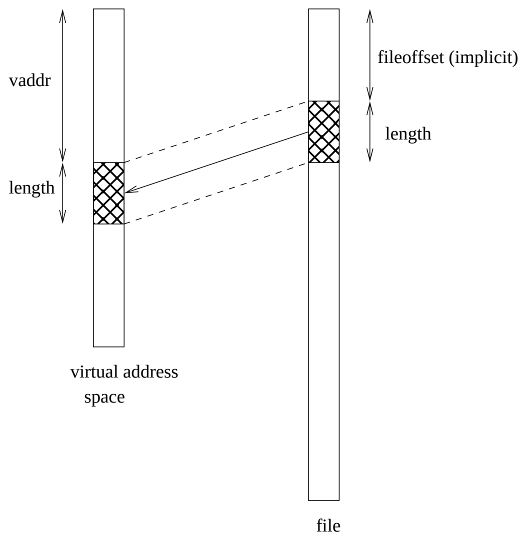
Files and File Systems

- files: persistent, named data objects
  - data consists of a sequence of numbered bytes
  - alternatively, a file may have some internal structure, e.g., a file may consist of sequence of numbered records
  - file may change size over time
  - file has associated meta-data (attributes), in addition to the file name
    - examples: owner, access controls, file type, creation and access timestamps
- file system: a collection of files which share a common name space
  - allows files to be created, destroyed, renamed, . . .

File Interface

- open, close
  - open returns a file identifier (or handle or descriptor), which is used in subsequent operations to identify the file. (Why is this done?)
- read, write
  - must specify which file to read, which part of the file to read, and where to put the data that has been read (similar for write).
  - often, file position is implicit (why?)
- seek
- get/set file attributes, e.g., Unix `fstat, chmod`

File Read




```
read(fileID, vaddr, length)
```


File Position

- may be associated with the file, with a process, or with a file descriptor (Unix style)
- read and write operations
  - start from the current file position
  - update the current file position
- this makes sequential file I/O easy for an application to request
- for non-sequential (random) file I/O, use:
  - `seek`, to adjust file position before reading or writing
  - a positioned read or write operation, e.g., Unix `pread, pwrite`: `pread(fileId,vaddr,length,filePosition)`

Sequential File Reading Example (Unix)

```
char buf[512];
int i;
int f = open("myfile",O_RDONLY);
for(i=0; i<100; i++) {
  read(f,(void *)buf,512);
}
close(f);
```

Read the first $100 * 512$ bytes of a file, 512 bytes at a time.

File Reading Example Using Seek (Unix)

```
char buf[512];
int i;
int f = open("myfile",O_RDONLY);
for(i=1; i<=100; i++) {
  lseek(f,(100-i)*512,SEEK_SET);
  read(f,(void *)buf,512);
}
close(f);
```

Read the first $100 * 512$ bytes of a file, 512 bytes at a time, in reverse order.

File Reading Example Using Positioned Read

```
char buf[512];
int i;
int f = open("myfile",O_RDONLY);
for(i=0; i<100; i+=2) {
  pread(f,(void *)buf,512,i*512);
}
close(f);
```

Read every second 512 byte chunk of a file, until 50 have been read.

Memory-Mapped Files

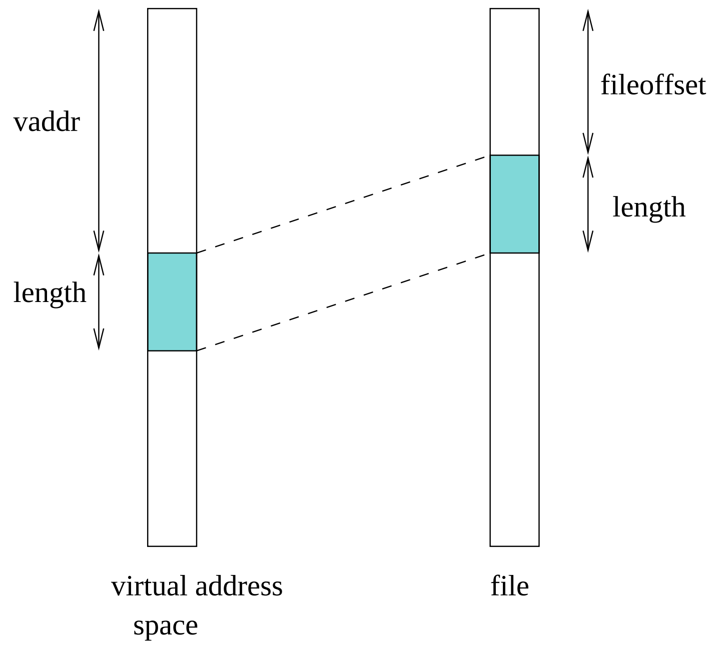
- generic interface:

  `vaddr ← mmap(file descriptor,fileoffset,length)`
  `munmap(vaddr,length)`

- `mmap` call returns the virtual address to which the file is mapped
- `munmap` call unmaps mapped files within the specified virtual address range

Memory-mapping is an alternative to the read/write file interface.

Memory Mapping Illustration



Memory Mapping Update Semantics

- what should happen if the virtual memory to which a file has been mapped is updated?
- some options:
  - prohibit updates (read-only mapping)
  - eager propagation of the update to the file (too slow!)
  - lazy propagation of the update to the file
    - user may be able to request propagation (e.g., Posix `msync()`
    - propagation may be guaranteed by `munmap()`
  - allow updates, but do not propagate them to the file

Memory Mapping Concurrency Semantics

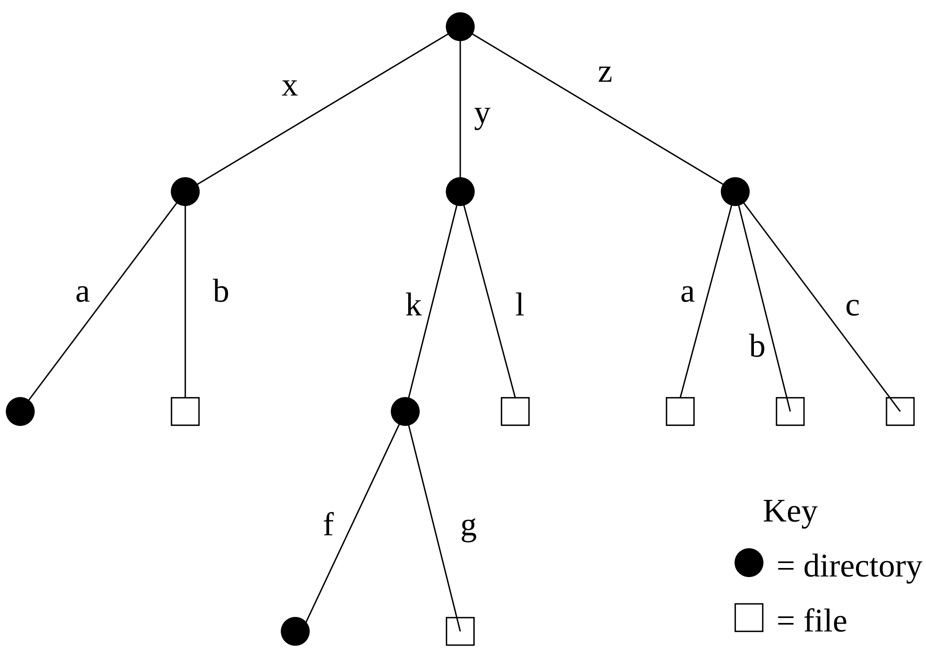
- what should happen if a memory mapped file is updated?
  - by a process that has mmapped the same file
  - by a process that is updating the file using a `write()` system call
- options are similar to those on the previous slide. Typically:
  - propagate lazily: processes that have mapped the file *may* eventually see the changes
  - propagate eagerly: other processes will see the changes
    - typically implemented by invalidating other process's page table entries

File Names

- application-visible objects (e.g., files, directories) are given names
- the file system is responsible for associating names with objects
- the namespace is typically structured, often as a tree or a DAG
- namespace structure provides a way for users and applications to organize and manage information
- in a structured namespace, objects may be identified by *pathnames*, which describe a path from a root object to the object being identified, e.g.:

  `/home/kmsalem/courses/cs350/notes/filesys.ps`

Hierarchical Namespace Example

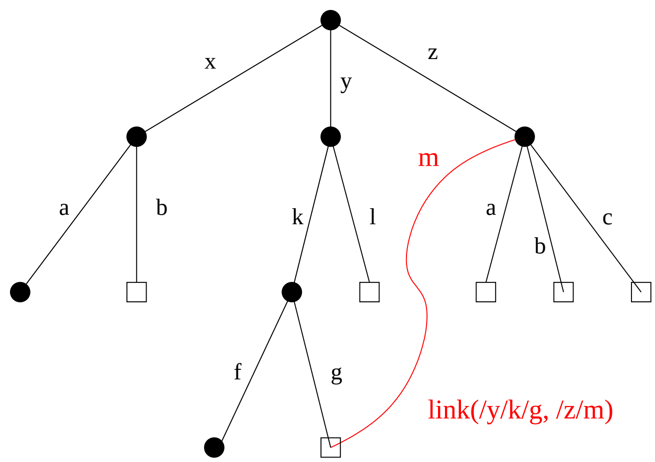


Hard Links

- a *hard link* is an association between a name and an underlying file (or directory)
- typically, when a file is created, a single link is created to the file as well (else the file would be difficult to use!)
  - POSIX example: `creat(pathname,mode)` creates both a new empty file object and a link to that object (using `pathname`)
- some file systems allow additional hard links to be made to existing files. This allows more than one name from the file system's namespace to refer the *same underlying object*.
  - POSIX example: `link(oldpath,newpath)` creates a new hard link, using `newpath`, to the underlying object identified by `oldpath`

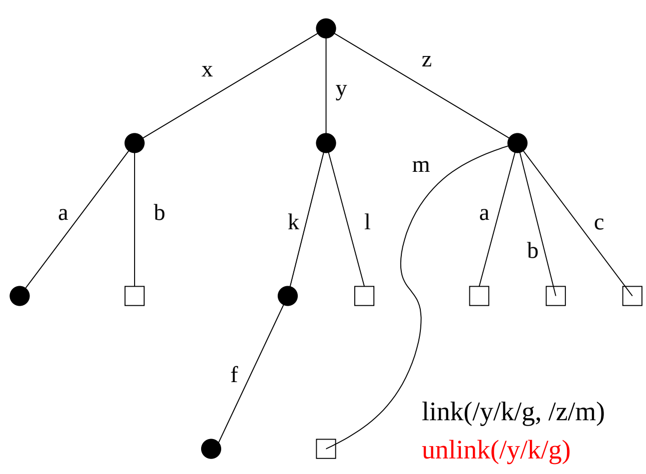
File systems ensure *referential integrity* for hard links. A hard link refers to the object it was created for until the link is explicitly destroyed. (What are the implications of this?)

Hard Link Illustration



Hard links are a way to create *non-hierarchical structure* in the namespace. Hard link creation may be restricted to restrict the kinds of structure that applications can create. Example: no hard links to directories.

Unlink Example

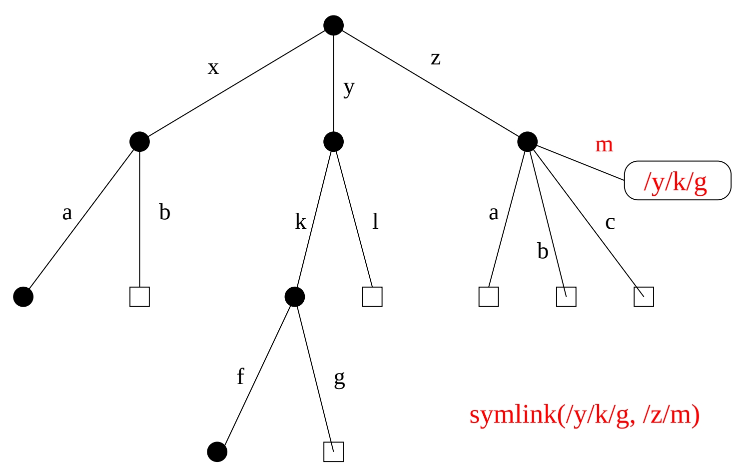


Removing the *last* link to a file causes the file itself to be deleted. Deleting a file that has a link would destroy the referential integrity of the link.

