

CS 350
Operating Systems
Course Notes

Spring 2007

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

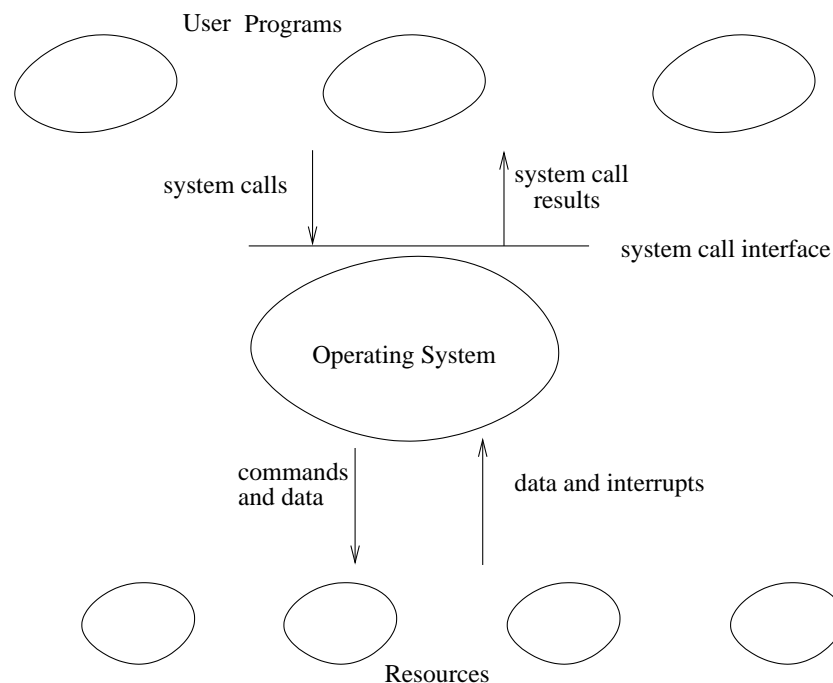
System View: The OS manages the hardware resources of a computer system.

- 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.

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
- Processes and Threads
- Concurrency and Synchronization
- Address Spaces and Virtual Memory
- Scheduling
- Devices and Device Management
- File Systems
- Interprocess Communication and Networking
- Security

What is a Process?

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

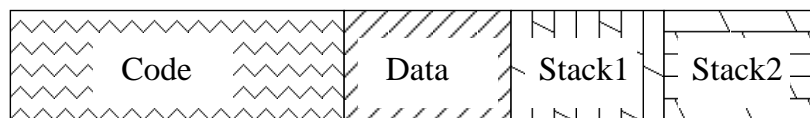
Answer 2: a process consists of

- an address space
- 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.

What is an Address Space?

- For now, think of an address space as a portion of the primary memory of the machine that is used to hold the code, data, and stack(s) of the running program.
- For example:



0 —————> max
addresses

- We will elaborate on this later.

What is a Thread?

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

The Operating System and the Kernel

- We will use the following 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

The OS Kernel

- Usually kernel code runs in a privileged execution mode, while the rest of the operating system does not.
- The kernel is a program. It has code and data like any other program.
- For now, think of the kernel as a program that resides in its own address space, separate from the address spaces of processes that are running on the system. Later, we will elaborate on the relationship between the kernel's address space and process address spaces.

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

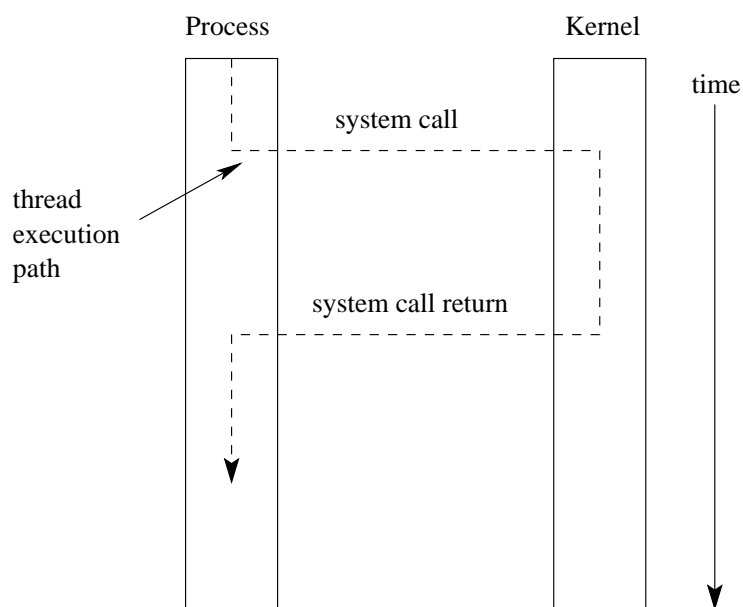
How System Calls Work

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

System Call Diagram



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 error, e.g, overflow
 - illegal instruction
 - memory protection violation
 - page fault (to be discussed later)
- exceptions are detected by the hardware

Exceptions (cont'd)

- when an exception occurs, control is transferred (by the hardware) to a fixed address in the kernel
- transfer of control happens in much the same way as it does for a system call. (In fact, a system call can be thought of as a type of exception, and they are sometimes implemented that way.)
- in the kernel, an exception handler determines which exception has occurred and what to do about it. For example, it may choose to destroy a process that attempts to execute an illegal instruction.

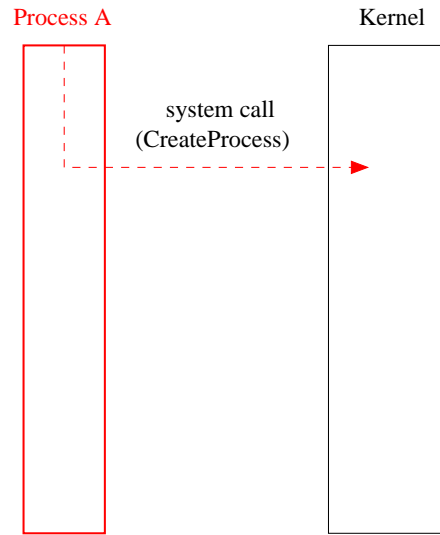
Interrupts

- 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.

Implementation of Processes

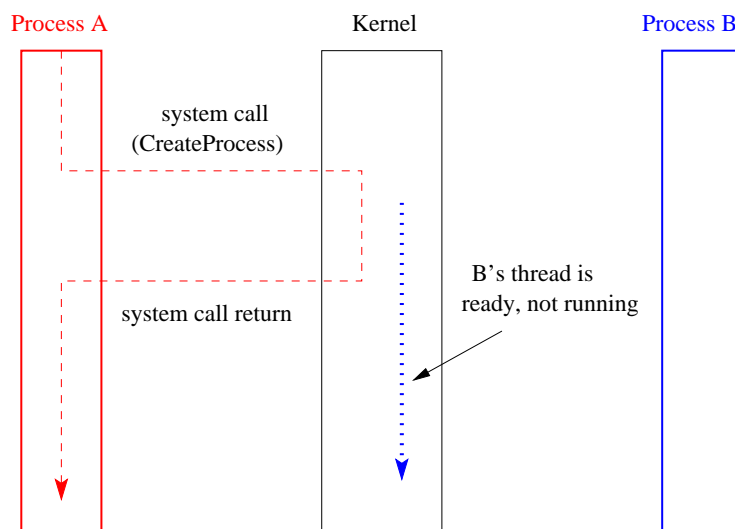
- The kernel maintains information about all of the processes in the system in a data structure often called the process table.
- Information about individual processes is stored in a structure that is sometimes called a *process control block (PCB)*. In practice, however, information about a process may not all be located in a single data structure.
- Per-process information may include:
 - process identifier and owner
 - current process state and other scheduling information
 - lists of available resources, such as open files
 - accounting information
 - and more

Process Creation Example (Part 1)



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

Process Creation Example (Part 2)

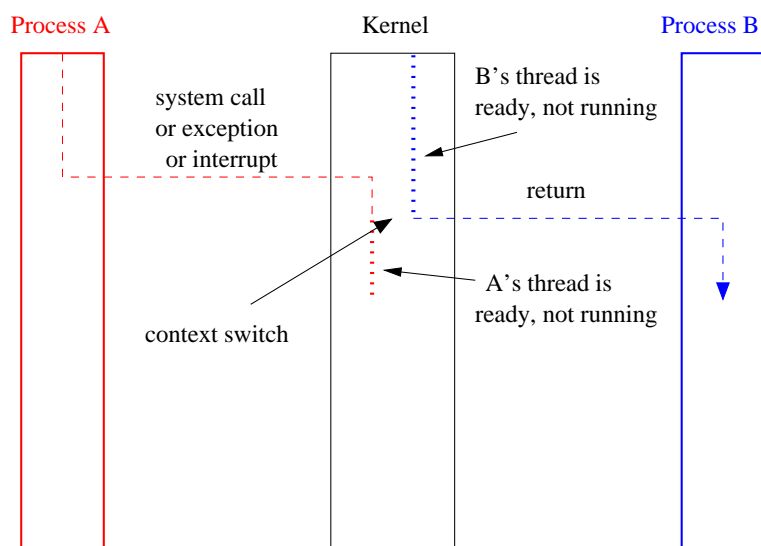


Kernel creates new process (Process B)

Multiprogramming

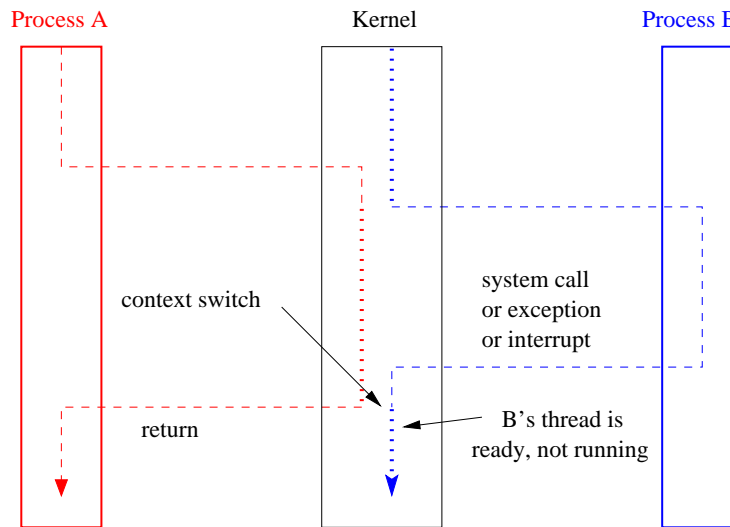
- multiprogramming means having multiple processes existing at the same time
- most modern, general purpose operating systems support multiprogramming
- all processes share the available hardware resources, with the sharing coordinated by the operating system:
 - Each process uses some of the available 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.

Timesharing Example (Part 1)



Kernel switches execution context to Process B.

Timesharing Example (Part 2)



Kernel switches execution context back to process A.

Summary of Hardware Features Used by the Kernel

Interrupts and Exceptions, such as timer interrupts, give the kernel the opportunity to regain control from user programs.

Memory management features, such as memory protection, allow the kernel to protect its address space from user programs.

Privileged execution mode allows the kernel to reserve critical machine functions (e.g, halt) for its own use.

Independent I/O devices allow the kernel to schedule other work while I/O operations are on-going.

Process Interface

- A running program may use process-related system calls to manipulate its own process, or other processes in the system.
- The process interface will usually include:
 - Creation:** make new processes, e.g., `Exec` in Nachos
 - Destruction:** terminate a process, e.g., `Exit` in Nachos
 - Synchronization:** wait for some event, e.g., `Join` in Nachos
 - Attribute Mgmt:** read or change process attributes, such as the process identifier or owner or scheduling priority

The Process Model

- Although the general operations supported by the process interface are straightforward, there are some less obvious aspects of process behaviour that must be defined by an operating system.
 - Process Initialization:** When a new process is created, how is it initialized? What is in the address space? What is the initial thread context? Does it have any other resources?
 - Multithreading:** Are concurrent processes supported, or is each process limited to a single thread?
 - Inter-Process Relationships:** Are there relationships among processes, e.g., parent/child? If so, what do these relationships mean?

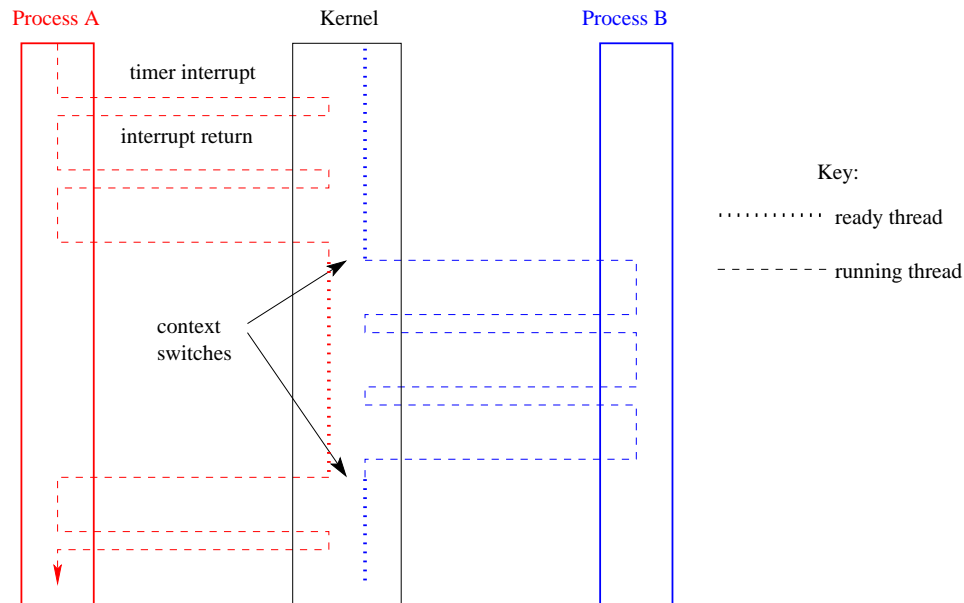
Processor Scheduling Basics

- Only one thread at a time can run on a processor.
- Processor scheduling means deciding how threads should share the available processor(s)
- Round-robin is a simple *preemptive* scheduling policy:
 - the kernel maintains a list of *ready* threads
 - the first thread on the list is *dispatched* (allowed to run)
 - when the running thread has run for a certain amount of time, called the scheduling quantum, it is *preempted*
 - the preempted thread goes to the back of the ready list, and the thread at the front of the list is dispatched.
- More on scheduling policies later.

Implementing Preemptive Scheduling

- 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.
- All interrupts transfer control from the running program to the kernel.
- In the case of a timer interrupt, this transfer of control gives the kernel the opportunity to preempt the running thread and dispatch a new one.

Preemptive Multiprogramming Example



Blocked Threads

- Sometimes a thread will need to wait for an event. Examples:
 - wait for data from a (relatively) slow disk
 - wait for input from a keyboard
 - wait for another thread to leave a critical section
 - wait for busy device to become idle
- The OS scheduler should only allocate the processor to threads that are not blocked, since blocked threads have nothing to do while they are blocked.

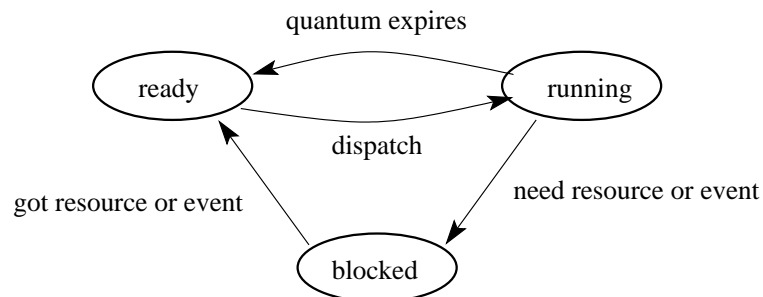
Multiprogramming makes it easier to keep the processor busy even though individual threads are not always ready.

Implementing Blocking

- The need for waiting normally arises during the execution of a system call by the thread, since programs use devices through the kernel (by making system calls).
- When the kernel recognizes that a thread faces a delay, it can *block* that thread. This means:
 - mark the thread as blocked, don't put it on the ready queue
 - choose a ready thread to run, and dispatch it
 - when the desired event occurs, put the blocked thread back on the ready queue so that it will (eventually) be chosen to run

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.

User-Level Threads

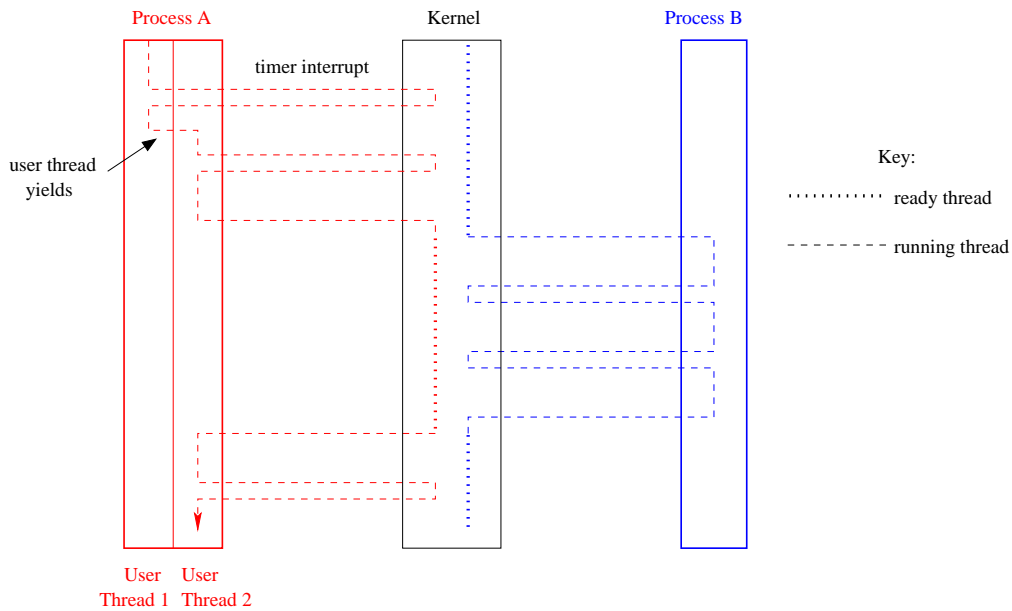
- It is possible to implement threading at the user level.
- This means threads are implemented outside of the kernel, within a process.
- Call these *user-level threads* to distinguish them from *kernel threads*, which are those implemented by the kernel.
- A user-level thread library will include procedures for
 - creating threads
 - terminating threads
 - yielding (voluntarily giving up the processor)
 - synchronization

In other words, similar operations to those provided by the operating system for kernel threads.

User-Level and Kernel Threads

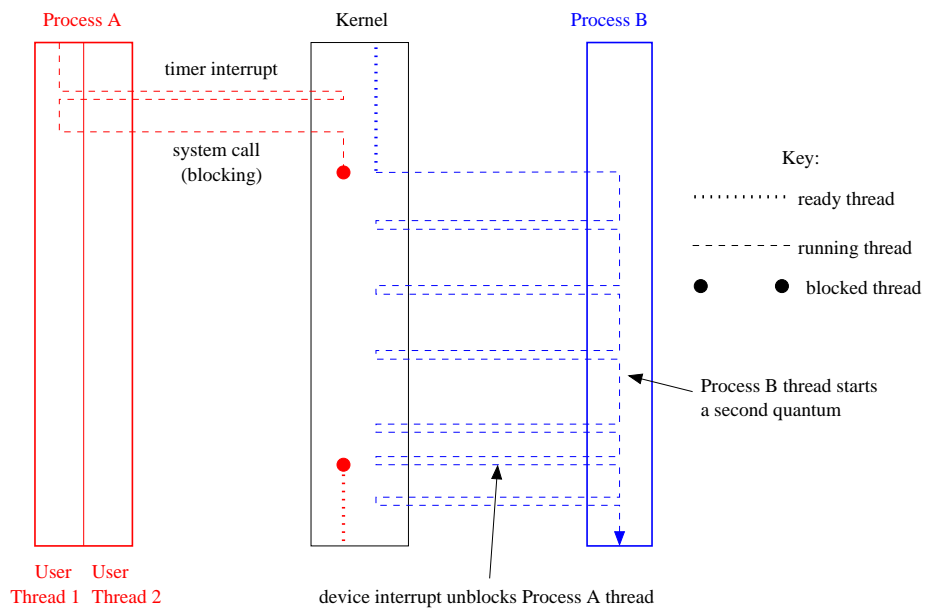
- There are two general ways to implement user-level threads
 1. Multiple user-level thread contexts in a process with one kernel thread. (N:1)
 - Kernel thread can “use” only one user-level thread context at a time.
 - Switching between user threads in the same process is typically non-preemptive.
 - Blocking system calls block the kernel thread, and hence all user threads in that process.
 - Can only use one CPU.
 2. Multiple user-level thread contexts in a process with multiple kernel threads. (N:M)
 - Each kernel thread “uses” one user-level thread context.
 - Switching between threads in the same process can be preemptive.
 - Process can make progress if at least one of its kernel threads is not blocked.
 - Can use multiple CPUs.

Two User Threads, One Kernel Thread (Part 1)



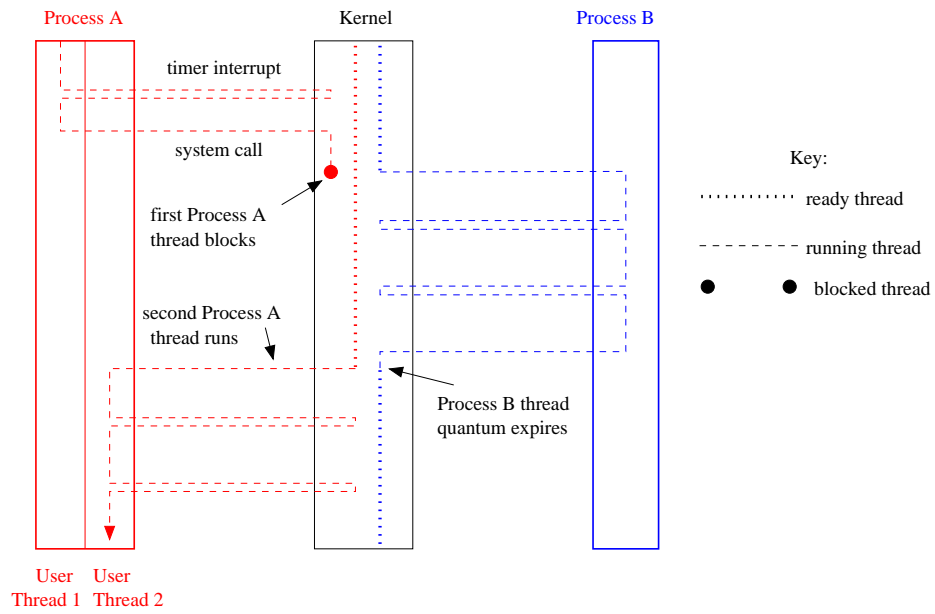
Process A has two user-level threads, but only one kernel thread.

Two User Threads, One Kernel Thread (Part 2)



Once Process A's thread blocks, only Process B's thread can run.

Two User Threads, Two Kernel Threads



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.
 - Threads in the same process share access to program variables in their process's address space.
- A common synchronization problem is to enforce *mutual exclusion*, which means making sure that only one thread at a time uses a shared object, e.g., a variable or a device.
- The part of a program in which the shared object is accessed is called a *critical section*.

Critical Section Example (Part 1)

```
int IntList::RemoveFront() {
    ListElement *element = first;
    ASSERT(!IsEmpty());
    int num = first->item;
    if (first == last) { first = last = NULL; }
    else { first = element->next; }
    numInList--;
    delete element;
    return num;
}
```

The RemoveFront method is a critical section. It may not work properly if two threads call it at the same time on the same IntList. (Why?)

Critical Section Example (Part 2)

```
void IntList::Append(int item) {
    ListElement *element = new ListElement(item);
    ASSERT(!IsInList(item));
    if (IsEmpty()) {
        first = element; last = element;
    } else {
        last->next = element; last = element;
    }
    numInList++;
}
```

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

Peterson's Mutual Exclusion Algorithm

```
boolean flag[2]; /* shared, initially false */
int turn;        /* shared */

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

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

flag[i] = false;
```

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

Mutual Exclusion Using Special Instructions

- Software solutions to the critical section problem (e.g., Peterson's algorithm) assume only atomic load and atomic store.
- Simpler algorithms are possible if more complex *atomic* operations are supported by the hardware. For example:
 - Test and Set:** set the value of a variable, and return the old value
 - Swap:** swap the values of two variables
- On uniprocessors, mutual exclusion can also be achieved by disabling interrupts during the critical section. (Normally, user programs cannot do this, but the kernel can.)

Mutual Exclusion with Test and Set

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

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

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

lock = false;
```

Works for any number of threads, but starvation is a possibility.

Semaphores

- A semaphore is a synchronization primitive that can be used to solve the critical section problem, and many other synchronization problems too
- 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 Binary Semaphore

```
binarySemaphore s; /* initial value is 1 */  
  
P(s);  
  
    critical section /* e.g., call to RemoveFront */  
  
V(s);
```

Producer/Consumer Using a Counting Semaphore

```
countingSemaphore s; /* initial value is 0 */  
item buffer[infinite]; /* huge buffer, initially empty */
```

Producer's Pseudo-code:

```
    add item to buffer  
    V(s);
```

Consumer's Pseudo-code:

```
    P(s);  
    remove item from buffer
```

If mutual exclusion is required for adding and removing items from the buffer, this can be provided using a second semaphore. (How?)

Producer/Consumer with a Bounded Buffer

```
countingSemaphore full; /* initial value is 0 */
countingSemaphore empty; /* initial value is N */
item buffer[N]; /* buffer with capacity N */
```

Producer's Pseudo-code:

```
P(empty);
  add item to buffer
V(full);
```

Consumer's Pseudo-code:

```
P(full);
  remove item from buffer
V(empty);
```

Implementing Semaphores

```
void P(s) {
  start critical section
  while (s == 0) { /* busy wait */
    end critical section
    start critical section }
  s = s - 1;
  end critical section }
```

```
void V(s) {
  start critical section
  s = s + 1;
  end critical section }
```

Any mutual exclusion technique can be used to protect the critical sections. However, starvation is possible with this implementation.

Implementing Semaphores in the Kernel

- Semaphores can be implemented at user level, e.g., as part of a user-level thread library.
- Semaphores can also be implemented by the kernel:
 - for its own use, for synchronizing threads in the kernel
 - for use by application programs, if a semaphore system call interface is provided
- As an optimization, semaphores can be integrated with the thread scheduler (easy to do for semaphores implemented in the kernel):
 - threads can be made to block, rather than busy wait, in the P operation
 - the V operation can make blocked threads ready

Nachos Semaphore Class

```
class Semaphore {
public:
    Semaphore(char* debugName, int initialValue);
    ~Semaphore();
    char* getName() { return name;}
    void P();
    void V();
    void SelfTest();
private:
    char* name;    // useful for debugging
    int value;    // semaphore value, always >= 0
    List<Thread *> *queue;
};
```

Nachos Semaphore P()

```
void Semaphore::P() {
    Interrupt *interrupt = kernel->interrupt;
    Thread *currentThread = kernel->currentThread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    if(value <= 0) {
        queue->Append(currentThread);
        currentThread->Sleep(FALSE);
    } else { value--; }
    (void) interrupt->SetLevel(oldLevel);
}
```

Nachos Semaphore V()

```
void Semaphore::V() {
    Interrupt *interrupt = kernel->interrupt;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    if (!queue->IsEmpty()) {
        kernel->scheduler->ReadyToRun(queue->RemoveFront());
    } else { value++; }
    (void) interrupt->SetLevel(oldLevel);
}
```

Monitors

- a monitor is a programming language construct that supports synchronized access to data
- a monitor is essentially an object for which
 - object state is accessible only through the object's methods
 - only one method may be active at a time
- if two threads attempt to execute methods at the same time, one will be blocked until the other finishes
- inside a monitor, so called *condition variables* can be declared and used

Condition Variable

- a condition variable is an object that support two operations:
 - wait:** causes the calling thread to block, and to release the monitor
 - signal:** if threads are blocked on the signaled condition variable then unblock one of them, otherwise do nothing
- a thread that has been unblocked by *signal* is outside of the monitor and it must wait to re-enter the monitor before proceeding.
- in particular, it must wait for the thread that signalled it

This describes Mesa-type monitors. There are other types on monitors, notably Hoare monitors, with different semantics for `wait` and `signal`.

Bounded Buffer Using a Monitor

```
item buffer[N]; /* buffer with capacity N */
int count; /* initially 0 */
condition notfull, notempty;

Produce(item) {
    while (count == N) { wait(notfull); }
    add item to buffer
    count = count + 1;
    signal(notempty);
}
```

Produce is implicitly executed atomically, because it is a monitor method.

Bounded Buffer Using a Monitor (cont'd)

```
Consume(item) {
    while (count == 0) { wait(notempty); }
    remove item from buffer
    count = count - 1;
    signal(notfull);
}
```

Consume is implicitly executed atomically, because it is a monitor method. Notice that while, rather than if, is used in both Produce and Consume. This is important. (Why?)

Simulating Monitors with Semaphores and Condition Variables

- Use a single binary semaphore (or Nachos “Lock”) to provide mutual exclusion.
- Each method must start by acquiring the mutex semaphore, and must release it on all return paths.
- Signal only while holding the mutex semaphore.
- Re-check the wait condition after each wait.
- Return only (the values of) variables that are local to the method.

Producer Implemented with Locks and Condition Variables (Example)

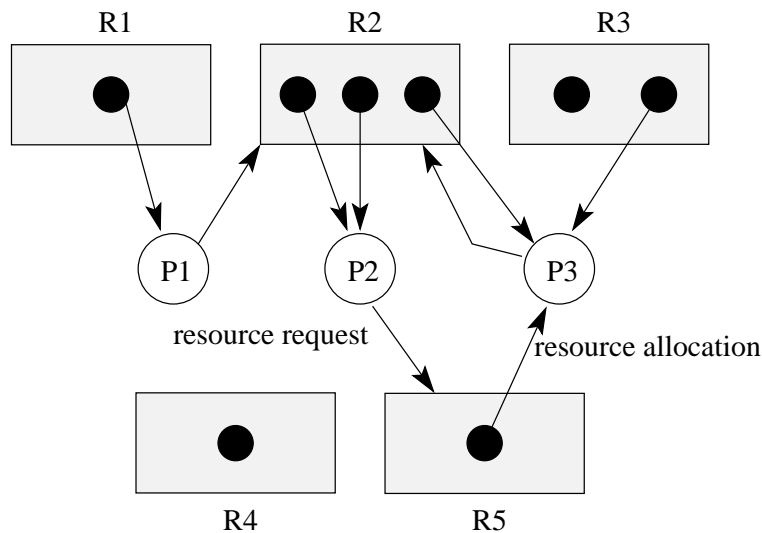
```
item buffer[N]; /* buffer with capacity N */
int count; /* initially 0 */
Condition notfull, notempty;
Lock mutex; /* for mutual exclusion */
Produce(item) {
    mutex.Acquire();
    while (count == N) {
        notfull.Wait(mutex);
    }
    add item to buffer
    count = count + 1;
    notempty.Signal(mutex);
    mutex.Release();
}
```


Deadlocks

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

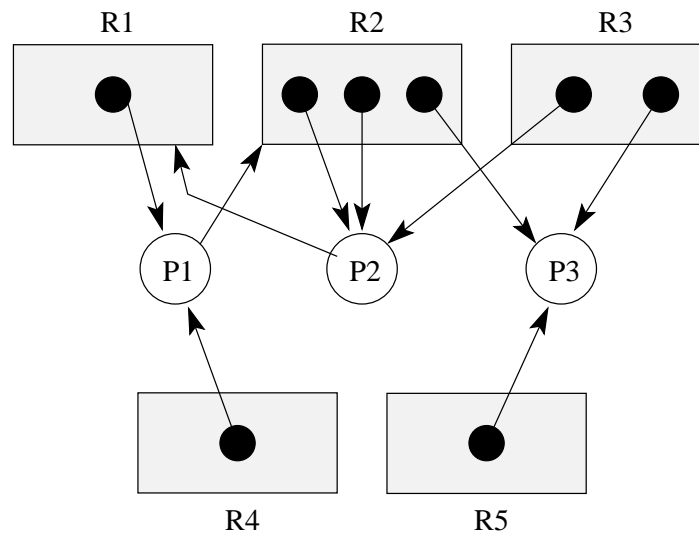
These two processes are *deadlocked* - neither process can make progress. Waiting will not resolve the deadlock. The processes are permanently stuck.

Resource Allocation Graph (Example)



Is there a deadlock in this system?

Resource Allocation Graph (Another Example)



Is there a deadlock in this system?

Deadlock Prevention

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

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

Resource Ordering: Order (e.g., number) the resource types, and require that each process acquire resources in increasing resource type order. That is, a process may make no requests for resources of type less than or equal to i once the process has requested 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 processes involved in the deadlock
- when to test for deadlocks? Can test on every blocked resource request, or can simply test periodically. Deadlocks persist, so periodic detection will not “miss” them.

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

Detecting Deadlock in a Resource Allocation Graph

- System State Notation:
 - R_i : request vector for process P_i
 - A_i : current allocation vector for process P_i
 - U : unallocated (available) resource vector
- Additional Algorithm Notation:
 - T : scratch resource vector
 - f_i : algorithm is finished with process P_i ? (boolean)

Detecting Deadlock (cont'd)

```

/* initialization */
T = U
fi is false if  $A_i > 0$ , else true
/* can each process finish? */
while  $\exists i ( \neg f_i \wedge (R_i \leq T) )$  {
    T = T +  $A_i$ ;
     $f_i = \text{true}$ 
}
/* if not, there is a deadlock */
if  $\exists i ( \neg f_i )$  then report deadlock
else report no deadlock

```

Deadlock Detection, Positive Example

- $R_1 = (0, 1, 0, 0, 0)$
- $R_2 = (0, 0, 0, 0, 1)$
- $R_3 = (0, 1, 0, 0, 0)$
- $A_1 = (1, 0, 0, 0, 0)$
- $A_2 = (0, 2, 0, 0, 0)$
- $A_3 = (0, 1, 1, 0, 1)$
- $U = (0, 0, 1, 1, 0)$

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

Deadlock Detection, Negative Example

- $R_1 = (0, 1, 0, 0, 0)$
- $R_2 = (1, 0, 0, 0, 0)$
- $R_3 = (0, 0, 0, 0, 0)$
- $A_1 = (1, 0, 0, 1, 0)$
- $A_2 = (0, 2, 1, 0, 0)$
- $A_3 = (0, 1, 1, 0, 1)$
- $U = (0, 0, 0, 0, 0)$

This system is not in deadlock. It is possible that the processes will run to completion in the order P_3, P_1, P_2 .

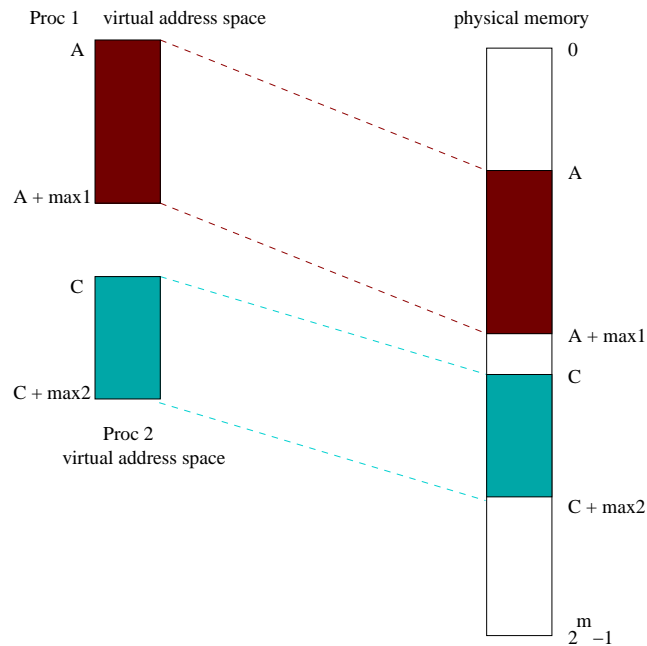
Virtual and Physical Addresses

- Physical addresses are provided directly by the machine.
 - one physical address space per machine
 - addresses typically range from 0 to some maximum, though some portions of this range are usually used by the OS and/or devices, and are not available for user processes
- Virtual addresses (or logical addresses) are addresses provided by the OS to processes.
 - one virtual address space per process
 - addresses typically start at zero, but not necessarily
 - space may consist of several *segments*
- Address translation (or address binding) means mapping virtual addresses to physical addresses.

Example 1: A Simple Address Translation Mechanism

- OS divides physical memory into partitions of different sizes.
- Each partition is made available by the OS as a possible virtual address space for processes.
- Properties:
 - virtual addresses are identical to physical addresses
 - address binding is performed by compiler, linker, or loader, not the OS
 - changing partitions means changing the virtual addresses in the application program
 - * by recompiling
 - * or by *relocating* if the compiler produces relocatable output
 - degree of multiprogramming is limited by the number of partitions
 - size of programs is limited by the size of the partitions

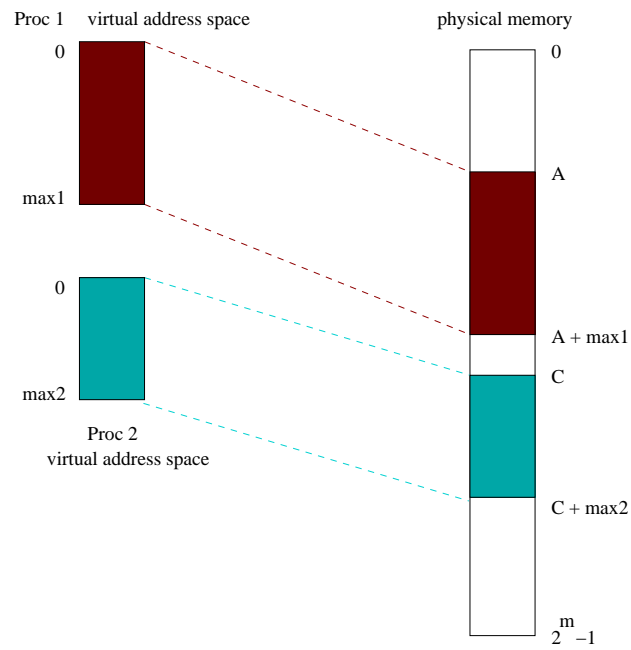
Example 1: Address Space Diagram



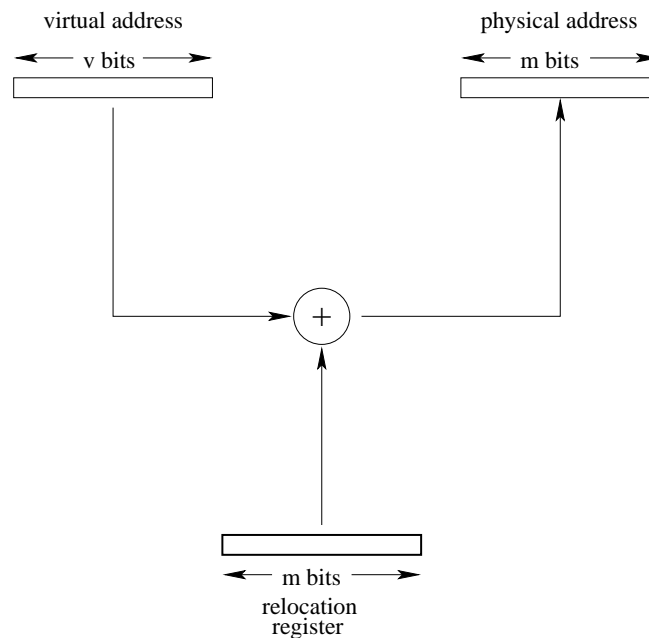
Example 2: Dynamic Relocation

- hardware provides a *memory management unit* which includes a *relocation register*
- *dynamic binding*: at run-time, the contents of the relocation register are added to each virtual address to determine the corresponding physical address
- OS maintains a separate relocation register value for each process, and ensures that relocation register is reset on each context switch
- Properties
 - all programs can have address spaces that start with address 0
 - OS can relocate a process without changing the process's program
 - OS can allocate physical memory dynamically (physical partitions can change over time), again without changing user programs
 - each virtual address space still corresponds to a contiguous range of physical addresses

Example 2: Address Space Diagram



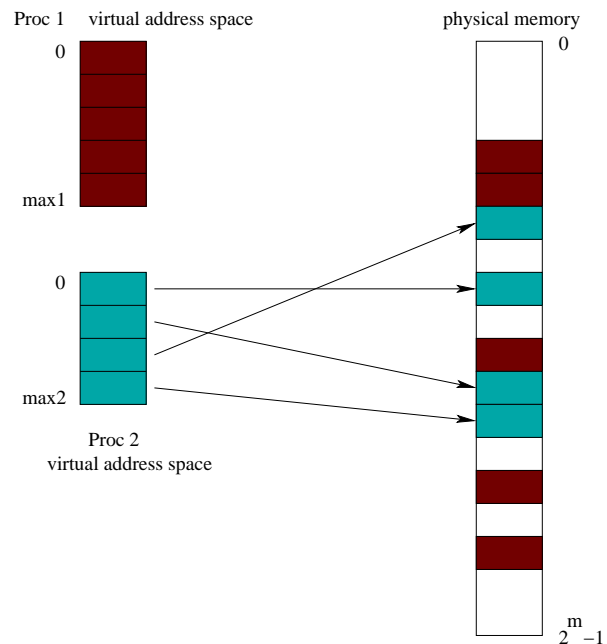
Example 2: Relocation Mechanism



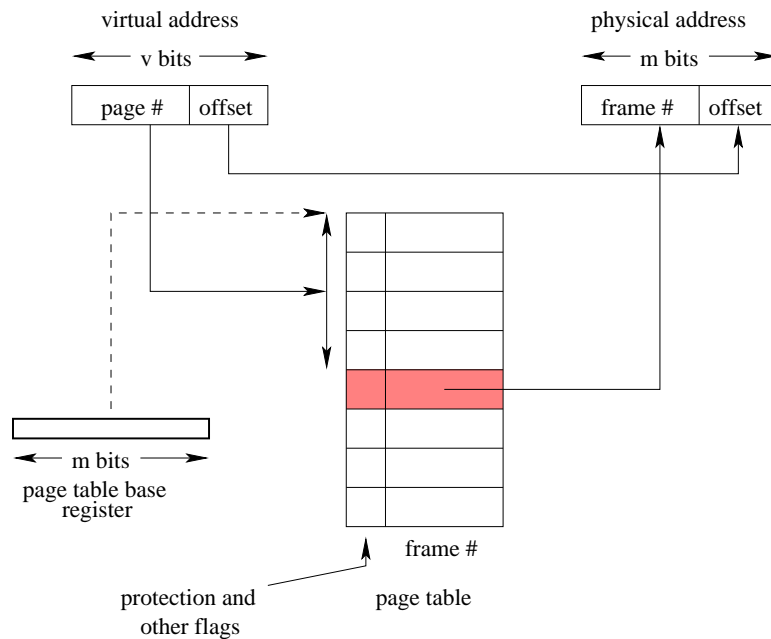
Example 3: Paging

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

Example 3: Address Space Diagram



Example 3: Page Table Mechanism



Summary of Binding and Memory Management Properties

address binding time:

- compile time: relocating program requires recompilation
- load time: compiler produces relocatable code
- dynamic (run time): hardware MMU performs translation

physical memory allocation:

- fixed or dynamic partitions
- fixed size partitions (frames) or variable size partitions

physical contiguity:

- virtual space is contiguous or un contiguous in physical space

Physical Memory Allocation

fixed allocation size:

- space tracking and placement are simple
- *internal* fragmentation

variable allocation size:

- space tracking and placement more complex
 - placement heuristics: first fit, best fit, worst fit
- *external* fragmentation

Memory Protection

- ensure that each process accesses only the physical memory that its virtual address space is bound to.
 - threat: virtual address is too large
 - solution: MMU *limit register* checks each virtual address
 - * for simple dynamic relocation, limit register contains the maximum virtual address of the running process
 - * for paging, limit register contains the maximum page number of the running process
 - MMU generates exception if the limit is exceeded
- restrict the use of some portions of an address space
 - example: read-only memory
 - approach (paging):
 - * include read-only flag in each page table entry
 - * MMU raises exception on attempt to write to a read-only page

Roles of the Operating System and the MMU (Summary)

- operating system:
 - save/restore MMU state on context switches
 - handle exceptions raised by the MMU
 - manage and allocate physical memory
- MMU (hardware):
 - translate virtual addresses to physical addresses
 - check for protection violations
 - 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

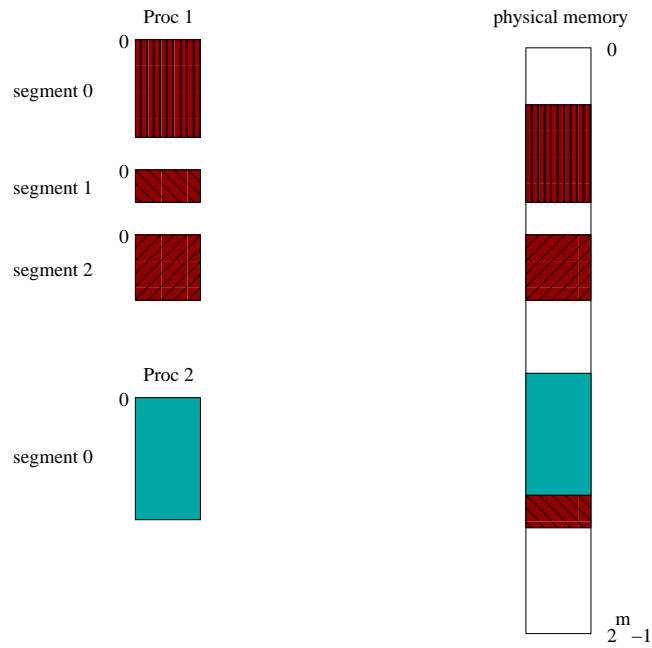
TLB

- Each entry in the TLB contains a (page number, frame number) pair, plus copies of some or all of the page's protection bits, use bit, and dirty bit.
- If address translation can be accomplished using a TLB entry, access to the page table is avoided.
- 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 (10^2 to 10^3 entries).
- Otherwise, translate through the page table, and add the resulting translation to the TLB, replacing an existing entry if necessary. In a *hardware controlled* TLB, this is done by the MMU. In a *software controlled* TLB, it is done by the kernel.
- On a context switch, the kernel must clear or invalidate the TLB. (Why?)

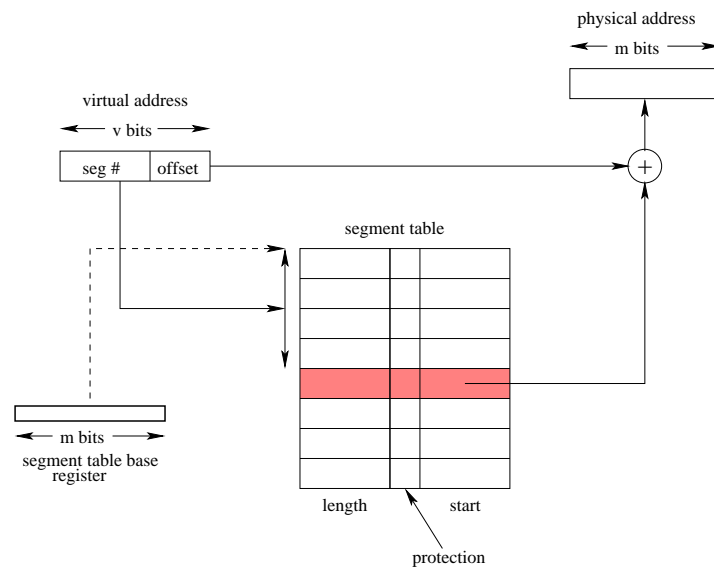
Segmentation

- An OS that supports segmentation (e.g., Multics, OS/2) can provide more than one address space to each process.
- The individual address spaces are called *segments*.
- A logical address consists of two parts:
(segment ID, address within segment)
- Each segment:
 - can grow or shrink independently of the other segments
 - has its own memory protection attributes
- For example, process could use separate segments for code, data, and stack.

Segmented Address Space Diagram

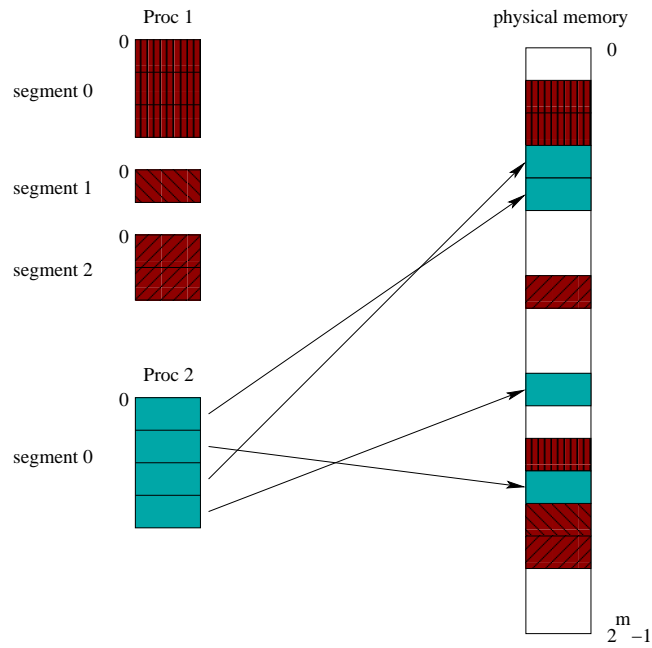


Mechanism for Translating Segmented Addresses

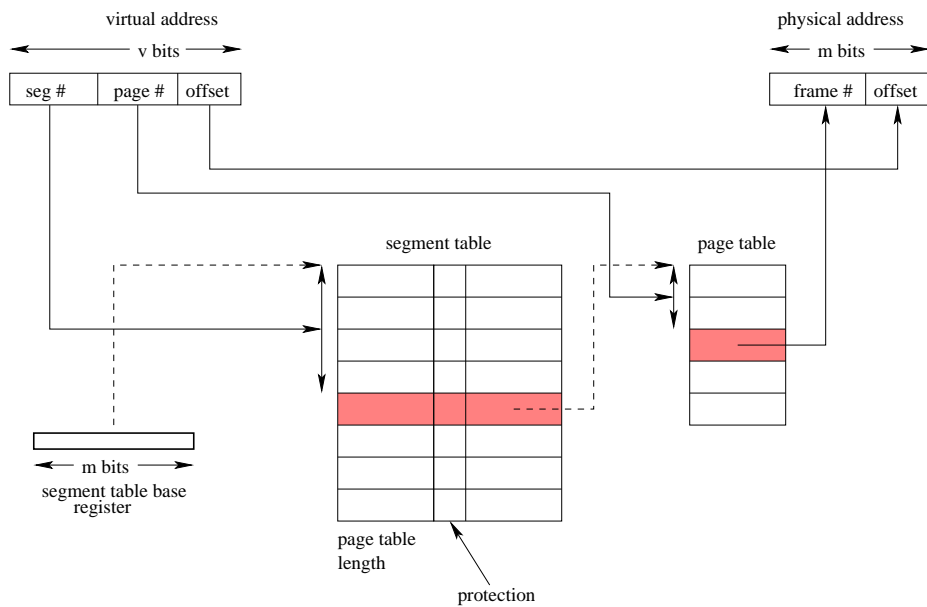


This translation mechanism requires physically contiguous allocation of segments.

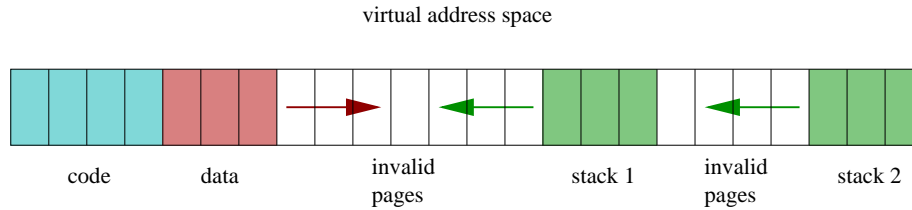
Combining Segmentation and Paging



Combining Segmentation and Paging: Translation Mechanism



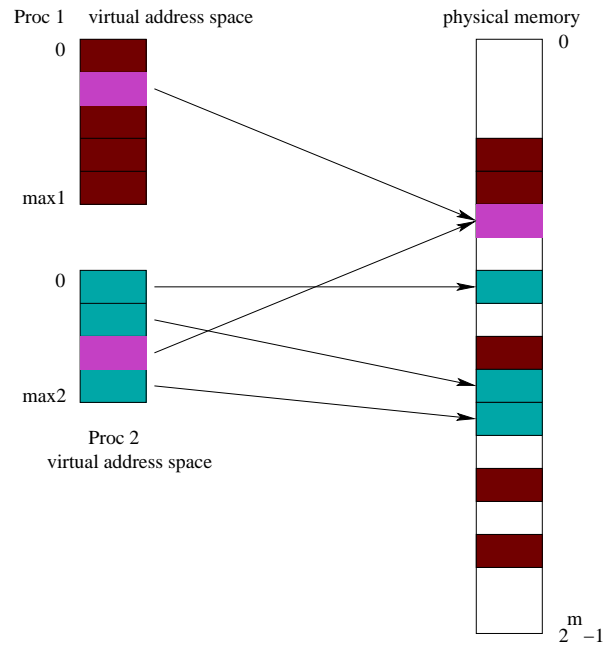
Simulating Segmentation with Paging



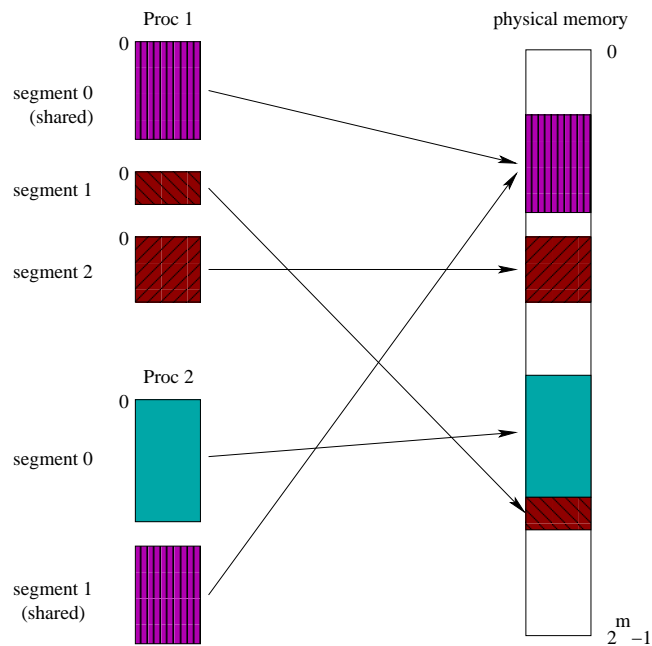
Shared Virtual Memory

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

Shared Pages Diagram



Shared Segments Diagram



An Address Space for the Kernel

Option 1: Kernel in physical space

- mechanism: disable MMU in system mode, enable it in user mode
- accessing process address spaces: OS must interpret process page tables
- OS must be entirely memory resident

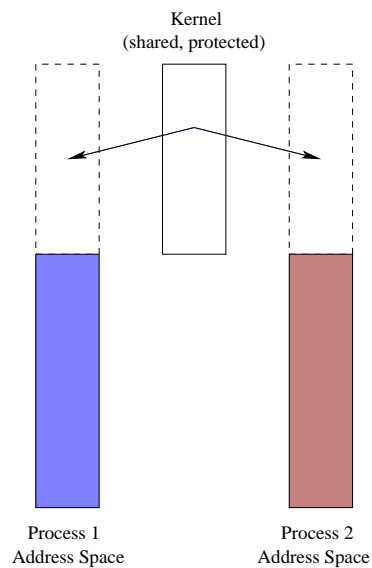
Option 2: Kernel in separate logical address space

- mechanism: MMU has separate state for user and system modes
- accessing process address spaces: difficult
- portions of the OS may be non-resident

Option 3: Kernel shares logical space with each process

- memory protection mechanism is used to isolate the OS
- accessing process address space: easy (process and kernel share the same address space)
- portions of the OS may be non-resident

The Kernel in Process' Address Spaces



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

Memory Management Interface

- much memory allocation is implicit, e.g.:
 - allocation for address space of new process
 - implicit stack growth on overflow
- OS may support explicit requests to grow/shrink address space, e.g., Unix `brk` system call.
- shared virtual memory (simplified Solaris example):
 - Create:** `shmid = shmget(key, size)`
 - Attach:** `vaddr = shmat(shmid, vaddr)`
 - Detach:** `shmdt(vaddr)`
 - Delete:** `shmctl(shmid, IPC_RMID)`

Virtual Memory

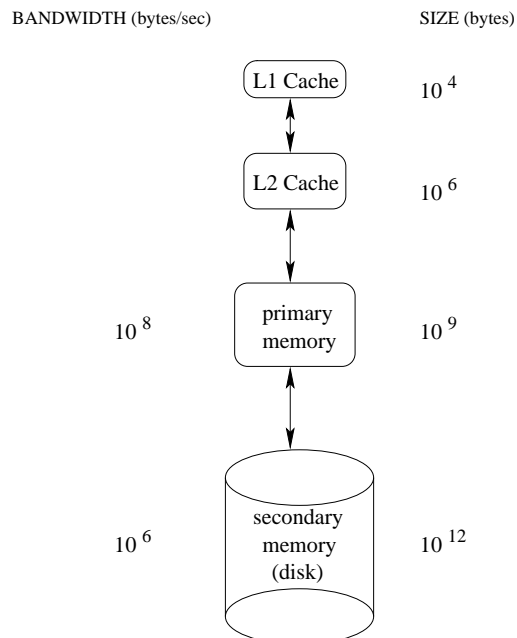
Goals:

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

Method:

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

The Memory Hierarchy



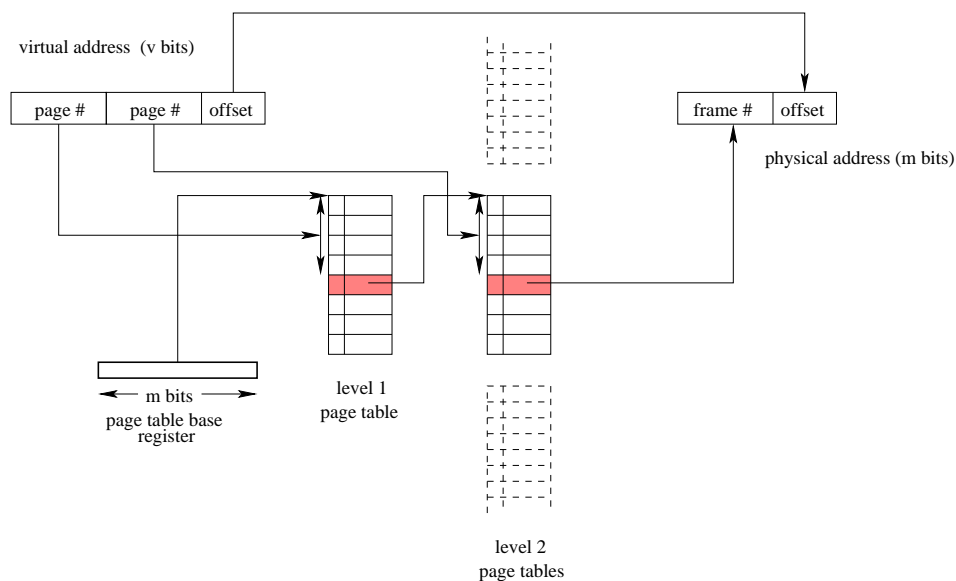
Large Virtual Address Spaces

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

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

- page tables must be in memory and physically contiguous
- some solutions:
 - multi-level page tables - page the page tables
 - inverted page tables

Two-Level Paging



Inverted Page Tables

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

Paging Policies

When to Page?:

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

What to Replace?:

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

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

Paging Mechanism

- A *valid* bit (V) in each page table entry is used to track which pages are in (primary) memory, and which are not.
 - $V = 1$: valid entry which can be used for translation
 - $V = 0$: invalid entry. If the MMU encounters an invalid page table entry, it raises a *page fault* exception.
- To handle a page fault exception, the operating system must:
 - Determine which page table entry caused the exception. (In NachOS, and in real MIPS processors, the MMU places the offending virtual address into the BadVAddrReg register.)
 - Ensure that that page is brought into memory.

On return from the exception handler, the instruction that resulted in the page fault will be retried.
- If (pure) segmentation is being used, there will a valid bit in each segment table entry to indicate whether the segment is in memory.

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.

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	

Other Replacement Policies

- FIFO is simple, but it does not consider:
 - Recency of Use:** when was a page last used?
 - Frequency of Use:** how often a page has been 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, LRU is often considered to be impractical for use as a replacement policy in virtual memory systems. Why?

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 updated by the operating system
- Page table entries in NachOS include a use bit.

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
```

Frequency-based Page Replacement

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

Page Cleanliness: the Dirty Bit

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

Enhanced Second Chance Replacement Algorithm

- Classify pages according to their use and dirty bits:
 - (0,0): not recently used, clean.
 - (0,1): not recently used, dirty.
 - (1,0): recently used, clean
 - (1,1): recently used, dirty
- Algorithm:
 1. Sweep once looking for (0,0) page. Don't clear use bits while looking.
 2. If none found, look for (0,0) or (0,1) page, this time clearing "use" bits while looking.

Page Cleaning

- A dirty page must be cleaned before it can be replaced, otherwise changes on that page will be lost.
- *Cleaning* a page means copying the page to secondary storage.
- Cleaning is distinct from replacement.
- Page cleaning may be *synchronous* or *asynchronous*:
 - synchronous cleaning:** happens at the time the page is replaced, during page fault handling. Page is first cleaned by copying it to secondary storage. Then a new page is brought in to replace it.
 - asynchronous cleaning:** happens before a page is replaced, so that page fault handling can be faster.
 - asynchronous cleaning may be implemented by dedicated OS *page cleaning threads* that sweep through the in-memory pages cleaning dirty pages that they encounter.

Prefetching

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

Page Size Tradeoffs

- larger pages mean:
 - + smaller page tables
 - + better TLB “coverage”
 - + more efficient I/O
 - greater internal fragmentation
 - increased chance of paging in unnecessary data

Belady's Anomaly

- FIFO replacement, 4 frames

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

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

More memory does not necessarily mean fewer page faults.

Stack Policies

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

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

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

Stack algorithms do not suffer from Belady's anomaly.

Global vs. Local Page Replacement

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

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

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

Detailed TLB and Paging Example

- Assume TLB and global page replacements are done using a round-robin algorithm with the next entries to replace being 1 and 3, respectively. Also assume a 4 KB page size. What happens in hardware and the kernel if program A performs the following operations in sequence: read 0x329, write 0x429, read 0x3691, read 0x271a, write 0x1741?

TLB						Page Table A						Page Table B						CoreMap					
VPN	PFN	D	U	V	RO	VPN	PFN	D	U	V	RO	VPN	PFN	D	U	V	RO	VPN	PGM				
0	0	3	0	0	1	0	0	0	3	0	0	1	0	0	0	1	0	0	1	0	0	4	A
1	1	2	1	1	0	1	1	1		0	0	0	1	1	1	2	0	0	1	1	1	0	B
2	3	5	1	1	0	0	2	2		0	0	0	0	2	2						2	1	B
							3	3	5	0	0	1	0	3	3						3	0	A
							4	4	0	0	0	1	0	4	4						4	7	B
														5	5	4	1	1	0	0	5	3	A
														6	6								
														7	7	4	0	0	1	0			

How Much 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 process pages that are located in memory.

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

Resident Set Sizes (Example)

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

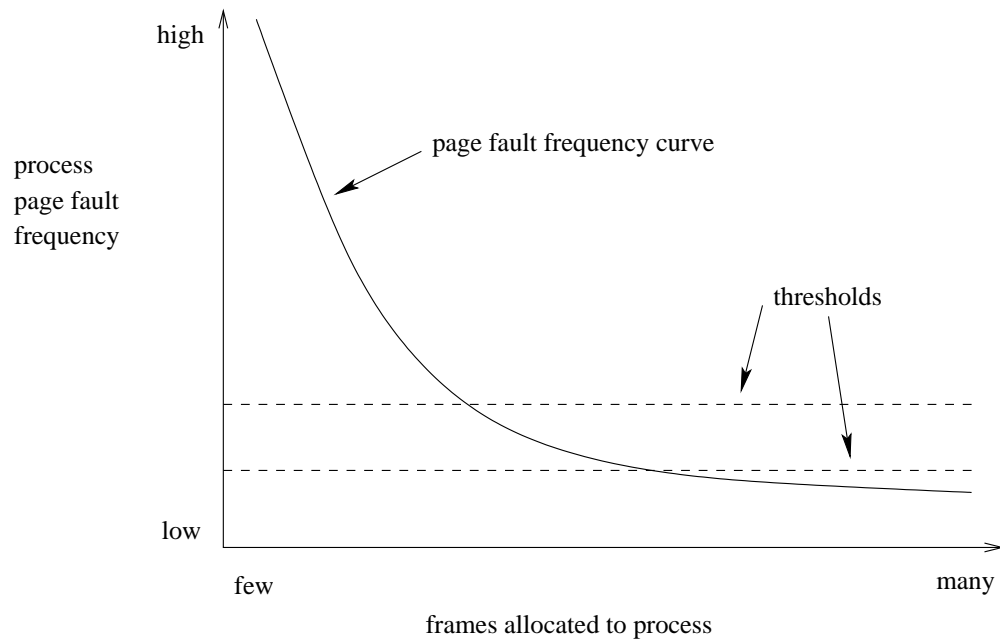
Refining the Working Set Model

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

Page Fault Frequency

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

A Page Fault Frequency Plot



Thrashing and Load Control

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

Swapping Out Processes

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

Scheduling Criteria

CPU utilization: keep the CPU as busy as possible

throughput: rate at which tasks are completed

response time/turnaround time: time required to finish a task

fairness

A “task” might be a single CPU burst, a thread, or an application-level service request.

The Nature of Program Executions

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

Preemptive vs. Non-Preemptive

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

FCFS and Round-Robin Scheduling

First-Come, First-Served (FCFS):

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

Round-Robin:

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

Shortest Job First (SJF) Scheduling

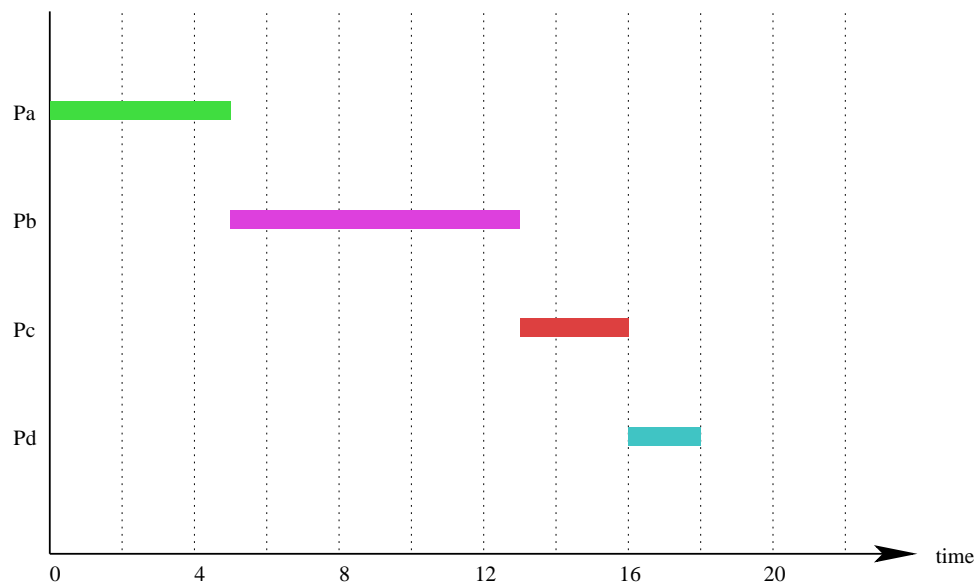
- non-preemptive
- ready threads are scheduled according to the length of their next CPU burst - thread with the shortest burst goes first
- SJF minimizes average waiting time, but can lead to starvation
- SJF requires knowledge of CPU burst lengths
 - Simplest approach is to estimate next burst length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

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

where B_i is the predicted length of the i th CPU burst, and b_i is its actual length, and $0 \leq \alpha \leq 1$.

- Shortest Remaining Time First is a preemptive variant of SJF. Preemption may occur when a new thread enters the ready queue.

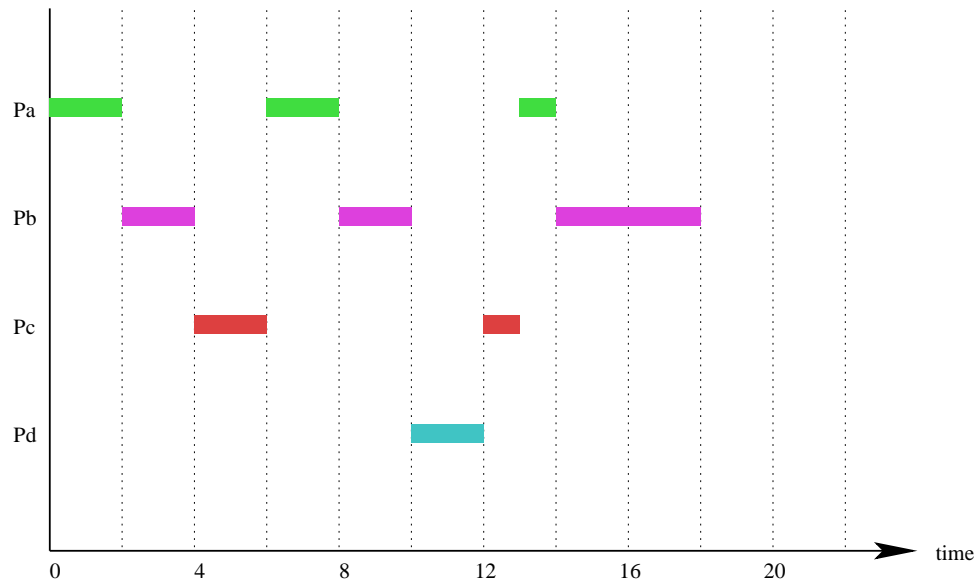
FCFS Gantt Chart Example



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

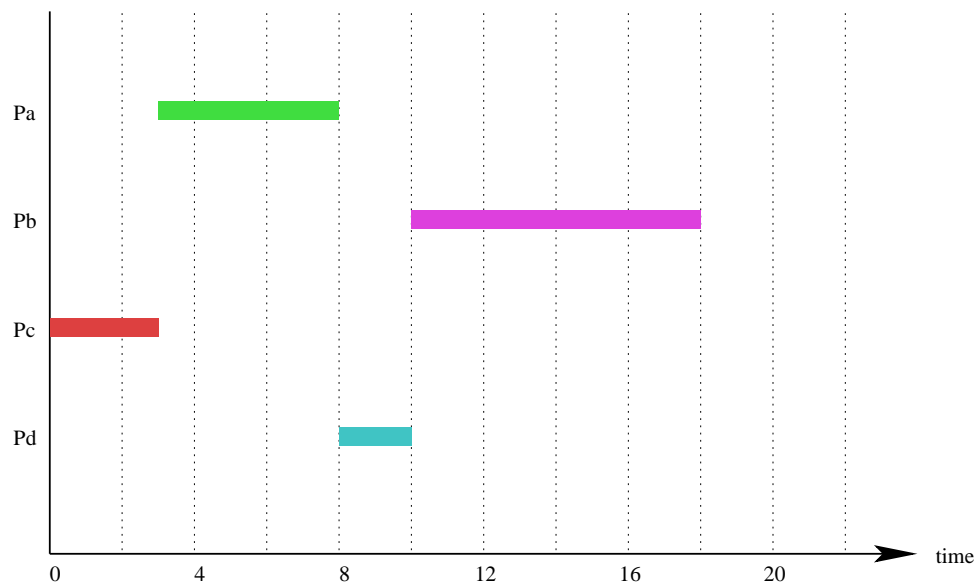
Thread Pd (=2) "arrives" at time 5

Round Robin Example



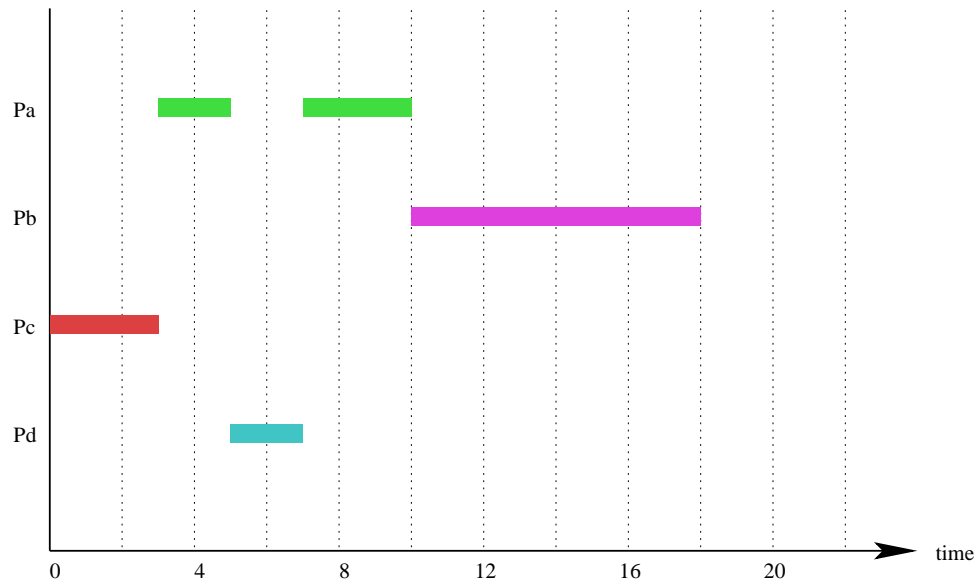
Initial ready queue: Pa = 5 Pb = 8 Pc = 3 Quantum = 2
 Thread Pd (=2) "arrives" at time 5

SJF Example



Initial ready queue: Pa = 5 Pb = 8 Pc = 3
 Thread Pd (=2) "arrives" at time 5

SRTF Example



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

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

Highest Response Ratio Next

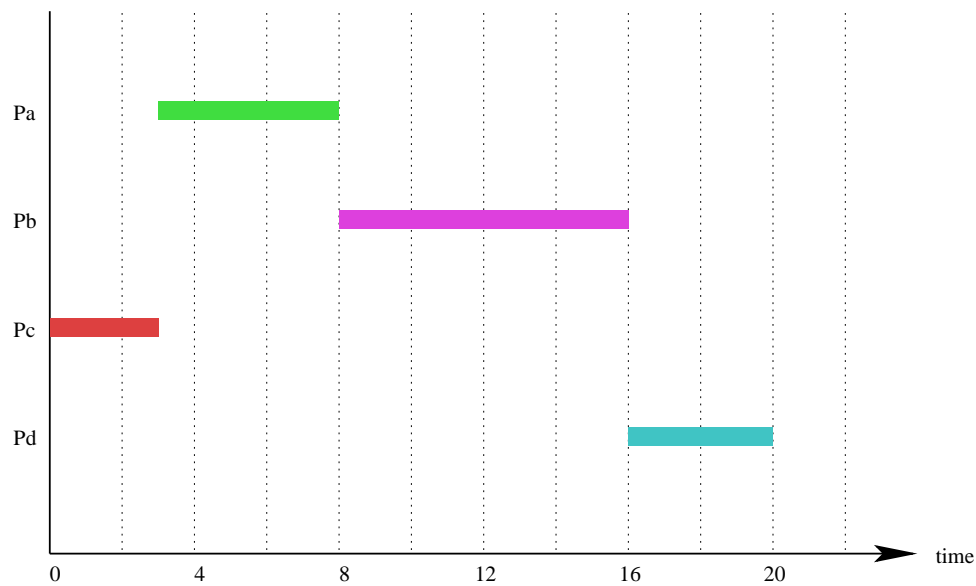
- non-preemptive
- response ratio is defined for each ready thread as:

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

where b is the estimated CPU burst time and w is the actual waiting time

- scheduler chooses the thread with the highest response ratio (choose smallest b in case of a tie)
- HRRN is an example of a heuristic that blends SJF and FCFS

HRRN Example



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

Thread Pd (=4) "arrives" at time 5

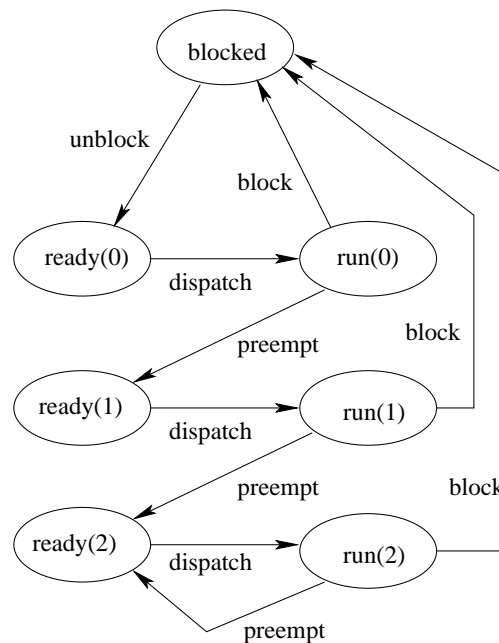
Prioritization

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

Multilevel Feedback Queues

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

3 Level Feedback Queue State Diagram



Lottery Scheduling

- randomized proportional share resource allocation
- resource rights represented by lottery tickets, allocation determined by lottery
 - resource granted to holder of winning ticket
- probabilistically fair with $p = t/T$
 - p = probability of allocation, t = tickets held, T = total tickets
 - avoid starvation by ensuring $t > 0$
- uniformly-distributed pseudo-random number generator (10 lines on MIPS)
- can proportionally assign other resources (e.g., memory, bandwidth)
- “Lottery Scheduling: Flexible Proportional-Share Resource Management”, Waldspurger and Weihl, Operating System Design and Implementation, 1994.

Processor Scheduling Summary

FCFS:

- + simple, low overhead
- + no starvation
- can give poor response times for interactive processes

RR:

- + no starvation
- + reduced waiting time variance
- + good response times for interactive processes

SJF and SRTF:

- + best response times
- depends on burst length estimates
- starvation is possible

Processor Scheduling Summary

HRRN:

- + no starvation
- + good response times
- depends on burst length estimates

Multilevel Feedback Queues:

- + flexible
- + good response times for interactive processes
- compute-intensive processes can starve

Other Scheduling Issues

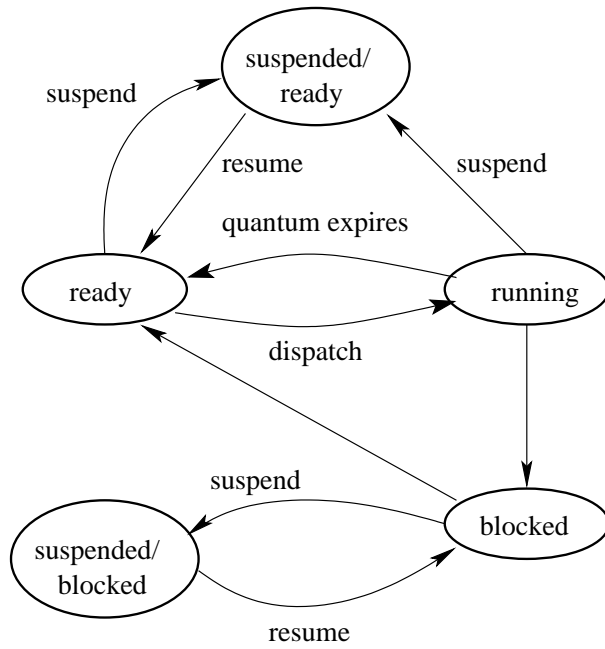
short term scheduling: what has been covered so far

medium term scheduling: suspension/resumption of partially executed processes

- usually because a resource, especially memory, is overloaded
- suspended process releases resources
- operating system may also provide mechanisms for applications or users to request suspension/resumption of processes

long term scheduling: process admission control to limit the degree of multiprogramming

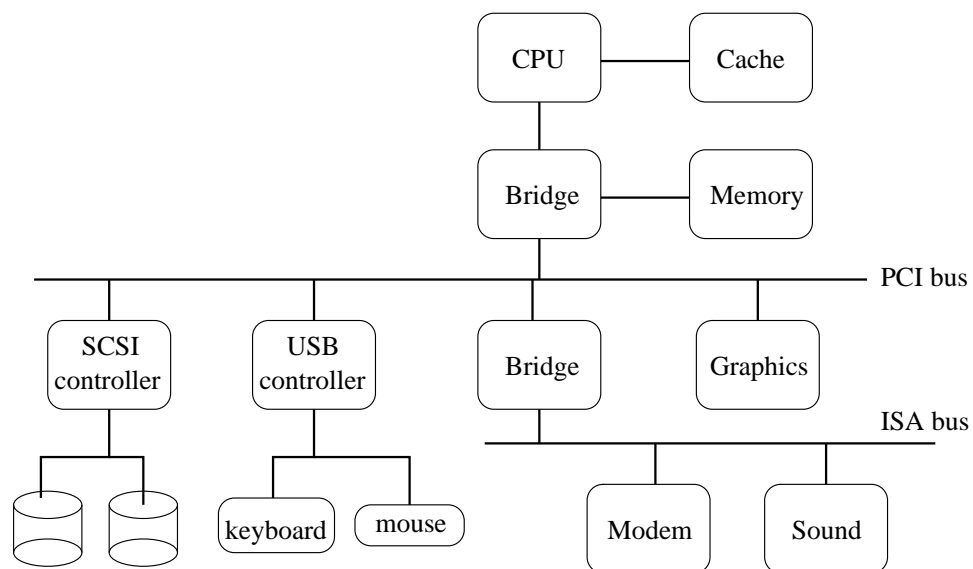
Scheduling States Including Suspend/Resume



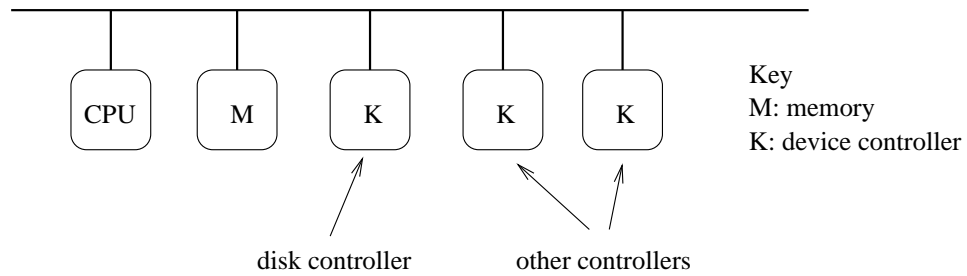
Devices and Device Controllers

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

Bus Architecture Example



Simplified Bus Architecture

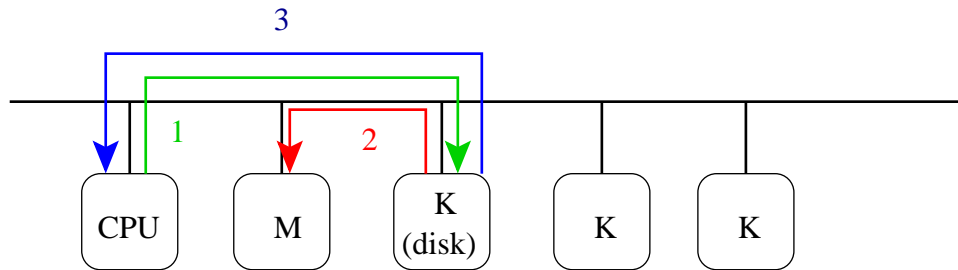


Device Interactions

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

Direct Memory Access (DMA)

- used for block data transfers between devices (e.g., disk controller) and memory
- CPU initiates DMA, but can do other work while the transfer occurs



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

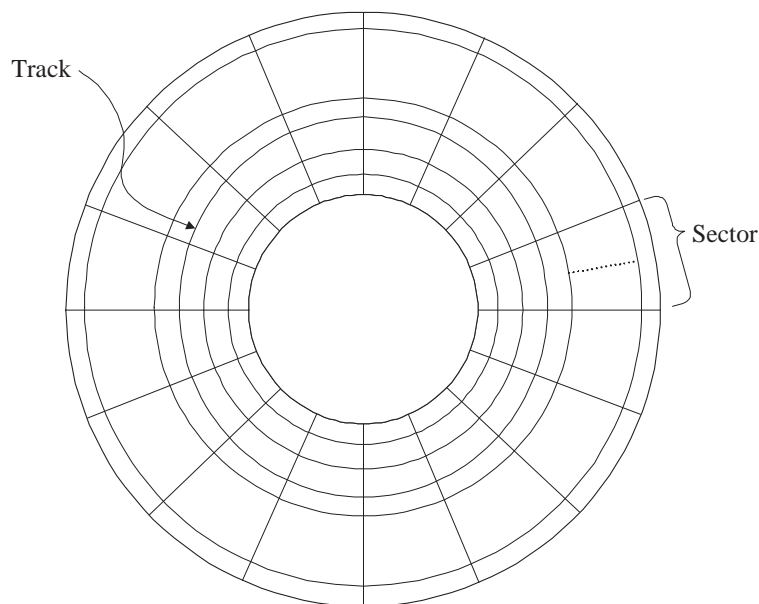
Applications and Devices

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

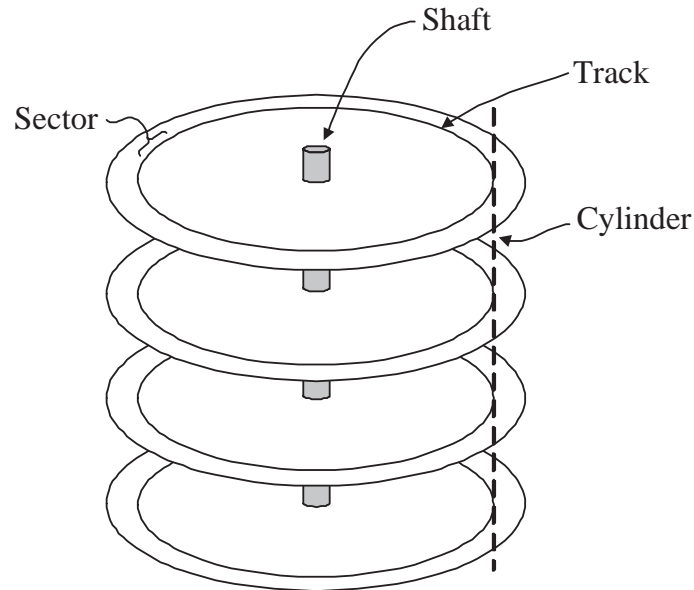
Logical View of a Disk Drive

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

A Disk Platter's Surface



Physical Structure of a Disk Drive



Simplified Cost Model for Disk Block Transfer

- moving data to/from a disk involves:
 - seek time:** move the read/write heads to the appropriate cylinder
 - rotational latency:** wait until the desired sectors spin to the read/write heads
 - transfer time:** wait while the desired sectors spin past the read/write heads
- request service time is the sum of seek time, rotational latency, and transfer time

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

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

Rotational Latency and Transfer Time

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

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

- expected rotational latency:

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

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

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

Seek Time

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

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

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, simple
 - : no optimization of seek times

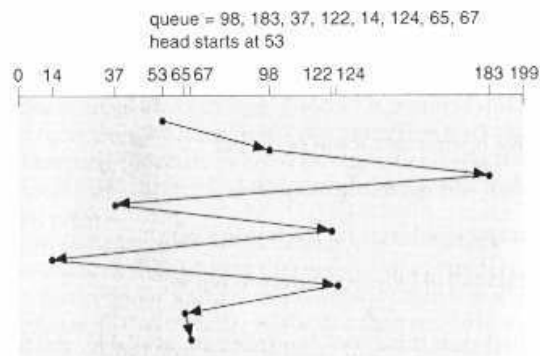


Figure 14.1 FCFS disk scheduling.

Figure from *Operating Systems Concepts, 6th Ed.*, Silberschatz, Galvin, Gagne. Wiley. 2003

Shortest Seek Time First (SSTF)

- choose closest request (a greedy approach)
 - + : seek times are reduced
 - : starvation of distant requests

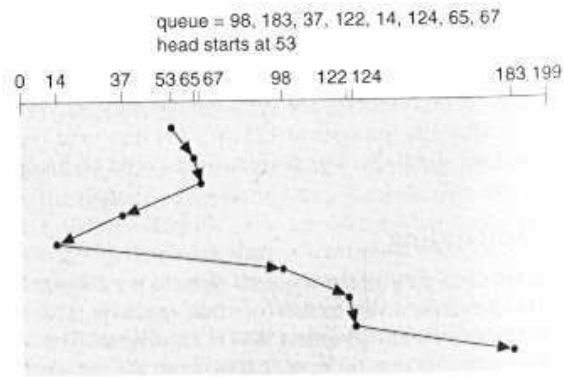


Figure 14.2 SSTF disk scheduling.

Figure from *Operating Systems Concepts, 6th Ed.*, Silberschatz, Galvin, Gagne. Wiley. 2003

SCAN and LOOK

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

SCAN Example

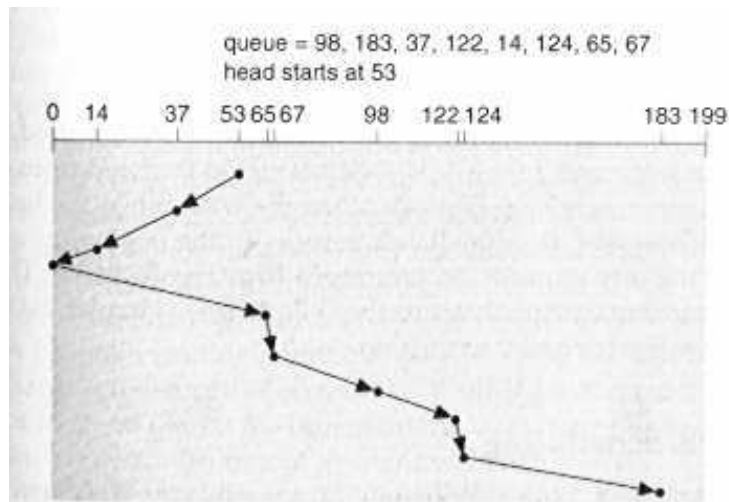


Figure 14.3 SCAN disk scheduling.

Figure from *Operating Systems Concepts, 6th Ed.*, Silberschatz, Galvin, Gagne. Wiley. 2003

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

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

C-LOOK Example

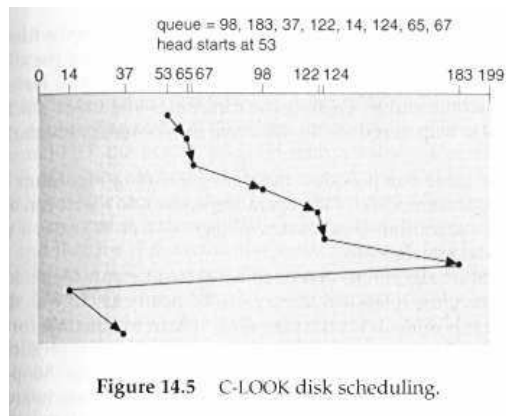


Figure from *Operating Systems Concepts, 6th Ed.*, Silberschatz, Galvin, Gagne. Wiley. 2003

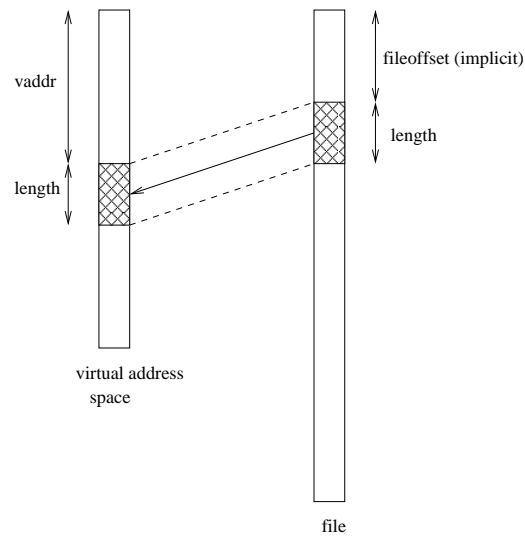
Files and File Systems

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

File Interface

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

File Read



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

File Position

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

Sequential File Reading Example (Unix)

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

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

File Reading Example Using Seek (Unix)

```
char buf[512];
int i;
int f = open("myfile",O_RDONLY);
lseek(f,99*512,SEEK_SET);
for(i=0; i<100; i++) {
    read(f,(void *)buf,512);
    lseek(f,-1024,SEEK_CUR);
}
close(f);
```

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

File Reading Example Using Positioned Read

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

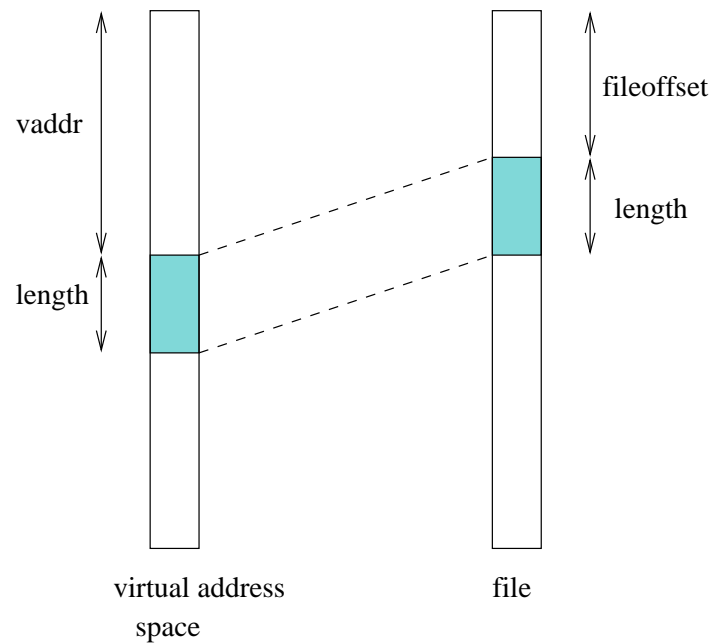
Read every second 512 byte chunk of a file, until 50 have been read.

Memory-Mapped Files

- generic interface:
vaddr ← mmap(file descriptor, fileoffset, length)
munmap(vaddr, length)
- mmap call returns the virtual address to which the file is mapped
- munmap call unmaps mapped files within the specified virtual address range

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

Memory Mapping Illustration



Memory Mapping Update Semantics

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

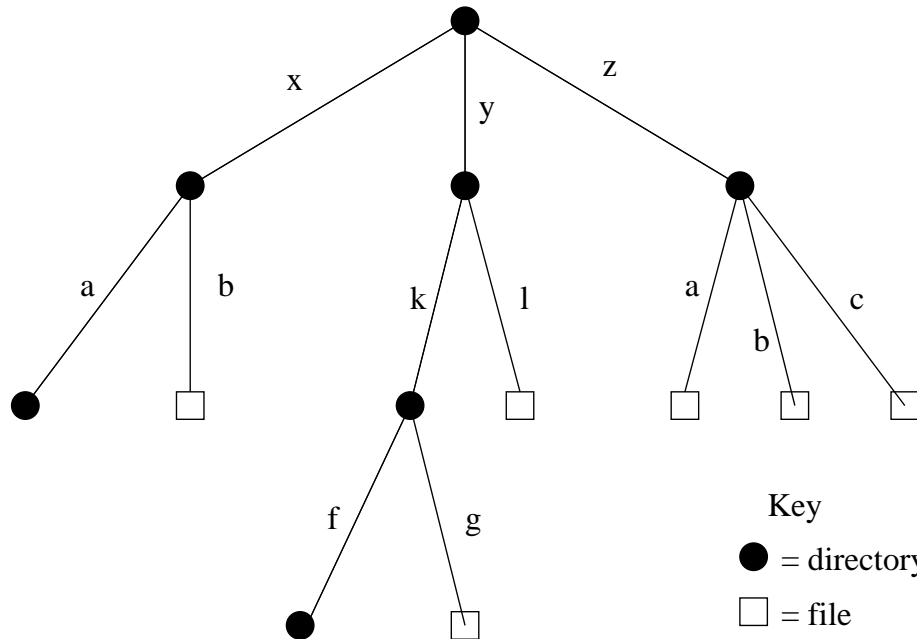
Memory Mapping Concurrency Semantics

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

File Names

- flat namespace
 - file names are simple strings
 - hierarchical namespace
 - directories (folders) can be used to organize files and/or other directories
 - directory inclusion graph is a tree
 - pathname: file or directory is identified by a *path* in the tree
- Unix:** `/home/kmsalem/courses/cs350/notes/filesys.ps`
Windows: `c:\kmsalem\cs350\schedule.txt`

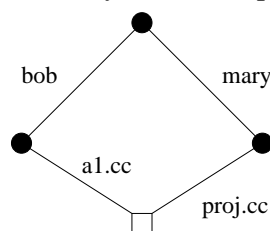
Hierarchical Namespace Example



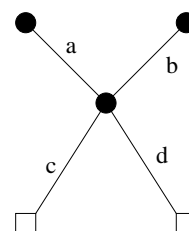
Acyclic File Namespaces

- directory inclusion graph can be a (rooted) DAG
- allows files/directories to have more than one pathname
 - increased flexibility for file sharing and file organization
 - file removal and some other file system operations are more complicated
- examples:

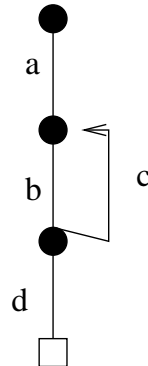
Rooted Acyclic Namespace



An Unrooted DAG



General File Namespaces

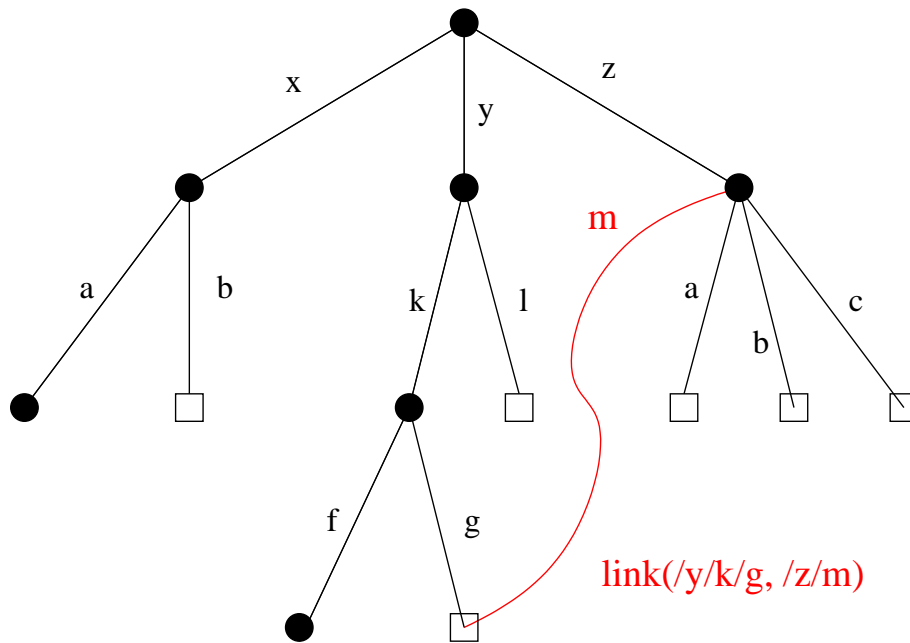


- no restriction on inclusion graph (except perhaps that it should have a designated root node)
- maximum flexibility
- additional complications, e.g.:
 - reference counts are no longer sufficient for implementing file deletion
 - pathnames can have an infinite number of components

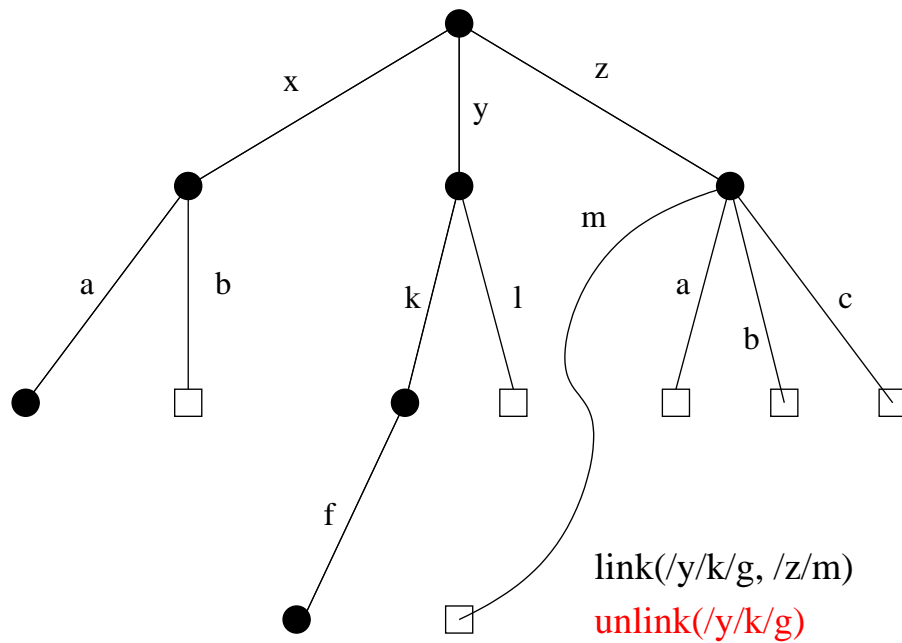
File Links

- typically, a new file or directory is linked to a single “parent” directory when it is created. This gives a hierarchical namespace.
- another mechanism can then be used to create additional links to existing files or directories, introducing non-hierarchical structure in the namespace.
- hard links
 - “first class” links, like the original link to a file
 - *referential integrity* is maintained (no “dangling” hard links)
 - scope usually restricted to a single file system
 - Unix: hard links can be made to files, but not to directories. This restriction is sufficient to avoid cycles. (Why?)
- soft links (a.k.a. “symbolic links”, “shortcuts”)
 - referential integrity is *not* maintained
 - flexible: may be allowed to span file systems, may link to directories and (possibly) create cycles

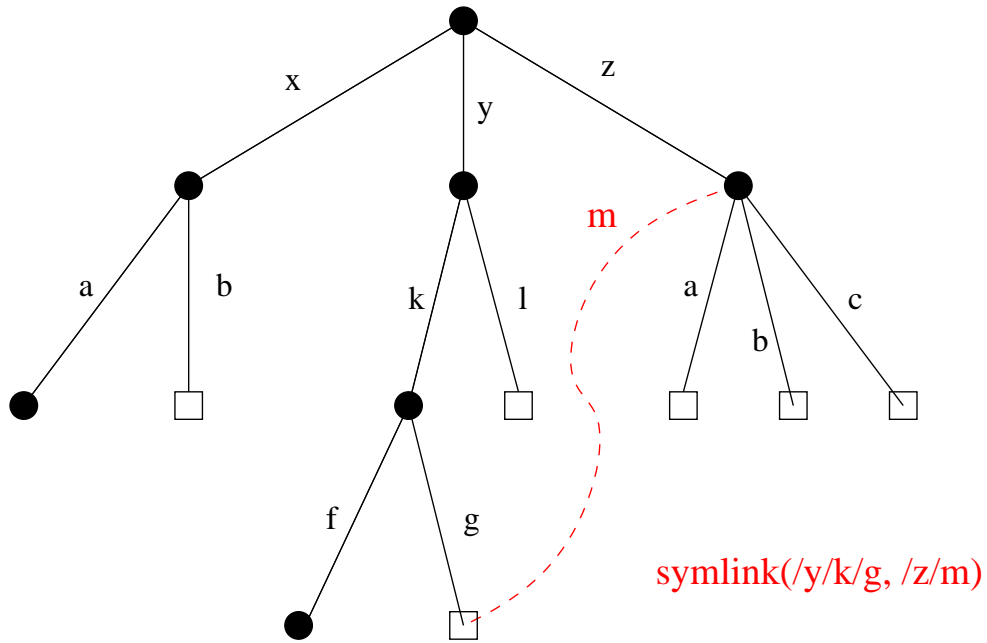
Hard Link Example (Part 1)



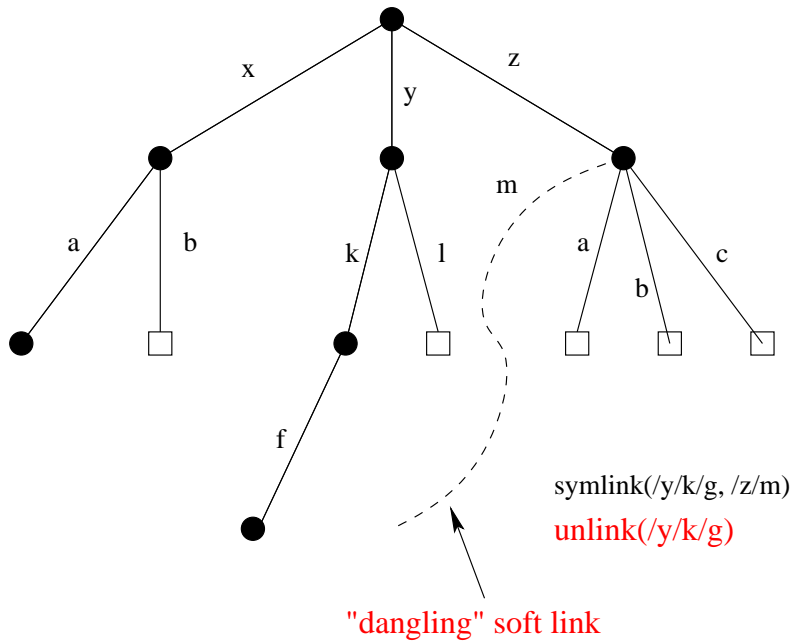
Hard Link Example (Part 2)



Soft Link Example (Part 1)



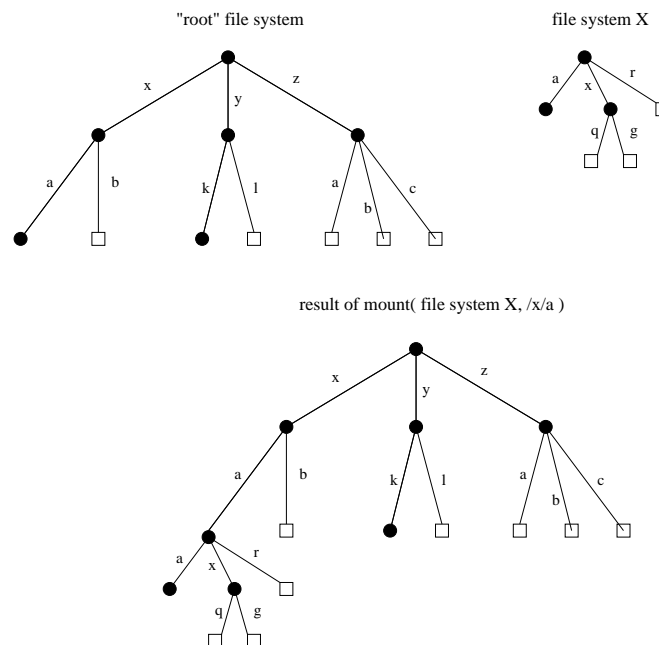
Soft Link Example (Part 2)



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:** use two-part file names: file system name,pathname
 - example: `C:\kmsalem\cs350\schedule.txt`
 - Unix:** merge file graphs into a single graph
 - Unix mount system call does this

Unix mount Example



File System Implementation

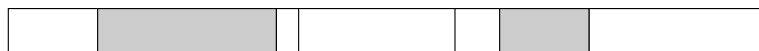
- space management
- file indexing (how to locate file data and meta-data)
- directories
- links
- buffering, in-memory data structures
- persistence

Space Allocation

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



fixed-size allocation



variable-size allocation

Space Allocation (continued)

- differences between primary and secondary memory
 - larger transfers are cheaper (per byte) than smaller transfers
 - sequential I/O is faster than random I/O
- both of these suggest that space should be allocated to files in large chunks, sometimes called *extents*

File Indexing

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

chaining:

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

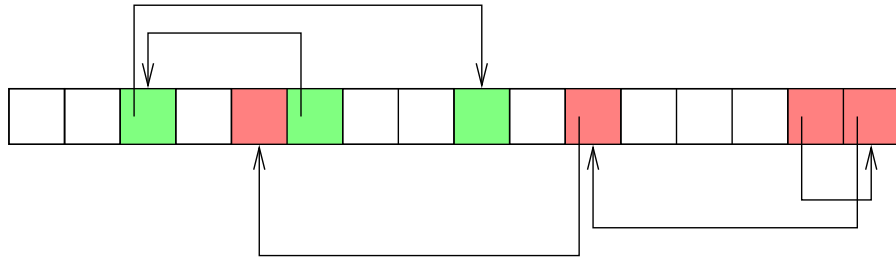
external chaining: DOS file allocation table (FAT), for example

- like chaining, but the chain is kept in an external structure

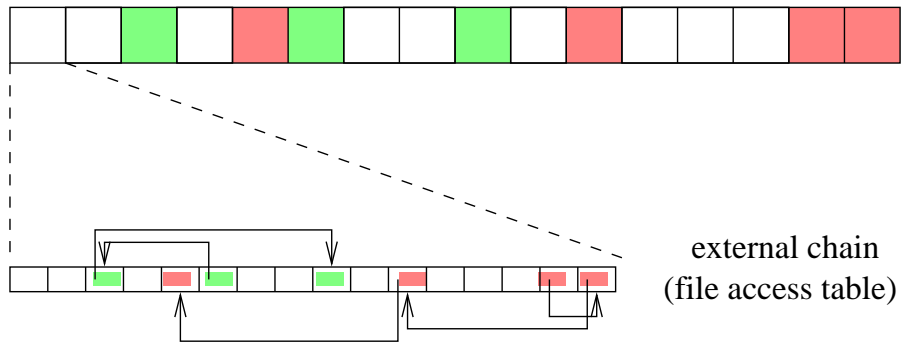
per-file index: Unix i-node and NachOS FileHeader, for example

- for each file, maintain a table of pointers to the file's blocks or extents

Chaining

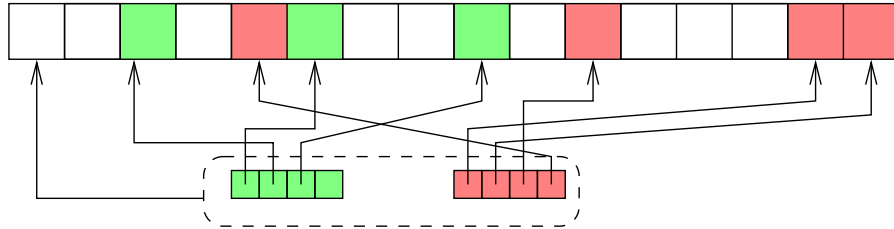


External Chaining (File Access Table)



external chain
(file access table)

Per-File Indexing



File Meta-Data and Other Information

- where to store file meta-data?
 - immediately preceding the file data
 - with the file index (if per-file indexing is being used)
 - with the directory entry for the file
 - * this is a problem if a file can have multiple names, and thus multiple directory entries

Unix i-nodes

- an i-node is a particular implementation of a per-file index
- each i-node is uniquely identified by an i-number, which determines its physical location on the disk
- an i-node is a fixed size record containing:

file attribute values

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

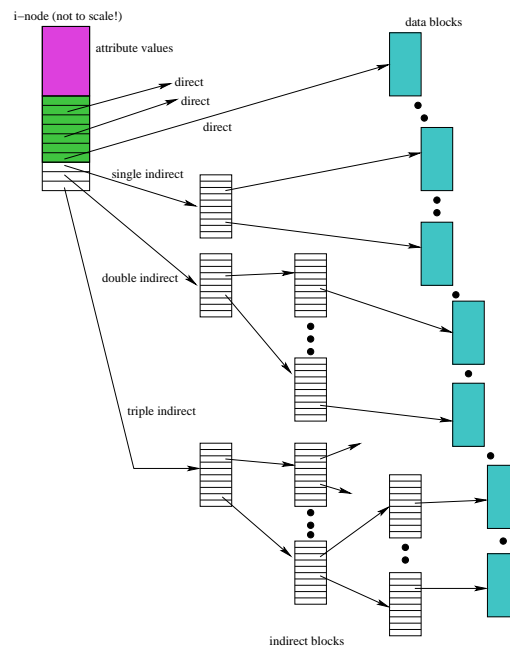
direct block pointers: approximately 10 of these

single indirect block pointer

double indirect block pointer

triple indirect block pointer

i-node Diagram



NachOS FileHeader

```
#define NumDirect ((SectorSize-2*sizeof(int))/sizeof(int))
class FileHeader {
public:
    // methods here
private:
    int numBytes;    // file size in bytes
    int numSectors; // file size in sectors
    int dataSectors[NumDirect]; // direct pointers
}
```

Directories

- A directory consists of a set of entries, where each entry is a record that includes:
 - a file name (component of a path name)
 - a file “locator”
 - * location of the first block of the file, if chaining or external chaining is used
 - * location of the file index, if per-file indexing is being used
- A directory can be implemented like any other file, except:
 - interface should allow reading of records (can be provided by a special system call or a library)
 - file should not be writable directly by application programs
 - directory records are updated by the kernel as files are created and destroyed

Implementing Hard Links (Unix)

- hard links are simply directory entries
- for example, consider:
`link(/y/k/g, /z/m)`
- to implement this:
 - create a new entry in directory /z
 - * file name in new entry is m
 - * file locator (i-number) in the new entry is the same as the i-number for entry g in directory /y/k

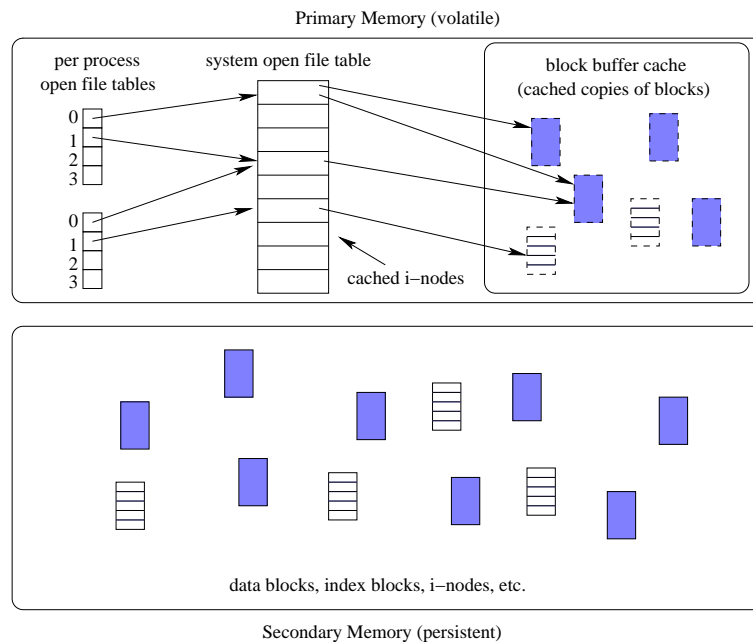
Implementing Soft Links (Unix)

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

File System Meta-Data

- file system must record:
 - location of file indexes or file allocation table
 - location of free list(s) or free space index
 - file system parameters, e.g., block size
 - file system identifier and other attributes
- example: Unix *superblock*
 - located at fixed, predefined location(s) on the disk
- example: NachOS free space bitmap and directory files
 - headers for these files are located in disk sectors 0 and 1

Main Memory Data Structures



A Simple Exercise

- Walk through the steps that the file system must take to implement `Open`.
 - which data structures (from the previous slide) are updated?
 - how much disk I/O is involved?

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 a failure, some but not all of these changes are reflected on the disk?

Fault Tolerance

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

Interprocess Communication Mechanisms

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

Signals

- signals permit asynchronous one-way communication
 - from a process to another process, or to a group of processes
 - 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

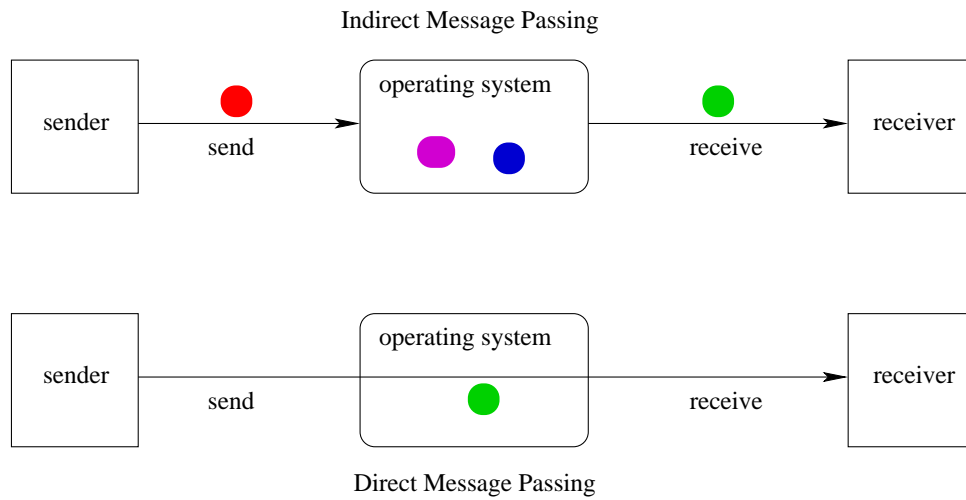
Examples of Signal Types

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`

Message Passing



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

Properties of Message Passing Mechanisms

Addressing: how to identify where a message should go

Directionality:

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

Message Boundaries:

datagram model: message boundaries

stream model: no boundaries

Properties of Message Passing Mechanisms (cont'd)

Connections: need to connect before communicating?

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

Reliability:

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

Sockets

- a socket is a communication *end-point*
- if two processes are to communicate, each process must create its own socket
- two common types of sockets
 - stream sockets:** support connection-oriented, reliable, duplex communication under the stream model (no message boundaries)
 - datagram sockets:** support connectionless, best-effort (unreliable), duplex communication under the datagram model (message boundaries)
- both types of sockets also support a variety of address domains, e.g.,
 - Unix domain:** useful for communication between processes running on the same machine
 - INET domain:** useful for communication between process running on different machines that can communicate using IP protocols.

Using Datagram Sockets (Receiver)

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

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

Using Datagram Sockets (Sender)

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

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

More on Datagram Sockets

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

A Socket Address Convention

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

Using Stream Sockets (Passive Process)

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

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

Notes on Using Stream Sockets (Passive Process)

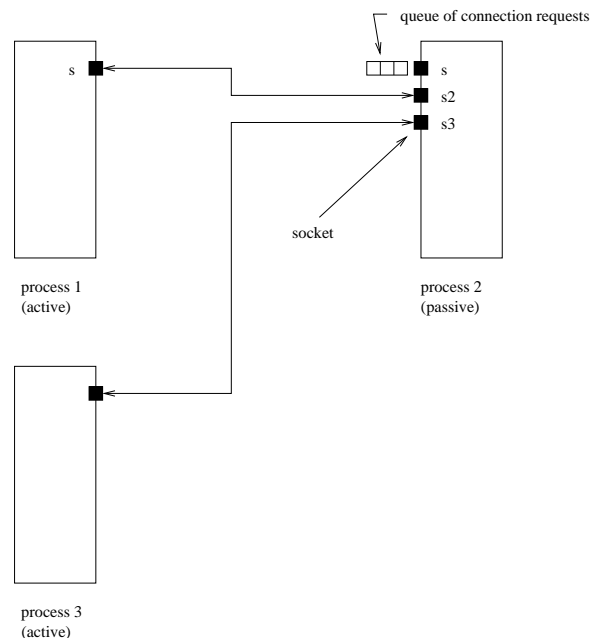
- `accept` creates a new socket (`ns`) for the new connection
- `sourceAddress` is an address buffer. `accept` fills it with the address of the socket that has made the connection request
- additional connection requests can be accepted using more `accept` calls on the original socket (`s`)
- `accept`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

Using Stream Sockets (Active Process)

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

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

Illustration of Stream Socket Connections



Socket Example: Client

```
#include "defs.h"

#define USAGE "client serverhost port#\n"
#define ERROR_STR_LEN  (80)

int
main(int argc, char *argv[])
{
    struct hostent *hostp;
    int sockfd, server_port, num;

    char error_str[ERROR_STR_LEN];
    char read_buf[BUF_LEN];
    char *hostname;
    struct sockaddr_in server_addr;
    struct in_addr tmp_addr;

    if (argc != 3) {
        fprintf(stderr, "%s", USAGE);
        exit(-1);
    }
}
```

Socket Example: Client (continued)

```
/* get hostname and port for the server */
hostname = argv[1];
server_port = atoi(argv[2]);

/* get the server hosts address */
if ((hostp = (struct hostent *)
    gethostbyname(hostname)) ==
    (struct hostent *) NULL) {
    sprintf(error_str,
        "client: gethostbyname fails for host %s",
        hostname);
    /* gethostbyname sets h_errno */
    perror(error_str);
    exit(-1);
}

/* create a socket to connect to server */
if ((sockfd = socket(DOMAIN, SOCK_STREAM, 0)) < 0) {
    perror("client: can't create socket ");
    exit(1);
}
```

Socket Example: Client (continued)

```
/* zero the socket address structure */
memset((char *) &server_addr, 0, sizeof(server_addr));

/* start constructing the server socket addr */
memcpy(&tmp_addr, hostp->h_addr_list[0],
       hostp->h_length);

printf("Using server IP addr = %s\n",
       inet_ntoa(tmp_addr));

/* set servers address field, port number and family */
memcpy((char *) &server_addr.sin_addr,
       (char *) &tmp_addr,
       (unsigned int) hostp->h_length);
server_addr.sin_port = htons(server_port);
server_addr.sin_family = DOMAIN;
```

Socket Example: Client (continued)

```
/* connect to the server */
if (connect(socketfd, (struct sockaddr *) &server_addr,
           sizeof(server_addr)) < 0) {
    perror("client: can't connect socket ");
    exit(1);
}

/* send from the client to the server */
num = write(socketfd, CLIENT_STR, CLIENT_BYTES);
if (num < 0) {
    perror("client: write to socket failed\n");
    exit(1);
}
assert(num == CLIENT_BYTES);
```

Socket Example: Client (continued)

```
/* receive data sent back by the server */
total_read = 0;
while (total_read < SERVER_BYTES) {
    num = read(socketfd, &read_buf[total_read],
        SERVER_BYTES - total_read);
    if (num < 0) {
        perror("client: read from socket failed\n");
        exit(1);
    }
    total_read += num;
}

printf("sent %s\n", CLIENT_STR);
printf("received %s\n", read_buf);

close(socketfd);
exit(0);
} /* main */
```

Socket Example: Server

```
#include "defs.h"

int
main()
{
    int serverfd, clientfd;
    struct sockaddr_in server_addr, client_addr;
    int size, num;
    char read_buf[BUF_LEN];
    struct sockaddr_in bound_addr;

    serverfd = socket(DOMAIN, SOCK_STREAM, 0);

    if (serverfd < 0) {
        perror("server: unable to create socket ");
        exit(1);
    }
}
```

Socket Example: Server (continued)

```
/* zero the server_addr structure */
memset((char *) &server_addr, 0, sizeof (server_addr));

/* set up addresses server will accept connections on */
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
server_addr.sin_port = htons(PORT);
server_addr.sin_family = DOMAIN;

/* assign address to the socket */
if (bind (serverfd, (struct sockaddr *) &server_addr,
    sizeof(server_addr)) < 0) {
    perror("server: unable to bind socket ");
    exit(1);
}

/* Willing to accept connections on this socket. */
/* Maximum backlog of 5 clients can be queued */
listen(serverfd, 5);
```

Socket Example: Server (continued)

```
for (;;) {
    /* wait for and return next completed connection */
    size = sizeof(client_addr);
    if ((clientfd = accept(serverfd,
        (struct sockaddr *) &client_addr, &size)) < 0) {
        perror("server: accept failed ");
        exit(1);
    }

    /* get the data sent by the client */
    total_read = 0;
    while (total_read < CLIENT_BYTES) {
        num = read(clientfd, &read_buf[total_read],
            CLIENT_BYTES - total_read);
        if (num < 0) {
            perror("server: read from client socket failed ");
            exit(1);
        }
        total_read += num;
    }
}
```


Socket Example: Server (continued)

```
/* process the client info / request here */
printf("client sent %s\n", read_buf);
printf("server sending %s\n", SERVER_STR);

/* send the data back to the client */
num = write(clientfd, SERVER_STR, SERVER_BYTES);
if (num < 0) {
    perror("server: write to client socket failed ");
    exit(1);
}
assert(num == SERVER_BYTES);

close(clientfd);
} /* for */
exit(0);
} /* main */
```

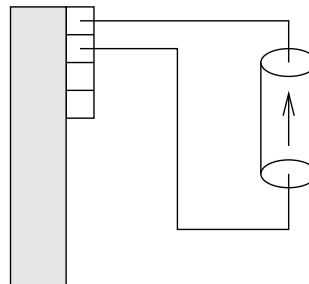
Pipes

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

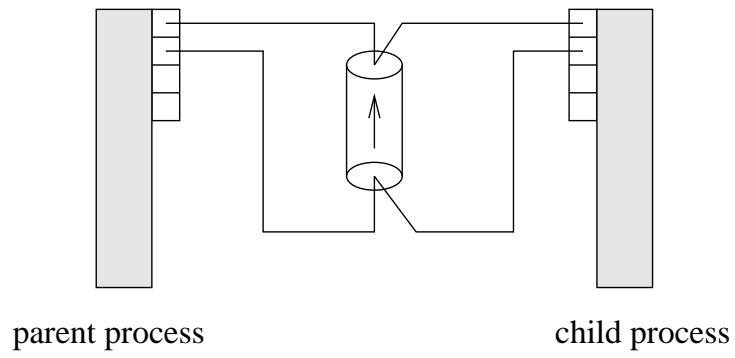
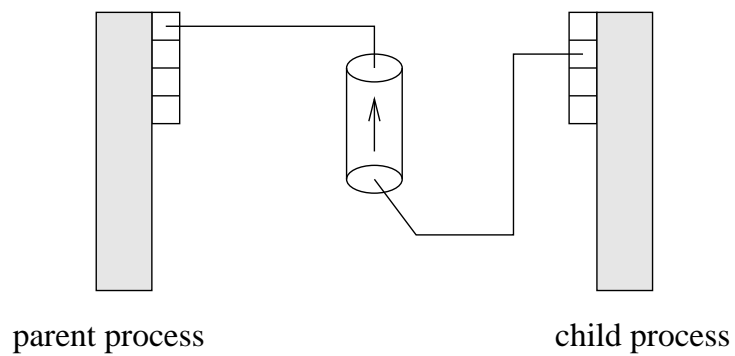
One-way Child/Parent Communication Using a Simplex Pipe

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

Illustration of Example (after pipe())



parent process

Illustration of Example (after `fork()`)**Illustration of Example (after `close()`)**

Examples of Other Interprocess Communication Mechanisms

named pipe:

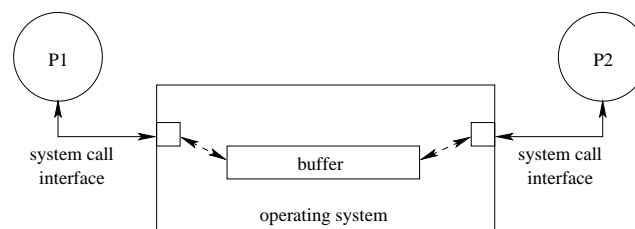
- similar to pipes, but with an associated name (usually a file name)
- name allows arbitrary processes to communicate by opening the same named pipe
- must be explicitly deleted, unlike an unnamed pipe

message queue:

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

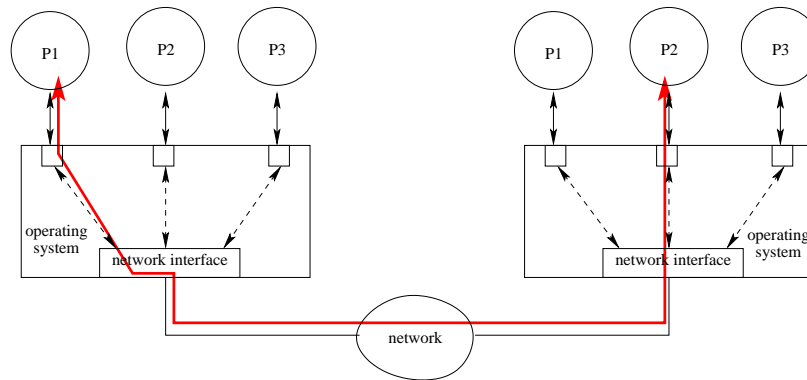
Implementing IPC

- 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

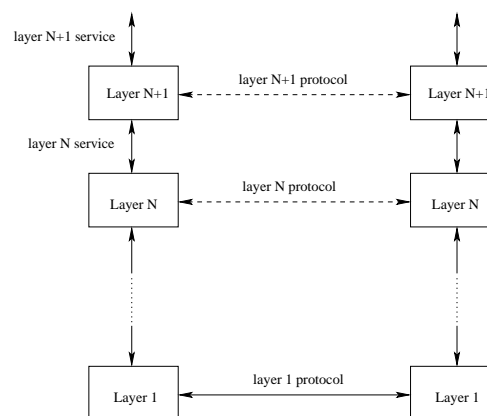
- some sockets can be used to connect processes that are running on different machine
- the kernel:
 - controls access to network interfaces
 - multiplexes socket connections across the network



Networking Reference Models

- ISO/OSI Reference Model

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



- Internet Model
 - layers 1-4 and 7

Internet Protocol (IP): Layer 3

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

IP Routing Table Example

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

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

- routing table key:
 - destination:** ultimate destination of packet
 - gateway:** next hop towards destination (or “-” if destination is directly reachable)
 - interface:** which network interface to use to send this packet

Internet Transport Protocols

TCP: transport control protocol

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

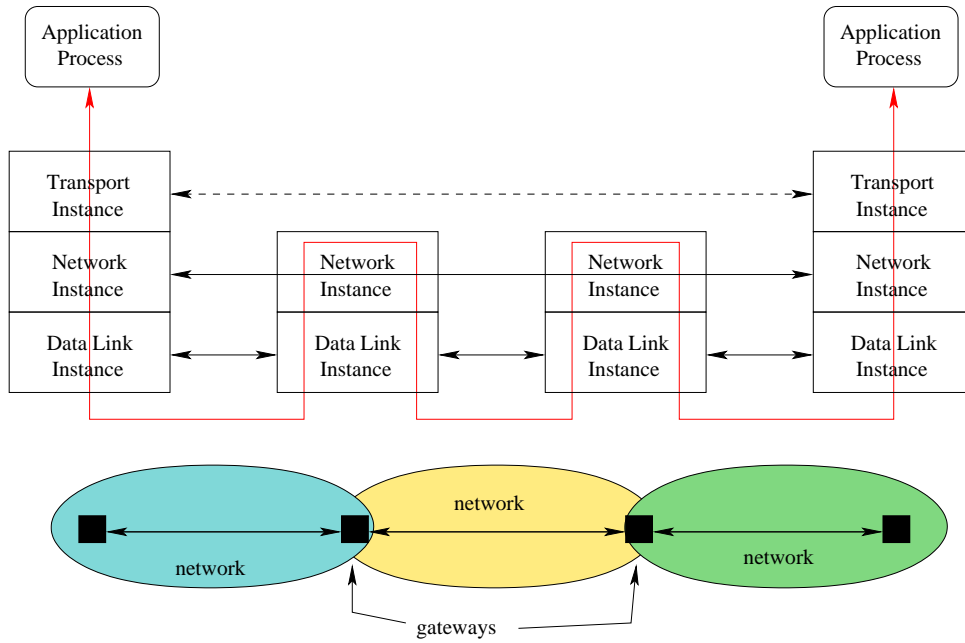
UDP: user datagram protocol

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

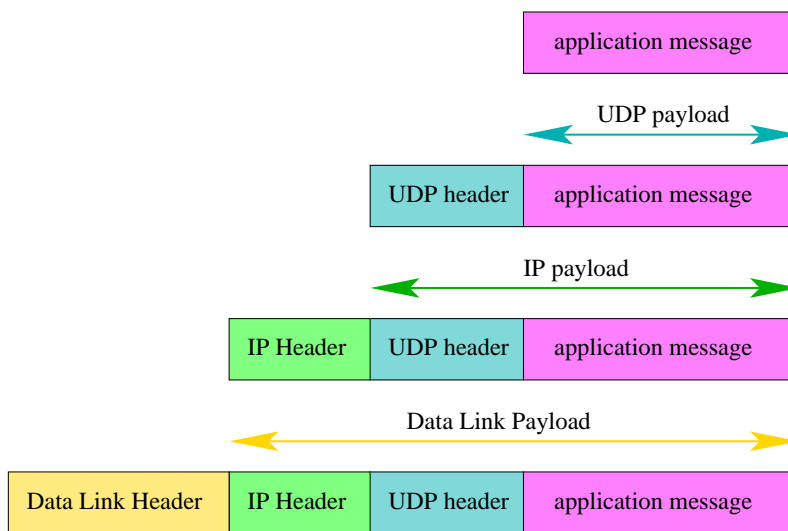
TCP and UDP Ports

- since there can be many TCP or UDP communications end points (sockets) on a single machine, there must be a way to distinguish among them
- each TCP or UDP address can be thought of as having two parts:
(machine name, port number)
- The machine name is the IP address of a machine, and the port number serves to distinguish among the end points on that machine.
- INET domain socket addresses are TCP or UDP addresses (depending on whether the socket is a stream socket or a datagram socket).

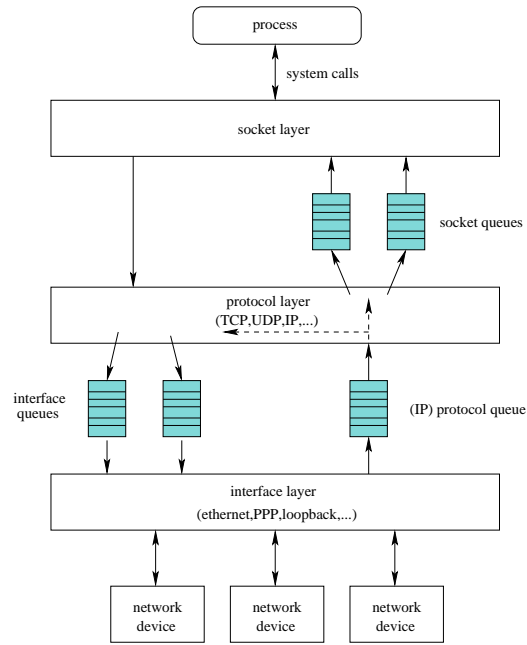
Example of Network Layers



Network Packets (UDP Example)



BSD Unix Networking Layers



Protection and Security

Protection: ensure controlled access to resources *internally* within a system

- OS provides mechanisms for policy enforcement
- Principle of least privilege: grant only enough privileges to complete task

Security: need to have adequate protection and consider *external* environment

- Security is hard because so many people try to break it.

Protection Domains

- Process should have access to specific objects and have the right to do specific things with each
- Rights should change to reflect what is needed at the time
- At any time, a process is operating in a protection domain that determines its access rights
- The domains should change to reflect actual needs (principle of least privilege)
- In most systems
 - domain changes are rare
 - more rights are granted than are needed

Protection Domains: Examples

- When you compile, should the compiler have access to all of your files?
E.g., mail, mp3 files, video
 - what is stopping it from transferring these files to another host
 - what is stopping it from deleting these files
- Often protection domain == user
All processes belonging to a user have the same rights
- Changing protection domains
UNIX `setuid`, effective user id becomes the same as file owner
Windows Server “execute as”
- Grant additional rights to specific programs, but not a solution to the first problem above

Protection: The Access Control Matrix

		Objects				
		1	2	3	4	5
Subjects	1		R		R	
	2			R,W		R,W
	3	R,X	R,W	R,W		
	4			R		

objects: the things to be protected, e.g., files, printers, etc.

subjects: users, groups, roles

matrix entries: access rights, i.e., operations allowed by a subject on an object

A common implementation is an *access control list* for each object.

Protection: Access Control Administration

- there must be a mechanism for changing the access rights described in the access control matrix
 - set of subjects is dynamic
 - set of objects is dynamic
 - access rights may need to change
- some approaches
 - encode access control change rights in the access control matrix
 - * add “owner” as a possible access right. Subject with owner rights on object x can change access rights in x 's column.
 - new users/subjects can inherit rights from others

Protection: Example – Access Rights in Unix

- subjects are users and groups (group membership is maintained separately)
- each object has an owner and a group
- access rights are specified for the owner, for the group, and for everyone else
- object access rights can be modified by the object owner
- major access rights are read, write, and execute
- access controls can be applied to files, devices, shared memory segments, and more.

Protection: Authentication

- object access is performed by processes
- to apply access controls, it is necessary to associate processes with users
- this requires user *authentication*
- some authentication techniques:
 - passwords
 - cryptographic (e.g., public key methods)
 - physical tokens (e.g., smart cards)
 - biometrics

Security

OS / Network threats:

- Trojan Horses
- Trap Doors
- Buffer Overflows
- Worms
- Viruses
- Other (specific examples)

Even Worse:

- Physical
- Human

Trojan Horses

- Two different meanings:
 1. A programs that intends to run and do something undesirable
E.g., download a program/game that also erases all of your files
 2. User tricked into running it
Unix PATH variable includes “.” as first entry
% cd /home/username; ls BUT
/home/username/ls is actually
cd \$HOME; /bin/rm -rf *

Fundamental problem: privilege of command determined by user

Trap Doors / Back Doors

Lets perpetrator do things they wouldn't normally be allowed to do.

For example,

- when run by root
- surprise/undocumented response to a special input
game cheats (type sequence of characters that make you invincible)
- special/alternate login program allows author access via special user name

Usually obscure, casual examination of source would miss it

Buffer Overflows

Send more data than is expected by executing program

Many programs have fixed-sized input buffers but don't limit input length

Data contains executable code

Consequences:

- Crash
- Much worse when buffer is on the stack (execute arbitrary code)

Very machine dependent but the X86 family is very wide-spread

Worms

- Program that replicates itself from one system to another (usually via Internet)
- Tend to consume resources leading to a Denial of Service (DOS) attack
- Arbitrarily bad behaviour
- Often use buffer overflows to compromise systems/users

Morris Worm:

- Spread using legitimate access of compromised users (e.g., .rhosts)
- 1988 – 3 yrs probation, 400 hrs community service, \$10K

Sobig:

- Mail to all users in the address book
- Modifies system parameters to restart worm when rebooted (registry)

Viruses

- Not a free standing program, but a fragment attached to a legitimate program
- Essentially a dynamic Trojan horse or trap door.
- Especially a problem on single user systems with weak or non-existent protection
 - Makes it easy to infect file systems
- Microsoft Office Macros and/or Active X lead to problems with Word files, email and web content
 - no need to reboot
 - execution not even expected
- Denial of Service often occurs at the same time (as a result of rapid spreading)

Specific Examples / Hacks

- Allocate virtual memory and look for data
- Illegal system calls or calls with invalid number of parameters or types
- Modify any OS data structures stored in user data e.g., `open ()` call
- Look for don't do XXX in the documentation
- Social Engineering

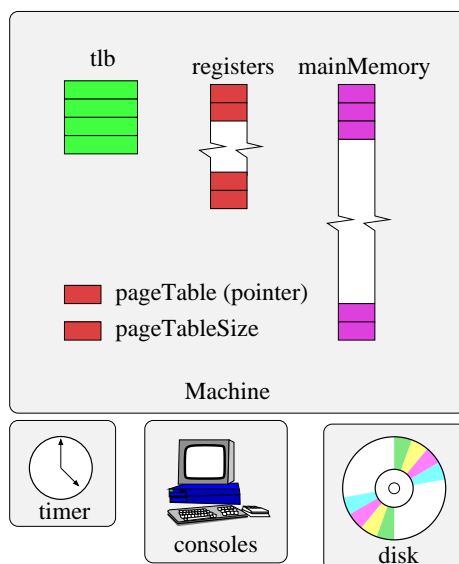
What is NachOS?

workstation simulator: the simulated workstation includes a MIPS processor, main memory, and a collection of devices including a timer, disk(s), a network interface, and input and output consoles.

operating system: the NachOS operating system manages the simulated workstation, and implements a set of system calls for user programs

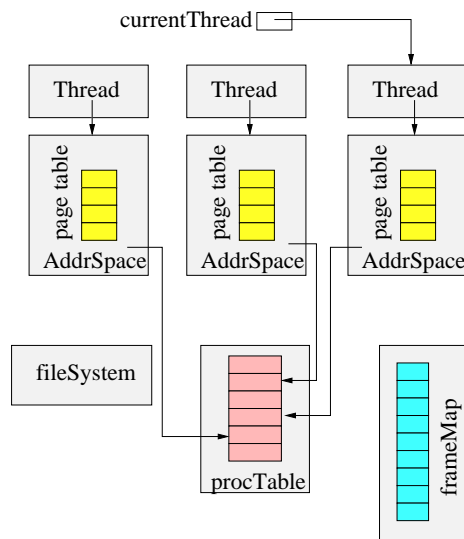
user programs: NachOS user programs run on the simulated machine, and use services provided by the NachOS operating system

The NachOS Machine Simulator



- registers include the program counter and stack pointer
- main memory consists of `NumPhysPages` frames, each of size `PageSize`
- some devices (e.g., network, second disk) are not shown
- simulator uses *either* the TLB or the `pageTable` and `pageTableSize` registers, but not both.

Some Components of The NachOS Kernel



- not all OS components are shown
- each NachOS process has an entry in ProcTable, a thread, and an address space
- address spaces are implemented by AddrSpace objects, which include a page table
- frameMap tracks which main memory frames are in use
- NachOS has two file system implementations

How does NachOS differ from a “real” OS?

- The NachOS operating system runs *beside* the simulated workstation, not *on* it. This means that the operating system and user programs (which run *on* the simulated workstation) do not share system resources.
- The NachOS operating system controls simulated devices through a set of abstract device interfaces. Instead of executing special I/O instructions or writing codes into device control registers, the operating system calls methods like `Disk::ReadRequest`.

Review: MIPS Register Usage

R0, \$0 =
R2, \$2 =
R4, \$4 =
R5, \$5 =
R6, \$6 =
R7, \$7 =
R29, \$29 =
R30, \$30 =
R31, \$31 =

System Calls

- to perform a system call, a user program executes a MIPS `syscall` instruction, as usual.
- to simulate the `syscall` instruction, the simulator's `Machine::Run` method (indirectly) calls the kernel's `ExceptionHandler` function. (`userprog/exception.cc`)
- `ExceptionHandler` performs any kernel operations that are needed to implement the system call.
- When `ExceptionHandler` returns, control goes back the `Machine::Run` and the user program simulation picks up from where it left off, just as in real life.

The call to `ExceptionHandler` is the switch from user mode to system mode. The return from `ExceptionHandler` to `Machine::Run` is the switch from system mode back to user mode.

How a System Call Works (example C program)

```
/* call.c
 * Show how a function/syscall is made.
 */
#include "syscall.h"

int
main()
{

    Write("Hello World\n", 12, 1);
}
```

How a System Call Works (compiled program - part 1)

```
.file 1 "call.c"
.rdata
.align 2
$LC0:
.ascii "Hello World\n\000"
.text
.align 2
.globl main
.ent main
```

How a System Call Works (compiled program - part 2)

```
main:
    .frame    $fp,24,$31
    .mask    0xc0000000,-4
    .fmask    0x00000000,0
    subu    $sp,$sp,24
    sw     $31,20($sp)
    sw     $fp,16($sp)
    move   $fp,$sp
    jal    __main
    la     $4,$LC0
    li     $5,0x0000000c
    li     $6,0x00000001
    jal    Write
```

How a System Call Works (compiled program - part 3)

```
$_L1:
    .set    noreorder
    move   $sp,$fp
    lw     $31,20($sp)
    lw     $fp,16($sp)
    j     $31
    addu   $sp,$sp,24
    .set   reorder
    .end   main
```

How a System Call Works (call stub from start.s)

```
        .globl Write
        .ent    Write
Write:
        addiu $2,$0,SC_Write
        syscall
        j      $31
        .end Write
```

Some System Call Codes (from start.s)

```
/* system call codes */
#define SC_Halt      0
#define SC_Exit     1
#define SC_Exec     2
#define SC_Join     3
#define SC_Create   4
#define SC_Open     5
#define SC_Read     6
#define SC_Write    7
#define SC_Close    8
#define SC_Fork     9
#define SC_Yield   10
```

How a System Call Works (NachOS exception handler)

```
void ExceptionHandler(ExceptionType which)
{
    int type = kernel->machine->ReadRegister(2);

    kernel->currentThread->SaveUserState();
    kernel->currentThread->space->SaveState();

    switch(which) {
    case SyscallException:
        switch(type) {

        case SC_Write:
            vaddr = kernel->machine->ReadRegister(4);
            len = kernel->machine->ReadRegister(5);
            fileID = kernel->machine->ReadRegister(6);
            retval = WriteHandler(fileID, vaddr, len);
            break;
```

Exceptions and Interrupts

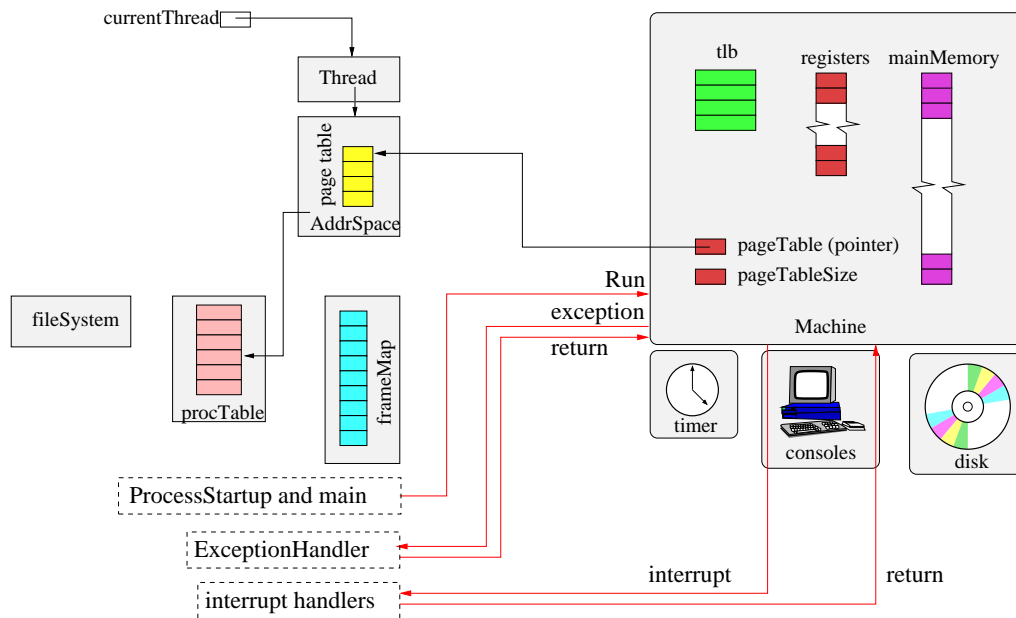
Exceptions: Exceptions are handled in the same way as system calls. If a user program instruction causes an exception, the simulator (`Machine::Run`) calls `ExceptionHandler` so that it can be handled by the kernel

Interrupts:

- The simulator keeps track of the simulation time at which device interrupts are supposed to occur.
- After simulating each user instruction, the simulator advances simulation time and determines whether interrupts are pending from any devices.
- If so, the simulator (`Machine::Run`) calls the kernel's handler for that interrupt before executing the next instruction.
- When the kernel's handler returns, the simulation continues executing instructions.

The kernel has a handler function for each type of interrupt (timer, disk, console input, console output, network).

Summary of Machine/Kernel Interactions in NachOS



NachOS Thread Facilities

Threads: new threads can be created, and threads can be destroyed. Each new thread executes a kernel procedure that is specified when the thread is created. (`threads/thread.*`)

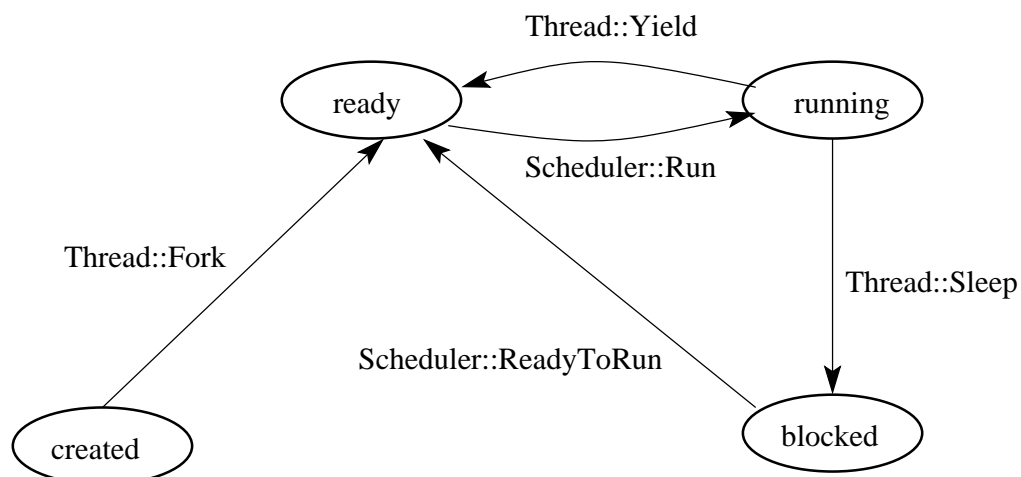
Scheduling: a round-robin ready queue for threads (`threads/scheduler.*`)

Synchronization: semaphores, locks, and condition variables. These are integrated with the scheduler: blocked threads are kept off of the ready queue, unblocked threads are placed back onto the ready queue. (`threads/synch.*`)

Context Switches in NachOS

- The user context of a thread can be saved in the Thread object.
- The thread's user context includes the values in the registers of the simulated machine, including the program counter and the stack pointer.
- When switching from one thread to another, the kernel:
 - saves the old thread's user context
 - restores the new thread's user context
- When the new thread returns to user mode, its own user context is in the simulated machine's registers.

NachOS Thread Scheduling



Birth of a NachOS Process

- the creator does:
 1. update the process table
 2. create and initialize an address space (allocate physical memory, set up page table, load user program and data into allocated space)
 3. create a new thread and put it on the ready queue. The new thread executes the kernel function `ProcessStartup`.
- the `ProcessStartup` function does:
 1. Initialize the registers of the simulated machine (page table pointer, program counter, stack pointer, and general registers)
 2. Call `Machine::Run`. This call never returns.

`Machine::Run` starts simulation of the user program. *This corresponds to an exception return in a real system.* The thread is now simulating the execution of user program code. That is, it is in user mode.

Starting Up a User Program (`__start`)

```
/* In Nachos all user programs are linked
 * with start.s, it begins at virtual address 0
 */
.globl __start
.ent    __start
__start:
    jal    main
    move   $4,$0
    jal    Exit
    .end  __start
```

NachOS Workstation Devices

- like many real devices, the NachOS workstation's simulated devices are *asynchronous*, which means that they use interrupts to notify the kernel that a requested operation has been completed, or that a new operation is possible. For example:
 - the input console (keyboard) generates an interrupt each time a new input character is available
 - the output console (display) can only output one character at a time. It generates an interrupt when it is ready to accept another character for output.
 - the disk accepts one read/write request at a time. It generates an interrupt when the request has been completed.
- the kernel implements *synchronous* interfaces to each of these devices
 - implemented using the synchronization primitives
 - synchronous interfaces are much easier for the rest of the kernel to use than the asynchronous interfaces. Use them!

Example: Synchronous Input Console

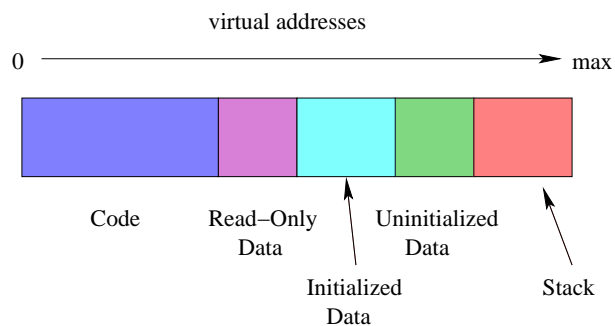
- `SynchConsoleInput::GetChar()` returns one character from the console, and causes the calling thread to *block* (until a character is available) if there are no available input characters.
- Implementation uses a single semaphore:
 - `SynchConsoleInput::GetChar()` does a `P()` before attempting to read a character from the input console.
 - Input console interrupt handler does a `V()`

Address Spaces

- One AddrSpace object per NachOS process.
- AddrSpace maintains the process page table, and provides methods for reading and writing data from virtual addresses.
- NachOS page table entry:

```
class TranslationEntry {
public:
    int virtualPage; // page number
    int physicalPage; // frame number
    bool valid; // is this entry valid?
    bool readOnly; // is page read-only?
    bool use; // used by replacement alg
    bool dirty; // used by replacement alg
};
```

Address Space Layout



- Size of each segment except stack is specified in NOFF file
- Code, read-only data and initialized data segments are initialized from the NOFF file. Remaining segments are initially zero-filled.
- Segments are page aligned.

C Code for Segments Example

```
#define N    (5)

unsigned int x = 0xdeadbeef;
int y = 0xbb;
const int blah = 0xff;
int data[N];

struct info {
    int x;
    int y;
};
```

C Code for Segments Example (cont'd)

```
main()
{
    int i;
    int j = 0xaa;
    int k;
    const int l = 0xee;
    char *str = "Hello World\n";

    for (i=0; i<N; i++) {
        data[i] = i;
    }
}
```

Coff2noff Output for Segments

Loading 4 sections:

```
".text"  filepos  52  (0x34)      mempos    0  (0x0)
         size 736 (0x2e0)
".rdata" filepos 788 (0x314)      mempos   768 (0x300)
         size  32 (0x20)
".data"  filepos 820 (0x334)      mempos   896 (0x380)
         size  16 (0x10)
".bss"   filepos -1 (0xffffffff) mempos  1024 (0x400)
         size  20 (0x14)
<not in file>
```

Some Output from objdump

```
Contents of section .rdata:
0300 ff000000 48656c6c 6f20576f 726c640a
     ....Hello World.
0310 00000000 00000000 00000000 00000000
     .....
Contents of section .data:
0380 efbeadde bb000000 00000000 00000000
     .....
```

The NachOS Stub File System

- NachOS has two file system implementations.
 - The real file system has very limited functionality. Files are stored on the workstation's simulated disk.
 - The “stub” file system stores files outside of the simulated machine, in the file system of the machine on which NachOS is running. Magic!
- Until Asst 3, the “stub” file system is used. This is why a file that is created by a NachOS user program appears on the machine on which NachOS is running. This is also why NachOS user programs can be stored in files on host machine, and not on the simulated workstation.
- The “stub” file system may seem unrealistic, however, a diskless workstation with network boot uses a similar mechanism.

The NachOS File System: disk.h

```
#define SectorSize      128 // bytes
#define SectorsPerTrack 32
#define NumTracks       64 // per disk
#define NumSectors (SectorsPerTrack * NumTracks)
                    // 32 * 64 = 2048
```

Disk Size = 2048 sectors * 128 bytes = 256 KB

The NachOS File System: `fileys.h`

```
#define FreeMapSector      0
#define DirectorySector   1

#define FreeMapFileSize (NumSectors / BitsInByte)
                        // 2048 / 8 = 256

#define NumDirEntries     10
#define DirectoryFileSize
    (sizeof(DirectoryEntry) * NumDirEntries)
    // 20 * 10 = 200 (1.5625 sectors)
```

The NachOS File System: `fileys.h` (cont'd)

```
class FileSystem {
    ...
private:
    // Bit map of free disk blocks,
    // represented as a file
    OpenFile* freeMapFile;

    // "Root" directory -- list of
    // file names, represented as a file
    OpenFile* directoryFile;
};
```

FreeMap file has 2048 entries and occupies 2 sectors (256 bytes), plus one sector for its header. Directory file has 10 entries, which requires 200 bytes (2 sectors), plus one sector for its header.

The NachOS File System: filehdr.h

```
#define NumDirect
  ((SectorSize - 2 * sizeof(int))
   / sizeof(int))
#define MaxFileSize
  (NumDirect * SectorSize)

class FileHeader {
  ...
private:
  int numBytes;
  int numSectors;           // data sectors
  int dataSectors[NumDirect]; // sector numbers
  ...
```

The NachOS File System: filehdr.h (cont'd)

- FileHeader fits in one sector = 128 bytes
- first two fields (numBytes and numSectors) use 8 bytes
- 120 bytes are left for block pointers
- each block pointer requires 4 bytes, so

$$\text{NumDirect} = \frac{128 - 2 * 4}{4} = 30$$

- maximum file size is:

$$\text{MaxFileSize} = \text{NumDirect} * \text{SectorSize} = 30 * 128 = 3840$$

The NachOS File System: directory.h

```
#define FileNameMaxLen 9
// for simplicity, we assume
// file names are <= 9 characters long

class DirectoryEntry {
public:
    bool inUse;
    int sector;
    char name[FileNameMaxLen + 1];
    // Text name for file, with +1 for
    // the trailing '\0'
};
```

4 bytes for inUse, 4 bytes for sector, 10 bytes for name.

File System Command Line Example

```
mobey 1% nachos -f
Machine halting!
```

```
Ticks: total 14010, idle 14000, system 10, user 0
Disk I/O: reads 3, writes 6
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```


File System Command Line Example (part 6)

Directory file header:

FileHeader contents. File size: 200. File blocks:
4 5

File contents:

```
\1\0\0\0\0\0\0\0\6File1\0\0\0\0\0\0\0\1\0\0\0\0\0\0\8
File2\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
```

Bitmap set:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9,

File System Command Line Example (part 7)

Directory contents:

Name: File1, Sector: 6

FileHeader contents. File size: 6. File blocks:
7

File contents:

Hello\a

Name: File2, Sector: 8

FileHeader contents. File size: 6. File blocks:
9

File contents:

World\a

Machine halting!

Using Memory

```
#include <stdio.h>
#include <stdlib.h>

struct foo_struct {
    int x;
    char a; char b; char c; char d;
};

int
main()
{
    int i;
    char a[40];
    int *iptr = (int *) a;
    struct foo_struct *sptr = (struct foo_struct *) a;
```

Using Memory

```
    for (i=0; i<40; i++) {
        a[i] = (char) i;
    }

    for (i=0; i<10; i++) {
        printf("%2d = 0x%08x\n", i, iptr[i]);
    }

    printf("x = 0x%08x  a = %d  b = %d  c = %d  d = %d\n",
        sptr[0].x, (int) sptr[0].a, (int) sptr[0].b,
        (int) sptr[0].c, (int) sptr[0].d);

    exit(0);
}
```

Using Memory: Example Output (x86)

```
0 = 0x03020100
1 = 0x07060504
2 = 0x0b0a0908
3 = 0x0f0e0d0c
4 = 0x13121110
5 = 0x17161514
6 = 0x1b1a1918
7 = 0x1f1e1d1c
8 = 0x23222120
9 = 0x27262524
x = 0x03020100  a = 4  b = 5  c = 6  d = 7
```

Using Memory: Example Output (Sparc)

```
0 = 0x00010203
1 = 0x04050607
2 = 0x08090a0b
3 = 0x0c0d0e0f
4 = 0x10111213
5 = 0x14151617
6 = 0x18191a1b
7 = 0x1c1d1e1f
8 = 0x20212223
9 = 0x24252627
x = 0x00010203  a = 4  b = 5  c = 6  d = 7
```

Writing to a File

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int
main()
{
    int i, rc, fd;
    int array[40];
```

Writing to a File

```
    for (i=0; i<40; i++) {
        array[i] = i;
    }

    fd = open("test-output", O_WRONLY | O_CREAT, S_IRWXU);

    if (fd < 0) {
        perror("open failed");
        exit(1);
    }
```


Writing to a File

```
rc = write(fd, array, sizeof(array));

if (rc < 0) {
    perror("write failed");
    exit(1);
}

close(fd);
exit(0);
}
```

Writing to a File: Example Output

```
% cat test-output
#@u
!$%
```

Writing to a File: Example Output (x86)

```
# Print offsets and values in Hex (x)
xxd test-output
0000000: 0000 0000 0100 0000 0200 0000 0300 0000  ....
0000010: 0400 0000 0500 0000 0600 0000 0700 0000  ....
0000020: 0800 0000 0900 0000 0a00 0000 0b00 0000  ....
0000030: 0c00 0000 0d00 0000 0e00 0000 0f00 0000  ....
0000040: 1000 0000 1100 0000 1200 0000 1300 0000  ....
0000050: 1400 0000 1500 0000 1600 0000 1700 0000  ....
0000060: 1800 0000 1900 0000 1a00 0000 1b00 0000  ....
0000070: 1c00 0000 1d00 0000 1e00 0000 1f00 0000  ....
0000080: 2000 0000 2100 0000 2200 0000 2300 0000  ...!...
0000090: 2400 0000 2500 0000 2600 0000 2700 0000  $...%...&
```

Writing to a File: Example Output (Sparc)

```
# Print offsets and values in Hex (x)
xxd test-output
0000000: 0000 0000 0000 0001 0000 0002 0000 0003  ....
0000010: 0000 0004 0000 0005 0000 0006 0000 0007  ....
0000020: 0000 0008 0000 0009 0000 000a 0000 000b  ....
0000030: 0000 000c 0000 000d 0000 000e 0000 000f  ....
0000040: 0000 0010 0000 0011 0000 0012 0000 0013  ....
0000050: 0000 0014 0000 0015 0000 0016 0000 0017  ....
0000060: 0000 0018 0000 0019 0000 001a 0000 001b  ....
0000070: 0000 001c 0000 001d 0000 001e 0000 001f  ....
0000080: 0000 0020 0000 0021 0000 0022 0000 0023  ... ..!
0000090: 0000 0024 0000 0025 0000 0026 0000 0027  ...$.%..
```

Arrays and Addresses

```
#include <stdio.h>
#include <stdlib.h>

static char *alpha = "abcdefghijklmnopqrstuvwxyz";

int
main()
{
    char a[12];
    char *b = 0;
    int i;
```

Arrays and Addresses

```
for (i=0; i<12; i++) {
    a[i] = alpha[i];
}

printf("addr of a = %p\n", &a);
printf("addr of a[0] = %p\n", &a[0]);
printf("*a = %c\n", *a);
printf("addr of b = %p\n", &b);
printf("addr of b[0] = %p\n", &b[0]);
printf("b = %p\n", b);
printf("\n");
```

Arrays and Addresses

```
b = a;
printf("addr of b = %p\n", &b);
printf("addr of b[0] = %p\n", &b[0]);
printf("b = %p\n", b);
printf("*b = %c\n", *b);
printf("\n");

b = &a[4];
printf("addr of b = %p\n", &b);
printf("addr of b[0] = %p\n", &b[0]);
printf("b = %p\n", b);
printf("*b = %c\n", *b);
printf("\n");

exit(0);
}
```

Arrays and Addresses: Example Output (x86)

```
addr of a = 0xbfe79b88
addr of a[0] = 0xbfe79b88
*a = a
addr of b = 0xbfe79b84
addr of b[0] = (nil)
b = (nil)

addr of b = 0xbfe79b84
addr of b[0] = 0xbfe79b88
b = 0xbfe79b88
*b = a

addr of b = 0xbfe79b84
addr of b[0] = 0xbfe79b8c
b = 0xbfe79b8c
*b = e
```

Additional Notes:

Additional Notes:

Additional Notes:

Additional Notes: