CS 350
Operating Systems
Course Notes

Fall 2015

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

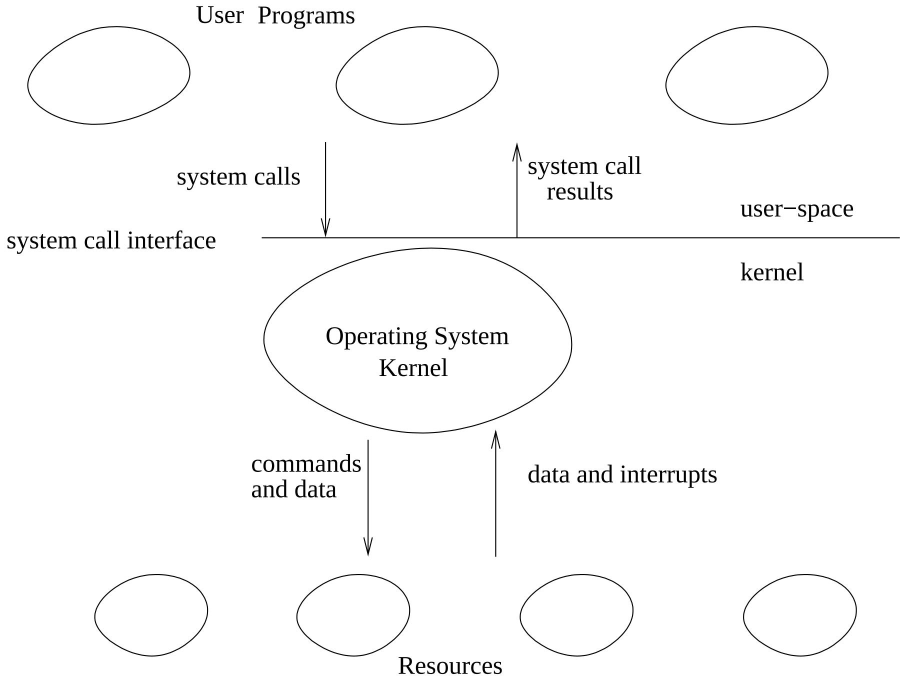
- Some terminology:

  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

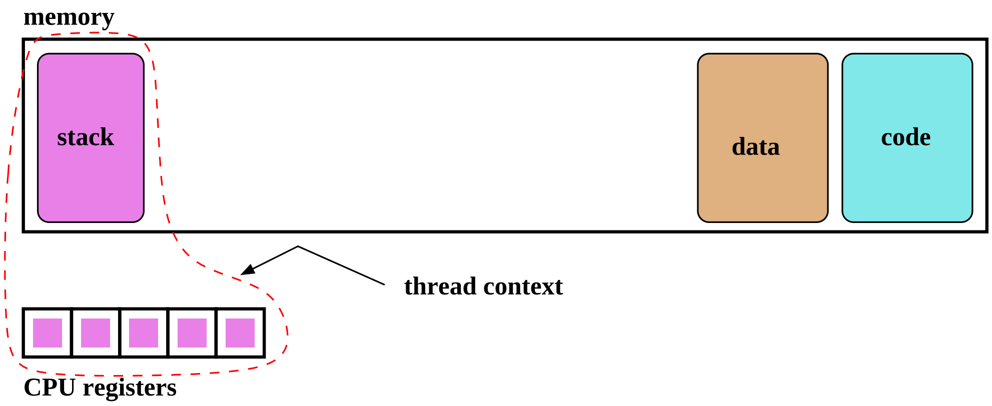
- Registers
  - program counter, stack pointer, . . .
- Memory
  - program code
  - program data
  - program stack containing procedure activation records
- CPU
  - fetches and executes instructions

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.

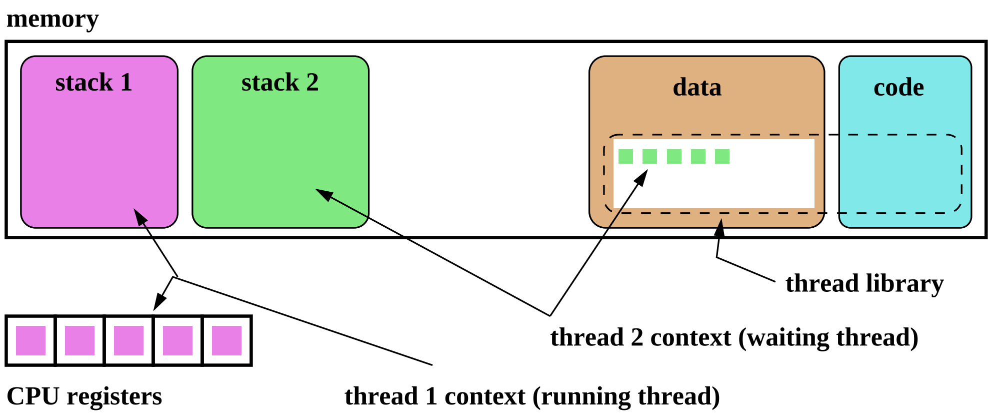
Thread Context



Concurrent Threads

- more than one thread may exist simultaneously (why might this be a good idea?)
- each thread has its own context, though they 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

Two Threads, One Running


memory
stack 1
stack 2
data
code
thread library
thread 2 context (waiting thread)
CPU registers
thread 1 context (running thread)

Thread Interface (Partial), With OS/161 Examples

- a *thread library* implements threads
- thread library provides a thread interface, used by program code to manipulate threads
- common thread interface functions include
  - create new thread

    ```
    int thread_fork(const char *name, struct proc *proc,
                    void (*func)(void *, unsigned long),
                    void *data1, unsigned long data2);
    ```
  - end (and destroy) the current thread

    ```
    void thread_exit(void);
    ```
  - cause current thread to *yield* (to be discussed later)

    ```
    void thread_yield(void);
    ```
- see `kern/include/thread.h`

Example: Creating Threads Using `thread_fork()`

```
for (index = 0; index < NumMice; index++) {
  error = thread_fork("mouse_simulation thread",
    NULL, mouse_simulation, NULL, index);
  if (error) {
    panic("mouse_simulation: thread_fork failed: %s\n",
     strerror(error));
  }
}
/* wait for all of the cats and mice to finish */
for(i=0;i<(NumCats+NumMice);i++) {
  P(CatMouseWait);
}
```

What `kern/synchprobs/catmouse.c` actually does is slightly more elaborate than this.

Example: Concurrent Mouse Simulation Threads (simplified)

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

  /* implicit thread_exit() on return from this function */
}
```

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. decide which thread will run next
  2. save the context of the currently running thread
  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?)

Scheduling

- scheduling means deciding which thread should run next
- scheduling is implemented by a *scheduler*, which is part of the thread library
- simple *round robin* 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 . . .

Causes of Context Switches

- a call to **thread_yield** by a running thread
  - running thread *voluntarily* allows other threads to run
  - yielding thread remains runnable, and on the ready queue
- a call to **thread_exit** by a running thread
  - running thread is terminated
- running thread *blocks*, via a call to `wchan_sleep`
  - thread is no longer runnable, moves off of the ready queue and into a wait channel
  - more on this later . . .
- running thread is *preempted*
  - running thread *involuntarily* stops running
  - remains runnable, and on the ready queue

Preemption

- without preemption, a running thread could potentially run forever, without yielding, blocking, or exiting
- to ensure *fair* access to the CPU for all threads, the thread library may preempt a 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 running thread 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.

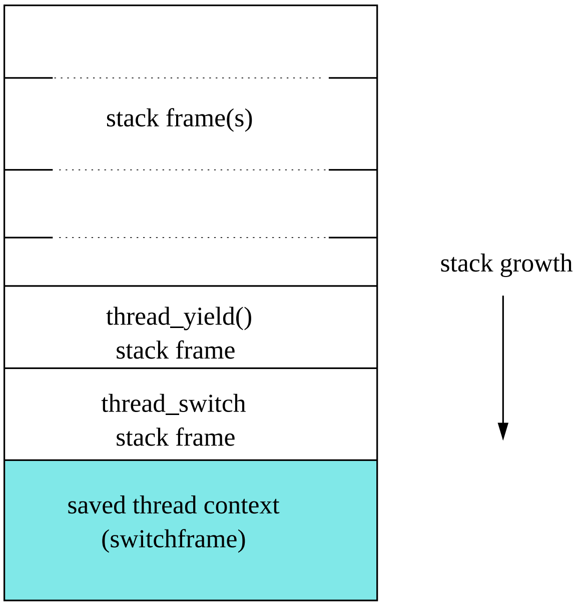
Preemptive Round-Robin Scheduling

- In preemptive round-robin scheduling, the thread library imposes a limit on the amount of time that a thread can run before being 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.
- the quantum is an *upper bound* on the amount of time that a thread can run once it has been dispatched
- the dispatched thread may run for less than the scheduling quantum if it yields, exits, or blocks before its quantum expires

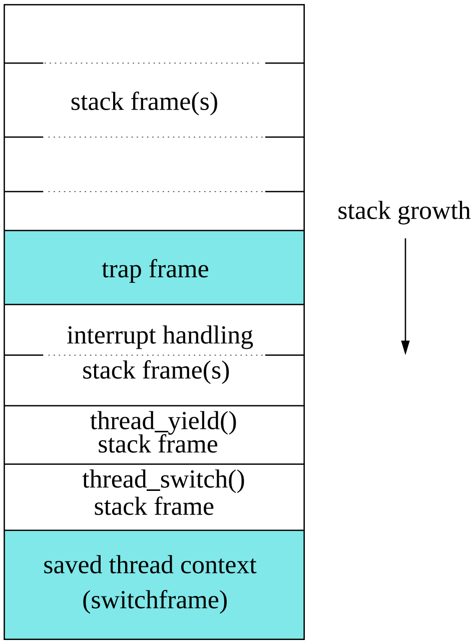
Implementing Preemptive Scheduling

- 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 `thread_yield` to cause a context switch

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



OS/161 Thread Stack after Preemption



Implementing Threads

- the 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.

The OS/161 `thread` Structure

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

struct thread {
 char *t_name;             /* Name of this thread */
 const char *t_wchan_name; /* Wait channel name, if sleeping */
 threadstate_t t_state;    /* State this thread is in */

 /* Thread subsystem internal fields. */
 struct thread_machdep t_machdep; /* Any machine-dependent goo */
 struct threadlistnode t_listnode; /* run/sleep/zombie lists */
 void *t_stack;                 /* Kernel-level stack */
 struct switchframe *t_context; /* Register context (on stack) */
 struct cpu *t_cpu;             /* CPU thread runs on */
 struct proc *t_proc;           /* Process thread belongs to */
 ...
```

Review: MIPS Register Usage

```
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)
```

Dispatching on the MIPS (1 of 2)

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

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

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

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

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

Dispatching on the MIPS (2 of 2)

```
/* Get the new stack pointer from the new thread */
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   s8, 28(sp)
lw   gp, 32(sp)
lw   ra, 36(sp)
nop                  /* delay slot for load */

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

Dispatching on the MIPS (Notes)

- Not all of the registers are saved during a context switch
- This is because the context switch code is reached via a call to `thread_switch` and by convention on the MIPS not all of the registers are required to be preserved across subroutine calls
- thus, after a call to `switchframe_switch` returns, the caller (`thread_switch`) does not expect all registers to have the same values as they had before the call - to save time, those registers are not preserved by the switch
- if the caller wants to reuse those registers it must save and restore them

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 0)

```
/* Note the use of volatile */
int volatile total = 0;

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

If one thread executes `add` and another executes `sub` what is the value of `total` when they have finished?

Critical Section Example (Part 0)

```
/* Note the use of volatile */
int volatile total = 0;

void add() {                   void sub() {
   loadaddr R8 total              loadaddr R10 total
   for (i=0; i<N; i++) {          for (i=0; i<N; i++) {
      lw R9 0(R8)                    lw R11 0(R10)
      add R9 1                       sub R11 1
      sw R9 0(R8)                    sw R11 0(R10)
   }                              }
}                              }
```

Critical Section Example (Part 0)

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

One possible order of execution.

Critical Section Example (Part 0)

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

Another possible order of execution. Many interleavings of instructions are possible. Synchronization is required to ensure a correct ordering.

The use of volatile

```
/* What if we DO NOT use volatile */
int ~~volatile~~ total = 0;

void add() {                      void sub() {
   loadaddr R8 total                 loadaddr R10 total
   lw R9 0(R8)                       lw R11 0(R10)
   for (i=0; i<N; i++) {             for (i=0; i<N; i++) {
      add R9 1                          sub R11 1
   }                                 }
   sw R9 0(R8)                       sw R11 0(R10)
}                                 }
```

Without volatile the compiler could optimize the code. If one thread executes `add` and another executes `sub`, what is the value of `total` when they have finished?

The use of volatile

```
/* What if we DO NOT use volatile */
int ~~volatile~~ total = 0;

void add() {                     void sub() {
   loadaddr R8 total                loadaddr R10 total
   lw R9 0(R8)                      lw R11 0(R10)
   add R9 N                         sub R11 N
   sw R9 0(R8)                      sw R11 0(R10)
}                                }
```

The compiler could aggressively optimize the code. Volatile tells the compiler that the object may change for reasons which cannot be determined from the local code (e.g., due to interaction with a device or because of another thread).

The use of volatile

```
/* Note the use of volatile */
int volatile total = 0;

void add() {                  void sub() {
   loadaddr R8 total             loadaddr R10 total
   for (i=0; i<N; i++) {         for (i=0; i<N; i++) {
      lw R9 0(R8)                   lw R11 0(R10)
      add R9 1                      sub R11 1
      sw R9 0(R8)                   sw R11 0(R10)
   }                             }
}                             }
```

The volatile declaration forces the compiler to load and store the value on every use. Using volatile is necessary but not sufficient for correct behaviour. Mutual exclusion is also required to ensure a correct ordering of instructions.

Ensuring Correctness with Multiple Threads

```
/* Note the use of volatile */
int volatile total = 0;

void add() {                   void sub() {
   int i;                         int i;
   for (i=0; i<N; i++) {          for (i=0; i<N; i++) {
     Allow one thread to execute and make others wait
        total++;                        total--;
     Permit one waiting thread to continue execution
   }                              }
}                              }
```

Threads must enforce mutual exclusion.

Another 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?)

Another 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*,
    - *compare-and-swap*, or
    - *load-link / store-conditional*,

    that are intended for this purpose
  - 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.

Interrupts in OS/161

This is one way that the OS/161 kernel enforces mutual exclusion on a single processor. There is a simple interface

- `spl0()` sets IPL to 0, enabling all interrupts.
- `splhigh()` sets IPL to the highest value, disabling all interrupts.
- `splx(s)` sets IPL to S, enabling whatever state S represents.

These are used by splx() and by the spinlock code.

- `splraise(int oldipl, int newipl)`
- `spllower(int oldipl, int newipl)`
- For splraise, `NEWIPL > OLDIPL`, and for spllower, `NEWIPL < OLDIPL`.

See `kern/include/spl.h` and `kern/thread/spl.c`

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??)

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
- abstractly, a test-and-set instruction works like the following function:

```
TestAndSet(addr,value)
  old = *addr;   // get old value at addr
  *addr = value; // write new value to addr
  return old;
```

  these steps happen *atomically*

- example: x86 `xchg` instruction:

```
xchg src,dest
```

  where `src` is typically a register, and `dest` is a memory address. Value in register `src` is written to memory at address `dest`, and the old value at `dest` is placed into `src`.

Alternatives to Test-And-Set

- Compare-And-Swap

```
CompareAndSwap(addr,expected,value)
  old = *addr;   // get old value at addr
  if (old == expected) *addr = value;
  return old;
```

- example: SPARC `cas` instruction

```
cas addr,R1,R2
```

  if value at `addr` matches value in `R1` then swap contents of `addr` and `R2`

- load-linked and store-conditional
  - Load-linked returns the current value of a memory location, while a subsequent store-conditional to the same memory location will store a new value only if no updates have occurred to that location since the load-linked.
  - more on this later . . .

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, in memory

  ```
  boolean volatile 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.

Spinlocks in OS/161

```
struct spinlock {
  volatile spinlock_data_t lk_lock; /* word for spin */
  struct cpu *lk_holder; /* CPU holding this lock */
};

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

Spinning happens in `spinlock_acquire`

Spinlocks in OS/161

```
spinlock_init(struct spinlock *lk)
{
  spinlock_data_set(&lk->lk_lock, 0);
  lk->lk_holder = NULL;
}

void spinlock_cleanup(struct spinlock *lk)
{
  KASSERT(lk->lk_holder == NULL);
  KASSERT(spinlock_data_get(&lk->lk_lock) == 0);
}

void spinlock_data_set(volatile spinlock_data_t *sd,
  unsigned val)
{
  *sd = val;
}
```

Acquiring a Spinlock in OS/161

```
void spinlock_acquire(struct spinlock *lk)
{
  /* note: code that sets lk->holder has been removed! */
  splraise(IPL_NONE, IPL_HIGH);
  while (1) {
    /* Do test-and-test-and-set to reduce bus contention */
    if (spinlock_data_get(&lk->lk_lock) != 0) {
      continue;
    }
    if (spinlock_data_testandset(&lk->lk_lock) != 0) {
      continue;
    }
    break;
  }
}
```

Using Load-Linked / Store-Conditional

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

  /* Test-and-set using LL/SC.
   * Load the existing value into X, and use Y to store 1.
   * After the SC, Y contains 1 if the store succeeded,
   * 0 if it failed. On failure, return 1 to pretend
   * that the spinlock was already held.
   */

  y = 1;
```

Using Load-Linked / Store-Conditional (Part 2)

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

Releasing a Spinlock in OS/161

```
void spinlock_release(struct spinlock *lk)
{
  /* Note: code that sets lk->holder has been removed! */
  spinlock_data_set(&lk->lk_lock, 0);
  spllower(IPL_HIGH, IPL_NONE);
}
```

Pros and Cons of Spinlocks

- Pros:
  - can be efficient for short critical sections
  - works on multiprocessors
- Cons:
  - CPU is busy (nothing else runs) while waiting for lock
  - starvation is possible

Thread Blocking

- Sometimes a thread will need to wait for an event. For example, if a thread needs to access a critical section that is busy, it must wait for the critical section to become free before it can enter
- 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
- With spinlocks, threads *busy wait* when they cannot enter a critical section. This means that a processor is busy doing useless work. If a thread may need to wait for a long time, it would be better to avoid busy waiting.
- To handle this, the thread scheduler can *block* threads.
- A blocked thread stops running until it is signaled to wake up, allowing the processor to run some other thread.

Thread Blocking in OS/161

- OS/161 thread library functions for blocking and unblocking threads:
  - `void wchan_lock(struct wchan *wc);`
  - `void wchan_unlock(struct wchan *wc);`
    - ∗ locks/unlocks the wait channel `wc`
  - `void wchan_sleep(struct wchan *wc);`
    - ∗ blocks calling thread on wait channel `wc`
    - ∗ channel must be locked, will be unlocked upon return
  - `void wchan_wakeall(struct wchan *wc);`
    - ∗ unblock all threads sleeping on wait channel `wc`
  - `void wchan_wakeone(struct wchan *wc);`
    - ∗ unblock one thread sleeping on wait channel `wc`

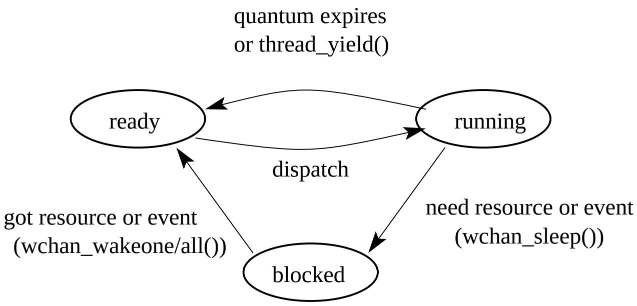
Note: current implementation is FIFO but not promised by the interface

Thread Blocking in OS/161

- `wchan_sleep()` is much like `thread_yield()`. The calling thread is voluntarily giving up the CPU, so the scheduler chooses a new thread to run, the state of the running thread is saved and the new thread is dispatched. However:
  - after a `thread_yield()`, the calling thread is *ready* to run again as soon as it is chosen by the scheduler
  - after a `wchan_sleep()`, the calling thread is *blocked*, and must not be scheduled to run again until after it has been explicitly unblocked by a call to `wchan_wakeone()` or `wchan_wakeall()`.

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.

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*.

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 */
```

A Simple Example using Semaphores

```
volatile int total = 0;

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

What type of semaphore can be used for `sem`?

OS/161 Semaphores

```
struct semaphore {
  char *sem_name;
  struct wchan *sem_wchan;
  struct spinlock sem_lock;
  volatile int sem_count;
};

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

see `kern/include/synch.h` and `kern/thread/synch.c`

OS/161 Semaphores: P() from kern/thread/synch.c

```
P(struct semaphore *sem)
{
  KASSERT(sem != NULL);
  KASSERT(curthread->t_in_interrupt == false);

  spinlock_acquire(&sem->sem_lock);
    while (sem->sem_count == 0) {
      /* Note: we don't maintain strict FIFO ordering */
      wchan_lock(sem->sem_wchan);
      spinlock_release(&sem->sem_lock);
      wchan_sleep(sem->sem_wchan);
      spinlock_acquire(&sem->sem_lock);
    }
    KASSERT(sem->sem_count > 0);
    sem->sem_count--;
  spinlock_release(&sem->sem_lock);
}
```

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

```
V(struct semaphore *sem)
{
  KASSERT(sem != NULL);

  spinlock_acquire(&sem->sem_lock);

    sem->sem_count++;
    KASSERT(sem->sem_count > 0);
    wchan_wakeone(sem->sem_wchan);

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

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 `Occupied` is used to count the number of occupied entries in the list (to ensure nothing is consumed if there are no occupied entries)
  - semaphore `Unoccupied` is used to count the number of unoccupied entries in the list (to ensure nothing is produced if there are no unoccupied entries)

```
struct semaphore *Occupied;
struct semaphore *Unoccupied;
occupied = sem_create("Occupied", 0);     /* initially = 0 */
unoccupied = sem_create("Unoccupied", N); /* initially = N */
```

Bounded Buffer Producer/Consumer Synchronization with Semaphores

```
Producer’s Pseudo-code:
  P(unoccupied);
  add item to the list (call list_append())
  V(occupied);

Consumer’s Pseudo-code:
  P(occupied);
  remove item from the list (call list_remove_front())
  V(unoccupied);
```

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. Once the thread is unblocked it reacquires the lock.

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 is true, it uses `signal` or `broadcast` to notify any threads that may be waiting

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

Waiting on Condition Variables

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

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

Bounded Buffer Producer Using Locks and Condition Variables

```
int volatile 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(itemType 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 Locks and Condition Variables

```
itemType 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);
  return(item);
}
```

Both `Produce()` and `Consume()` call `cv_wait()` inside of a `while` loop. Why?

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.

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.

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)
  - . . .

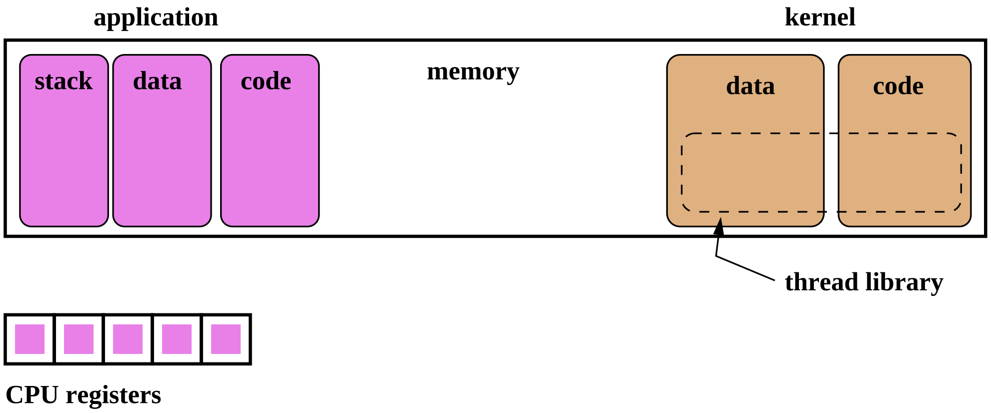
> A process with one thread is a *sequential* process. A process with more than one thread is a *concurrent* process.

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



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.
- Some examples:

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

How System Calls Work

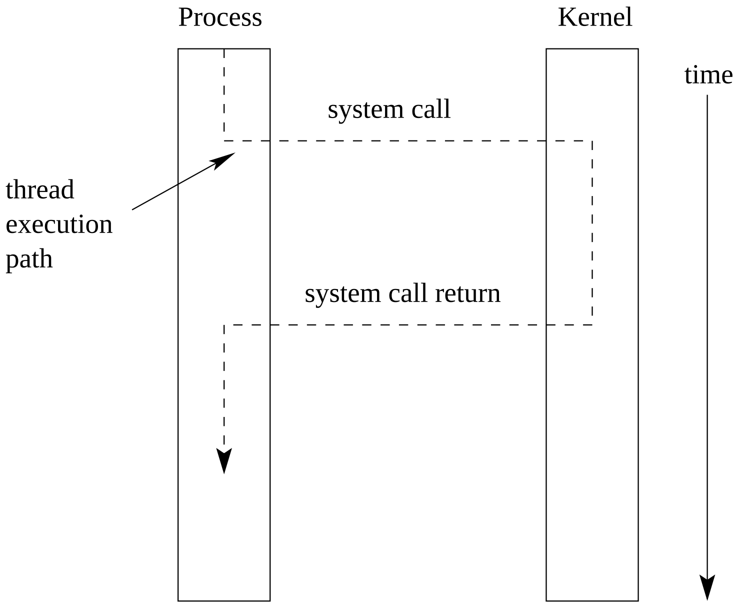
- The hardware provides a mechanism that a running program can use to cause a system call. Often, it is a special instruction, e.g., the MIPS `syscall` instruction.
- What happens on a system call:
  - the processor is switched to system (privileged) execution mode
  - key parts of the current thread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

System Call Execution and Return

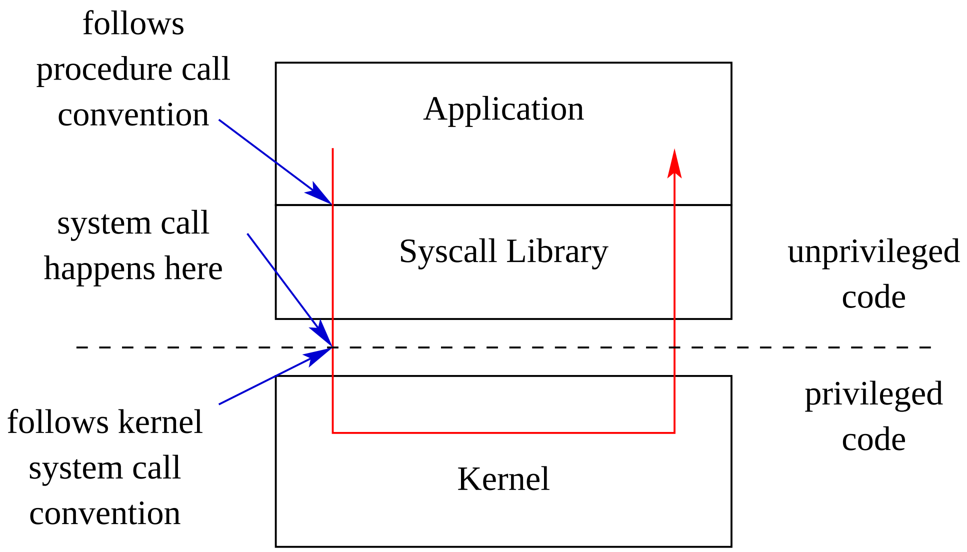
- Once a system call occurs, the calling thread will be executing a system call handler, which is part of the kernel, in privileged mode.
- The kernel's handler determines which service the calling process wanted, and performs that service.
- When the kernel is finished, it returns from the system call. 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



System Call Software Stack



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: user/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 user/uw-testbin/syscall

```
00400050 <main>:
  400050: 27bdffe8  addiu sp,sp,-24
  400054: afbf0010  sw  ra,16(sp)
  400058: 0c100077  jal 4001dc <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
```

MIPS procedure call convention: arguments in a0,a1,. . ., return value in v0.

The above can be obtained using `cs350-objdump -d`.

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

```
/* Contains a number for every more-or-less standard  */
/* Unix system call (you will implement some subset). */
...
#define SYS_close        49
#define SYS_read         50
#define SYS_pread        51
//#define SYS_readv      52  /* won’t be implementing */
//#define SYS_preadv     53  /* won’t be implementing */
#define SYS_getdirentry  54
#define SYS_write        55
...
```

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

System Call Wrapper Functions from the Standard Library

```
...
004001dc <close>:
  4001dc: 08100030  j 4000c0 <__syscall>
  4001e0: 24020031  li  v0,49

004001e4 <read>:
  4001e4: 08100030  j 4000c0 <__syscall>
  4001e8: 24020032  li  v0,50
...
```

The above is disassembled code from the standard C library (libc), which is linked with `user/uw-testbin/syscall.o`.

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

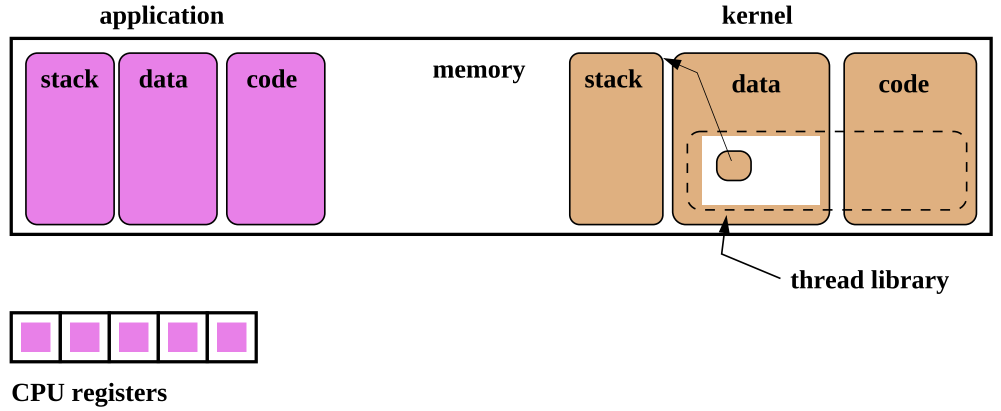
```
common_exception:
  mfc0 k0, c0_status /* Get status register */
  andi k0, k0, CST_KUp /* Check the we-were-in-user-mode bit */
  beq  k0, $0, 1f /* If clear, from kernel, already have stack */
                  /* 1f is branch forward to label 1: */
  nop             /* delay slot */

  /* Coming from user mode - find kernel stack */
  mfc0 k1, c0_context    /* we keep the CPU number here */
  srl k1, k1, CTX_PTBASESHIFT /* shift to get the CPU number */
  sll k1, k1, 2          /* shift back to make array index */
  lui k0, %hi(cpustacks) /* get base address of cpustacks[] */
  addu k0, k0, k1        /* index it */
  move k1, sp            /* Save previous stack pointer */
  b 2f                   /* Skip to common code */
  lw sp, %lo(cpustacks)(k0) /* Load kernel sp (in delay slot) */
```

OS/161 MIPS Exception Handler

```
1:
  /* Coming from kernel mode - just save previous stuff */
  move k1, sp      /* Save previous stack in k1 (delay slot) */
2:

  /* At this point:
   * Interrupts are off. (The processor did this for us.)
   * k0 contains the value for curthread, to go into s7.
   * k1 contains the old stack pointer.
   * sp points into the kernel stack.
   * All other registers are untouched.
   */
```

When the `syscall` instruction occurs, the MIPS transfers control to address `0x80000080`. This kernel exception handler lives there. See `kern/arch/mips/locore/exception-mips1.S`

OS/161 User and Kernel Thread Stacks



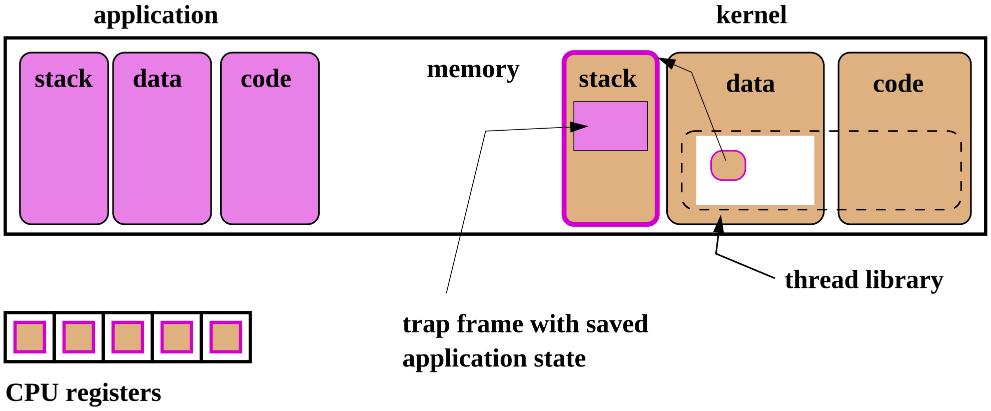
Each OS/161 thread has two stacks, one that is used while the thread is executing unprivileged application code, and another that is used while the thread is executing privileged kernel code.

OS/161 MIPS Exception Handler (cont'd)

The `common_exception` code then 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 the location specified by the application program counter when the `syscall` occurred (i.e., exception PC) 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 (`syscall`) in the case of a system call.
- `mips_trap` can be found in `kern/arch/mips/locore/trap.c`.

Interrupts and exceptions will be presented shortly

OS/161 System Call Handler

```
syscall(struct trapframe *tf)
{  callno = tf->tf_v0; retval = 0;
   switch (callno) {
     case SYS_reboot:
       err = sys_reboot(tf->tf_a0);
       break;
     case SYS___time:
       err = sys___time((userptr_t)tf->tf_a0,
         (userptr_t)tf->tf_a1);
       break;

     /* Add stuff here */

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

syscall checks system call code and invokes a handler for the indicated system call. See kern/arch/mips/syscall/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 */
  KASSERT(curthread->t_curspl == 0);
  /* ...or leak any spinlocks */
  KASSERT(curthread->t_iplhigh_count == 0);
}
```

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 kernel handler creates a *trap frame* and uses it to saves the application thread context so that the handler code can be executed on the CPU.
- Just before the kernel handler finishes executing, it restores the application thread context from the trap frame, before returning control to the application.

In OS/161, trap frames are placed on the *kernel stack* of the thread performed the system call, or of the thread that was running when the interrupt or exception occurred

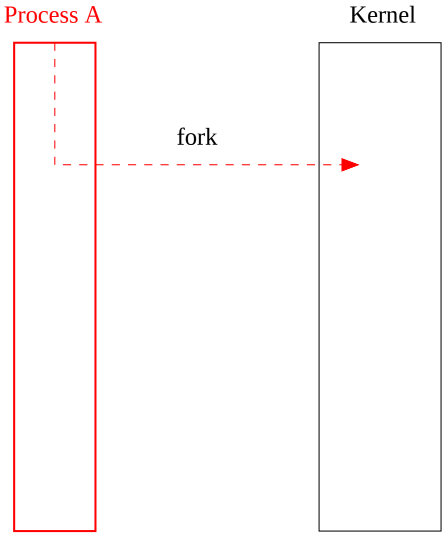
System Calls for Process Management

	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 fork, _exit, getpid and waitpid system calls

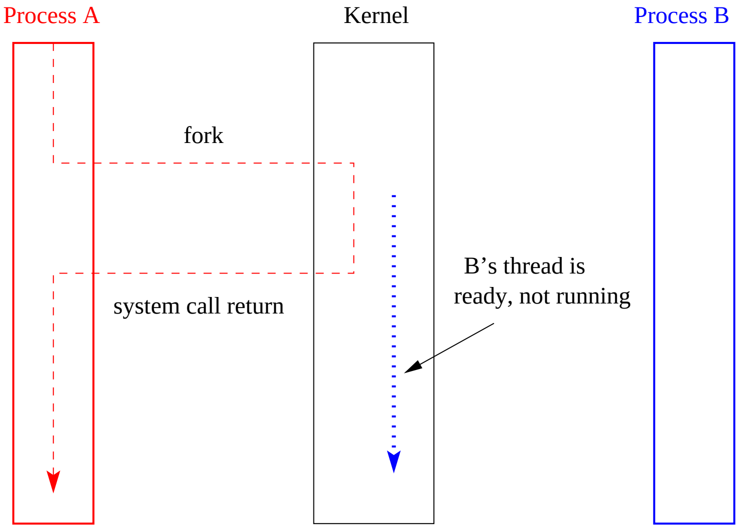
```
main()
{
   rc = fork();  /* returns 0 to child, pid to parent */
   if (rc == 0) {
      my_pid = getpid();
      x = child_code();
      _exit(x);
   } else {
     child_pid = rc;
     parent_code();
     child_exit = waitpid(child_pid);
     parent_pid = getpid();
   }
}
```

Process Creation Example (Part 1)



Parent process (Process A) requests creation of a new process.

Process Creation Example (Part 2)



Kernel creates new process (Process B)

The execv system call

```
int main()
{
  int rc = 0;
  char *args[4];

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

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

Combining fork and execv

```
main()
{
   char *args[4];
   /* set args here */
   rc = fork();  /* returns 0 to child, pid to parent */
   if (rc == 0) {
     status = execv("/testbin/argtest",args);
     printf("If you see this execv failed\n");
     printf("status = %d errno = %d\n", status, errno);
     exit(0);
   } else {
     child_pid = rc;
     parent_code();
     child_exit = waitpid(child_pid);
   }
}
```

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
  - the address space for the process
  - threads belonging to the process
  - lists of resources allocated to the process, such as open files
  - accounting information

OS/161 Process

```
/* From kern/include/proc.h */
struct proc {
  char *p_name;  /* Name of this process */
  struct spinlock p_lock; /* Lock for this structure */
  struct threadarray p_threads; /* Threads in process */

  struct addrspace *p_addrspace; /* virtual address space */
  struct vnode *p_cwd; /* current working directory */

  /* add more material here as needed */
};
```

OS/161 Process

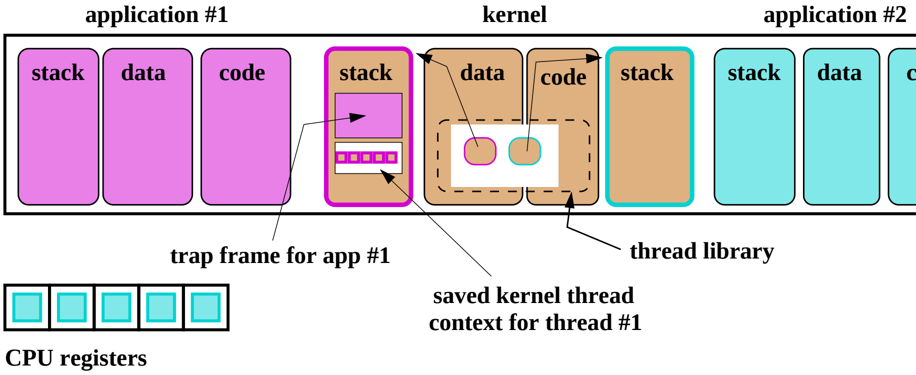
```
/* From kern/include/proc.h */
/* Create a fresh process for use by runprogram() */
struct proc *proc_create_runprogram(const char *name);

/* Destroy a process */
void proc_destroy(struct proc *proc);

/* Attach a thread to a process */
/* Must not already have a process */
int proc_addthread(struct proc *proc, struct thread *t);

/* Detach a thread from its process */
void proc_remthread(struct thread *t);
...
```

Implementing Timesharing

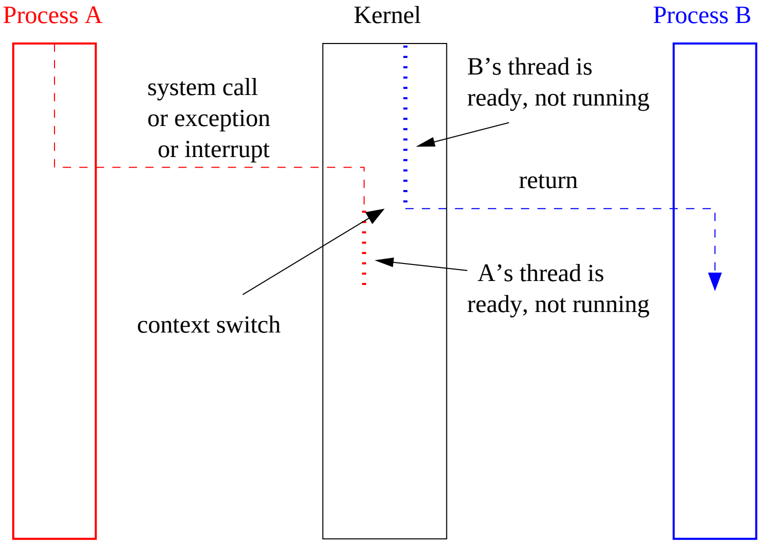
- whenever a system call, exception, or interrupt occurs, control is transferred from the running program to the kernel
- at these points, the kernel has the ability to cause a context switch from the running process' thread to another process' thread
- notice that these context switches always occur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

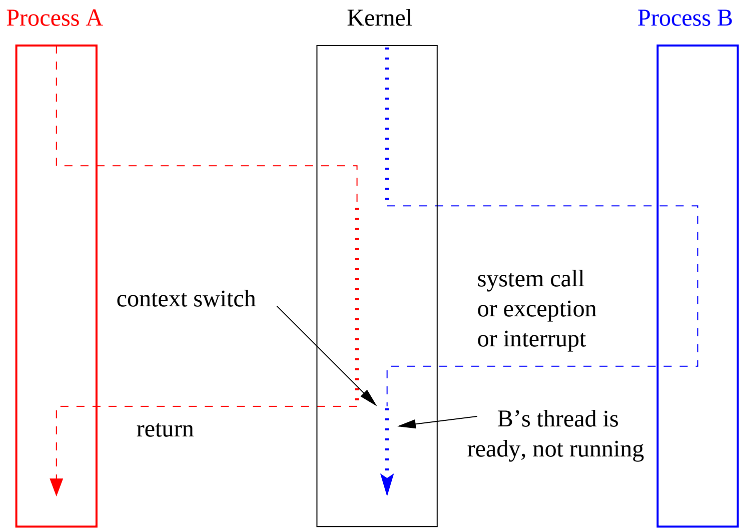


Timesharing Example (Part 1)



Kernel switches execution context to Process B.

Timesharing Example (Part 2)

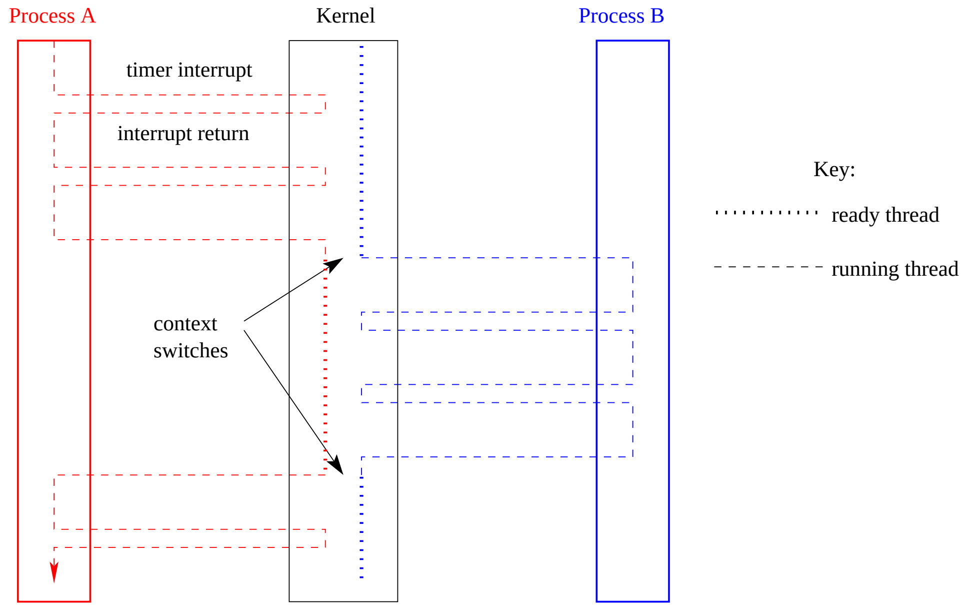


Kernel switches execution context back to process A.

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



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.

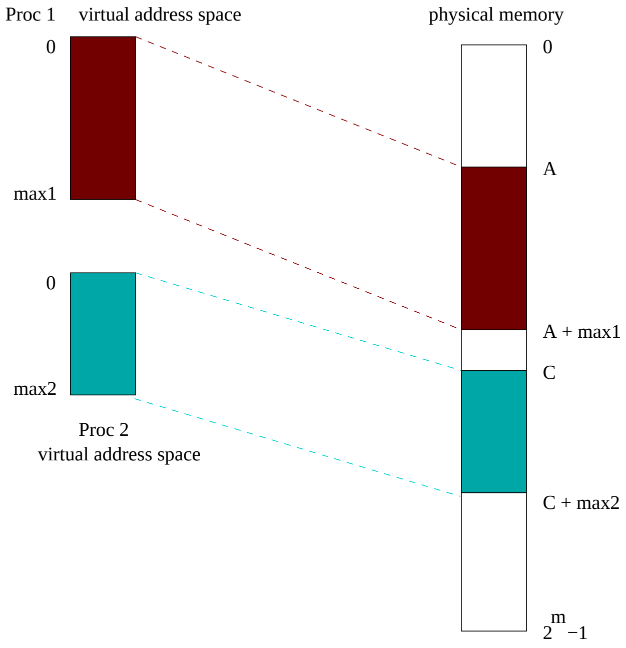
Simple Address Translation: Dynamic Relocation

- hardware provides a *memory management unit (MMU)* which includes a *relocation register* and a *limit register* (or *bound register*).
- to translate a virtual address to a physical address, the MMU:
  - checks whether the virtual address is larger than the limit in the limit register
  - if it is, the MMU raises an *exception*
  - otherwise, the MMU adds the base address (stored in the relocation register) to the virtual address to produce the physical address
- The OS maintains a separate base address and limit for each process, and ensures that the relocation and limit registers in the MMU always contain the base address and limit of the currently-running process.
- To ensure this, the OS must normally change the values in the MMU's registers during each address space switch (i.e., when context switching to a thread of a different process).

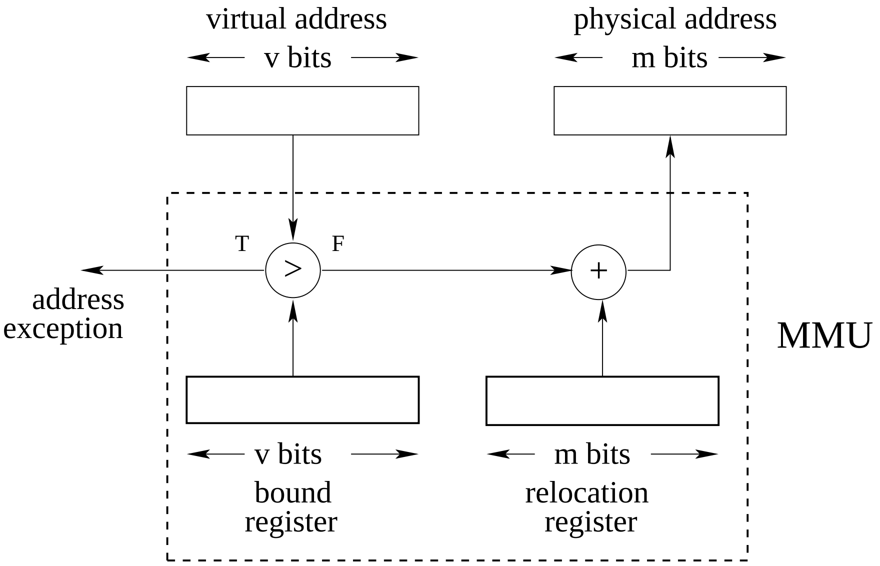
Properties of Dynamic Relocation

- each virtual address space corresponds to a *contiguous range of physical addresses*
- the OS is responsible for deciding *where* each virtual address space should map to in physical memory
  - the OS must track which parts of physical memory are in use, and which parts are free
  - since different address spaces may have different sizes, the OS must allocate/deallocate variable-sized chunks of physical memory
  - this creates the potential for *external fragmentation* of physical memory: wasted, unallocated space
- the MMU is responsible for performing all address translations, using base and limit information provided to it by the the OS

Dynamic Relocation: Address Space Diagram



Dynamic Relocation Mechanism



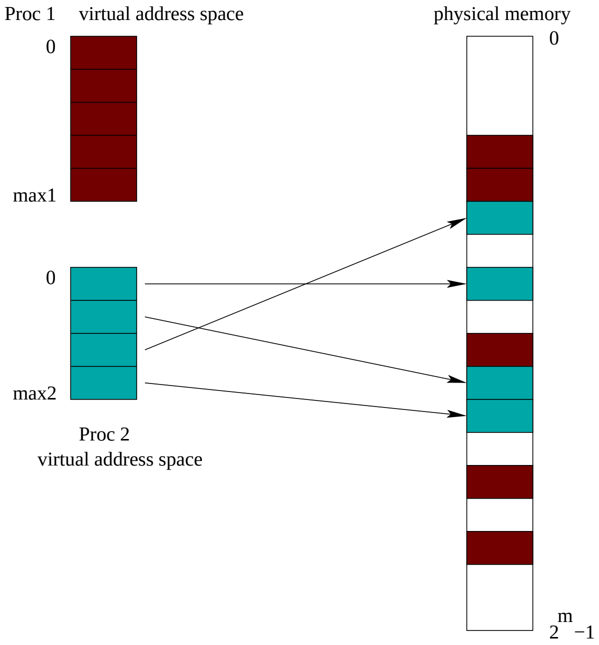
Address Translation: Dynamic Relocation Example

```
Bound register:              0x0011 8000
Relocation register:         0x0010 0000
Process 1: virtual addr      0x000A 1024
Process 1: physical addr =   ___________ ?
Process 1: virtual addr      0x0010 E048
Process 1: physical addr =   ___________ ?

Bound register:              0x0001 0000
Relocation register:         0x0030 0000
Process 2: virtual addr      0x0001 8090
Process 2: physical addr =   ___________ ?

Bound register:              0x000A 1184
Relocation register          0x0020 0000
Process 3: virtual addr      0x000A 1024
Process 3: physical addr =   ___________ ?
```

Address Translation: Paging

- Each virtual address space is divided into fixed-size chunks called *pages*
- 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.

Address Space Diagram for Paging



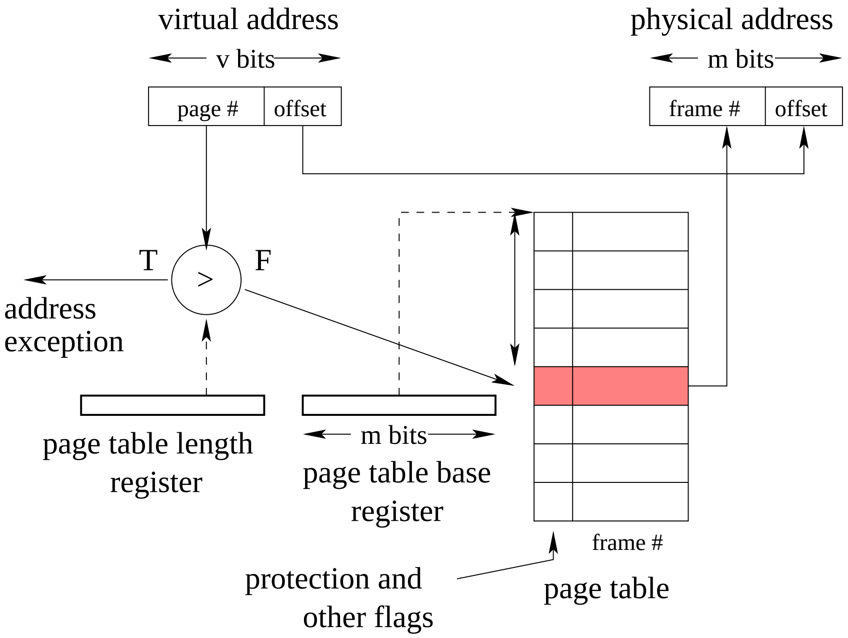
Properties of Paging

- OS is responsible for deciding which frame will hold each page
  - 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.
- MMU is responsible for performing all address translations using the page table that is created and maintained by the OS.
- The OS must normally change the values in the MMU registers on each address space switch, so that they refer to the page table of the currently-running process.

How the MMU Translates Virtual Addresses

- The MMU includes a *page table base register* and a *page table length register*.
  - the base register contains the (physical) address of the first page table entry for the currently-running process
  - the length register contains the number of entries in the page table of the currently running process.
- To translate a virtual address, the MMU:
  - determines the *page number* and *offset* of the virtual address
  - checks whether the page number is larger than the value in the page table length register
  - if it is, the MMU raises an exception
  - otherwise, the MMU uses the page table to determine the *frame number* of the frame that holds the virtual page, and combines the frame number and offset to determine the physical address

Paging Mechanism



virtual address

v bits

page #

offset

physical address

m bits

frame #

offset

T

>

F

address exception

page table length register

m bits

page table base register

frame #

protection and other flags

page table

Address Translation: Paging Example

- Virtual addr = 32 bits, physical addr = 32 bits, offset = 12 bits

Page Table Proc 1 0x0010 0000		Page Table Proc 2 0x0050 0000	
Page	Frame	Page	Frame
0	0x5 0002	0	0x4 0001
1	0x5 0001	1	0x7 9005
2	0x5 0000	2	0x7 FADD
3	0x6 0002		

	Process 1	Process 2
Page Table base register	0x0010 0000	0x0050 0000
Page Table Len register	0x0000 0004	0x0000 0003
Virtual addr	0x0000 2004	0x0000 2004
Physical addr =	____________ ?	____________ ?
Virtual addr	0x0000 31A4	0x0000 31A4
Physical addr =	____________ ?	____________ ?

Page Table Entries

- the primary payload of each page table entry (PTE) is a frame number
- PTEs typically contain other information as well, such as
  - information provided by the kernel to control address translation by the MMU, such as:
    - valid bit: is the process permitted to use this part of the address space?
    - present bit: is this page mapped into physical memory (useful with page replacement, to be discussed later)
    - protection bits: to be discussed
  - information provided by the MMU to help the kernel manage address spaces, such as:
    - reference (use) bit: has the process used this page recently?
    - dirty bit: has the process changed the contents of this page?

Validity and Protection

- during address translation, the MMU checks that the page being used by the process has a *valid* page table entry
  - typically, each PTE contains a *valid bit*
  - invalid PTEs indicate pages that the process is not permitted to use
- the MMU may also enforce other protection rules, for example
  - each PTE may contain a *read-only* bit that indicates whether the corresponding page is read-only, or can be modified by the process
- if a process attempts to access an invalid page, or violates a protection rule, 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.

Summary: Roles of the Kernel and the MMU

- Kernel:
  - manage MMU state on address space switches (context switch from thread in one process to thread in a different process)
  - 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

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.
  - This is called a *TLB hit*.
- Otherwise, translate through the page table.
  - This is called a *TLB miss*.
- 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 on each address space switch. (Why?)

TLB Management

- A TLB may be *hardware-controlled* or *software-controlled*
- In a hardware-controlled TLB, when there is a TLB miss:
  - The MMU (hardware) finds the frame number by performing a page table lookup, translates the virtual address, and adds the translation (page number, frame number pair) to the TLB.
  - If the TLB is full, the MMU evicts an entry to make room for the new one.
- In a software-controlled TLB, when there is a TLB miss:
  - the MMU simply causes an exception, which triggers the kernel exception handler to run
  - the kernel must determine the correct page-to-frame mapping and load the mapping into the TLB (evicting an entry if the TLB is full), before returning from the exception
  - after the exception handler runs, the MMU retries the instruction that caused the exception.

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:

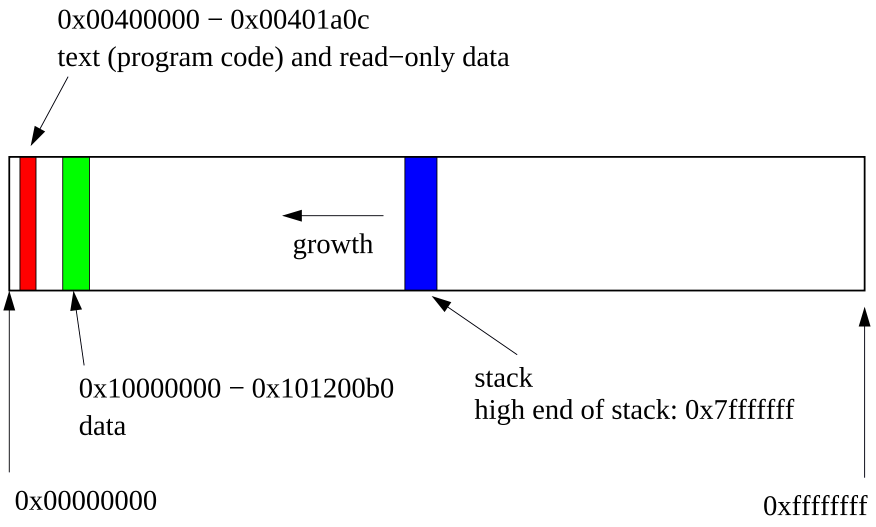
  TLB_Write: modify a specified TLB entry

  TLB_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?



0x00400000 − 0x00401a0c
text (program code) and read−only data

growth

0x10000000 − 0x101200b0
data

stack
high end of stack: 0x7fffffff

0x00000000

0xffffffff

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

Address Translation In OS/161: 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
};
```

- Notice that each segment must be mapped contiguously into physical memory.

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/vm/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 exception. This time, it may succeed.

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

Address Translation: OS/161 dumbvm Example

- Note: in OS/161 the stack is 12 pages and the page size is 4 KB = 0x1000.

Variable/Field	Process 1	Process 2
as_vbase1	0x0040 0000	0x0040 0000
as_pbase1	0x0020 0000	0x0050 0000
as_npages1	0x0000 0008	0x0000 0002
as_vbase2	0x1000 0000	0x1000 0000
as_pbase2	0x0080 0000	0x00A0 0000
as_npages2	0x0000 0010	0x0000 0008
as_stackpbase	0x0010 0000	0x00B0 0000

```
                     Process 1        Process 2
Virtual addr        0x0040 0004      0x0040 0004
Physical addr =     ___________ ?    ___________ ?
Virtual addr        0x1000 91A4      0x1000 91A4
Physical addr =     ___________ ?    ___________ ?
Virtual addr        0x7FFF 41A4      0x7FFF 41A4
Physical addr =     ___________ ?    ___________ ?
Virtual addr        0x7FFF 32B0      0x2000 41BC
Physical addr =     ___________ ?    ___________ ?
```

Initializing 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

> OS/161 *pre-loads* the address space before the program runs. Many other OS load pages *on demand*. (Why?)

- 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

  `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

- The ELF file contains a header describing the segments and segment *images*.
- Each ELF segment describes a contiguous region of the virtual address space.
- The header includes an entry for each segment 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 segment image in the ELF file (if present)
  - the length of the segment image in the ELF file (if present)
- 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 OS/161 kernel copies segment 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/syscall/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 user/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 user/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`.

An Address Space for the Kernel

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

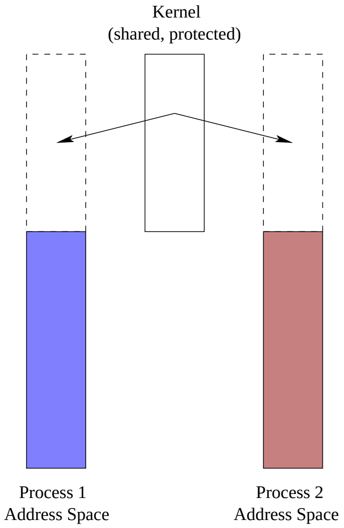
Kernel in physical space: disable address translation in privileged system execution mode, enable it in unprivileged mode

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

Kernel mapped into portion of address space of *every process*: OS/161, Linux, and other operating systems use this approach

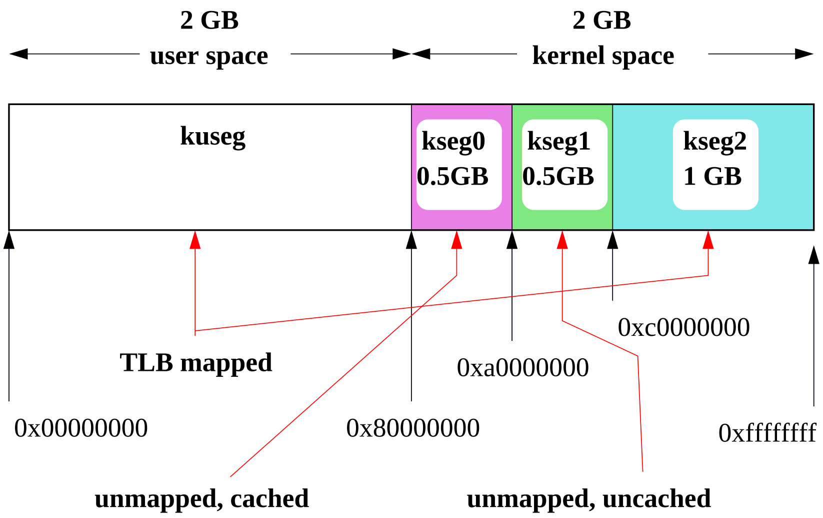
- memory protection mechanism is used to isolate the kernel from applications
- one advantage of this approach: application virtual addresses (e.g., system call parameters) are easy for the kernel to use

The Kernel in Process' Address Spaces



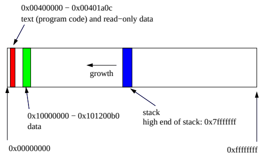
Attempts to access kernel code/data in user mode result in memory protection exceptions, not invalid address exceptions.

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.

The Problem of Sparse Address Spaces



- Consider the page table for `user/testbin/sort`, assuming a 4 Kbyte page:
  - need 2^{19} page table entries (PTEs) to cover the bottom half of the virtual address space (2GB).
  - the text segment occupies 2 pages, the data segment occupies 289 pages, and OS/161 sets the initial stack size to 12 pages, so there are only 303 *valid* pages (of 2^{19}).
- If dynamic relocation is used, the kernel will need to map a 2GB address space contiguously into physical memory, even though only a tiny fraction of that address space is actually used by the program.
- If paging is used, the kernel will need to create a page table with 2^{19} PTEs, almost all of which are marked as not valid.

Handling Sparse Address Spaces

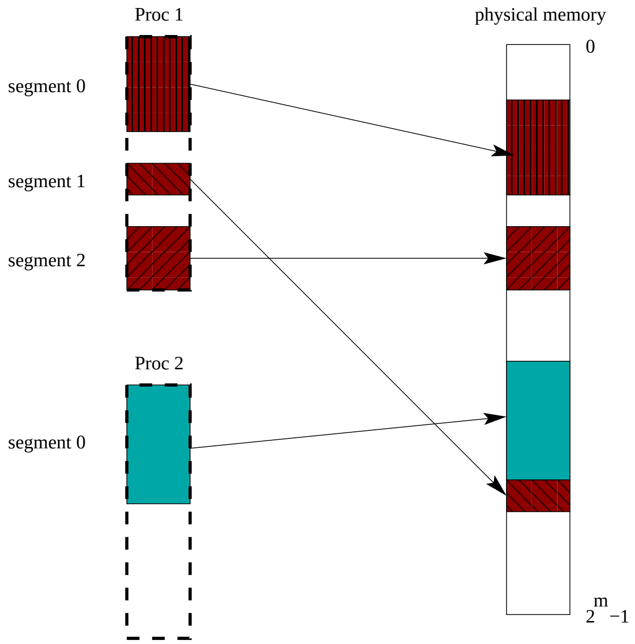
- Use dynamic relocation, but provide separate base and length for each valid segment of the address space. Do not map the rest of the address space.
  - OS/161 `dumbvm` uses a simple variant of this idea, which depends on having a software-managed TLB.
  - A more general approach is *segmentation*.
- A second approach is to use *multi-level paging*
  - replace the single large linear page table with a hierarchy of smaller page tables
  - a sparse address space can be mapped by a sparse tree hierarchy
  - easier to manage several smaller page tables than one large one (remember: each page table must be continguous in physical memory!)

Segmentation

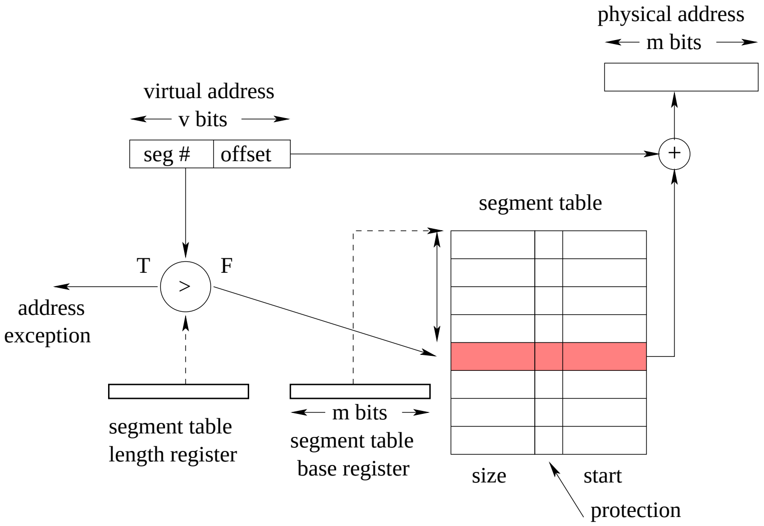
- Often, programs (like `sort`) need several virtual address segments, e.g, for code, data, and stack.
- With segmentation, a virtual address can be thought of as having two parts:

  (segment ID, address within segment)
- Each segment also has a length.

Segmented Address Space Diagram



Mechanism for Translating Segmented Addresses



This translation mechanism requires physically contiguous allocation of segments.

Address Translation: Segmented Addresses Example

- Virtual addr = 32 bits, physical addr = 32 bits, offset = 28 bits

Segment Table 1 @ 0x0010 0000				Segment Table 2 @ 0x0050 0000			
Seg	Size	Prot	Start	Seg	Size	Prot	Start
0	0x6000	X-	0x7 0000	0	0x9000	X-	0x90 0000
1	0x1000	-W	0x6 0000	1	0xD000	-W	0xAF 0000
2	0xC000	-W	0x5 0000	2	0xA000	-W	0x70 0000
3	0xB000	-W	0x8 0000				

	Process 1	Process 2
Seg Table base reg	0x0010 0000	0x0050 0000
Seg Table Len	0x0000 0004	0x0000 0003
Virtual addr	0x0000 2004	0x1000 B481
Physical addr =	___________ ?	___________ ?
Virtual addr	0x2000 31A4	0x2000 B1A4
Physical addr =	___________ ?	___________ ?

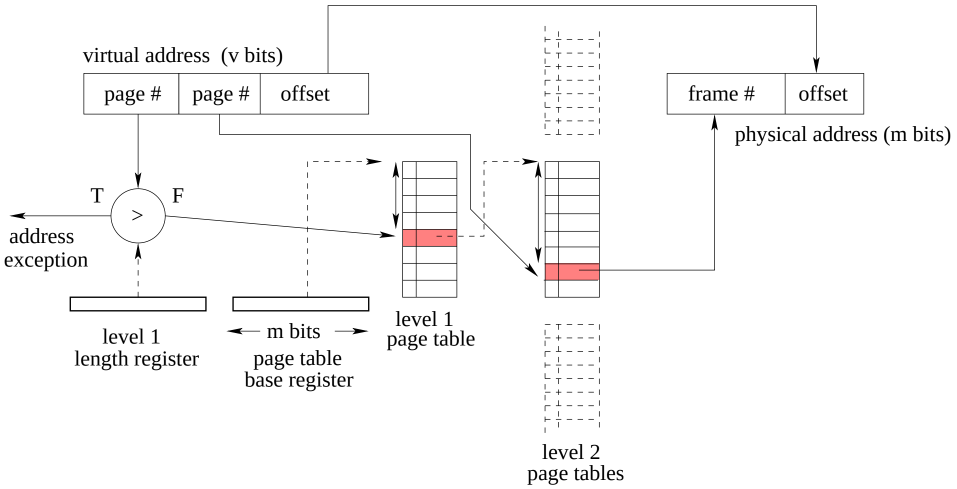
Handling Sparse Paged Virtual Address Spaces

- Large paged 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

Two-Level Paging



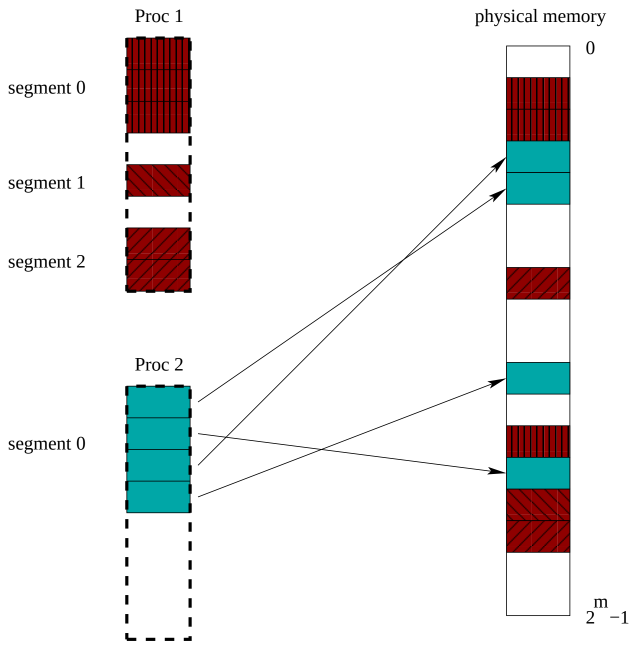
Address Translation: Two-Level Paging Example

- Bits: Virtual addr = 32, physical addr = 32, level 1 = 4, level 2 = 12

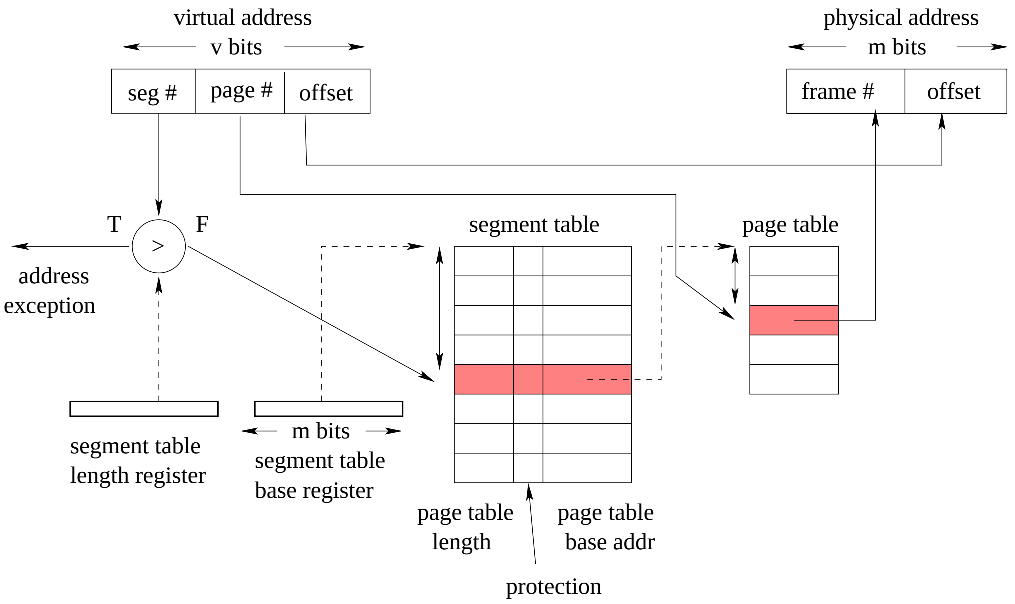
Level 1			Level 2		Level 2		Level 2	
0x0010 0000			0x0005 0000		0x0004 0000		0x0007 0000	
S	Len	Address	P	Frame	P	Frame	P	Frame
0	3	0x7 0000	0	0x8001	0	0x4001	0	0x200A
1	3	0x5 0000	1	0x9005	1	0x7005	1	0x200B
2	4	0x4 0000	2	0x6ADD	2	0x7A00	2	0x200C
					3	0x1023		
		...		...		...		...

Base register	0x0010 0000	0x0010 0000
Level 1 Len	0x0000 0003	0x0000 0003
Virtual addr	0x1000 2004	0x3004 020F
Physical addr =	______________ ?	______________ ?
Virtual addr	0x2003 31A4	0x1003 20A8
Physical addr =	______________ ?	______________ ?
Virtual addr	0x0002 A049	
Physical addr =	______________ ?	

Combining Segmentation and Paging



Combining Segmentation and Paging: Translation Mechanism



Address Translation: Segmentation and Paging Example

- Bits: Virtual addr = 32, physical addr = 32, Segment = 8, Page = 8

Segment Table 1 0x0020 0000				Page Table 0x0003 0000		Page Table 0x0001 0000		Page Table 0x0002 0000	
S	Len	Prot	Address	P	Frame	P	Frame	P	Frame
0	3	X-	0x3 0000	0	0x8001	0	0x4001	0	0x200A
1	4	-W	0x1 0000	1	0x9005	1	0x7005	1	0x200B
2	3	-W	0x2 0000	2	0x6ADD	2	0x7A00	2	0x200C
						3	0x1023		

```
Base register    0x0020 0000     0x0020 0000
Seg Table Len    0x0000 0003     0x0000 0003
Virtual addr     0x0100 2004     0x0102 C104
Physical addr =  ___________ ?   ___________ ?
Virtual addr     0x0203 31A4     0x0304 020F
Physical addr =  ___________ ?   ___________ ?
Virtual addr     0x0002 A049     0x0104 20A8
Physical addr =  ___________ ?   ___________ ?
```

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 storage, e.g., on disks, as well as primary memory.
- Move pages (or segments) between secondary storage and primary memory so that they are in primary memory when they are needed.

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.

Page Faults

- When paging is used, some valid pages may be loaded into memory, and some may not be.
- To account for this, each PTE may contain a *present* bit, to indicate whether the page is or is not loaded into memory
  - $V = 1, P = 1$: page is valid and in memory (no exception occurs)
  - $V = 1, P = 0$: page is valid, but is not in memory (exception!)
  - $V = 0, P = x$: invalid page (exception!)
- If $V = 1$ and $P = 0$, the MMU will generate an exception if a process tries to access the page. This is called a *page fault*.
- To handle a page fault, the kernel must:
  - issue a request to the disk to copy the missing page into memory,
  - *block* the faulting process
  - once the copy has completed, set $P = 1$ in the PTE and unblock the process
- the processor will then retry the instrution that caused the page fault

Page Faults in OS/161

- things are a bit different in systems with software-managed TLBs, such as OS/161 on the MIPS processor
- MMUs with software-managed TLBs never check page tables, and thus do not interpret P bits in page table entries
- In an MMU with a software-managed TLB, either there is a valid translation for a page in the TLB, or there is not.
  - If there is not, the MMU generates an exception. It is up to the kernel to determine the reason for the exception. Is this:
    - an access to a valid page that is not in memory (a page fault)?
    - an access to a valid page that is in memory?
    - an access to an invalid page?
  - The kernel should ensure that a page has a translation in the TLB *only* if the page is valid and in memory. (Why?)

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?

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, true LRU is difficult to implement 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 page table 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 cleared by the operating system
- The use bit provides a small amount of efficiently-maintainable usage information that can be exploited by a page replacement algorithm.

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 page table entry of a page when a process *writes* to a virtual address on that page, i.e., when the page is changed.
- The operating system can clear 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.

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
```

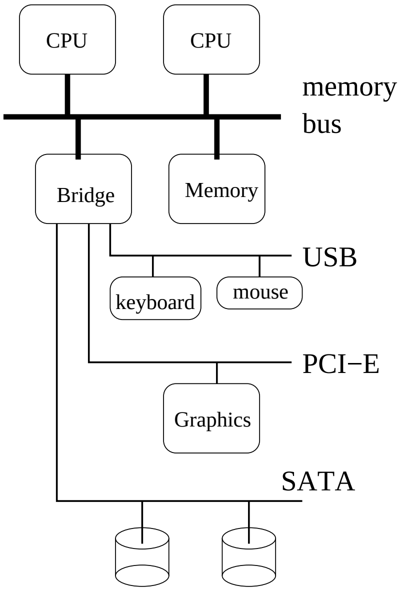
Thrashing and Load Control

- 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)

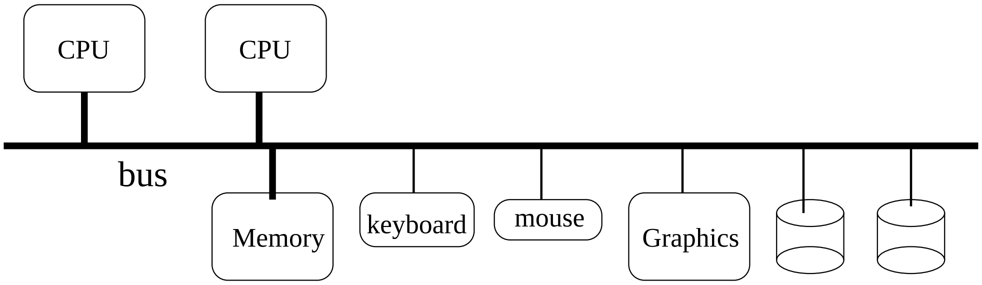
Swapping Out Processes

- 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.

Bus Architecture Example



Simplified Bus Architecture



Sys/161 LAMEbus Device Examples

- 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

Device Register Example: Sys/161 timer/clock

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

Device Register Example: Sys/161 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

Device Drivers

- a device driver is a part of the kernel that interacts with a device
- example: write character to serial output device

```
write character to device data register
write output command to device command register
repeat {
    read device status register
} until device status is ‘‘completed’’
clear the device status register
```

- this example illustrates *polling*: the driver repeatedly checks whether the device is finished, until it is finished.

Another Polling Example

```
write target sector number into sector number register
write output data (512 bytes) into transfer buffer
write ‘‘write’’ command into status register
repeat {
   read status register
} until status is ‘‘completed’’ (or error)
clear the status register
```

Disk operations are slow. The device driver may have to poll for a long time.

Using Interrupts to Avoid Polling

- polling can be avoided if the device can use interrupts to indicate that it is finished
- example: disk write operation using interrupts:

```
write target sector number into sector number register
write output data (512 bytes) into transfer buffer
write ’’write’’ command into status register
block until device generates completion interrupt
read status register to check for errors
clear status register
```

- while thread running the driver is blocked, the CPU is free to run other threads
- kernel synchronization primitives (e.g., semaphores) can be used to implement blocking

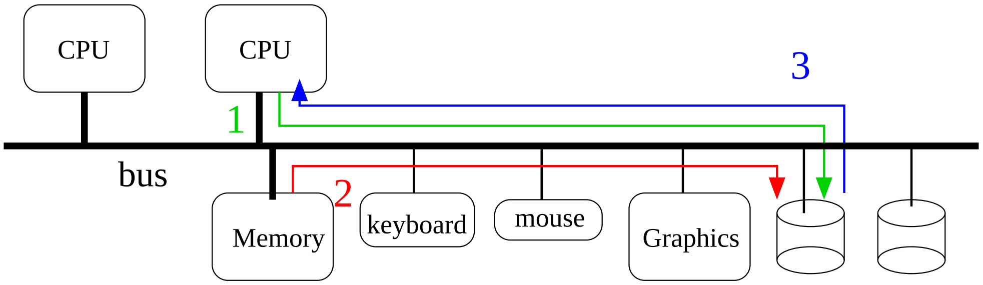
Device Data Transfer

- Sometimes, a device operation will involve a large chunk of data - much larger than can be moved with a single instruction.
  - example: disk read or write operation
- Devices may have data buffers for such data - but how to get the data between the device and memory?
  - Option 1: *program-controlled I/O*
    The device driver moves the data iteratively, one word at a time.
    - ∗ Simple, but the CPU is *busy* while the data is being transferred.
  - Option 2: *direct memory access (DMA)*
    - ∗ CPU is *not busy* during data transfer, and is free to do something else.

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

Device Driver for Disk Write with DMA

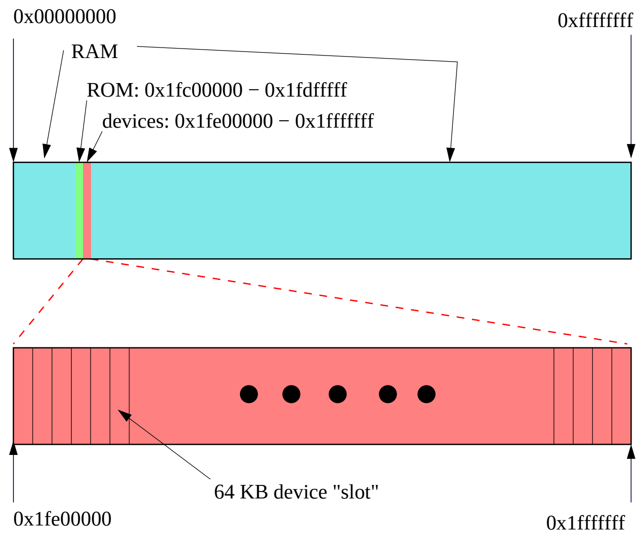
```
write target disk sector number into sector number register
write source memory address into address register
write ’’write’’ command into status register
block (sleep) until device generates completion interrupt
read status register to check for errors
clear status register
```

Note: driver no longer copies data into device transfer buffer

Accessing Devices

- how can a device driver access device registers?
- Option 1: special I/O instructions
  - such as `in` and `out` instructions on x86
  - device registers are assigned “port” numbers
  - instructions transfer data between a specified port and a CPU register
  -
- Option 2: memory-mapped I/O
  - each device register has a physical memory address
  - device drivers can read from or write to device registers using normal load and store instructions, as though accessing memory

MIPS/OS161 Physical Address Space

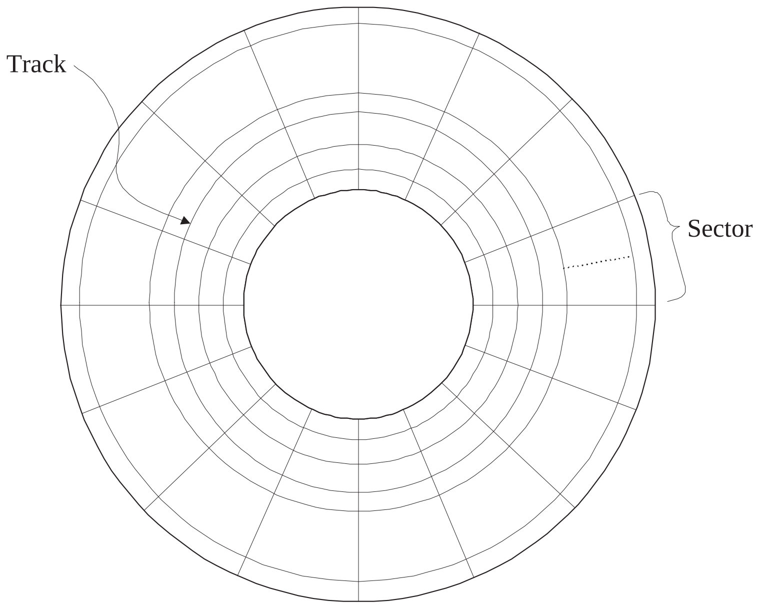


Each device is assigned to one of 32 64KB device “slots”. A device’s registers and data buffers are memory-mapped into its assigned slot.

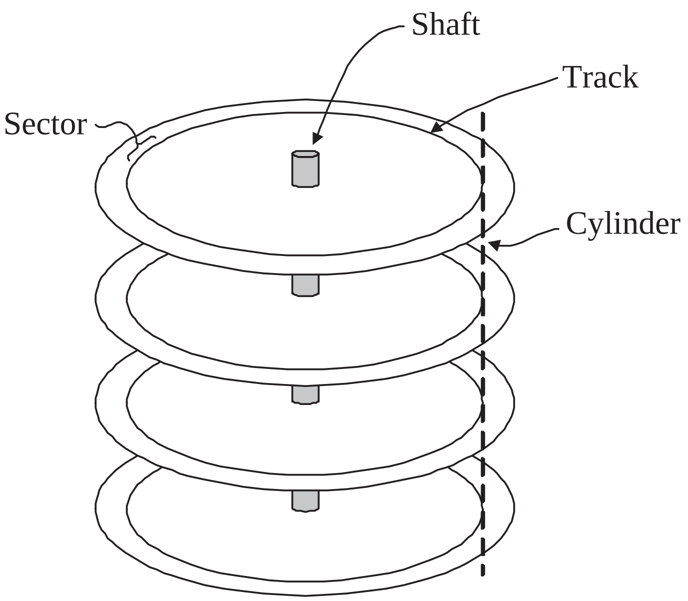
Logical View of a Disk Drive

- 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

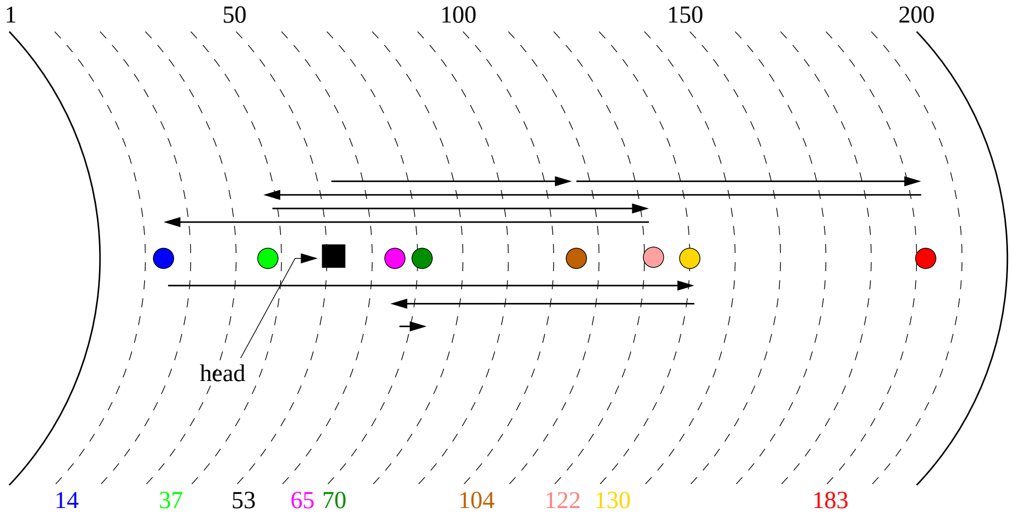
- larger transfers to/from a disk device are *more efficient* than smaller ones. That is, the cost (time) per byte is smaller for larger transfers. (Why?)
- sequential I/O is faster than non-sequential I/O
  - sequential I/O operations eliminate the need for (most) seeks
  - disks use other techniques,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

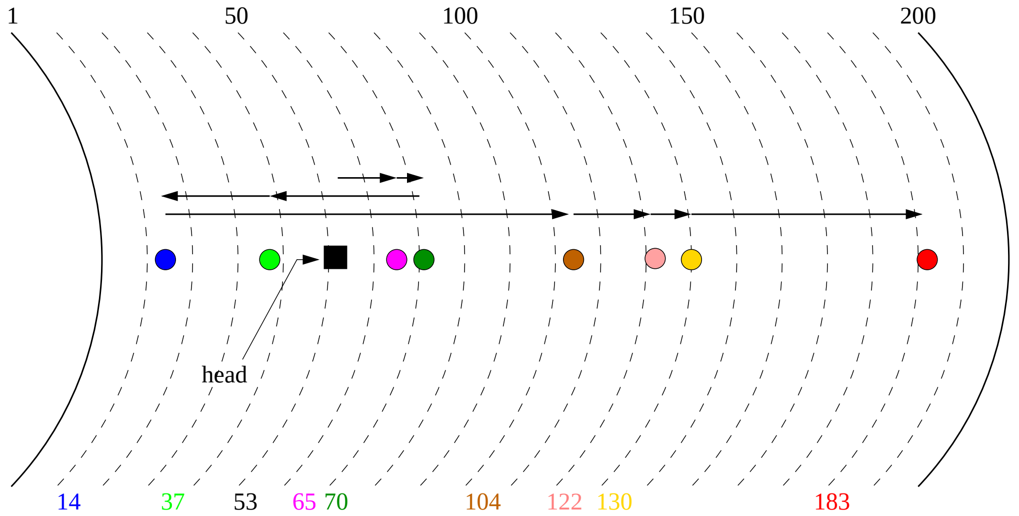
- handle requests in the order in which they arrive
- fair and simple, but no optimization of seek times



arrival order: 104 183 37 122 14 130 65 70

Shortest Seek Time First (SSTF)

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

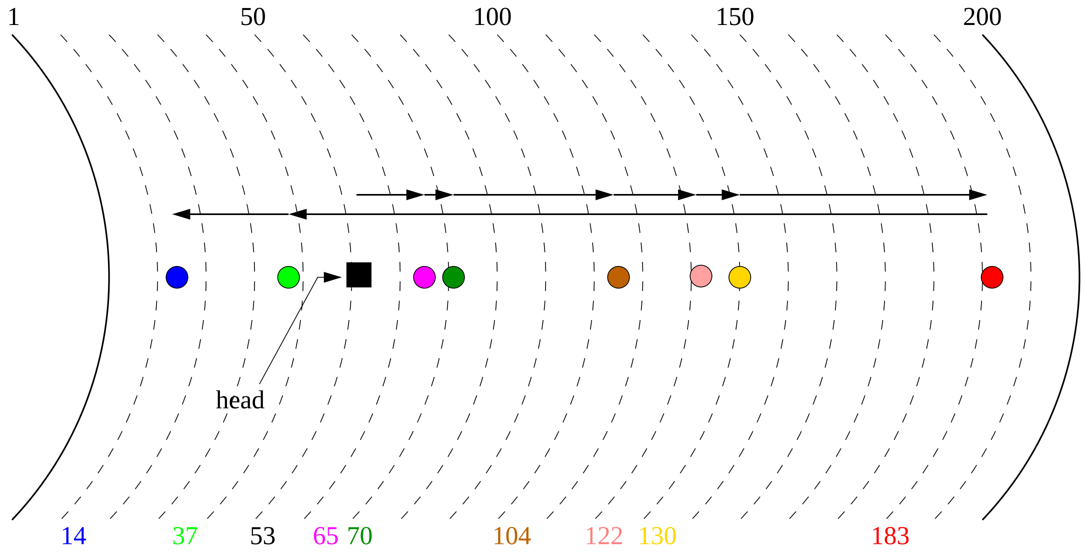


arrival order: 104 183 37 122 14 130 65 70

Elevator Algorithms (SCAN)

- Under SCAN, aka the elevator algorithm, the disk head moves in one direction until there are no more requests in front of it, then reverses direction.
- there are many variations on this idea
- SCAN reduces seek times (relative to FCFS), while avoiding starvation

SCAN Example



arrival order: 104 183 37 122 14 130 65 70

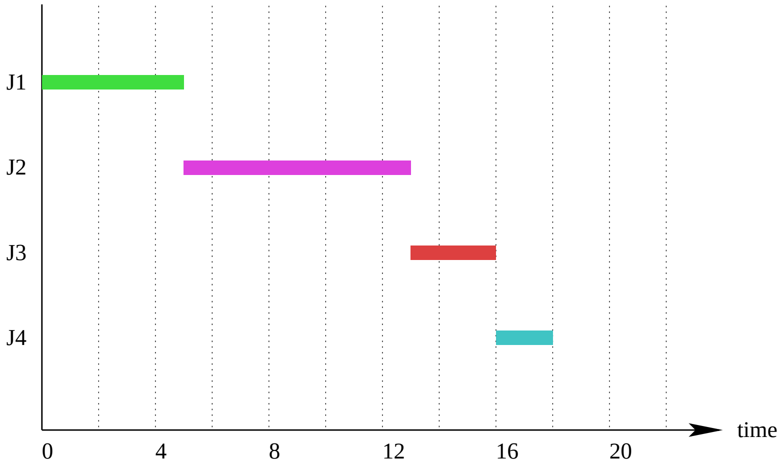
Job Scheduling Model

- problem scenario: a set of *jobs* needs to be executed using a single server, on which only one job at a time may run
- for the ith job, we have an arrival time a_i and a run time r_i
- after the ith job has run on the server for total time r_i, it finishes and leaves the system
- a job *scheduler* decides which job should be running on the server at each point in time
- let s_i ($s_i \geq a_i$) represent the time at which the ith job first runs, and let f_i represent the time at which the ith job finishes
    - the *turnaround time* of the ith job is $f_i - a_i$
    - the *response time* of the ith job is $s_i - a_i$

Basic Non-Preemptive Schedulers: FCFS and SJF

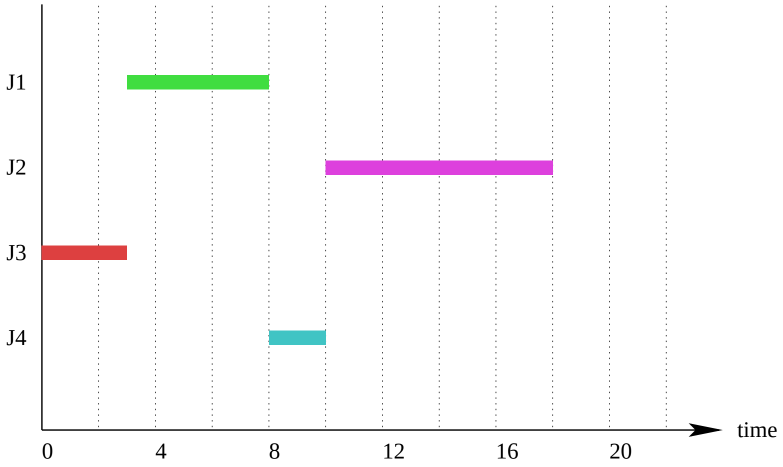
- FCFS: runs jobs in arrival time order.
    - simple, avoids starvation
    - pre-emptive variant: round-robin
- SJF: shortest job first - run jobs in increasing order of r_i
    - minimizes average *turnaround* time
    - long jobs may starve
    - pre-emptive variant: SRTF (shortest remaining time first)

FCFS Gantt Chart Example



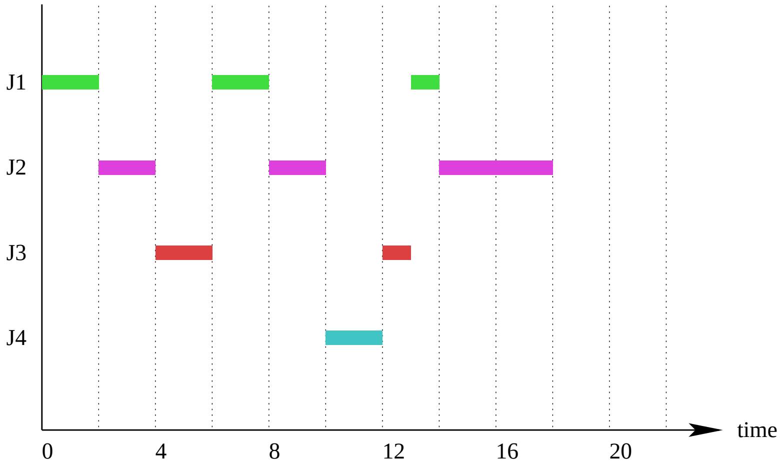
Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

SJF Example



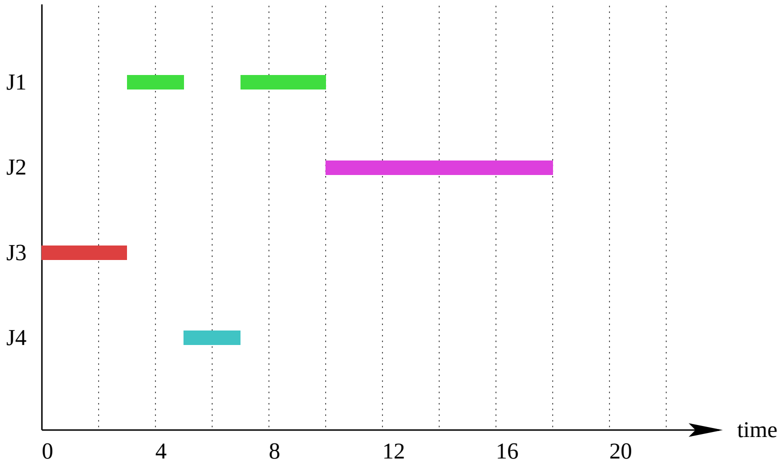
Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

Round Robin Example



Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

SRTF Example



Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

CPU Scheduling

- CPU scheduling is job scheduling where:
  - the server is a CPU (or a single core of a multi-core CPU)
  - the jobs are *ready threads*
    - a thread "arrives" when it becomes ready, i.e., when it is first created, or when it wakes up from sleep
    - the run-time of the thread is the amount of time that it will run before it either finishes or blocks
  - thread run times are typically *not known* in advance by the scheduler
- typical scheduler objectives
  - responsiveness - low *response time* for some or all threads
  - "fair" sharing of the CPU
  - efficiency - there is a cost to switching

Prioritization

- CPU schedulers are often expected to consider process or thread priorities
- priorities may be
  - specified by the application (e.g., Linux `setpriority/sched_setscheduler`)
  - chosen by the scheduler
  - some combination of these
- two approaches to scheduling with priorites
  1. schedule the highest priority thread
  2. weighted fair sharing
     - let p_i be the priority of the ith thread
     - try to give each thread a "share" of the CPU in proportion to its priority:

$$\frac{p_i}{\sum_j p_j} \tag{1}$$

Multi-level Feedback Queues

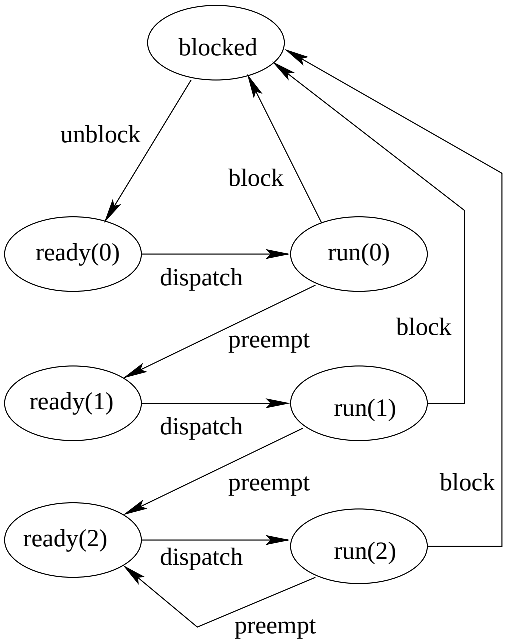
- objective: good responsiveness for *interactive* processes
  - threads of interactive processes block frequently, have short run times
- idea: gradually diminish priority of threads with long run times and infrequent blocking
  - if a thread blocks before its quantum is used up, *raise* its priority
  - if a thread uses its entire quantum, *lower* its priority

Multi-level Feedback Queues (Algorithm)

- scheduler maintains several round-robin ready queues
  - highest priority threads in queue Q_0, lower priority in Q_1, still lower in Q_2, and so on.
- scheduler always chooses thread from the lowest non-empty queue
- threads in queue Q_i use quantum q_i, and $q_i \leq q_j$ if $i < j$
- newly ready threads go into ready queue Q_0
- a level i thread that is preempted goes into queue Q_{i+1}

This basic algorithm may *starve* threads in lower queues. Various enhancements can avoid this, e.g, periodically migrate all threads into Q_0.

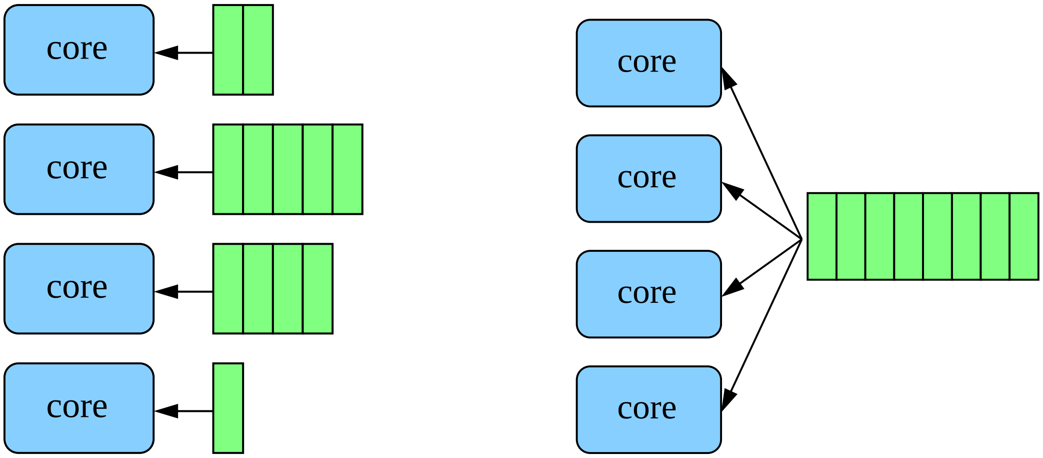
3 Level Feedback Queue State Diagram



Linux Completely Fair Scheduler (CFS) - Key Ideas

- “Completely Fair Scheduling” - a weighted fair sharing approach
- suppose that c_i is the actual amount of time that the scheduler has allowed the ith thread to run.
- on an *ideally shared* processor, we would expect $c_0 \frac{\sum_j p_j}{p_0} = c_1 \frac{\sum_j p_j}{p_1} = \cdots$
- CFS calls $c_i \frac{\sum_j p_j}{p_i}$ the *virtual runtime* of the ith thread, and tracks it for each thread
- CFS chooses the thread with the lowest virtual runtime, and runs it until some other thread’s virtual runtime is lower (subject to a minimum runtime quantum)
  - virtual runtime advances more slowly for higher priority threads, so they get longer time slices
  - all ready threads run regularly, so good responsiveness

Scheduling on Multi-Core Processors



per core ready queue(s) vs. shared ready queue(s)

Scalability and Cache Affinity

- Contention and Scalability
  - access to shared ready queue is a critical section, mutual exclusion needed
  - as number of cores grows, contention for ready queue becomes a problem
  - per core design *scales* to a larger number of cores
- CPU cache affinity
  - as thread runs, data it accesses is loaded into CPU cache(s)
  - moving the thread to another core means data must be reloaded into that core’s caches
  - as thread runs, it acquires an *affinity* for one core because of the cached data
  - per core design benefits from affinity by keeping threads on the same core
  - shared queue design does not

Load Balancing

- in per-core design, queues may have different lengths
- this results in *load imbalance* across the cores
  - cores may be idle while others are busy
  - threads on lightly loaded cores get more CPU time than threads on heavily loaded cores
- not an issue in shared queue design
- per-core designs typically need some mechanism for *thread migration* to address load imbalances
  - migration means moving threads from heavily loaded cores to lightly loaded cores

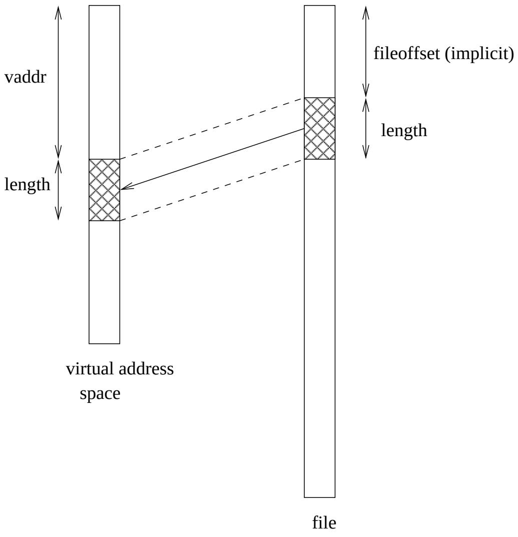
Files and File Systems

- files: persistent, named data objects
  - data consists of a sequence of numbered bytes
  - file may change size over time
  - file has associated meta-data
    - ∗ 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, seek
  - read copies data from a file into a virtual address space
  - write copies data from a virtual address space into a file
  - seek enables non-sequential reading/writing
- get/set file meta-data, e.g., Unix `fstat, chmod`

File Read



`read(fileID, vaddr, length)`

File Position

- each file descriptor (open file) has an associated file position
- 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:
  - a seek operation (`lseek`) 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);
```

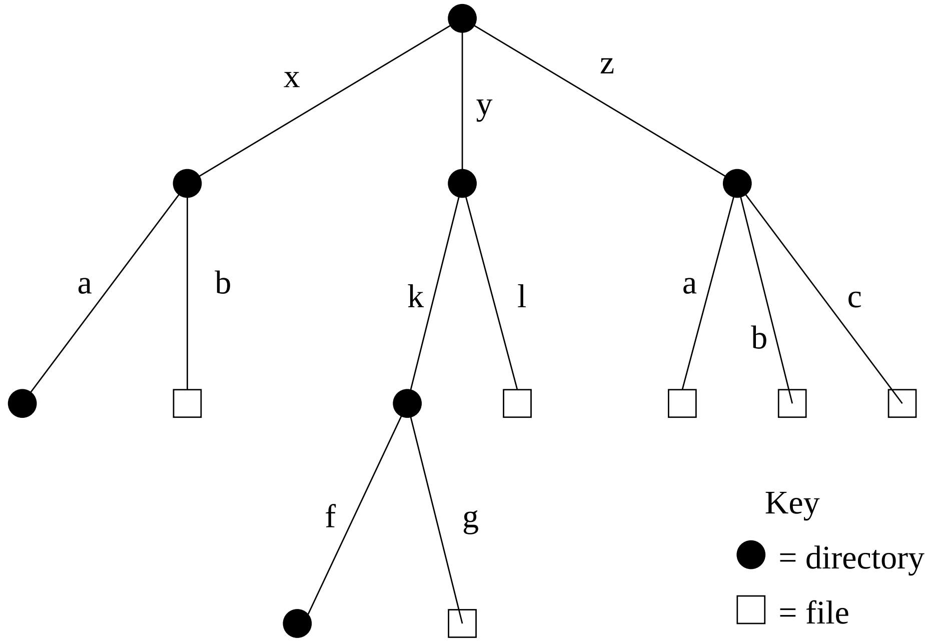
Read every second 512 byte chunk of a file, until 50 have been read.

Directories and File Names

- A directory maps *file names* (strings) to *i-numbers*
  - an i-number is a unique (within a file system) identifier for a file or directory
  - given an i-number, the file system can find the data and meta-data for the file
- Directories provide a way for applications to group related files
- Since directories can be nested, a filesystem's directories can be viewed as a tree, with a single *root* directory.
- In a directory tree, files are leaves
- Files may be identified by *pathnames*, which describe a path through the directory tree from the root directory to the file, e.g.:

  `/home/user/courses/cs350/notes/filesys.pdf`
- Directories also have pathnames
- Applications refer to files using pathnames, not i-numbers

Hierarchical Namespace Example



Hard Links

- a *hard link* is an association between a name (string) and an i-number
  - each entry in a directory is a hard link
- when a file is created, so is a hard link to that file
  - `open(/a/b/c,O_CREAT|O_TRUNC)`
  - this creates a new file if a file called `/a/b/c` does not already exist
  - it also creates a hard link to the file in the directory `/a/b`
- Once a file is created, *additional* hard links can be made to it.
  - example: `link(/x/b,/y/k/h)` creates a new hard link `h` in directory `/y/k`. The link refers to the i-number of file `/x/b`, which must exist.
- linking to an existing file creates a new pathname for that file
  - each file has a unique i-number, but may have multiple pathnames
- Not possible to `link` to a directory (to avoid cycles)

Unlinking and Referential Integrity

- hard links can be removed:
  - `unlink(/x/b)`
- the file system ensures that hard links have *referential integrity*, which means that if the link exists, the file that it refers to also exists.
  - When a hard link is created, it refers to an existing file.
  - There is no system call to delete a file. Instead, a file is deleted when its last hard link is removed.

Symbolic Links

- a *symbolic link*, or *soft link*, is an association between a name (string) and a pathname.
  - `symlink(/z/a,/y/k/m)` creates a symbolic link `m` in directory `/y/k`. The symbolic link refers to the pathname `/z/a`.
- If an application attempts to open `/y/k/m`, the file system will
  1. recognize `/y/k/m` as a symbolic link, and
  2. attempt to open `/z/a` instead
- referential integrity is *not* preserved for symbolic links
  - in the example above, `/z/a` need not exist!

UNIX/Linux Link Example (1 of 3)

```
% cat > file1
This is file1.
<cntl-d>
% ls -li
685844 -rw------- 1 user group 15 2008-08-20 file1
% ln file1 link1
% ln -s file1 sym1
% ln not-here link2
ln: not-here: No such file or directory
% ln -s not-here sym2
```

Files, hard links, and soft/symbolic links.

UNIX/Linux Link Example (2 of 3)

```
% ls -li
685844 -rw------- 2 user group 15 2008-08-20 file1
685844 -rw------- 2 user group 15 2008-08-20 link1
685845 lrwxrwxrwx 1 user group  5 2008-08-20 sym1 -> file1
685846 lrwxrwxrwx 1 user group  8 2008-08-20 sym2 -> not-here
% cat file1
This is file1.
% cat link1
This is file1.
% cat sym1
This is file1.
% cat sym2
cat: sym2: No such file or directory
% /bin/rm file1
```

Accessing and manipulating files, hard links, and soft/symbolic links.

UNIX/Linux Link Example (3 of 3)

```
% ls -li
685844 -rw------- 1 user group 15 2008-08-20 link1
685845 lrwxrwxrwx 1 user group  5 2008-08-20 sym1 -> file1
685846 lrwxrwxrwx 1 user group  8 2008-08-20 sym2 -> not-here
% cat link1
This is file1.
% cat sym1
cat: sym1: No such file or directory
% cat > file1
This is a brand new file1.
<cntl-d>
% ls -li
685847 -rw------- 1 user group 27 2008-08-20 file1
685844 -rw------- 1 user group 15 2008-08-20 link1
685845 lrwxrwxrwx 1 user group  5 2008-08-20 sym1 -> file1
685846 lrwxrwxrwx 1 user group  8 2008-08-20 sym2 -> not-here
% cat link1
This is file1.
% cat sym1
This is a brand new file1.
```

Different behaviour for hard links and soft/symbolic 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:

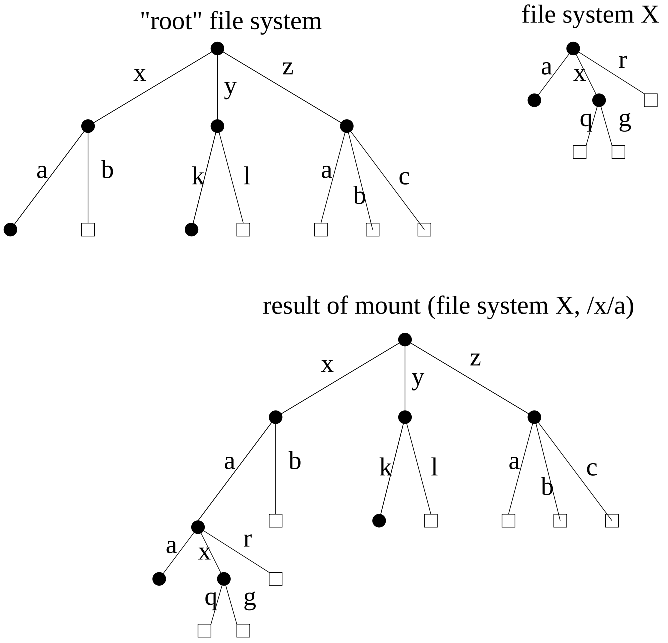
  DOS/Windows: use two-part file names: file system name, pathname within file system

  - example: `C:\user\cs350\schedule.txt`

  Unix: create single hierarchical namespace that combines the namespaces of two file systems

  - Unix `mount` system call does this
- mounting does *not* make two file systems into one file system
  - it merely creates a single, hierarchical namespace that combines the namespaces of two file systems
  - the new namespace is temporary - it exists only until the file system is unmounted

Unix `mount` Example



Links and Multiple File Systems

- hard links cannot cross file system boundaries
  - each hard link maps a name to an i-number, which is unique only *within* a file system
- 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

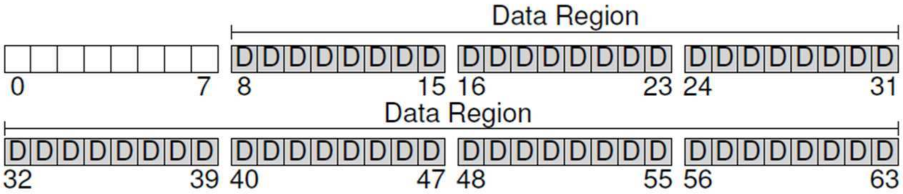
- what needs to be stored persistently?
  - file data
  - file meta-data
  - directories and links
  - file system meta-data
- non-persistent information
  - open files per process
  - file position for each open file
  - *cached* copies of persistent data

File System Example

- Use an extremely small disk as an example:
  - 256 KB disk!
  - Most disks have a sector size of 512 bytes
    - ∗ Memory is usually *byte addressable*
    - ∗ Disk is usually “sector addressable”
  - 512 total sectors on this disk
- Group every 8 consecutive sectors into a block
  - Better spatial locality (fewer seeks)
  - Reduces the number of block pointers (we’ll see what this means soon)
  - 4 KB block is a convenient size for demand paging
  - 64 total blocks on this disk

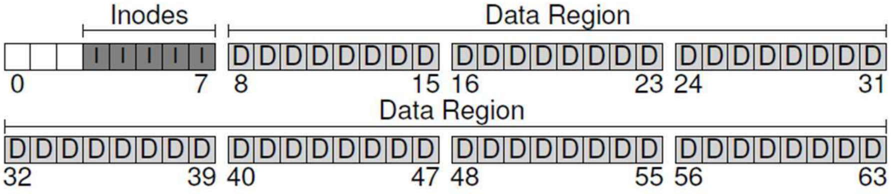
VSFS: Very Simple File System (1 of 5)

- Most of the blocks should be for storing user data (last 56 blocks)



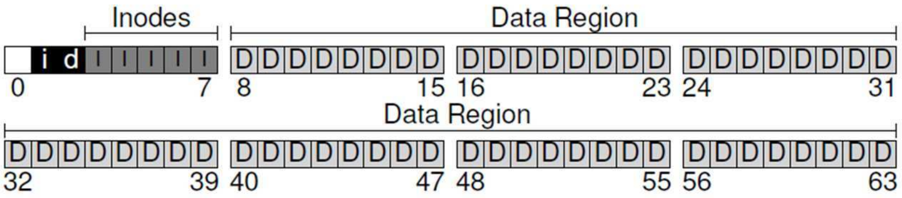
VSFS: Very Simple File System (2 of 5)

- Need some way to map files to data blocks
- Create an array of i-nodes, where each i-node contains the meta-data for a file
  - The index into the array is the file’s index number (i-number)
- Assume each i-node is 256 bytes, and we dedicate 5 blocks for i-nodes
  - This allows for 80 total i-nodes/files



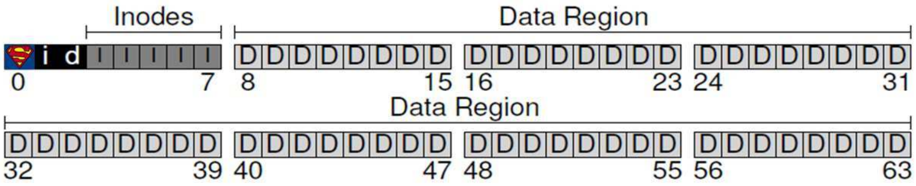
VSFS: Very Simple File System (3 of 5)

- We also need to know which i-nodes and blocks are unused
- Many ways of doing this:
  - In VSFS, we use a bitmap for each
  - Can also use a free list instead of a bitmap
- A block size of 4 KB means we can track 32K i-nodes and 32K blocks
  - This is far more than we actually need

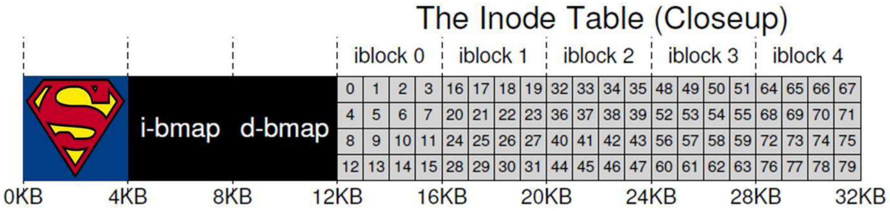


VSFS: Very Simple File System (4 of 5)

- Reserve the first block as the **superblock**
- A superblock contains meta-information about the entire file system
  - e.g., how many i-nodes and blocks are in the system, where the i-node table begins, etc.



VSFS: Very Simple File System (5 of 5)



i-nodes

- An i-node is a *fixed size* index structure that holds both file meta-data and a small number of pointers to data blocks
- i-node fields include:
  - file type
  - file permissions
  - file length
  - number of file blocks
  - time of last file access
  - time of last i-node update, last file update
  - number of hard links to this file
  - direct data block pointers
  - single, double, and triple indirect data block pointers

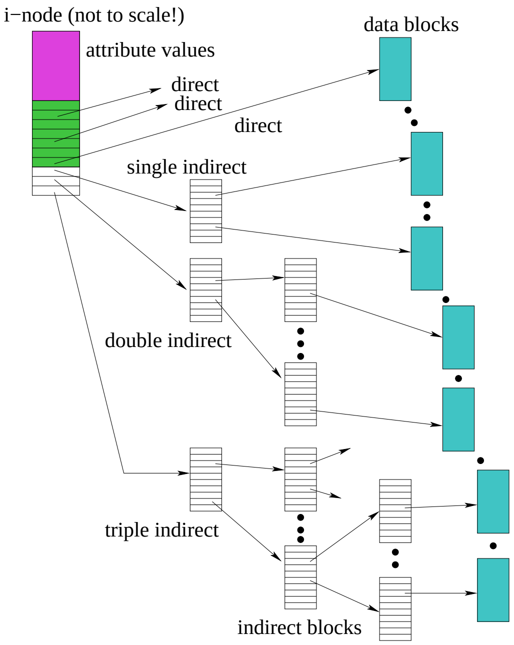
VSFS: i-node

- Assume disk blocks can be referenced based on a 4 byte address
  - 2^{32} blocks, 4 KB blocks
  - Maximum disk size is 16 TB
- In VSFS, an i-node is 256 bytes
  - Assume there is enough room for 12 direct pointers to blocks
  - Each pointer points to a different block for storing user data
  - Pointers are ordered: first pointer points to the first block in the file, etc.
- What is the maximum file size if we only have direct pointers?
  - 12 * 4 KB = 48 KB
- Great for small files (which are common)
- Not so great if you want to store big files

VSFS: Indirect Blocks

- In addition to 12 direct pointers, we can also introduce an **indirect pointer**
  - An indirect pointer points to a block full of direct pointers
- 4 KB block of direct pointers = 1024 pointers
  - Maximum file size is: (12 + 1024) * 4 KB = 4144 KB
- Better, but still not enough
- Add a **double indirect pointer**
  - Points to a 4 KB block of indirect pointers
  - (12 + 1024 + 1024 * 1024) * 4 KB
  - Just over 4 GB in size (is this enough?)

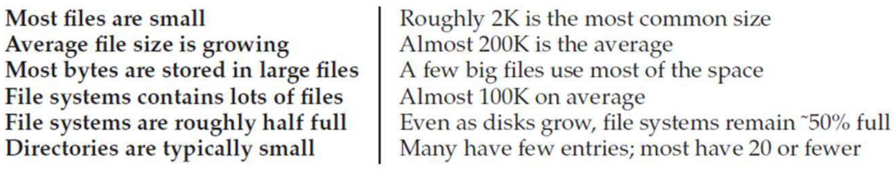
i-node Diagram



File System Design

- File system parameters:
  - *How many i-nodes should a file system have?*
  - *How many direct and indirect blocks should an i-node have?*
  - *What is the "right" block size?*

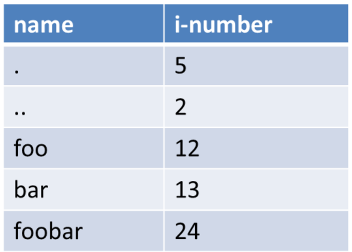
- For a general purpose file system, design it to be efficient for the common case



Most files are small	Roughly 2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of the space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain ˜50% full
Directories are typically small	Many have few entries; most have 20 or fewer

Directories

- Implemented as a special type of file.
- Directory file contains directory entries, each consisting of
  - a file name (component of a path name) and the corresponding i-number



name	i-number
.	5
..	2
foo	12
bar	13
foobar	24

- Directory files can be read by application programs (e.g., `ls`)
- Directory files are only updated by the kernel, in response to file system operations, e.g, create file, create link
- Application programs cannot write directly to directory files. (Why not?)

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:

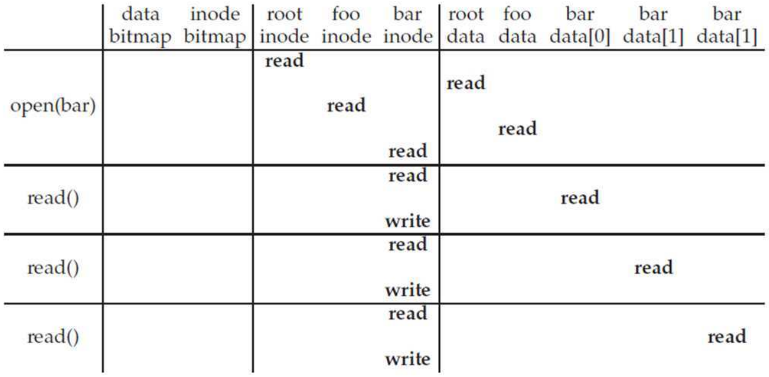
  `symlink(/y/k/g,/z/m)`
- to implement this:
  - create a new *symlink* file
  - add a new entry in directory `/z`
    - file name in new entry is `m`
    - i-number in the new entry is the i-number of the new symlink file
  - store the pathname string “/y/k/g” as the contents of the new symlink file

Free Space Management

- Use the bitmaps to find a free i-node and free blocks
  - Each bit represents the availability of an i-node or block
- There are often many free blocks to choose from
  - To improve spatial locality and reduce fragementation, a file system may want to select a free block that is followed by a sequence of other free blocks

Reading From a File (/foo/bar)

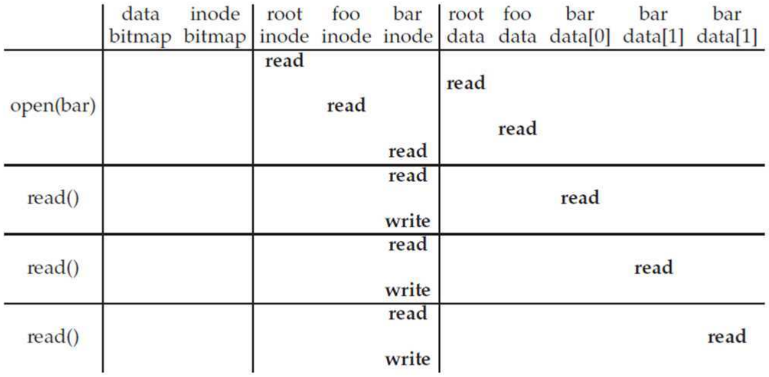
- First read the root i-node
  - At “well known” position (i-node 2)
  - i-node 1 is usually for tracking bad locks



	data	inode	root	foo	bar	root	foo	bar	bar	bar
	bitmap	bitmap	inode	inode	inode	data	data	data[0]	data[1]	data[1]
open(bar)			read							
						read				
				read						
							read			
					read					
read()					read					
								read		
					write					
read()					read					
									read	
					write					
read()					read					
										read
					write					

Reading From a File (/foo/bar)

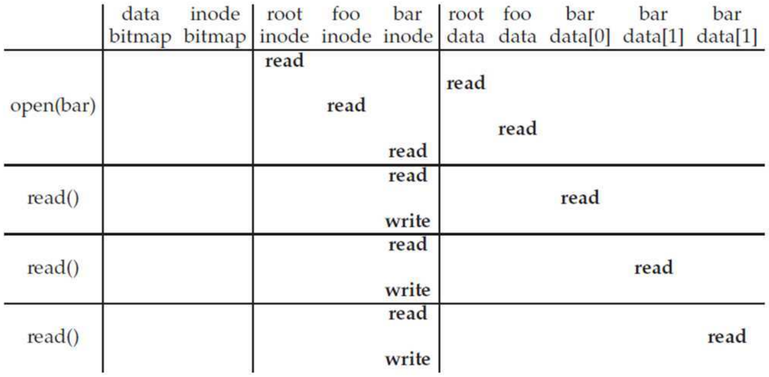
- Read the directory information from root
  - Find the i-number for foo
  - Read the foo i-node



	data	inode	root	foo	bar	root	foo	bar	bar	bar
	bitmap	bitmap	inode	inode	inode	data	data	data[0]	data[1]	data[1]
open(bar)			read							
						read				
				read						
							read			
					read					
read()					read					
								read		
					write					
read()					read					
									read	
					write					
read()					read					
										read
					write					

Reading From a File (/foo/bar)

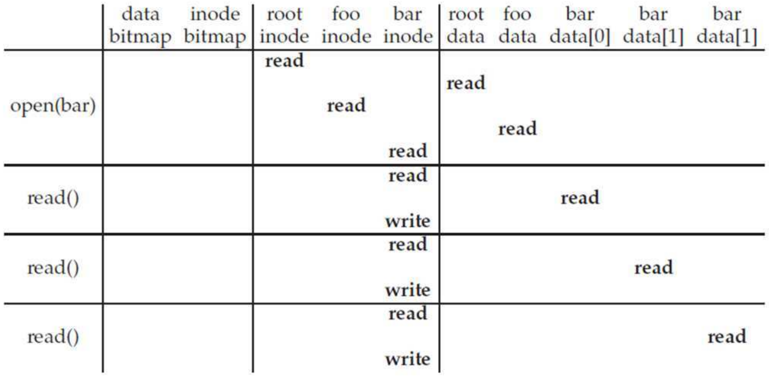
- Read the directory information from foo
  - Find the i-number for bar
  - Read the bar i-node



	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[1]
open(bar)			read							
						read				
				read						
							read			
					read					
read()					read					
								read		
					write					
read()					read					
									read	
					write					
read()					read					
										read
					write					

Reading From a File (/foo/bar)

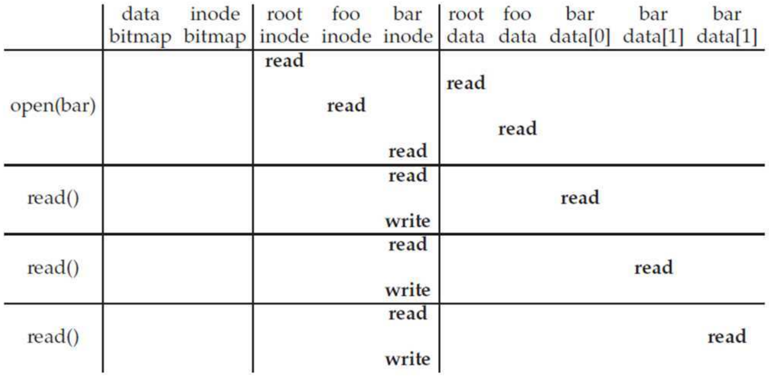
- Permission check (is the user allowed to read this file?)
- Allocate a file descriptor in the per-process descriptor table
- Increment the counter for this i-number in the global open file table



	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[1]
open(bar)			read							
						read				
				read						
							read			
					read					
read()					read					
								read		
					write					
read()					read					
									read	
					write					
read()					read					
										read
					write					

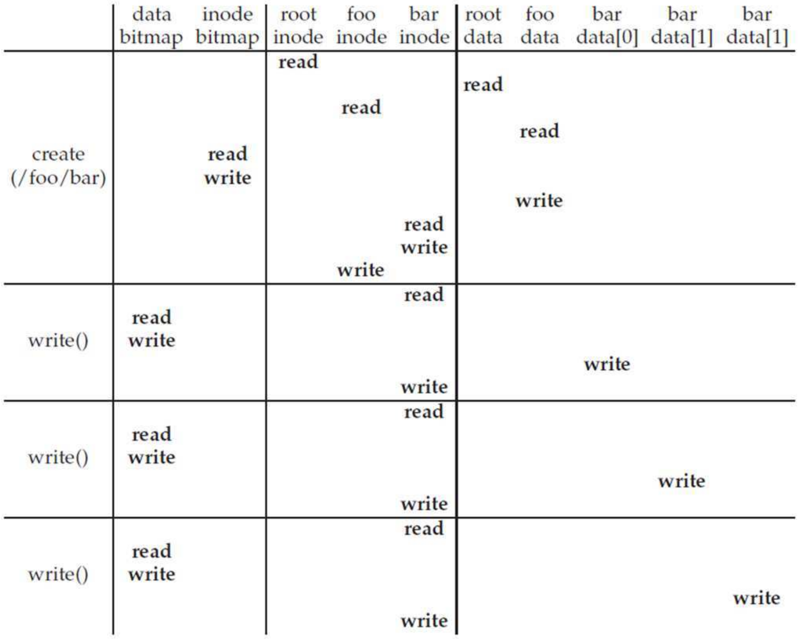
Reading From a File (/foo/bar)

- Find the block using a direct/indirect pointer and read the data
- Update the i-node with a new access time
- Update the file position in the per-process descriptor table
- Closing a file deallocates the file descriptor and decrements the counter for this i-number in the global open file table



	data	inode	root	foo	bar	root	foo	bar	bar	bar
	bitmap	bitmap	inode	inode	inode	data	data	data[0]	data[1]	data[1]
open(bar)			read							
						read				
				read						
							read			
					read					
read()					read					
								read		
					write					
read()					read					
									read	
					write					
read()					read					
										read
					write					

Creating a File (/foo/bar)



	data	inode	root	foo	bar	root	foo	bar	bar	bar
	bitmap	bitmap	inode	inode	inode	data	data	data[0]	data[1]	data[1]
create (/foo/bar)			read							
						read				
				read						
							read			
		read								
		write								
							write			
					read					
					write					
				write						
write()					read					
	read									
	write									
								write		
					write					
write()					read					
	read									
	write									
									write	
					write					
write()					read					
	read									
	write									
										write
					write					

In-Memory (Non-Persistent) Structures

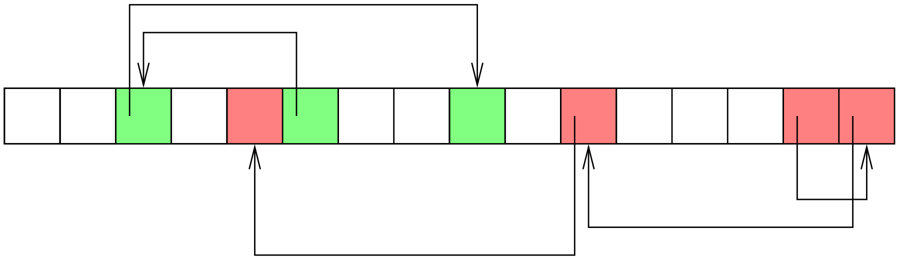
- per process
  - descriptor table
    - which file descriptors does this process have open?
    - to which file does each open descriptor refer?
    - what is the current file position for each descriptor?
- system wide
  - open file table
    - which files are currently open (by any process)?
  - i-node cache
    - in-memory copies of recently-used i-nodes
  - block cache
    - in-memory copies of data blocks and indirect blocks

Chaining

- VSFS uses a per-file index (direct and indirect pointers) to access blocks
- Two alternative approaches:
  - Chaining:
    - Each block includes a pointer to the next block
  - External chaining:
    - The chain is kept as an external structure
    - Microsoft's File Allocation Table (FAT) uses external chaining

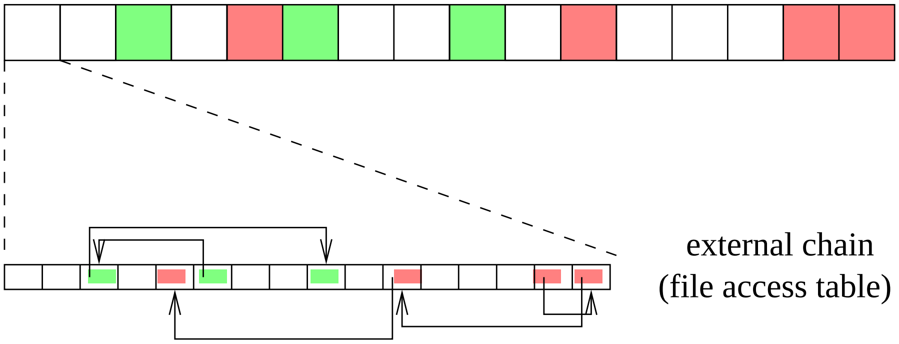
Chaining

- Directory table contains the name of the file, and each file's starting block
- Acceptable for sequential access, very slow for random access (why?)



External Chaining

- Introduces a special file access table that specifies all of the file chains



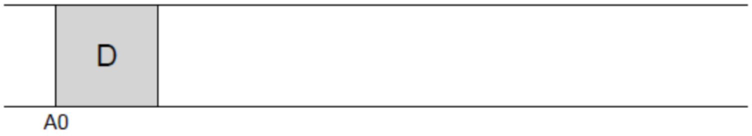
external chain
(file access table)

Log-Structured File System (LFS)

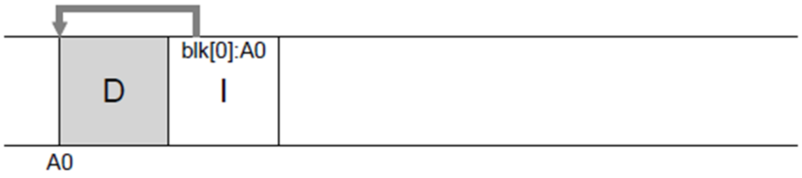
- LFS is built on a very different set of assumptions:
  - Memory sizes are growing
    - Most reads will be served from cache
    - Reads therefore are fast and do not require any seeks
  - Large gap between random I/O and sequential I/O performance
- Main idea: Make all writes sequential writes

Log-Structured File System

- Write data block D at address A0



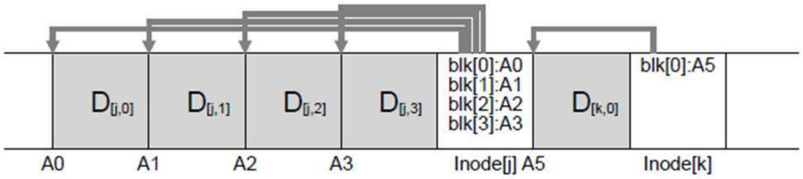
- Write i-node I for the file and point it to block D



- The next block write will be placed after i-node I
- To overwrite data, just write a **new** block and i-node at the next position

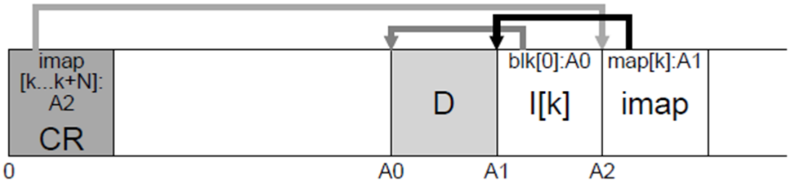
Log-Structured File System

- Write buffering is used to collect multiple writes to the same file together
  - Reduces the number of i-nodes that need to be written



Log-Structured File System

- Problem:
  - How do we find the i-node for a file on a cache miss?
- i-node map (imap) takes an i-node number as an input, and returns the disk address of the most recent version of the i-node
- Write a portion of the i-node map every time an i-node is updated



- How do we find the i-node maps?
  - Maintain a periodically updated checkpoint region with pointers to the latest imaps
  - Checkpoint region always updated during graceful shutdown

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?

- system failure will destroy in-memory file system structures
- persistent structures should be *crash consistent*, i.e., should be consistent when system restarts after a failure

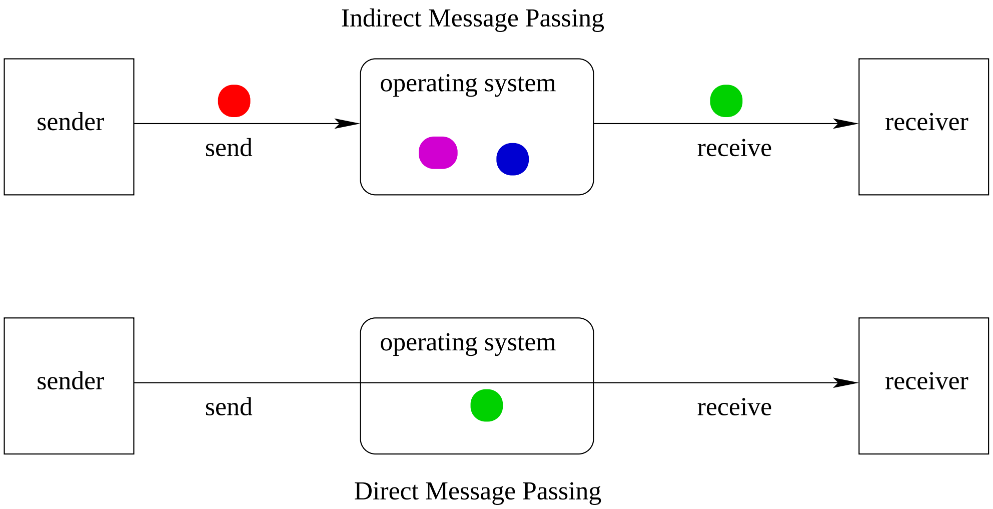
Fault Tolerance

- 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
  - shared virtual memory
  - shared files
- message-based
  - sockets
  - pipes
  - signals
  - . . .

Message Passing



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

Properties of Message Passing Mechanisms

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

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? reordered? 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.

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)

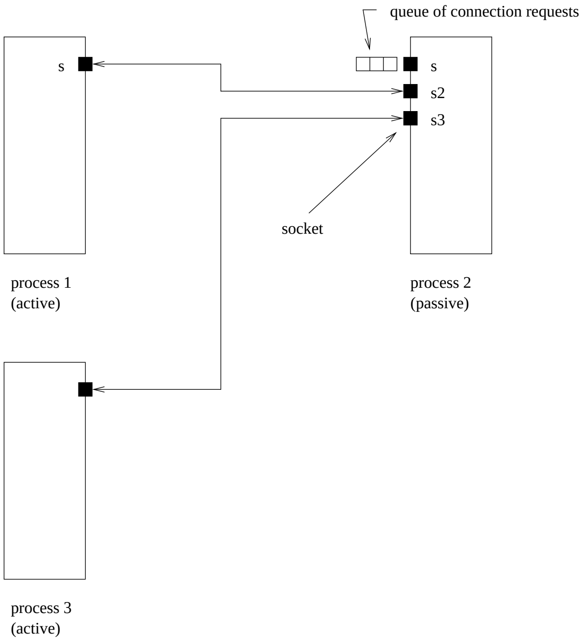
- `accept` creates a new socket (`ns`) for the new connection
- `sourceAddress` is an address buffer. `accept` fills it with the address of the socket that has made the connection request
- additional connection requests can be accepted using more `accept` calls on the original socket (`s`)
- `accept` 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



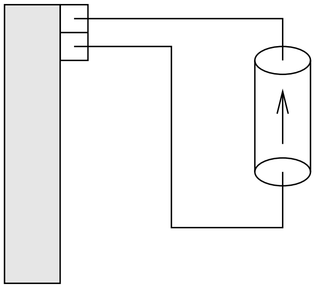
Pipes

- 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

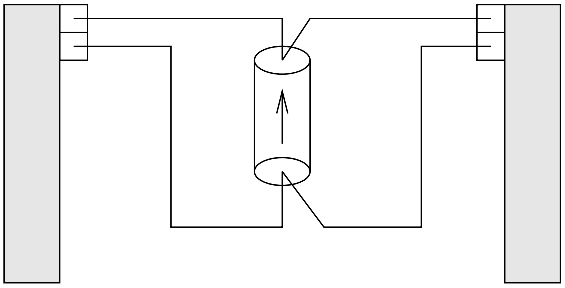
```
int fd[2];
char m[] = "message for parent";
char y[100];
pipe(fd); // create pipe
pid = fork(); // create child process
if (pid == 0) {
  // child executes this
  close(fd[0]); // close read end of pipe
  write(fd[1],m,19);
  ...
} else {
  // parent executes this
  close(fd[1]); // close write end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

Illustration of Example (after `pipe()`)



parent process

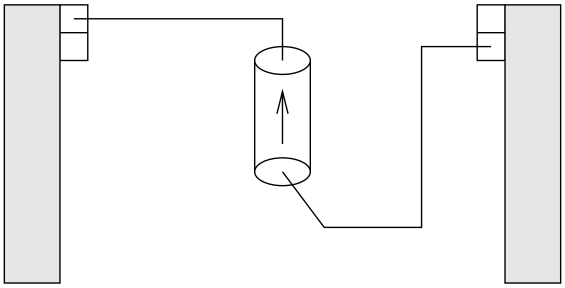
Illustration of Example (after `fork()`)



parent process

child process

Illustration of Example (after `close()`)



parent process

child process

Examples of Other Interprocess Communication Mechanisms

named pipe:

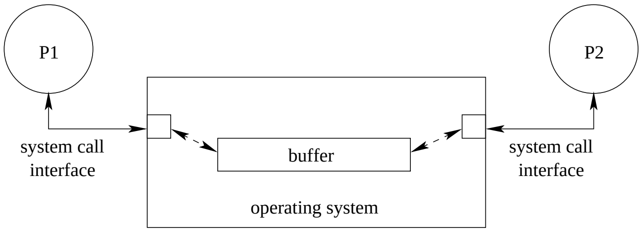
- similar to pipes, but with an associated name (usually a file name)
- name allows arbitrary processes to communicate by opening the same named pipe
- must be explicitly deleted, unlike an unnamed pipe

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

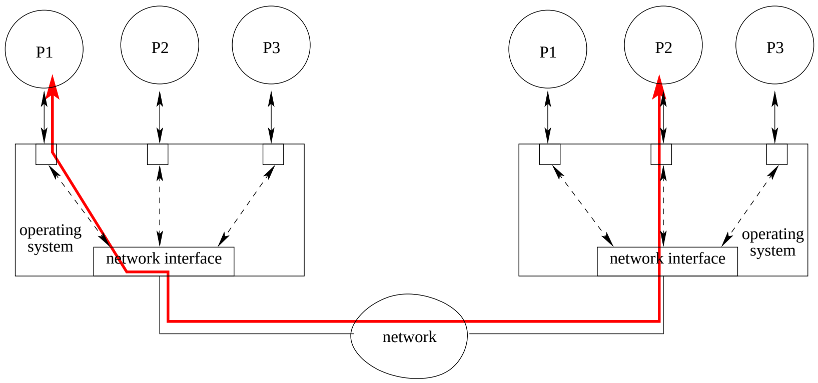
Implementing IPC

- 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



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`