Symbolic Links

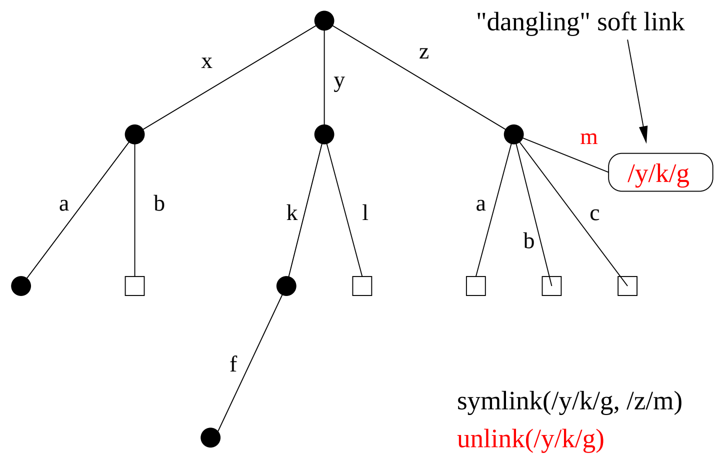
- a *Symbolic link*, or *soft link*, is an association between two names in the file namespace. Think of it is a way of defining a synonym for a filename.
  - `symlink(oldpath,newpath)` creates a symbolic link from `newpath` to `oldpath`, i.e., `newpath` becomes a synonym for `oldpath`.
- symbolic links relate filenames to filenames, while hard links relate filenames to underlying file objects!
- referential integrity is *not* preserved for symbolic links, e.g., the system call above can succeed even if there is no object named `oldpath`

Soft Link Example



`/y/k/g` still has only one hard link after the `symlink` call. A new symlink object records the association between `/z/m` and `/y/k/g`. `open(/z/m)` will now have the same effect as `open(/y/k/g)`.

Soft Link Example with Unlink



A file is deleted by this `unlink` call. An attempt to `open(/z/m)` after the `unlink` will result in an error. If a *new* file called `/y/k/g` is created, a subsequent `open(/z/m)` will open the new file.

Linux Link Example (1 of 2)

```
% cat > file1
This is file1.
% ls -li
685844 -rw------- 1 kmsalem kmsalem 15 2008-08-20 file1
% ln file1 link1
% ln -s file1 sym1
% ls -li
685844 -rw------- 2 kmsalem kmsalem 15 2008-08-20 file1
685844 -rw------- 2 kmsalem kmsalem 15 2008-08-20 link1
685845 lrwxrwxrwx 1 kmsalem kmsalem  5 2008-08-20 sym1 -> file1
% cat file1
This is file1.
% cat link1
This is file1.
% cat sym1
This is file1.
```

A file, a hard link, a soft link.

Linux Link Example (2 of 2)

```
% /bin/rm file1
% ls -li
685844 -rw------- 1 kmsalem kmsalem 15 2008-08-20 link1
685845 lrwxrwxrwx 1 kmsalem kmsalem  5 2008-08-20 sym1 -> file1
% cat link1
This is file1.
% cat sym1
cat: sym1: No such file or directory
% cat > file1
This is a brand new file1.
% ls -li
685846 -rw------- 1 kmsalem kmsalem 27 2008-08-20 file1
685844 -rw------- 1 kmsalem kmsalem 15 2008-08-20 link1
685845 lrwxrwxrwx 1 kmsalem kmsalem  5 2008-08-20 sym1 -> file1
% cat link1
This is file1.
% cat sym1
This is a brand new file1.
```

Different behaviour for hard links and soft links.

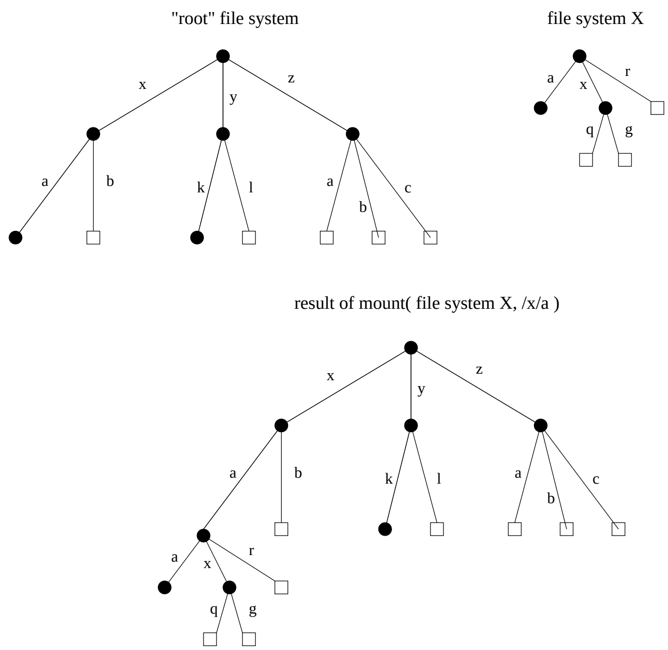
Multiple File Systems

- it is not uncommon for a system to have multiple file systems
- some kind of global file namespace is required
- two examples:

  DOS/Windows: use two-part file names: file system name, pathname
  - example: `C:\kmsalem\cs350\schedule.txt`

  Unix: merge file graphs into a single graph
  - Unix `mount` system call does this

Unix `mount` Example



Links and Multiple File Systems

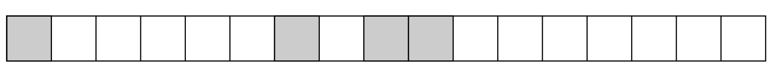
- a hard link associates a name in the file system namespace with a file in that file system
- typically, hard links cannot cross file system boundaries
- for example, even after the mount operation illustrated on the previous slide, `link(/x/a/x/g,/z/d)` would result in an error, because the new link, which is in the root file system refers to an object in file system X
- soft links do not have this limitation
- for example, after the mount operation illustrated on the previous slide:
  - `symlink(/x/a/x/g,/z/d)` would succeed
  - `open(/z/d)` would succeed, with the effect of opening `/z/a/x/g`.
- even if the `symlink` operation were to occur *before* the `mount` command, it would succeed

File System Implementation

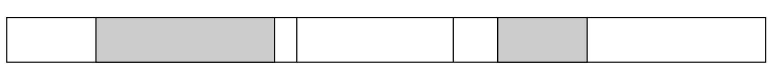
- space management
- file indexing (how to locate file data and meta-data)
- directories
- links
- buffering, in-memory data structures
- persistence

Space Allocation and Layout

- space may be allocated in fixed-size chunks, or in chunks of varying size
- fixed-size chunks: simple space management, but internal fragmentation
- variable-size chunks: external fragmentation



fixed−size allocation



variable−size allocation

- *layout* matters! Try to lay a file out sequentially, or in large sequential extents that can be read and written efficiently.

File Indexing

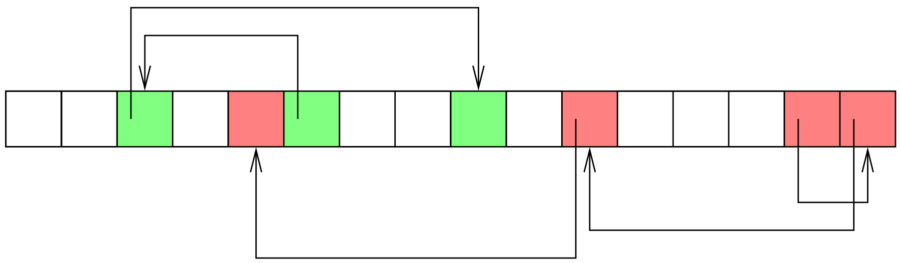
- in general, a file will require more than one chunk of allocated space
- this is especially true because files can grow
- how to find all of a file's data?

  chaining:
  - each chunk includes a pointer to the next chunk
  - OK for sequential access, poor for random access

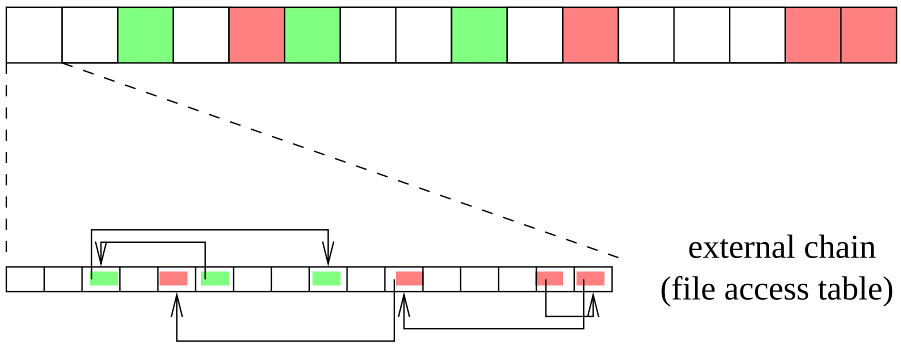
  external chaining: DOS file allocation table (FAT), for example
  - like chaining, but the chain is kept in an external structure

  per-file index: Unix i-node, for example
  - for each file, maintain a table of pointers to the file's blocks or extents

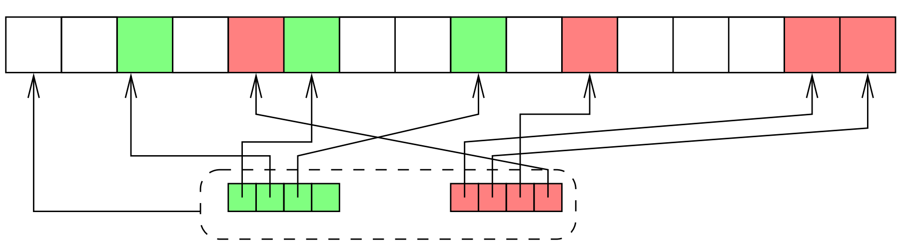
Chaining



External Chaining (File Access Table)



Per-File Indexing



Internal File Identifiers

- typically, a file system will assign a unique internal identifier to each file, directory or other object
- given an identifer, the file system can *directly* locate a record containing key information about the file, such as:
  - the per-file index to the file data (if per-file indexing is used), or the location of the file's first data block (if chaining is used)
  - file meta-data (or a reference to the meta-data), such as
    - file owner
    - file access permissions
    - file acesss timestamps
    - file type
- for example, a file identifier might be a number which indexes an on-disk array of file records

Example: Unix i-nodes

- an i-node is a record describing a file
- each i-node is uniquely identified by an i-number, which determines its physical location on the disk
- an i-node is a *fixed size* record containing:

file attribute values

- file type
- file owner and group
- access controls
- creation, reference and update timestamps
- file size

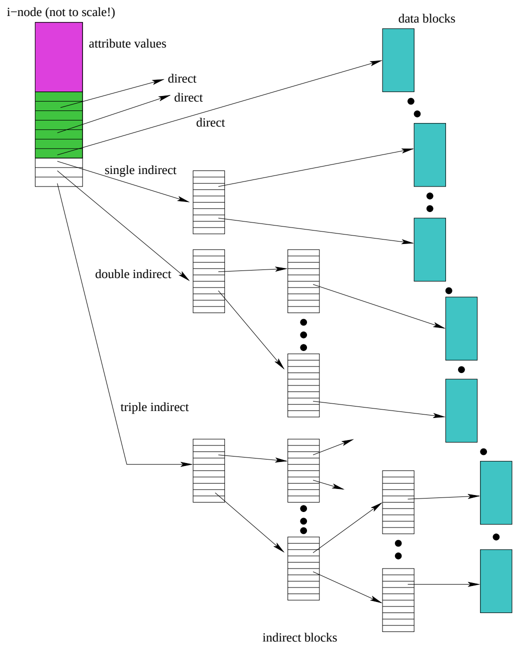
direct block pointers: approximately 10 of these

single indirect block pointer

double indirect block pointer

triple indirect block pointer

i-node Diagram



Directories

- A directory consists of a set of entries, where each entry is a record that includes:
  - a file name (component of a path name)
  - the internal file identifier (e.g., i-number) of the file
- A directory can be implemented as a special type of file. The directory entries are the contents of the file.
- The file system should not allow directory files to be directly written by application programs. Instead, the directory is updated by the file system as files are created and destroyed

Implementing Hard Links

- hard links are simply directory entries
- for example, consider:

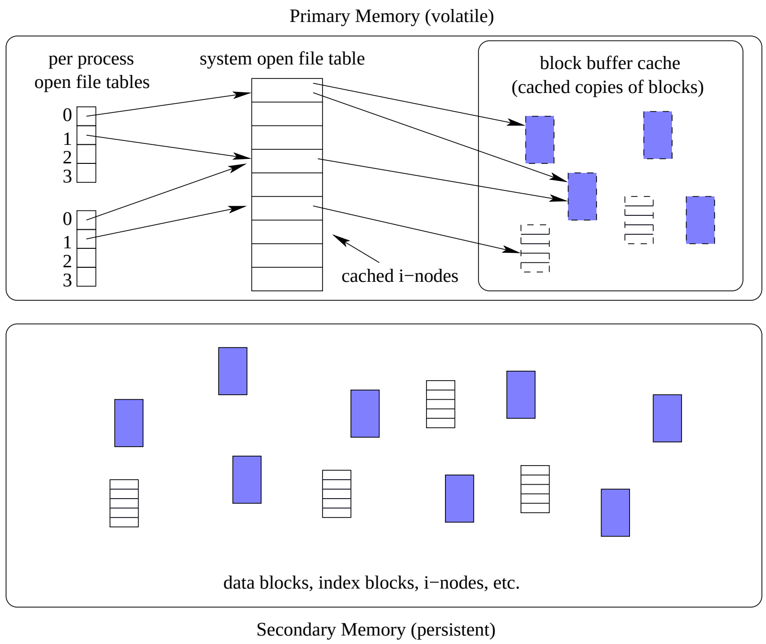
  `link(/y/k/g,/z/m)`
- to implement this:
  1. find out the internal file identifier for `/y/k/g`
  2. create a new entry in directory `/z`
     - file name in new entry is `m`
     - file identifier (i-number) in the new entry is the one discovered in step 1

Implementing Soft Links

- soft links can be implemented as a special type of file
- for example, consider:

  `symlink(/y/k/g,/z/m)`
- to implement this:
  - create a new *symlink* file
  - add a new entry in directory `/z`
    * file name in new entry is `m`
    * i-number in the new entry is the i-number of the new symlink file
  - store the pathname string “/y/k/g” as the contents of the new symlink file
- change the behaviour of the `open` system call so that when the symlink file is encountered during `open(/z/m)`, the file `/y/k/g` will be opened instead.

Main Memory Data Structures



Problems Caused by Failures

- a single logical file system operation may require several disk I/O operations
- example: deleting a file
  - remove entry from directory
  - remove file index (i-node) from i-node table
  - mark file’s data blocks free in free space index
- what if, because of a failure, some but not all of these changes are reflected on the disk?

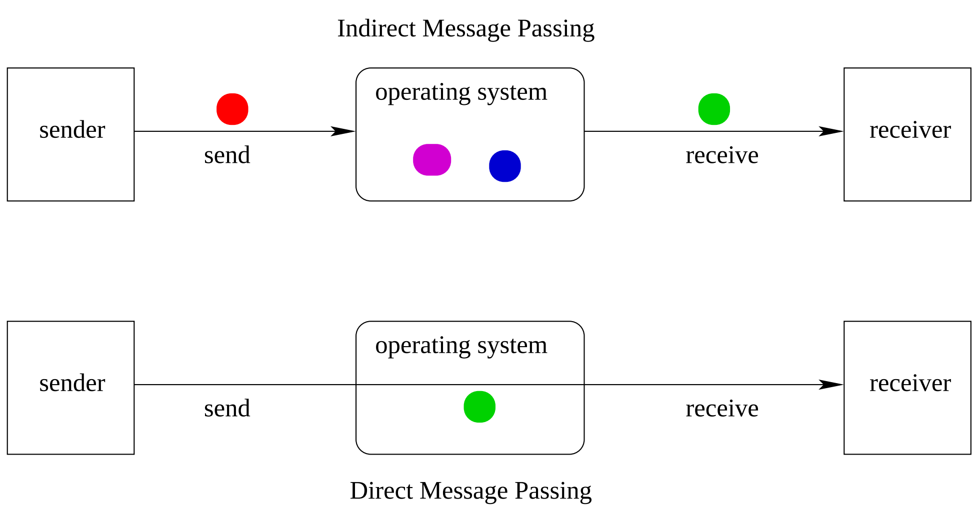
Fault Tolerance

- special-purpose consistency checkers (e.g., Unix `fsck` in Berkeley FFS, Linux ext2)
  - runs after a crash, before normal operations resume
  - find and attempt to repair inconsistent file system data structures, e.g.:
    * file with no directory entry
    * free space that is not marked as free
- journaling (e.g., Veritas, NTFS, Linux ext3)
  - record file system meta-data changes in a journal (log), so that sequences of changes can be written to disk in a single operation
  - *after* changes have been journaled, update the disk data structures (*write-ahead logging*)
  - after a failure, redo journaled updates in case they were not done before the failure

Interprocess Communication Mechanisms

- shared storage
  - These mechanisms have already been covered. examples:
    * shared virtual memory
    * shared files
  - processes must agree on a name (e.g., a file name, or a shared virtual memory key) in order to establish communication
- message based
  - signals
  - sockets
  - pipes
  - ...

Message Passing



If message passing is indirect, the message passing system must have some capacity to buffer (store) messages.

Properties of Message Passing Mechanisms

Addressing: how to identify where a message should go

Directionality:

- simplex (one-way)
- duplex (two-way)
- half-duplex (two-way, but only one way at a time)

Message Boundaries:

datagram model: message boundaries

stream model: no boundaries

Properties of Message Passing Mechanisms (cont'd)

Connections: need to connect before communicating?

- in connection-oriented models, recipient is specified at time of connection, not by individual send operations. All messages sent over a connection have the same recipient.
- in connectionless models, recipient is specified as a parameter to each send operation.

Reliability:

- can messages get lost?
- can messages get reordered?
- can messages get damaged?

Sockets

- a socket is a communication *end-point*
- if two processes are to communicate, each process must create its own socket
- two common types of sockets

  stream sockets: support connection-oriented, reliable, duplex communication under the stream model (no message boundaries)

  datagram sockets: support connectionless, best-effort (unreliable), duplex communication under the datagram model (message boundaries)
- both types of sockets also support a variety of address domains, e.g.,

  Unix domain: useful for communication between processes running on the same machine

  INET domain: useful for communication between process running on different machines that can communicate using IP protocols.

Using Datagram Sockets (Receiver)

```
s = socket(addressType, SOCK_DGRAM);
bind(s,address);
recvfrom(s,buf,bufLength,sourceAddress);
...
close(s);
```

- `socket` creates a socket
- `bind` assigns an address to the socket
- `recvfrom` receives a message from the socket
  - `buf` is a buffer to hold the incoming message
  - `sourceAddress` is a buffer to hold the address of the message sender
- both `buf` and `sourceAddress` are filled by the `recvfrom` call

Using Datagram Sockets (Sender)

```
s = socket(addressType, SOCK_DGRAM);
sendto(s,buf,msgLength,targetAddress)
...
close(s);
```

- `socket` creates a socket
- `sendto` sends a message using the socket
  - `buf` is a buffer that contains the message to be sent
  - `msgLength` indicates the length of the message in the buffer
  - `targetAddress` is the address of the socket to which the message is to be delivered

More on Datagram Sockets

- `sendto` and `recvfrom` calls *may* block
  - `recvfrom` blocks if there are no messages to be received from the specified socket
  - `sendto` blocks if the system has no more room to buffer undelivered messages
- datagram socket communications are (in general) unreliable
  - messages (datagrams) may be lost
  - messages may be reordered
- The sending process must know the address of the receive process's socket. How does it know this?

A Socket Address Convention

```
Service          Port       Description
-------------------------------------------------------
echo             7/udp
systat           11/tcp
netstat          15/tcp
chargen          19/udp
ftp              21/tcp
ssh              22/tcp     # SSH Remote Login Protocol
telnet           23/tcp
smtp             25/tcp
time             37/udp
gopher           70/tcp     # Internet Gopher
finger           79/tcp
www              80/tcp     # WorldWideWeb HTTP
pop2             109/tcp    # POP version 2
imap2            143/tcp    # IMAP
```


Using Stream Sockets (Passive Process)

```
s = socket(addressType, SOCK_STREAM);
bind(s,address);
listen(s,backlog);
ns = accept(s,sourceAddress);
recv(ns,buf,bufLength);
send(ns,buf,bufLength);
...
close(ns); // close accepted connection
close(s);  // don't accept more connections
```

- `listen` specifies the number of connection requests for this socket that will be queued by the kernel
- `accept` accepts a connection request and creates a new socket (`ns`)
- `recv` receives up to `bufLength` bytes of data from the connection
- `send` sends `bufLength` bytes of data over the connection.

Notes on Using Stream Sockets (Passive Process)

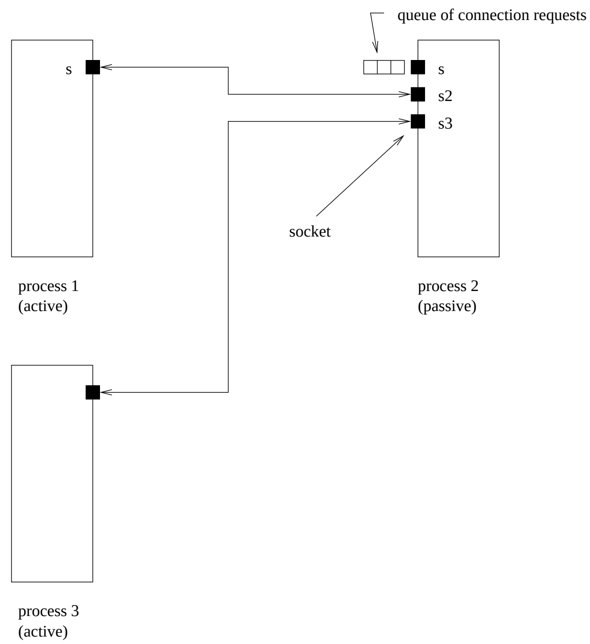
- `accept` creates a new socket (`ns`) for the new connection
- `sourceAddress` is an address buffer. `accept` fills it with the address of the socket that has made the connection request
- additional connection requests can be accepted using more `accept` calls on the original socket (`s`)
- `accept` blocks if there are no pending connection requests
- connection is duplex (both `send` and `recv` can be used)

Using Stream Sockets (Active Process)

```
s = socket(addressType, SOCK_STREAM);
connect(s,targetAddress);
send(s,buf,bufLength);
recv(s,buf,bufLength);
...
close(s);
```

- `connect` sends a connection request to the socket with the specified address
  - `connect` blocks until the connection request has been accepted
- active process may (optionally) bind an address to the socket (using `bind`) before connecting. This is the address that will be returned by the `accept` call in the passive process
- if the active process does not choose an address, the system will choose one

Illustration of Stream Socket Connections



Pipes

- pipes are communication objects (not end-points)
- pipes use the stream model and are connection-oriented and reliable
- some pipes are simplex, some are duplex
- pipes use an implicit addressing mechanism that limits their use to communication between *related* processes, typically a child process and its parent
- a `pipe()` system call creates a pipe and returns two descriptors, one for each end of the pipe
  - for a simplex pipe, one descriptor is for reading, the other is for writing
  - for a duplex pipe, both descriptors can be used for reading and writing

One-way Child/Parent Communication Using a Simplex Pipe

```
int fd[2];
char m[] = "message for parent";
char y[100];
pipe(fd); // create pipe
pid = fork(); // create child process
if (pid == 0) {
  // child executes this
  close(fd[0]); // close read end of pipe
  write(fd[1],m,19);
  ...
} else {
  // parent executes this
  close(fd[1]); // close write end of pipe
  read(fd[0],y,100);
  ...
}
```


Illustration of Example (after `pipe()`)

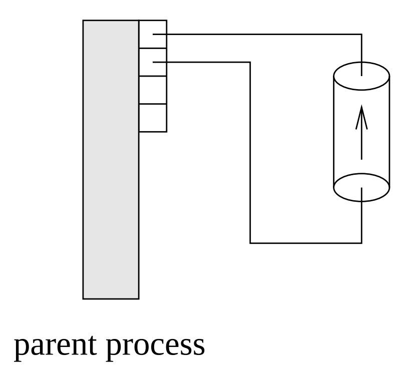


Illustration of Example (after `fork()`)

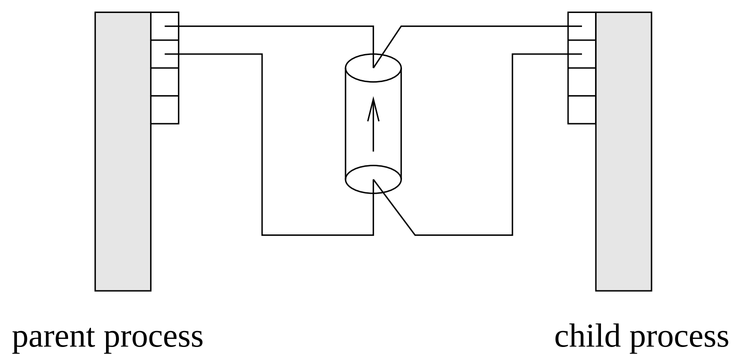
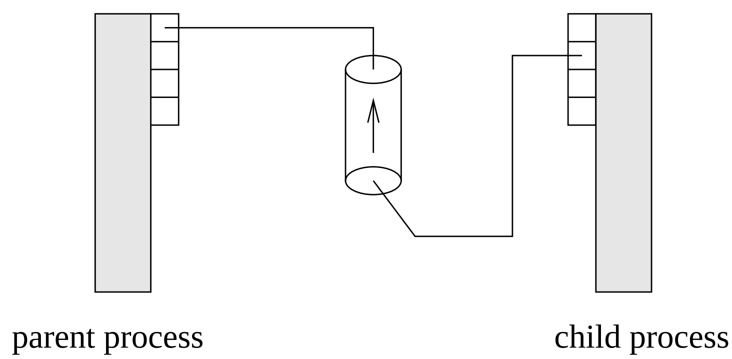


Illustration of Example (after `close()`)



Examples of Other Interprocess Communication Mechanisms

named pipe:

- similar to pipes, but with an associated name (usually a file name)
- name allows arbitrary processes to communicate by opening the same named pipe
- must be explicitly deleted, unlike an unnamed pipe

message queue:

- like a named pipe, except that there are message boundaries
- `msgsend` call sends a message into the queue, `msgrecv` call receives the next message from the queue

Signals

- signals permit asynchronous one-way communication
  - from a process to another process, or to a group of processes, via the kernel
  - from the kernel to a process, or to a group of processes
- there are many types of signals
- the arrival of a signal may cause the execution of a *signal handler* in the receiving process
- there may be a different handler for each type of signal

Examples of Signal Types

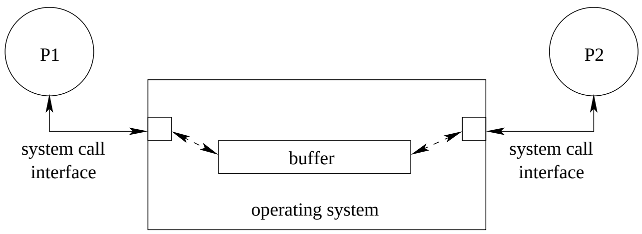
Signal	Value	Action	Comment
SIGINT	2	Term	Interrupt from keyboard
SIGILL	4	Core	Illegal Instruction
SIGKILL	9	Term	Kill signal
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGBUS	10,7,10	Core	Bus error
SIGXCPU	24,24,30	Core	CPU time limit exceeded
SIGSTOP	17,19,23	Stop	Stop process

Signal Handling

- operating system determines default signal handling for each new process
- example default actions:
  - ignore (do nothing)
  - kill (terminate the process)
  - stop (block the process)
- a running process can change the default for some types of signals
- signal-related system calls
  - calls to set non-default signal handlers, e.g., Unix `signal`, `sigaction`
  - calls to send signals, e.g., Unix `kill`

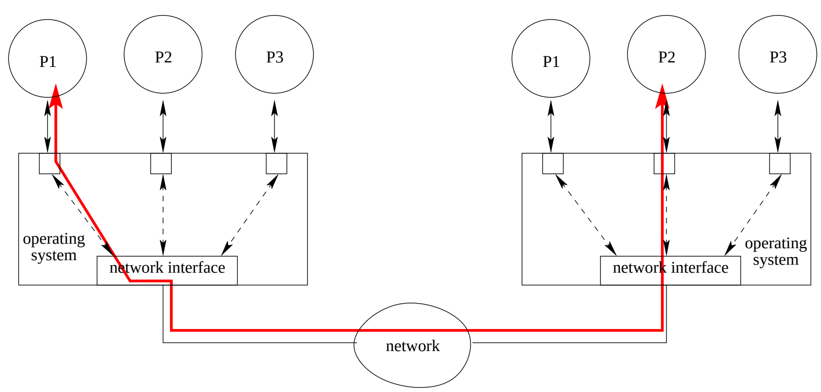
Implementing IPC

- application processes use descriptors (identifiers) provided by the kernel to refer to specific sockets and pipes, as well as files and other objects
- kernel *descriptor tables* (or other similar mechanism) are used to associate descriptors with kernel data structures that implement IPC objects
- kernel provides bounded buffer space for data that has been sent using an IPC mechanism, but that has not yet been received
  - for IPC objects, like pipes, buffering is usually on a per object basis
  - IPC end points, like sockets, buffering is associated with each endpoint



Network Interprocess Communication

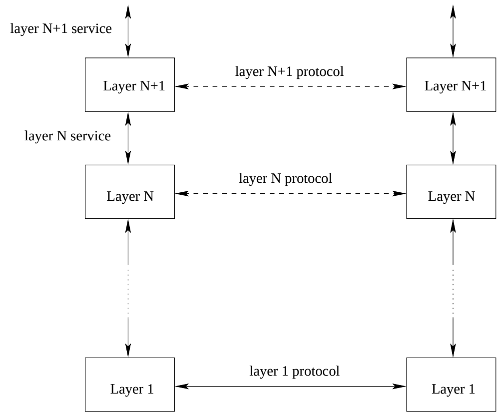
- some sockets can be used to connect processes that are running on different machines
- the kernel:
  - controls access to network interfaces
  - multiplexes socket connections across the network



Networking Reference Models

- ISO/OSI Reference Model

7	Application Layer
6	Presentation Layer
5	Session Layer
4	Transport Layer
3	Network Layer
2	Data Link Layer
1	Physical Layer



- Internet Model
  - layers 1-4 and 7

Internet Protocol (IP): Layer 3

- every machine has one (or more) IP address, in addition to its data link layer address(es)
- In IPv4, addresses are 32 bits, and are commonly written using “dot” notation, e.g.:
  - cpu06.student.cs = 129.97.152.106
  - www.google.ca = 216.239.37.99 or 216.239.51.104 or ...
- IP moves packets (datagrams) from one machine to another machine
- principal function of IP is *routing*: determining the network path that a packet should take to reach its destination
- IP packet delivery is “best effort” (unreliable)

IP Routing Table Example

- Routing table for zonker.uwaterloo.ca, which is on three networks, and has IP addresses 129.97.74.66, 172.16.162.1, and 192.168.148.1 (one per network):

Destination	Gateway	Interface
172.16.162.*	-	vmnet1
129.97.74.*	-	eth0
192.168.148.*	-	vmnet8
default	129.97.74.1	eth0

- routing table key:

destination: ultimate destination of packet

gateway: next hop towards destination (or "-" if destination is directly reachable)

interface: which network interface to use to send this packet

Internet Transport Protocols

TCP: transport control protocol

- connection-oriented
- reliable
- stream
- congestion control
- used to implement INET domain stream sockets

UDP: user datagram protocol

- connectionless
- unreliable
- datagram
- no congestion control
- used to implement INET domain datagram sockets

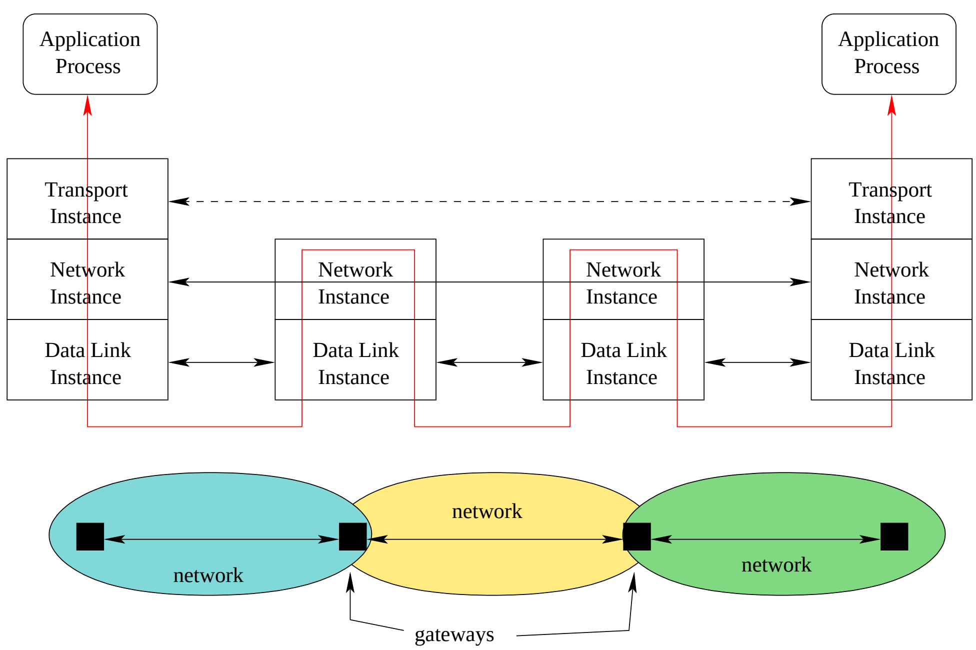
TCP and UDP Ports

- since there can be many TCP or UDP communications end points (sockets) on a single machine, there must be a way to distinguish among them
- each TCP or UDP address can be thought of as having two parts:

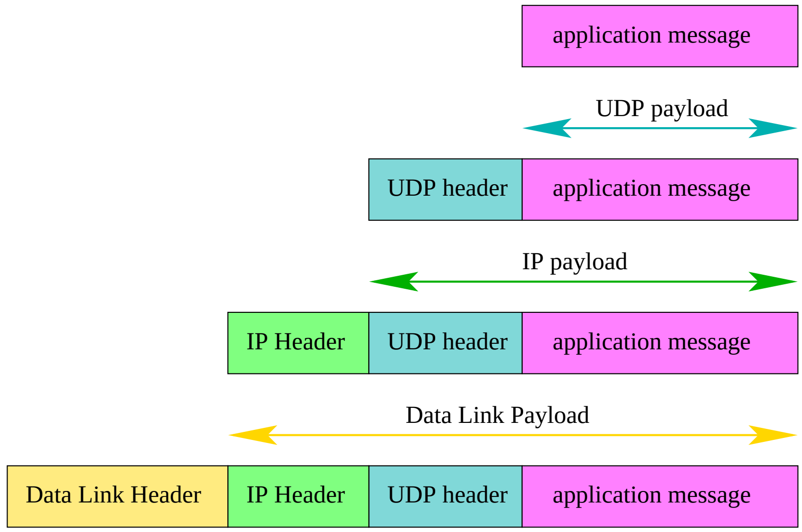
  (machine name, port number)

- The machine name is the IP address of a machine, and the port number serves to distinguish among the end points on that machine.
- INET domain socket addresses are TCP or UDP addresses (depending on whether the socket is a stream socket or a datagram socket).

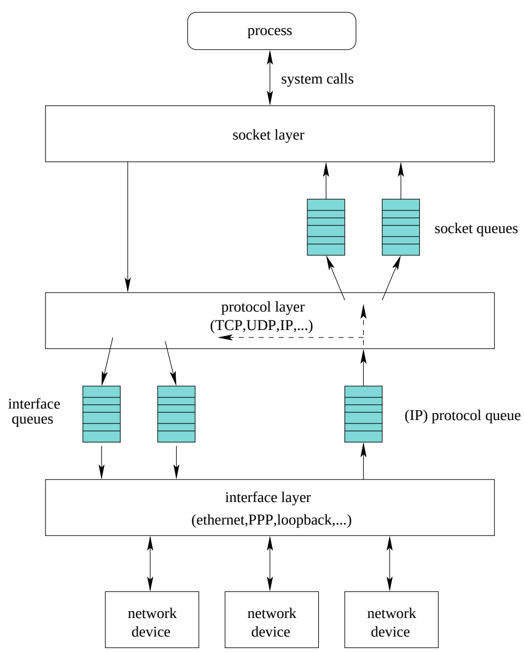
Example of Network Layers



Network Packets (UDP Example)



BSD Unix Networking Layers



Additional Notes:

Additional Notes:

Additional Notes:

Additional Notes: