# OPERATING SYSTEM
# 5CS4-03

Dr. SONAM MITTAL

Associate Professor

Dept. of Information Technology

B K Birla Institute of Engineering & Technology, Pilani

## What is an Operating System?

- An *operating system* (OS) is the interface between the user and the hardware
  - It implements a virtual machine that is easier to program than bare hardware
- An OS provides standard **services** (an interface) which are implemented on the hardware, including:
  - Processes, CPU scheduling, memory management, file system, networking
- The OS **coordinates** multiple applications and users (multiple processes) in a fair and efficient manner

↪The goal in OS development is to make the machine **convenient** to use (a software engineering problem) and **efficient** (a system and engineering problem)

## Why Study Operating Systems?

- Abstraction — how do you give the users the illusion of infinite resources (CPU time, memory, file space)?
- System design —tradeoffs between:
  - performance and convenience of these abstractions
  - performance and simplicity of OS
  - functionality in hardware or software
- Primary intersection point — OS is the point where hardware, software, programming languages, data structures, and algorithms all come together
- Curiosity — “look under the hood”
- “Operating systems are among the most complex pieces of software yet developed”, William Stallings, 1994

## Modern OS Functionality

- Concurrency
  - Multiple processes active at once
  - Processes can communicate
  - Processes may require mutually-exclusive access to some resource
  - CPU scheduling, resource management
- Memory management — allocate memory to processes, move processes between disk and memory
- File system — allocate space for storage of programs and data on disk
- Networks and distributed computing — allow computers to work together
- Security & protection

## What is an Operating System?

- A **magician** — provides each user with the illusion of a dedicated machine with infinite memory and CPU time
- A **government** — allocates resources efficiently and fairly, protects users from each other, provides safe and secure communication
- A **parent** — always there when you need it, never breaks, always succeeds
- A **fast food restaurant** — provides a service everyone needs, always works the same everywhere (standardization)
- A **complex system** — but keep it as simple as possible so that it will work

## What is an Operating System? (Review)

- An *operating system* (OS) is the interface between the user and the hardware
  - It implements a virtual machine that is easier to program than bare hardware
- An OS provides standard **services** (an interface) which are implemented on the hardware, including:
  - Processes, CPU scheduling, memory management, file system, networking
- The OS **coordinates** multiple applications and users (multiple processes) in a fair and efficient manner

➥The goal in OS development is to make the machine **convenient** to use (a software engineering problem) and **efficient** (a system and engineering problem)

## History of Operating Systems

- Phase 0 — hardware is a very expensive experiment; no operating systems exist
  1. One user at console
     - One function at a time (computation, I/O, user think/response)
     - Program loaded via card deck
       - Libraries of device drivers (for I/O)
     - User debugs at console
- Phase 1 — hardware is expensive, humans are cheap
  2. Simple batch processing: load program, run, print results, dump, repeat
     - User gives program (cards or tape) to the operator, who schedules the jobs
     - *Resident monitor* automatically loads, runs, dumps user jobs
     - Requires memory management (relocation) and protection
     - More efficient use of hardware, but debugging is more difficult (from dumps)

## History of Operating Systems (cont.)

- Phase 1 — hardware is expensive, humans are cheap
  3. Overlapped CPU & I/O operations
     - First: buffer slow I/O onto fast tape drives connected to CPU, replicate I/O devices
     - Later: *spool* data to disk
  4. Multiprogrammed batch systems
     - Multiple jobs are on the disk,waiting to run
     - *Multiprogramming* — run **<u>several</u>** programs at the "same" time
       - Pick some jobs to run (*scheduling*), and put them in memory (*memory management*)
       - Run one job; when it waits on something (tape to be mounted, key to be pressed), switch to another job in memory
     - First big failures:
       - MULTICS announced in 1963, not released until 1969
       - IBM's OS/360 released with 1000 known bugs
     - OS design should be a science, not an art

## History of Operating Systems (cont.)

- Phase 2 — hardware is less expensive than before, humans are expensive
  5. Interactive *timesharing*
     - Lots of cheap terminals, one computer
       - All users interact with system at once
       - Debugging is much easier
     - Disks are cheap, so put programs and data online
       - 1 punch card = 100 bytes
       - 1MB = 10K cards
       - OS/360 was several feet of cards
     - New problems:
       - Need *preemptive scheduling* to maintain adequate *response time*
       - Need to avoid *thrashing* (swapping programs in and out of memory too often)
       - Need to provide adequate security measures
     - Success: UNIX developed at Bell Labs so a couple of computer nerds (Thompson, Ritchie) could play Star Trek on an unused PDP-7 minicomputer

## History of Operating Systems (cont.)

- Phase 3 — hardware is very cheap, humans are expensive
  6. Personal computing
     - CPUs are cheap enough to put one in each terminal, yet powerful enough to be useful
       - Computers for the masses!
     - Return to simplicity; make OS simpler by getting rid of support for multiprogramming, concurrency, and protection

---

- Modern operating systems are:
  - Enormous
    - Small OS = 100K lines of code
    - Big OS = 10M lines
  - Complex (100-1000 person year of work)
  - Poorly understood (outlives its creators, too large for one person to comprehend)

## History Lessons

- None of these operating systems were particularly bad; each depended on tradeoffs made at that point in time
  - Technology changes drive OS changes
- Since 1953, there has been about 9 orders of magnitude of change in almost every computer system component
  - Unprecedented! In past 200 years, gone from horseback (10 mph) to Concorde (1000 mph), only 2 orders of magnitude
- Changes in "typical" academic computer:

| | 1981 | 1996 |
|---|---|---|
| MIPS | 1 | 400 |
| price / MIPS | $100,000 | $50 |
| memory | 128 KByte | 64 MByte |
| disk | 10 MByte | 4 GByte |
| network | 9600 bit/sec | 155 Mb/s |
| address bits | 16 | 64 |

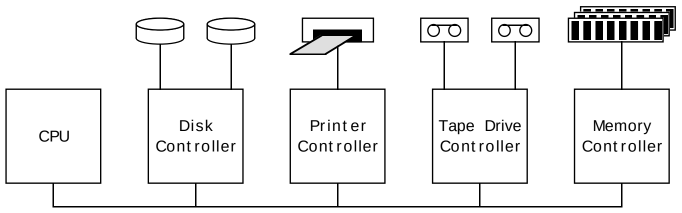
## Modern OS Functionality (Review)

- Concurrency
  - Multiple processes active at once
  - Processes can communicate
  - Processes may require mutually-exclusive access to some resource
  - CPU scheduling, resource management
- Memory management — allocate memory to processes, move processes between disk and memory
- File system — allocate space for storage of programs and data on disk
- Networks and distributed computing — allow computers to work together
- Security & protection

## More Recent Developments

- Parallel operating systems
  - Shared memory, shared clock
  - Large number of tightly-coupled processors
  - Appearance of single operating system
- Distributed operating systems
  - No shared memory, no shared clock
  - Small number of loosely-coupled processors
  - Appearance of single operating system is ideal goal, but not realized in practice
  - May try to simulate a shared memory
- Real-time operating systems
  - Meet hard / soft real-time constraints on processing of data

## Input / Output (I/O)



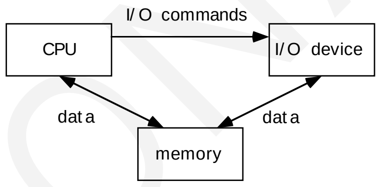
- CPU and device controllers all use a common bus for communication
- Software-polling synchronous I/O
  - CPU starts an I/O operation, and continuously *polls* (checks) that device until the I/O operation finishes
  - Device controller contains registers for communication with that device
    - Input register, output register — for data
    - Control register — to tell it what to do
    - Status register — to see what it's done
  - Why not connect all I/O devices directly to CPU? Why not… memory…?

## Input / Output (I/O) (cont.)

- Terminology
  - *Synchronous* I/O — CPU execution waits while I/O proceeds
  - *Asynchronous* I/O — I/O proceeds concurrently with CPU execution
- Interrupt-based asynchronous I/O
  - Device controller has its own processor, and executes asynchronously with CPU
    - Device controller puts an interrupt signal on the bus when it needs CPU's attention
  - When CPU receives an interrupt:
    1. It saves the CPU state and invokes the appropriate interrupt handler using the *interrupt vector* (addresses of OS routines to handle various events)
    2. Handler must save and restore software state (e.g., registers it will modify)
    3. CPU restores CPU state

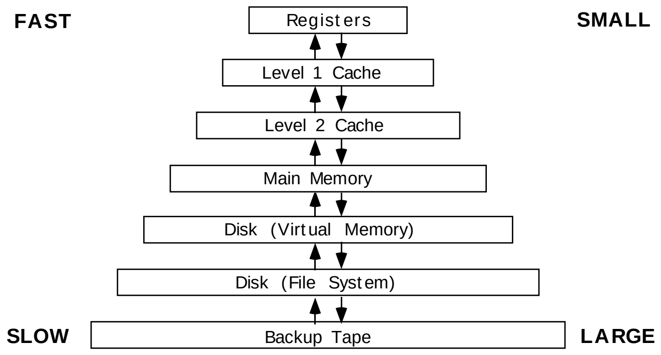
## Input / Output (I/O) (cont.)

- Memory-mapped I/O
  - Uses direct memory access (DMA) — I/O device can transfer block of data to / from memory without going through CPU



  - OS allocates buffer in memory, tells I/O device to use that buffer
  - I/O device operates asynchronously with CPU, interrupts CPU when finished
  - Used for most high-speed I/O devices (e.g., disks, communication interfaces)

## Storage Structures



- At a given level, memory may not be as big or as fast as you'd like it be
  - Tradeoffs between size and speed
- *Principle of Locality of Reference* leads to *caching*
  - When info is needed, look on this level
  - If it's not on this level, get it from the next slower level, and save a copy here in case it's needed again sometime soon

## Magnetic Disks

- Provide secondary storage for system (after main memory)
- Technology
  - Covered with magnetic material
  - Read / write head “floats” just above surface of disk
  - Hierarchically organized as platters, tracks, sectors (blocks)
- Devices
  - Hard (moving-head) disk — one or more platters, head moves across tracks
  - Floppy disk — disk covered with hard surface, read / write head sits on disk, slower, smaller, removable, rugged
  - CDROM — uses laser, read-only, high-density
    - Optical —read / write

## Protection

- Multiprogramming and timesharing require that the memory and I/O of the OS and user processes be **protected** against each other
  - Note that most PCs do not support this kind of protection
- Provide protection via two modes of CPU execution: *user mode* and *kernel mode*
  - In kernel / privileged / supervisor mode, *privileged instructions* can:
    - Access I/O devices, control interrupts
    - Manipulate the state of the memory (page table, TLB, etc.)
    - Halt the machine
    - Change the mode
  - Requires architectural support:
    - Mode bit in a protected register
    - Privileged instructions, which can only be executed in kernel mode

## I/O Protection

- To prevent illegal I/O, or simultaneous I/O requests from multiple processes, perform all I/O via privileged instructions
  - User programs must make a *system call* to the OS to perform I/O
- When user process makes a system call:
  - A *trap* (software-generated interrupt) occurs, which causes:
    - The appropriate trap handler to be invoked using the trap vector
    - Kernel mode to be set
  - Trap handler:
    - Saves state
    - Performs requested I/O (if appropriate)
    - Restores state, sets user mode, and returns to calling program

## Memory Protection

- Must protect OS’s memory from user programs (can’t overwrite, can’t access)
  - Must protect memory of one process from another process
  - Must not protect memory of user process from OS
- Simplest and most common technique:
  - *Base register* —smallest legal address
  - *Limit register* — size of address range
  - Base and limit registers are loaded by OS before running a particular process
  - CPU checks each address (instruction & data) generated in user mode
    - Any attempt to access memory outside the legal range results in a trap to the OS
- Additional hardware support is provided for virtual memory

## CPU Protection

- Use a timer to prevent CPU from being hogged by one process (either maliciously, or due to an infinite loop)
  - Set timer to interrupt OS after a specified period (small fraction of a second)
  - When interrupt occurs, control transfers to OS, which decides which process to execute for next time interval (maybe the same process, maybe another one)
- Also use timer to implement time sharing
  - At end of each time interval, OS switches to another process
  - *Context switch* = save state of that process, update Process Control Block for each of the two processes, restore state of next process

## Computer Architecture & OS

- Need for OS services often drives inclusion of architectural features in CPU:

| OS Service | Hardware Support |
|---|---|
| I/O | interrupts<br>memory-mapped I/O<br>caching |
| Data access | memory hierarchies<br>file systems |
| Protection | system calls<br>kernel & user mode<br>privileged instructions<br>interrupts & traps<br>base & limit registers |
| Scheduling &<br>Error recovery | timers |

## Process Management

- OS manages many kinds of activities:
  - User programs
  - System programs: printer spoolers, name servers, file servers, etc.
- Each is encapsulated in a *process*
  - A process includes the complete execution context (code, data, PC, registers, OS resources in use, etc.)
  - A *process* is ***not*** a *program*
    - A process is ***one*** instance of a program ***in execution***; many processes can be running the same program
- OS must:
  - Create, delete, suspend, resume, and schedule processes
  - Support inter-process communication and synchronization, handle deadlock

## Memory Management

- Primary (Main) Memory
  - Provides direct access storage for CPU
  - Processes must be in main memory to execute
- OS must:
  - Mechanics
    - Keep track of memory in use
    - Keep track of unused (“free”) memory
    - Protect memory space
    - Allocate, deallocate space for processes
    - Swap processes: memory <–> disk
  - Policies
    - Decide when to load each process into memory
    - Decide how much memory space to allocate each process
    - Decide when a process should be removed from memory

## File System Management

- File System
  - Disks (secondary storage) provide long-term storage, but are awkward to use directly
  - *File system* provides files and various operations on files
    - A *file* is a long-term storage entity, a named collection of persistent information that can be read or written
    - A file system supports directories, which contain files and other directories
      - Name, size, date created, date last modified, owner, etc.
- OS must:
  - Create and delete files and directories
  - Manipulate files and directories
    - Read, write, extend, rename, copy, protect
  - Provide general higher-level services
    - Backups, accounting, quotas

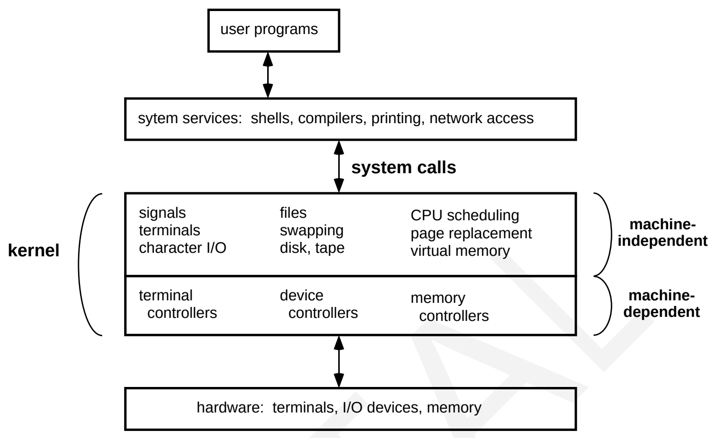
## Disk Management

- Disk
  - The actual hardware that sits underneath the file system
  - Large enough to store all user programs and data, application programs, entire OS
  - Persistent — endures system failures
- OS must:
  - Manage disk space at low level:
    - Keep track of used spaces
    - Keep track of unused (free) space
    - Keep track of “bad blocks”
  - Handle low-level disk functions, such as:
    - Scheduling of disk operations
    - Head movement
  - Note fine line between disk management and file system management

## System Calls

- Process control
  - end/abort this program
  - load/execute another program
  - create/terminate a process
    - get/set attributes
    - wait specified time
    - wait for event, signal event
- File manipulation
  - create/open/read/write/close/delete file
  - get/set attributes
- Device manipulation
  - request/read/write/release device
- Information
  - get/set time/date

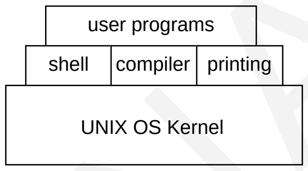
## One OS Structure: Large Kernel



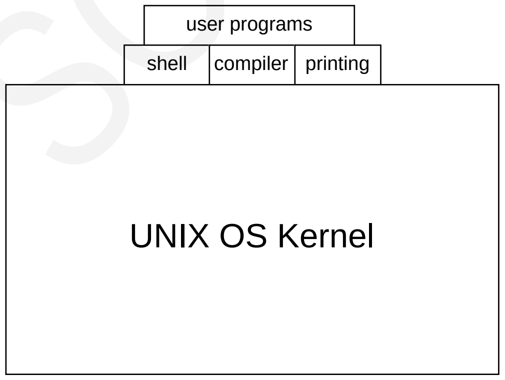
- The *kernel* is the protected part of the OS that runs in kernel mode
  - Critical OS data structures and device registers are protected from user programs
  - Can use privileged instructions

## Coping with Hugeness

- Ideal:



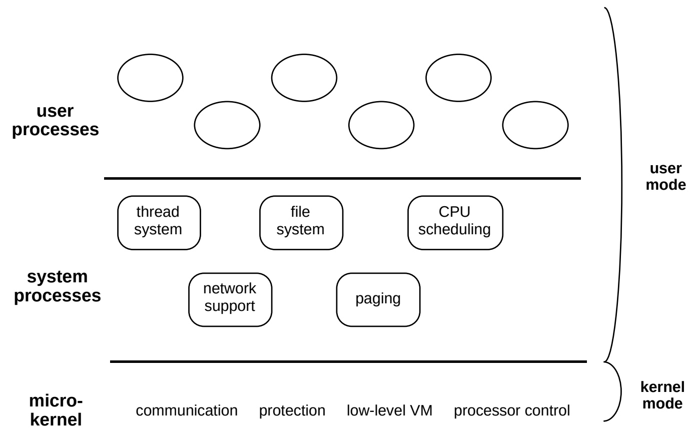
- Reality:



## OS Design Issues

- Another approach: layered OS
  - Divide OS into layers
  - Each layer uses services provided by next lower layer
    - User programs
    - Shell & compilers
    - CPU scheduling & memory management
    - Device drivers
    - Hardware
  - Advantages: modularity, simplicity
    - Disadvantages: performance
- Big tradeoff in OS design:
  - ➥ **simplicity** versus **performance**
  - ➥ Always strive for simplicity…
  - ➥ …Unless you have a strong reason to believe that complication is needed to achieve acceptable performance

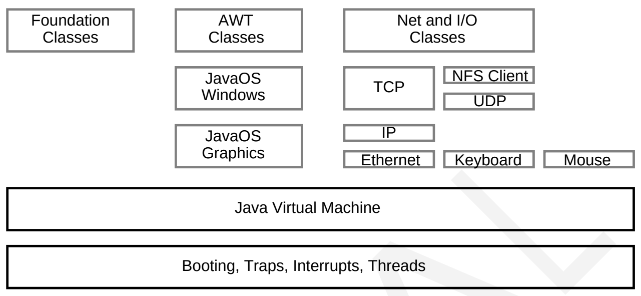
## Another OS Structure: Microkernel



- Goal is to minimize what goes in the kernel, implementing as much of the OS as possible in user-mode processes
  - Better reliability, easier extension
  - Lower performance (unfortunately)
- Examples: Mach (US), Chorus (France)

## The Future? Network Operating Systems

- Sun's JavaOS architecture:



  Details at http://java.sun.com/doc/white_papers.html

- No disk
- OS can only run a net browser
  - Get whatever the OS needs over the net
  - Get whatever application programs are needed over the net (as Java *applets*)

## Process

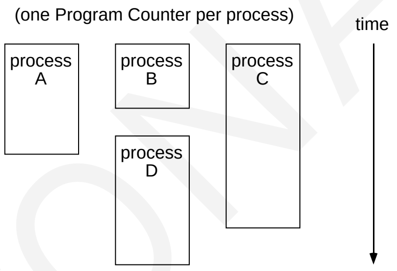
- A *process* (sometimes called a *task,* or a *job*) is, informally, a program in execution
- "Process" is not the same as "program"
  - We distinguish between a passive program stored on disk, and an actively executing process
    - Multiple people can run the same program; each running copy corresponds to a distinct process
  - The program is only part of a process; the process also contains the execution state
- List processes (HP UNIX):
  - ps — my processes, little detail
  - ps -fl — my processes, more detail
  - ps -efl — all processes, more detail
- Note user processes and OS processes

## Process Creation / Termination

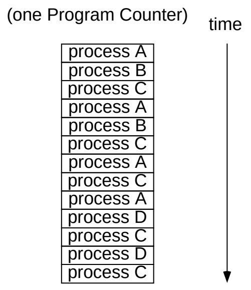
- Reasons for process creation
  - User logs on
  - User starts a program
  - OS creates process to provide a service (e.g., printer daemon to manage printer)
  - Program starts another process (e.g., netscape calls xv to display a picture)
- Reasons for process termination
  - Normal completion
  - Arithmetic error, or data misuse (e.g., wrong type)
  - Invalid instruction execution
  - Insufficient memory available, or memory bounds violation
  - Resource protection error
  - I/O failure

## Process Execution

- Conceptual model of 4 processes executing:

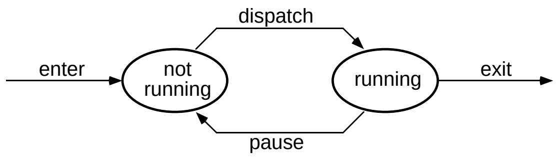


- Actual interleaved execution of the 4 processes:

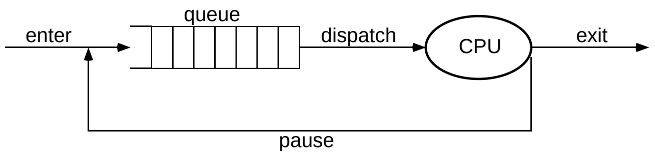


## A Two-State Process Model

- This process model says that either a process is *running*, or it is *not running*
- State transition diagram:



- Queuing diagram:



- CPU scheduling (round-robin)
  - Queue is first-in, first-out (FIFO) list
  - CPU scheduler takes process at head of queue, runs it on CPU for one time slice, then puts it back at tail of queue

## Process Transitions in the Two-State Process Model

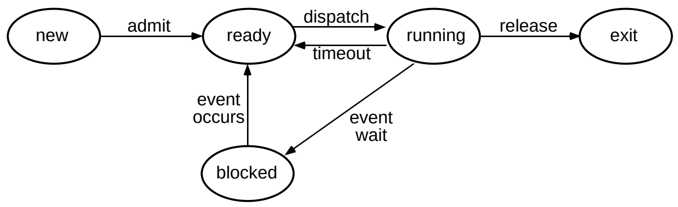
- When the OS creates a new process, it is initially placed in the **not-running** state
  - It's waiting for an opportunity to execute
- At the end of each time slice, the *CPU scheduler* selects a new process to run
  - The previously running process is *paused* — moved from the **running** state into the **not-running** state (at tail of queue)
  - The new process (at head of queue) is *dispatched* — moved from the **not-running** state into the **running** state
    - If the running process completes its execution, it exits, and the CPU scheduler is invoked again
    - If it doesn't complete, but its time is up, it gets moved into the **not-running** state anyway, and the CPU scheduler chooses a new process to execute

## Waiting on Something to Happen…

- Some reasons why a process that might otherwise be running needs to wait:
  - Wait for user to type the next key
  - Wait for output to appear on the screen
  - Program tried to read a file — wait while OS decides which disk blocks to read, and then actually reads the requested information into memory
  - Netscape tries to follow a link (URL) — wait while OS determines address, requests data, reads packets, displays requested web page
- OS must distinguish between:
  - Processes that are ready to run and are waiting their turn for another time slice
  - Processes that are waiting for something to happen (OS operation, hardware event, etc.)

## A Five-State Process Model

- The *not-running* state in the two-state model has now been split into a *ready* state and a *blocked* state
  - *Running* — currently being executed
  - *Ready* — prepared to execute
  - *Blocked* — waiting for some event to occur (for an I/O operation to complete, or a resource to become available, etc.)
  - *New* — just been created
  - *Exit* — just been terminated
- State transition diagram:



## State Transitions in Five-State Process Model

- new → ready
  - Admitted to ready queue; can now be considered by CPU scheduler
- ready → running
  - CPU scheduler chooses that process to execute next, according to some scheduling algorithm
- running → ready
  - Process has used up its current time slice
- running → blocked
  - Process is waiting for some event to occur (for I/O operation to complete, etc.)
- blocked → ready
  - Whatever event the process was waiting on has occurred

## Process State

- The *process state* consists of (at least):
  - Code for the program
  - Program's static and dynamic data
  - Program's procedure call stack
  - Contents of general purpose registers
  - Contents of Program Counter (PC) —address of next instruction to be executed
  - Contents of Stack Pointer (SP)
  - Contents of Program Status Word (PSW) — interrupt status, condition codes, etc.
  - OS resources in use (e.g., memory, open files, connections to other programs)
  - Accounting information

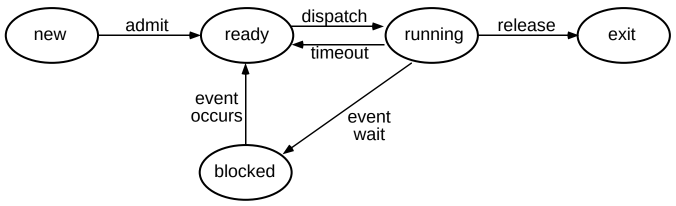
↪Everything necessary to resume the process' execution if it is somehow put aside temporarily

## Process Control Block (PCB)

- For every process, the OS maintains a *Process Control Block* (*PCB*), a data structure that represents the process and its state:
  - Process id number
  - Userid of owner
  - Memory space (static, dynamic)
  - Program Counter, Stack Pointer, general purpose registers
  - Process state (running, not-running, etc.)
  - CPU scheduling information (e.g., priority)
  - List of open files
  - I/O states, I/O in progress
  - Pointers into CPU scheduler's state queues (e.g., the waiting queue)
  - …

## A Five-State Process Model (Review)

- The *not-running* state in the two-state model has now been split into a *ready* state and a *blocked* state
  - *Running* — currently being executed
  - *Ready* — prepared to execute
  - *Blocked* — waiting for some event to occur (for an I/O operation to complete, or a resource to become available, etc.)
  - *New* — just been created
  - *Exit* — just been terminated
- State transition diagram:



## UNIX Process Model

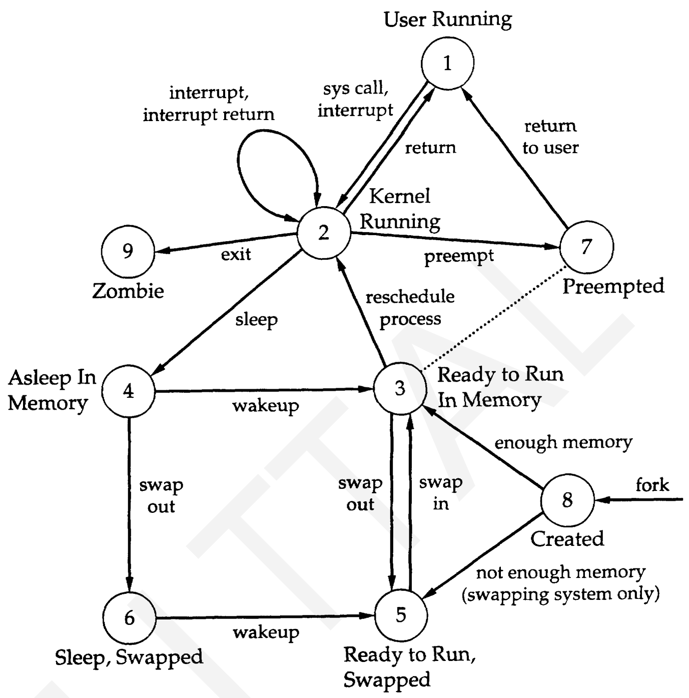


**FIGURE 3.16 UNIX process state transition diagram [BACH86]**

Figure from *Operating Systems*, 2nd edition, Stallings, Prentice Hall, 1995

Original diagram from *The Design of the UNIX Operating System*, M. Bach, Prentice Hall, 1986

## UNIX Process Model (cont.)

- Start in **Created**, go to either:
  - **Ready to Run, in Memory**
  - or **Ready to Run, Swapped** (Out) if there isn't room in memory for the new process
  - **Ready to Run, in Memory** is basically same state as **Preempted** (dotted line)
    - **Preempted** means process was returning to user mode, but the kernel switched to another process instead
- When scheduled, go to either:
  - **User Running** (if in user mode)
  - or **Kernel Running** (if in kernel mode)
  - Go from **U.R.** to **K.R.** via system call
- Go to **Asleep in Memory** when waiting for some event, to **RtRiM** when it occurs
- Go to **Sleep, Swapped** if swapped out

## Process Creation in UNIX

- One process can create another process, perhaps to do some work for it
  - The original process is called the *parent*
  - The new process is called the *child*
  - The child is an (almost) identical **copy** of parent (same code, same data, etc.)
  - The parent can either wait for the child to complete, or continue executing in parallel (*concurrently*) with the child
- In UNIX, a process creates a child process using the system call *fork( )*
  - In child process, fork( ) returns 0
  - In parent process, fork( ) returns process id of new child
- Child often uses *exec( )* to start another completely different program

## Example of UNIX Process Creation

```
#include <sys/types.h>
#include <stdio.h>

int a = 6;              /* global (external) variable */

int main(void)
{
   int b;               /* local variable */
   pid_t pid;           /* process id */

   b = 88;
   printf("..before fork\n");

   pid = fork();
   if (pid == 0) {      /* child */
      a++;  b++;
   } else               /* parent */
      wait(pid);

   printf("..after fork, a = %d, b = %d\n", a, b);
   exit(0);
}

aegis> fork
..before fork
..after fork, a = 7, b = 89
..after fork, a = 6, b = 88
```

## Context Switching

- Stopping one process and starting another is called a *context switch*
  - When the OS stops a process, it stores the hardware registers (PC, SP, etc.) and any other state information in that process' PCB
  - When OS is ready to execute a waiting process, it loads the hardware registers (PC, SP, etc.) with the values stored in the new process' PCB, and restores any other state information
  - Performing a context switch is a relatively expensive operation
    - However, time-sharing systems may do 100–1000 context switches a second
    - Why so often?
    - Why not more often?

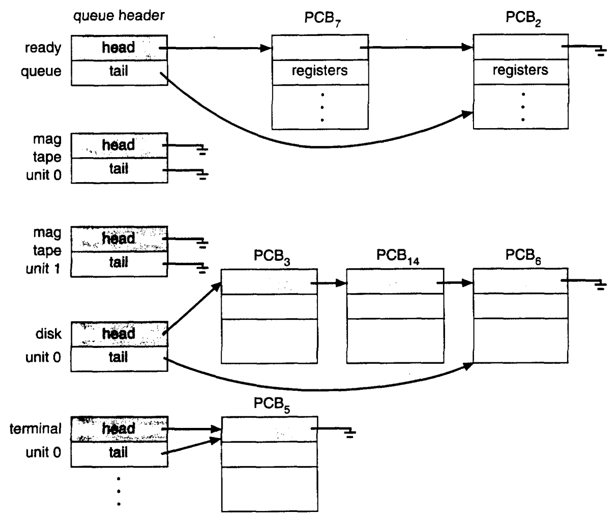
## Schedulers

- Long-term scheduler (job scheduler)
  - Selects job from spooled jobs, and loads it into memory
  - Executes infrequently, maybe only when process leaves system
  - Controls degree of multiprogramming
    - Goal: good mix of CPU-bound and I/O-bound processes
  - Doesn't really exist on most modern time-sharing systems
- Medium-term scheduler
  - On time-sharing systems, does some of what long-term scheduler used to do
  - May swap processes out of memory temporarily
  - May suspend and resume processes
  - Goal: balance load for better throughput

## Schedulers (cont.)

- Short-term scheduler (CPU scheduler)
  - Executes frequently, about one hundred times per second (every 10ms)
  - Runs whenever:
    - Process is created or terminated
    - Process switches from running to blocked
    - Interrupt occurs
  - Selects process from those that are ready to execute, allocates CPU to that process
  - Goals:
    - Minimize response time (e.g., program execution, character to screen)
    - Minimize variance of average response time — predictability may be important
    - Maximize throughput
      - Minimize overhead (OS overhead, context switching, etc.)
      - Efficient use of resources
    - Fairness — share CPU in an equitable fashion

## Ready Queue and Various I/O Device Queues



From *Operating System Concepts*, Silberschatz & Galvin., Addison-Wesley, 1994

- OS organizes all waiting processes (their PCBs, actually) into a number of queues
  - Queue for ready processes
  - Queue for processes waiting on each device (e.g., mouse) or type of event (e.g., message)

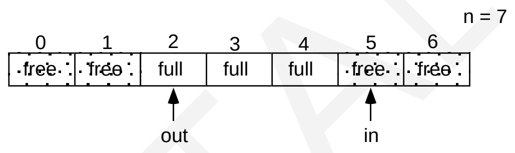
## Cooperating Processes

- Processes can cooperate with each other to accomplish a single task.
- Cooperating processes can:
  - Improve performance by overlapping activities or performing work in parallel
  - Enable an application to achieve a better program structure as a set of cooperating processes, where each is smaller than a single monolithic program
  - Easily share information
- Issues:
  - How do the processes communicate?
  - How do the processes share data?

## The Producer-Consumer Problem

- One process is a producer of information; another is a consumer of that information
- Processes communicate through a bounded (fixed-size) circular buffer

```
var buffer:  array[0..n-1] of items;     /* circular array */
in = 0
out = 0
```



```
/* producer */                         /* consumer */
repeat forever                         repeat forever
    …                                      while (in == out)
    produce item nextp                         do nothing
    …                                      nextc = buffer[out]
    while (in+1 mod n == out)              out = out+1 mod n
        do nothing                         …
    buffer[in] = nextp                     consume item nextc
    in = in+1 mod n                        …
end repeat                             end repeat
```

## Message Passing using Send & Receive

- Blocking send:
  - send(*destination-process*, *message*)
  - Sends a message to another process, then *blocks* (i.e., gets suspended by OS) until message is received
- Blocking receive:
  - receive(*source-process*, *message*)
  - Blocks until a message is received (may be minutes, hours, …)
- Producer-Consumer problem:

```
/* producer */                  /* consumer */
repeat forever                  repeat forever
    …                               receive(producer,nextc)
    produce item nextp              …
    …                               consume item nextc
    send(consumer, nextp)           …
end repeat                      end repeat
```

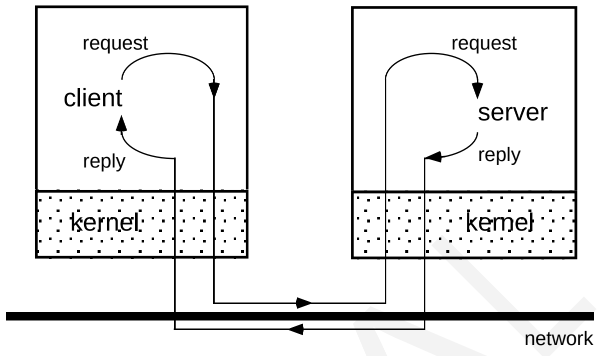
## Direct vs. Indirect Communication

- Direct communication — explicitly name the process you're communicating with
  - send(*destination-process*, *message*)
  - receive(*source-process*, *message*)
  - Link is associated with exactly two processes
    - Between any two processes, there exists at most one link
    - The link may be unidirectional, but is usually bidirectional
- Indirect communication — communicate using mailboxes (owned by receiver)
  - send(*mailbox*, *message*)
  - receive(*mailbox*, *message*)
  - Link is associated with two or more processes that share a mailbox
    - Between any two processes, there may be a number of links
    - The link may be either unidirectional or bidirectional

## Buffering

- Link may have some capacity that determines the number of message that can be temporarily queued in it
- Zero capacity: (queue of length 0)
  - No messages wait
  - Sender must wait until receiver receives the message — this synchronization to exchange data is called a *rendezvous*
- Bounded capacity: (queue of length *n*)
  - If receiver's queue is not full, new message is put on queue, and sender can continue executing immediately
  - If queue is full, sender must block until space is available in the queue
- Unbounded capacity: (infinite queue)
  - Sender can always continue

## Client / Server Model using Message Passing

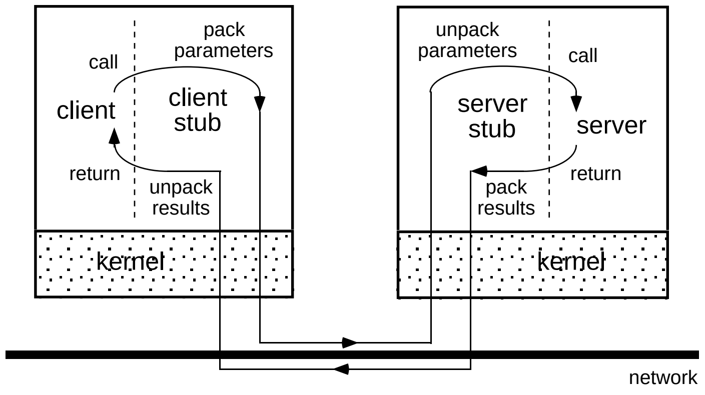


- Client / server model
  - *Server* = process (or collection of processes) that provides a *service*
    - Example: name service, file service
  - *Client* — process that uses the service
  - Request / reply protocol:
    - Client sends **request** message to server, asking it to perform some service
    - Server performs service, sends **reply** message containing results or error code

## Remote Procedure Call (RPC)

- RPC mechanism:
  - Hides message-passing I/O from the programmer
  - Looks (almost) like a procedure call — but client invokes a procedure on a server
- RPC invocation (high-level view):
  - Calling process (client) is suspended
  - Parameters of procedure are passed across network to called process (server)
  - Server executes procedure
  - Return parameters are sent back across network
  - Calling process resumes
- Invented by Birrell & Nelson at Xerox PARC, described in February 1984 *ACM Transactions on Computer Systems*

## Client / Server Model using Remote Procedure Calls (RPCs)



- Each RPC invocation by a client process calls a *client stub*, which builds a message and sends it to a *server stub*
- The server stub uses the message to generate a local procedure call to the server
- If the local procedure call returns a value, the server stub builds a message and sends it to the client stub, which receives it and returns the result(s) to the client

## RPC Invocation (More Detailed)

1. Client procedure calls the client stub
2. Client stub packs parameters into message and traps to the kernel
3. Kernel sends message to remote kernel
4. Remote kernel gives message to server stub
5. Server stub unpacks parameters and calls server
6. Server executes procedure and returns results to server stub
7. Server stub packs result(s) in message and traps to kernel
8. Remote kernel sends message to local kernel
9. Local kernel gives message to client stub
10. Client stub unpacks result(s) and returns them to client

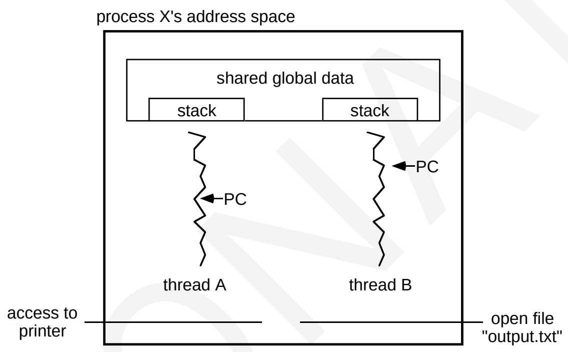
## Conventional View of Processes

- A process can be viewed two ways:
  - A unit of **resource ownership**
    - A process has an address space, containing program code and data
    - A process may have open files, may be using an I/O device, etc.
  - A unit of **scheduling**
    - The CPU scheduler dispatches one process at a time onto the CPU
    - Associated with a process are values in the PC, SP, and other registers
- Insight (~1988) — these two are usually linked, but they don't have to be
- In many recent operating systems (UNIX, Windows NT), the two are independent:
  - Process = unit of resource ownership
  - Thread = unit of scheduling

## Processes vs. Threads

- *Process* = unit of resource ownership
  - A process (sometimes called a *heavyweight process*) has:
    - Address space
    - Program code
    - Global variables, heap, stack
    - OS resources (files, I/O devices, etc.)
- *Thread* = unit of scheduling
  - A thread (sometimes called a *lightweight process*) is a single sequential execution stream within a process
  - A thread ***shares*** with other threads:
    - Address space, program code
    - Global variables, heap
    - OS resources (files, I/O devices)
  - A thread has its own:
    - Registers, Program Counter (PC)
    - Stack, Stack Pointer (SP)

## Processes vs. Threads



- A thread is bound to a particular process
  - A process may contain multiple threads of control inside it
  - Threads can block, create children, etc.
- All of the threads in a process:
  - Share address space, program code, global variables, heap, and OS resources
  - Execute concurrently (has its own register, PC, SP, etc. values)

## Why Threads?

- A process with multiple threads makes a great server (e.g., printer server):
  - Have one server process, many “worker” threads — if one thread blocks (e.g., on a read), others can still continue executing
  - Threads can share common data; don't need to use inter-process communication
  - Can take advantage of multiprocessors
- Threads are cheap!
  - Cheap to create — only need a stack and storage for registers
  - Use very little resources — don't need new address space, global data, program code, or OS resources
  - Context switches are fast — only have to save / restore PC, SP, and registers
- But… no protection between threads!

## What Kinds of Programs Can Be Multithreaded?

- Good programs to multithread:
  - Programs with multiple independent tasks (debugger needs to run and monitor program, keep its GUI active, and display an interactive data inspector and dynamic call grapher)
  - Server which needs to process multiple requests simultaneously
  - Repetitive numerical tasks — break large problem, such as weather prediction, down into small pieces and assign each piece to a separate thread
- Programs difficult to multithread:
  - Programs that don't require any multiprocessing (99% of all programs)
  - Programs that require multiple processes (maybe one needs to run as root)

## User-Level Threads

- User-level threads = provide a library of functions to allow user processes to create and manage their own threads
  - ✔ Doesn't require modification to the OS
  - ✔ Simple representation — each thread is represented simply by a PC, registers, stack, and a small control block, all stored in the user process' address space
  - ✔ Simple management — creating a new thread, switching between threads, and synchronization between threads can all be done without intervention of the kernel
  - ✔ Fast — thread switching is not much more expensive than a procedure call
  - ✔ Flexible — CPU scheduling (among those threads) can be customized to suit the needs of the algorithm

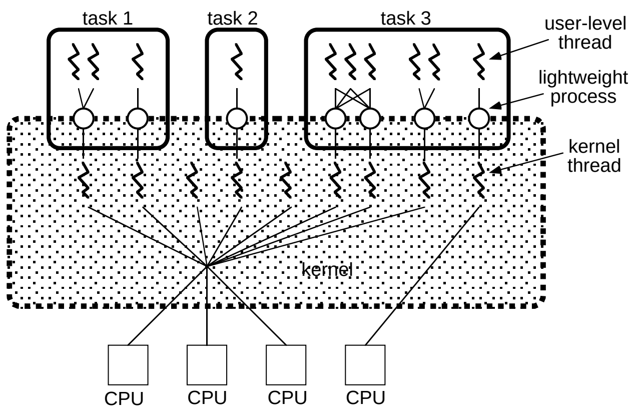
## User-Level Threads (cont.)

- User-level threads = provide a library of functions to allow user processes to create and manage their own threads
  - ✘ Lack of coordination between threads and OS kernel
    - Process as a whole gets one time slice
    - Same time slice, whether process has 1 thread or 1000 threads
    - Also — up to each thread to relinquish control to other threads in that process
  - ✘ Requires non-blocking system calls (i.e., a multithreaded kernel)
    - Otherwise, entire process will blocked in the kernel, even if there are runnable threads left in the process
  - ✘ If one thread causes a page fault, the entire process blocks

## Kernel-Level Threads

- Kernel-level threads = kernel provides system calls to create and manage threads
  - ✔ Kernel has full knowledge of all threads
    - Scheduler may choose to give a process with 10 threads more time than process with only 1 thread
  - ✔ Good for applications that frequently block (e.g., server processes with frequent interprocess communication)
  - ✘ Slow — thread operations are 100s of times slower than for user-level threads
  - ✘ Significant overhead and increased kernel complexity — kernel must manage and schedule threads as well as processes
    - Requires a full thread control block (TCB) for each thread

## Two-Level Thread Model (Digital UNIX, Solaris, IRIX, HP-UX)



- User-level threads for user processes
  - "Lightweight process" (LWP) serves as a "virtual CPU" where user threads can run
- Kernel-level threads for use by kernel
  - One for each LWP
  - Others perform tasks not related to LWPs
- OS supports multiprocessor systems

## Nachos

- Nachos is an instructional operating system developed at UC Berkeley
- Nachos consists of two main parts:
  - Operating system
    - This is the part of the code that you will study and modify
    - This code is in the **threads**, **userprog**, and **filesys** directories
    - We will not study networking, so the **network** directory has been removed
  - Machine emulator — simulates a (slightly old) MIPS CPU, registers, memory, timer (clock), console, disk drive, and network
    - You will study this code, but will not be allowed to modify it
    - This code is in the **machine** directory
- The OS and machine emulator run together as a single UNIX process

## Preparing for the First Project

- Reading assignment:
  - Read about Nachos, & skim the material on the emulated machine and threads
    - Don’t worry about synchronization, user programs, or the file system
  - Read old Appendix A of the text (online as “Overview Paper”)
  - Skim Section 2 “Nachos Machine” and Section 3 “Nachos Threads” in Narten’s “A Road Map Through Nachos” (online)
  - Skim material on threads in Kalra’s “Salsa — An OS Tutorial” (online)
  - Start looking at the code in the **threads** and **machine** directories
  - Road Map plus printouts of all code are available in the MCS office for $4.50
- If you are not familiar with C++ or the gdb debugger, see the class web page

## Preparing for the First Project (cont.)

- Compiling the code
  - Nachos source code is available in ~walker/pub
  - Read ~walker/pub/README
  - Decide where you want to work, so you can copy files from the appropriate directory into your account
    - ~walker/pub/nachos-3.4-hp
      - For HP workstations (aegis, intrepid)
      - Recommended
    - ~walker/pub/nachos-3.4-sparc
      - For Sun workstations (nimitz)
    - ~walker/pub/nachos-3.4-orig
      - The original, unmodified version
  - Read “Project 1 — Getting an Early Start” on the class web page to find out how to copy the necessary files to your account, and compile an executable copy of Nachos into the **threads** directory

## Nachos — The Emulated Machine

- Code is in the **machine** directory
- **machine.h**, **machine.cc** — emulates the part of the machine that executes user programs: main memory, processor registers, etc.
- **mipssim.cc** — emulates the integer instruction set of a MIPS R2/3000 CPU.
- **interrupt.h**, **interrupt.cc** — manages enabling and disabling interrupts as part of the machine emulation.
- **timer.h**, **timer.cc** — emulates a clock that periodically causes an interrupt to occur.
- **stats.h** — collects interesting statistics.

## Nachos — The Operating System

- For now, we will mostly be concerned with code in the **threads** directory
- **main.cc**, **threadtest.cc** — a simple test of the thread routines.
- **system.h**, **system.cc** — Nachos startup/shutdown routines.
- **thread.h**, **thread.cc** — thread data structures and thread operations such as thread fork, thread sleep and thread finish.
- **scheduler.h**, **scheduler.cc** — manages the list of threads that are ready to run.
- **list.h**, **list.cc** — generic list management.
- **utility.h**, **utility.cc** — some useful definitions and debugging routines.

## Nachos Threads

- As distributed, Nachos does not support multiple processes, only threads
  - All threads share / execute the same code (the Nachos source code)
  - All threads share the same global variables (have to worry about synch.)
- Some interesting functions:
  - Thread::Fork( ) — create a new thread to run a specified function with a single argument, and put it on the ready queue
  - Thread::Yield( ) — if there are other threads waiting to run, suspend this thread and run another
  - Thread::Sleep( ) — this thread is waiting on some event, so suspend it, and hope someone else wakes it up later
  - Thread::Finish( ) — terminate the currently running thread

## Manipulating Threads in Nachos

```
void
Thread::Fork(VoidFunctionPtr func, int arg)
{
    DEBUG('t',"Forking thread \"%s\" with
        func = 0x%x, arg = %d\n",
        name, (int) func, arg);

    StackAllocate(func, arg);

    IntStatus oldLevel = interrupt->
        SetLevel(IntOff);
    scheduler->ReadyToRun(this);
    (void) interrupt->SetLevel(oldLevel);
}
```

## Manipulating Threads in Nachos (cont.)

```
void
Thread::Yield ()
{
    Thread *nextThread;

    IntStatus oldLevel = interrupt->
        SetLevel(IntOff);

    ASSERT(this == currentThread);
    DEBUG('t', "Yielding thread \"%s\"\n",
        getName());

    nextThread = scheduler->
        FindNextToRun();
    if (nextThread != NULL) {
        scheduler->ReadyToRun(this);
        scheduler->Run(nextThread);
    }
    (void) interrupt->SetLevel(oldLevel);
}
```

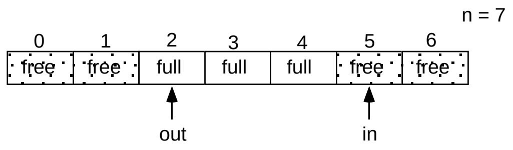
## Manipulating Threads in Nachos (cont.)

```
void
Thread::Sleep ()
{
    Thread *nextThread;

    ASSERT(this == currentThread);
    ASSERT(interrupt->getLevel() == IntOff);
    DEBUG('t', "Sleeping thread \"%s\"\n",
        getName());

    status = BLOCKED;
    while ((nextThread = scheduler->
        FindNextToRun()) == NULL)
        interrupt->Idle();

    scheduler->Run(nextThread);
}
```

## The Producer-Consumer Problem (Review from Lecture 07)

- One thread is a producer of information; another is a consumer of that information
  - They share a bounded circular buffer
  - Processes — OS must support shared memory between processes
  - Threads — all memory is shared

```
var buffer:  array[0..n-1] of items;     /* circular array */
in = 0
out = 0
```



```
/* producer */                          /* consumer */
repeat forever                          repeat forever
    …                                       while (in == out)
    produce item nextp                          do nothing
    …                                       nextc = buffer[out]
    while (in+1 mod n == out)               out = out+1 mod n
        do nothing                          …
    buffer[in] = nextp                      consume item nextc
    in = in+1 mod n                         …
end repeat                              end repeat
```

## Too Much Milk!

| Time | You | Your Roommate |
|---|---|---|
| 3:00 | Arrive home | |
| 3:05 | Look in fridge, no milk | |
| 3:10 | Leave for grocery | |
| 3:15 | | Arrive home |
| 3:20 | Arrive at grocery | Look in fridge, no milk |
| 3:25 | Buy milk, leave | Leave for grocery |
| 3:30 | | |
| 3:35 | Arrive home | Arrive at grocery |
| 3:36 | Put milk in fridge | |
| 3:40 | | Buy milk, leave |
| 3:45 | | |
| 3:50 | | Arrive home |
| 3:51 | | Put milk in fridge |

3:51 Oh, no! ***Too much milk!!***

- The problem here is that the lines:
  "Look in fridge, no milk"
  through
  "Put milk in fridge"

  are not an **atomic** operation

## Another Example

```
Thread A                    Thread B

i = 0                       i = 0
while (i < 10)              while (i > –10)
    i = i + 1                   i = i – 1
print "A wins"              print "B wins"
```

- Assumptions:
  - Memory load and store are atomic
  - Increment and decrement at not atomic
- Questions:
  - Who wins?
  - Is it guaranteed that someone wins?
  - What if both threads have their own CPU, running concurrently at exactly the same speed? Is it guaranteed that it goes on forever?
  - What if they are sharing a CPU?

## Synchronization Terminology

- *Synchronization* — using *atomic* (indivisible) operations to ensure cooperation between threads
- *Mutual exclusion* — ensures that only one thread does a particular activity at a time — all other threads are *excluded* from doing that activity
- *Critical section* (*region*) — code that only one thread can execute at a time (e.g., code that modifies shared data)
- *Lock* — mechanism that prevents another thread from doing something:
  - *Lock* before entering a critical section
  - *Unlock* when leaving a critical section
  - Thread wanting to enter a locked critical section must **wait** until it's unlocked

## Enforcing Mutual Exclusion

- Methods to enforce mutual exclusion
  - Up to user — threads have to explicitly coordinate with each other
  - Up to OS — OS provides support for mutual exclusion
  - Up to hardware — hardware provides architectural support for mutual exclusion
- Solution must:
  - Avoid *starvation* — if a thread starts trying to gain access to the critical section, then it should eventually succeed
  - Avoid *deadlock* — if **some** threads are trying to enter their critical sections, then **one** of them must eventually succeed
- We will assume that a thread may halt in its non-critical-section, but not in its critical section

## Algorithm 1

- Informal description:
  - Igloo with blackboard inside
    - Only one person (thread) can fit in the igloo at a time
    - In the igloo is a blackboard, which is large enough to hold only one value
  - A thread that wants to execute the critical section enters the igloo, and examines the blackboard
    - If its number is not on the blackboard, it leaves the igloo, goes outside, and runs laps around the igloo
      - After a while, it goes back inside, and checks the blackboard again
      - This "busy waiting" continues until eventually its number is on the blackboard
    - If its number is on the blackboard, it leaves the igloo and goes on to the critical section
    - When it returns from the critical section, it enters the igloo, and writes the other thread's number on the blackboard

## Algorithm 1 (cont.)

- Code:

```
t1 ( )   {
    while (true)   {
        while (turn != 1)
            ;     /* do nothing */
        … critical section of code …
        turn = 2;
        … other non-critical code …
    }
}

t2 ( )   {
    while (true)   {
        while (turn != 2)
            ;     /* do nothing */
        … critical section of code …
        turn = 1;
        … other non-critical code …
    }
}
```

## Algorithm 2a

- Informal description:
  - Each thread has its own igloo
    - A thread can examine and alter its own blackboard
    - A thread can examine, but not alter, the other thread's blackboard
    - "true" on blackboard = that thread is in the critical section
  - A thread that wants to execute the critical section enters the other thread's igloo, and examines the blackboard
    - It looks for "false" on that blackboard, indicating that the other thread is not in the critical section
      - When that happens, it goes back to its own igloo, and writes "true" on its own blackboard, and then goes on to the critical section
    - When it returns from the critical section, it enters the igloo, and writes "false" on the blackboard

## Algorithm 2a (cont.)

- Code:

```
t1 ( )   {
    while (true)   {
        while (t2_in_crit == true)
            ;     /* do nothing */
        t1_in_crit = true;
        … critical section of code …
        t1_in_crit = false;
        … other non-critical code …
    }
}

t2 ( )   {
    while (true)   {
        while (t1_in_crit == true)
            ;     /* do nothing */
        t2_in_crit = true;
        … critical section of code …
        t2_in_crit = false;
        … other non-critical code …
    }
}
```

## Algorithm 2b

- Code:

```
t1 ( )   {
    while (true)   {
        t1_in_crit = true;
        while (t2_in_crit == true)
            ;     /* do nothing */
        … critical section of code …
        t1_in_crit = false;
        … other non-critical code …
    }
}

t2 ( )   {
    while (true)   {
        t2_in_crit = true;
        while (t1_in_crit == true)
            ;     /* do nothing */
        … critical section of code …
        t2_in_crit = false;
        … other non-critical code …
    }
}
```

## Algorithm 3

- Think of this algorithm as using a referee who keeps track of whose "turn" it is
  - Anytime the two disagree about whose turn it is, they ask the referee, who keeps track of whose turn it is to have priority
  - This is called Peterson's algorithm (1981)
    - The original (but more complicated) solution to this problem is Dekker's algorithm (1965)
- For n processes, we can use Lamport's Bakery algorithm (1974)
  - When a thread tries to enter the critical section, it get assigned a number higher than anyone else's number
  - Thread with lowest number gets in
  - If two threads get the same number, the one with the lowest process id gets in

## Algorithm 3 (cont.)

- Code:

```
t1 ( )  {
     while (true)   {
          t1_in_crit = true;
          turn = 2;
          while (t2_in_crit == true && turn != 1)
               ;    /* do nothing */
          … critical section of code …
           t1_in_crit = false;
          … other non-critical code …
     }
}

t2 ( )  {
     while (true)   {
          similar…
     }
}
```

## Semaphores — OS Support for Mutual Exclusion

- Semaphores were invented by Dijkstra in 1965, and can be thought of as a generalized locking mechanism
  - A semaphore supports two **atomic** operations, **P / wait** and **V / signal**
    - The semaphore initialized to 1
    - Before entering the critical section, a thread calls "**P(semaphore)**", or sometimes "**wait(semaphore)**"
    - After leaving the critical section, a thread calls "**V(semaphore)**", or sometimes "**signal(semaphore)**"
- Too much milk:

```
Thread A                  Thread B

milk–>P( );               milk–>P( );
if (noMilk)               if (noMilk)
     buy milk;                 buy milk;
milk–>V( );               milk–>V( );
```

## Details of Semaphore Operation

- Semaphore "s" is initially 1
- Before entering the critical section, a thread calls "**P(s)**" or "**wait(s)**"
  - wait (s):
    - s = s – 1
    - if (s < 0)
      block the thread that called wait(s) on a queue associated with semaphore s
    - otherwise
      let the thread that called wait(s) continue into the critical section
- After leaving the critical section, a thread calls "**V(s)**" or "**signal(s)**"
  - signal (s):
    - s = s + 1
    - if (s ≤ 0), then
      wake up one of the threads that called wait(s), and run it so that it can continue into the critical section

## Semaphore Operation

- Informal description:
  - Single igloo, containing a blackboard and a very large freezer
  - Wait — thread enters the igloo, checks the blackboard, and decrements the value shown there
    - If new value is 0, thread goes on to the critical section
    - If new value is negative, thread crawls in the freezer and hibernates (making room for others to enter the igloo)
  - Signal — thread enters igloo, checks blackboard, and increments the value there
    - If new value is 0 or negative, there's a thread waiting in the freezer, so it thaws out a frozen thread, which then goes on to the critical section

## Using Semaphores

- Code using semaphores:

```
t1 ( )  {
    while (true)  {
        wait (s);
        … critical section of code …
        signal (s);
        … other non-critical code …
    }
}

t2 ( )  {
    while (true)  {
        wait (s);
        … critical section of code …
        signal (s);
        … other non-critical code …
    }
}
```

## Semaphore Operation & Values

- Semaphores (simplified slightly):

```
wait (s):                      signal (s):

s = s – 1                      s = s + 1
if (s < 0)                     if (s ≤ 0)
    block the thread               wake up & run one of
    that called wait(s)            the waiting threads
otherwise
    continue into CS
```

- Semaphore values:
  - Positive semaphore = number of (additional) threads that can be allowed into the critical section
  - Negative semaphore = number of threads blocked (note — there's also one in CS)
  - *Binary semaphore* has an initial value of 1
  - *Counting semaphore* has an initial value greater than 1

## Using Semaphores for Mutual Exclusion

- Too much milk:

```
Thread A                  Thread B

milk–>P( );               milk–>P( );
if (noMilk)               if (noMilk)
    buy milk;                 buy milk;
milk–>V( );               milk–>V( );
```

  - "noMilk" is a semaphore initialized to 1

- Execution:

| After: | s | queue | A | B |
|---|---|---|---|---|
| | 1 | | | |
| A: noM->P( ); | 0 | | in CS | |
| B: noM->P( ); | -1 | B | in CS | waiting |
| A: noM->V( ); | 0 | | finish | ready, in CS |
| B: noM->V( ); | 1 | | | finish |

## The Coke Machine (Bounded-Buffer Producer-Consumer)

```
/* number of full slots (Cokes) in machine */
semaphore fullSlot = 0;
/* number of empty slots in machine */
semaphore emptySlot = 100;
/* only one person accesses machine at a time */
semaphore mutex = 1;

DeliveryPerson()
{
    emptySlot->P( );        /* empty slot avail? */
    mutex->P( );            /* exclusive access */
    put 1 Coke in machine
    mutex->V( );
    fullSlot->V( );         /* another full slot! */
}

ThirstyPerson()
{
    fullSlot->P( );         /* full slot (Coke)? */
    mutex->P( );            /* exclusive access */
    get 1 Coke from machine
    mutex->V( );
    emptySlot->V( );        /* another empty slot! */
}
```

## Two Versions of Semaphores

■ Semaphores from last time (simplified):

```
wait (s):                   signal (s):

s = s – 1                   s = s + 1
if (s < 0)                  if (s ≤ 0)
    block the thread            wake up one of
    that called wait(s)         the waiting threads
otherwise
    continue into CS
```

■ "Classical" version of semaphores:

```
wait (s):                   signal (s):

if (s ≤ 0)                  if (a thread is waiting)
    block the thread            wake up one of
    that called wait(s)         the waiting threads
s = s – 1                   s = s + 1
continue into CS
```

■ Do both work?  What is the difference??

## Implementing Semaphores

■ Implementing semaphores using *busy-waiting*:

```
wait (s):                   signal (s):

while (s ≤ 0)                   s = s + 1
   do nothing;
s = s – 1
```

■ Evaluation:

✘ Doesn’t support queue of blocked threads waiting on the semaphore

✘ Waiting threads wastes time *busy-waiting* (doing nothing useful, wasting CPU time)

✘ The code inside wait(s) and signal(s) is a critical section also, and it’s not protected

## Implementing Semaphores (cont.)

■ Implementing semaphores (not fully) by *disabling interrupts*:

```
wait (s):                   signal (s):

disable interrupts          disable interrupts
while (s ≤ 0)               s = s + 1
   do nothing;
s = s – 1
enable interrupts           enable interrupts
```

■ Evaluation:

✘ Doesn’t support queue of blocked threads waiting on the semaphore

✘ Waiting threads wastes time *busy-waiting* (doing nothing useful, wasting CPU time)

✘ Doesn’t work on multiprocessors

✘ Can interfere with timer, which might be needed by other applications

✘ OK for OS to do this, but users aren’t allowed to disable interrupts!  (Why not?)

## Implementing Semaphores (cont.)

■ Implementing semaphores (not fully) using a *test&set instruction*:

```
wait (s):                      signal (s):

while (test&set(lk)!=0)        while (test&set(lk)!=0)
   do nothing;                        do nothing;
while (s ≤ 0)                  s = s + 1
   do nothing;
s = s – 1
lk = 0                         lk = 0
```

■ Operation:

- Lock “lk” has an initial value of 0
- If “lk” is free (lk=0), test&set atomically:
  - reads 0, sets value to 1, and returns 0
  - loop test fails, meaning lock is now busy
- If “lk” is busy (lk=1), test&set atomically:
  - reads 1, sets value to 1, and returns 1
  - loop test is true, so loop continues until someone releases the lock

## Implementing Semaphores (cont.)

- Test&set is an example of an atomic *read-modify-write* (RMW) instruction
  - RMW instructions atomically read a value from memory, modify it, and write the new value to memory
    - Test&set — on most CPUs
    - Exchange — Intel x86 — swaps values between register and memory
    - Compare&swap — Motorola 68xxx — read value, if value matches value in register r1, exchange register r1 and value
- Evaluation:
  - ✔ Can be made to work, even on multiprocessors (although there may be some cache consistency problems)
  - ✘ Doesn't support queue of blocked threads waiting on the semaphore
  - ✘ Waiting threads wastes time *busy-waiting* (doing nothing useful, wasting CPU time)

## Semaphores in Nachos

- The class Semaphore is defined in **threads/synch.h** and **synch.cc**
  - The classes Lock and Condition are also defined , but their member functions are empty (implementation left as exercise)
- Interesting functions:
  - Semaphores:
    - Semaphore::Semaphore( ) — creates a semaphore with specified name & value
    - Semaphore::P( ) — semaphore wait
    - Semaphore::V( ) — semaphore signal
  - Locks:
    - Lock::Acquire( )
    - Lock::Release( )
  - Condition variables:
    - Condition::Wait( )
    - Condition::Signal( )

## Semaphores in Nachos

```
void
Semaphore::P()
{
   IntStatus oldLevel = interrupt->
      SetLevel(IntOff);    // disable interrupts

   while (value == 0) {        // sema not avail
      queue->                  // so go to sleep
          Append((void *)currentThread);
      currentThread->Sleep();
   }

   value--;              // semaphore available,
                         // consume its value

   (void) interrupt->    // re-enable interrupts
      SetLevel(oldLevel);
}
```

## Semaphores in Nachos (cont.)

```
void
Semaphore::V()
{
   Thread *thread;

   IntStatus oldLevel = interrupt->
      SetLevel(IntOff);

   thread = (Thread *)queue->Remove();
   if (thread != NULL) // make thread ready,
             // consuming the V immediately
      scheduler->ReadyToRun(thread);

   value++;

   (void) interrupt->SetLevel(oldLevel);
}
```

## Semaphores — OS Support for Mutual Exclusion (Review)

- Even with semaphores, some synchronization errors can occur:

| Honest Mistake | Careless Mistake |
|---|---|

```
Honest Mistake          Careless Mistake

milk–>V( );             milk–>P( );
if (noMilk)             if (noMilk)
    buy milk;               buy milk;
milk–>P( );             milk–>P( );
```

  - Other variations possible
- Solution — new language constructs
  - (Conditional) Critical region
    - **region** v **when** B **do** S;
    - Variable v is a shared variable that can only be accessed inside the critical region
    - Boolean expression B governs access
    - Statement S ( critical region) is executed only if B is true; otherwise it blocks until B does become true
  - Monitor

## From Semaphores to Locks and Condition Variables

- A semaphore serves two purposes:
  - Mutual exclusion — protect shared data
    - mutex in Coke machine
    - milk in Too Much Milk
    - Always a binary semaphore
  - Synchronization — temporally coordinate events (one thread waits for something, other thread signals when it's available)
    - fullSlot and emptySlot in Coke machine
    - Either a binary or counting semaphore
- Idea — two separate constructs:
  - *Locks* — provide mutually exclusion
  - *Condition variables* — provide synchronization
  - Like semaphores, locks and condition variables are language-independent, and are available in many programming environments

## Locks

- *Locks* provide mutually exclusive access to shared data:
  - A lock can be "locked" or "unlocked" (sometimes called "busy" and "free")
- Operations on locks (Nachos syntax):
  - Lock(*name) — create a new (initially unlocked) Lock with the specified name
  - Lock::Acquire( ) — wait (block) until the lock is unlocked; then lock it
  - Lock::Release( ) — unlock the lock; then wake up (signal) any threads waiting on it in Lock::Acquire( )
- Can be implemented:
  - Trivially by binary semaphores (create a private lock semaphore, use P and V)
  - By lower-level constructs, much like semaphores are implemented

## Locks (cont.)

- Conventions:
  - Before accessing shared data, call Lock::Acquire( ) on a specific lock
    - Complain (via ASSERT) if a thread tries to Acquire a lock it already has
  - After accessing shared data, call Lock:: Release( ) on that same lock
    - Complain if a thread besides the one that Acquired a lock tries to Release it
- Example of using locks for mutual exclusion (here, "milk" is a lock):

```
Thread A                Thread B

milk–>Acquire( );       milk–>Acquire( );
if (noMilk)             if (noMilk)
    buy milk;               buy milk;
milk–>Release( );       milk–>Release( );
```

  - The test in threads/threadtest.cc should work exactly the same if locks are used instead of semaphores

## Locks vs. Condition Variables

- Consider the following code:

```
Queue::Add( ) {              Queue::Remove( ) {
    lock->Acquire( );            lock->Acquire( );
    add item                     if item on queue
    lock->Release( );                remove item
}                                lock->Release( );
                                 return item;
                             }
```

  - Queue::Remove will only return an item if there's already one in the queue
- If the queue is empty, it might be more desirable for Queue::Remove to wait until there is something to remove
  - Can't just go to sleep — if it sleeps while holding the lock, no other thread can access the shared queue, add an item to it, and wake up the sleeping thread
  - Solution: **condition variables** will let a thread sleep <u>inside</u> a critical section, by releasing the lock while the thread sleeps

## Condition Variables

- *Condition variables* coordinate events
- Operations on condition variables (Nachos syntax):
  - Condition(*name) — create a new instance of class Condition (a condition variable) with the specified name
    - After creating a new condition, the <u>programmer</u> must call Lock::Lock( ) to create a lock that will be associated with that condition variable
  - Condition::Wait(conditionLock) — release the lock and wait (sleep); when the thread wakes up, immediately try to re-acquire the lock; return when it has the lock
  - Condition::Signal(conditionLock) — if threads are waiting on the lock, wake up <u>one</u> of those threads and put it on the ready list; otherwise do nothing

## Condition Variables (cont.)

- Operations (cont.):
  - Condition::Broadcast(conditionLock) — if threads are waiting on the lock, wake up <u>all</u> of those threads and put them on the ready list; otherwise do nothing
- **Important**: a thread <u>**must**</u> hold the lock before calling Wait, Signal, or Broadcast
- Can be implemented:
  - Carefully by higher-level constructs (create and queue threads, sleep and wake up threads as appropriate)
  - Carefully by binary semaphores (create and queue semaphores as appropriate, use P and V to synchronize)
    - Does this work? More on this in a few minutes…
  - Carefully by lower-level constructs, much like semaphores are implemented

## Using Locks and Condition Variables

- Associated with a data structure is both a lock and a condition variable
  - Before the program performs an operation on the data structure, it acquires the lock
  - If it needs to wait until another operation puts the data structure into an appropriate state, it uses the condition variable to wait
- Unbounded-buffer producer-consumer:

```
Lock *lk;                  int avail = 0;
Condition *c;
                           /* consumer */
/* producer */             while (1) {
while (1) {                    lk-> Acquire( );
    lk->Acquire( );            if (avail==0)
    produce next item              c->Wait(lk);
    avail++;                   consume next item
    c->Signal(lk)              avail--;
    lk->Release( );            lk->Release( );
}                          }
```

## Using Locks and Condition Variables (Review)

- Associated with a data structure is both a lock and a condition variable
  - Before the program performs an operation on the data structure, it acquires the lock
  - If it needs to wait until another operation puts the data structure into an appropriate state, it uses the condition variable to wait
- Unbounded-buffer producer-consumer:

```
Lock *lk;                  int avail = 0;
Condition *c;
                           /* consumer */
/* producer */             while (1) {
while (1) {                     lk-> Acquire( );
     lk->Acquire( );            if (avail==0)
     produce next item               c->Wait(lk);
     avail++;                   consume next item
     c->Signal(lk)              avail--;
     lk->Release( );            lk->Release( );
}                          }
```

## Comparing Semaphores and Condition Variables

- Semaphores and condition variables are pretty similar — perhaps we can build condition variables out of semaphores
- Does this work?

```
Condition::Wait( ) {          Condition::Signal( ) {
     sema->P( );                   sema->V( );
}                             }
```

  - No, we're going to use these condition operations inside a lock. What happens if we use semaphores inside a lock?
- How about this?

```
Condition::Wait( ) {          Condition::Signal( ) {
     lock->Release( );             sema->V( );
     sema->P( );              }
     lock->Acquire( );
}
```

  - How do semaphores and condition variables differ with respect to keeping track of history?

## Comparing Semaphores and Condition Variables (cont.)

```
Condition::Wait( ) {          Condition::Signal( ) {
     lock->Release( );             sema->V( );
     sema->P( );              }
     lock->Acquire( );
   }
```

- Semaphores have a value, CVs do not!
- On a **<u>semaphore</u>** signal (a V), the value of the semaphore is always incremented, even if no one is waiting
  - Later on, if a thread does a semaphore wait (a P), the value of the semaphore is decremented and the thread **<u>continues</u>**
- On a **<u>condition variable</u>** signal, if no one is waiting, the signal has no effect
  - Later on, if a thread does a condition variable wait, it **<u>waits</u>** (it **always** waits!)
  - It doesn't matter how many signals have been made beforehand

## Two Kinds of Condition Variables

- Hoare-style (named after C.A.R. Hoare, used in most textbooks including *OSC*):
  - When a thread performs a Signal( ), it gives up the lock (and the CPU)
    - The waiting thread <u>is picked as the next thread that gets to run</u>
  - Previous example uses Hoare-style CVs
- Mesa-style (used in Mesa, Nachos, and most real operating systems):
  - When a thread performs a Signal( ), it keeps the lock (and the CPU)
    - The waiting thread <u>gets put on the ready queue</u> with no special priority
      - There is **<u>no guarantee</u>** that it will be picked as the next thread that gets to run
      - Wore yet, another thread may even run and acquire the lock before it does!
  - When using Mesa-style CVs, **<u>always</u>** surround the Wait( ) with a "while" loop

## Monitors

- A *monitor* is a programming-language abstraction that automatically associates locks and condition variables with data
  - A monitor includes private data and a set of atomic operations (member functions)
    - Only one thread can execute (any function in) monitor code at a time
    - Monitor functions access monitor data only
    - Monitor data cannot be accessed outside
  - A monitor also has a lock, and (optionally) one or more condition variables
    - Compiler automatically inserts an acquire operation at the beginning of each function, and a release at the end
- Special languages that supported monitors were popular with some OS people in the 1980s, but no longer
  - Now, most OSs (OS/2, Windows NT, Solaris) just provide locks and CVs

## The Dining Philosophers

- 5 philosophers live together, and spend most of their lives thinking and eating (primarily spaghetti)
  - They all eat together at a large table, which is set with 5 plates and 5 forks
  - To eat, a philosopher goes to his or her assigned place, and uses the two forks on either side of the plate to eat spaghetti
  - When a philosopher isn't eating, he or she is thinking
- Problem: devise a ritual (an algorithm) to allow the philosophers to eat
  - Must satisfy *mutual exclusion* (i.e., only one philosopher uses a fork at a time)
  - Avoids *deadlock* (e.g., everyone holding the left fork, and waiting for the right one)
  - Avoids *starvation* (i.e., everyone eventually gets a chance to eat)

## The Dining Philosophers (Using Semaphores)

- First solution — doesn't work: (why not?)

```
philosopher-i ( )
    while (true)
        think;
        P(fork[i]);
        P(fork[i+1 mod 5]);
        eat;            /* critical section */
        V(fork[i]);
        V(fork[i+1 mod 5]);
```

- Second solution — only 4 eat at a time:

```
philosopher-i ( )
    while (true)
        think;
        P(room_at_table);
        P(fork[i]);
        P(fork[i+1 mod 5]);
        eat;            /* critical section */
        V(fork[i]);
        V(fork[i+1 mod 5]);
        V(room_at_table);
```

## The Dining Philosophers (Using Locks and CVs)

```
mutex:  lock;
self:  array [0..N–1] of condition;
state:  array [0..N–1] of (thinking,hungry,eating)
                    initially all thinking

pickup (int i) {              putdown (int i) {
   acquire(mutex);                 acquire(mutex);
   state[i] = hungry;            state[i] = thinking;
   test(i);                      test(i+N–1 mod N);
   if (state[i] != eat)          test(i+1 mod N);
      wait(self[i]);             release(mutex);
   release(mutex);               }
}

test (int k) {
   if ((state[k+N–1 mod N] != eat) &&
      (state[k] == hungry) &&
      state[k+1 mod N] != eat)) {
        state[i] = eat;
        signal(self[i]);
    }
}
```

## The Readers/Writers Problem (Courtois, 1971)

- Models access to a database, such as an airline reservation system
  - Multiple readers (customers) want to read the database — access schedules, flight availability, etc.
  - Multiple writers (the airline) want to write the database — update the schedules
- It is acceptable to have multiple readers at the same time
  - But while a writer is writing to (updating) the database, no one else can access the database — either to read or to write
- Two versions:
  - Readers have priority over writers
  - Writers have priority over readers
  - In most solutions, the non-priority group can starve

## Readers/Writers w/ Readers Priority (Using Semaphores)

**Initialization:**

```
semaphore mutex = 1;  /* for mutual exclusion */
semaphore write = 1;  /* for mut. ex. & synch. */
int readers = 0;      /* number of readers */
```

**Reader:**

```
P(mutex);
readers++;
if (readers == 1)
    P(write);
V(mutex);
read database
P(mutex);
readers– –;
if (readers == 0)
    V(write);
V(mutex);
```

**Writer:**

```
P(write);
write database
V(write);
```

## Notes on R/W w/ Readers Priority (Using Semaphores)

- Reader:
  - Needs mutually exclusive access while manipulating “readers” variable
  - Does not need mutually exclusive access while reading database
  - If this reader is the first reader, it has to wait if there is an active writer (which has exclusive access to the database)
    - First reader did a “P(write)”
  - If other readers come along while the first one is waiting, they wait at the “P(mutex)”
  - If other readers come along while the first one is actively reading the database, they can also get in to read the database
  - When the last reader finishes, if there are waiting writers, it must wake one up

## Notes on R/W w/ Readers Priority (Using Semaphores) (cont.)

- Writer:
  - If there is an active writer, this writer has to wait (the active writer has exclusive access to database)
  - If there are active readers, this writer has to wait (readers have priority)
    - First reader did a “P(write)”
  - The writer only gets in to write to the database when there are no other active readers or writers
  - When the writer finishes, it wakes up someone (either a reader or a writer — it’s up to the CPU scheduler)
  - If a reader gets to go next, then once it goes through the “V(mutex)” and starts reading the database, then all other readers waiting at the top “P(mutex)” get to get in and read the database as well

## Readers/Writers w/ Writers Priority (Using Semaphores)

**Reader:**

```
P(mutex);
if (AW+WW > 0)
    WR++;
else {
    V(OKToRead);
    AR++;
}
V(mutex);
P(OKToRead);

read database

P(mutex);
AR– –;
if (AR == 0 &&
    WW > 0) {
    V(OKToWrite);
    AW++; WW– –;
}
V(mutex);
```

**Writer:**

```
P(mutex);
if (AW+AR > 0)
   WW++;
else {
   V(OKToWrite);
   AW++;
}
V(mutex);
P(OKToWrite);

write database

P(mutex);
AW– –;
if (WW > 0) {
   V(OKToWrite);
   AW++; WW– –;
} else if (WR > 0) {
   V(OKToRead);
   AR++;  WR– –;
}
V(mutex);
```

## Notes on R/W w/ Writers Priority (Using Semaphores)

- Reader:
  - If there are active or waiting writers, this reader has to wait (writers have priority)
  - Otherwise, this reader can read (possibly along with other readers)
  - When the last reader finishes, if there are waiting writers, it must wake one up
- Writer:
  - If there are active readers or writers, this writer has to wait (everyone has to finish before writer can update database)
  - Otherwise, this writer can write (and has exclusive access to database)
  - When the writer finishes,
    - (first choice) if there are waiting writers, it must wake one up (writers have priority)
    - (second choice) if there are waiting readers, it must wake one up

## Readers/Writers w/ Writers Priority (Using Locks and CVs)

**Reader:**

```
acquire(mutex);
while (AW+WW > 0) {
    WR++;
    wait(OKToRead);
    WR– –;
}
AR++;
release(mutex);

read database

acquire(mutex);
AR– –;
if (AR == 0 &&
    WW > 0)
    signal(OKToWrite);
release(mutex);
```

**Writer:**

```
acquire(mutex);
while (AW+AR > 0) {
   WW++;
   wait(OKToWrite);
   WW– –;
}
AW++;
release(mutex);

write database

acquire(mutex);
AW– –;
if (WW > 0)
   signal(OKToWrite);
else
   br'cast(OKToRead);
release(mutex);
```

## Readers/Writers w/ Writers Priority (Using Locks and CVs, Solution 2)

**Reader:**

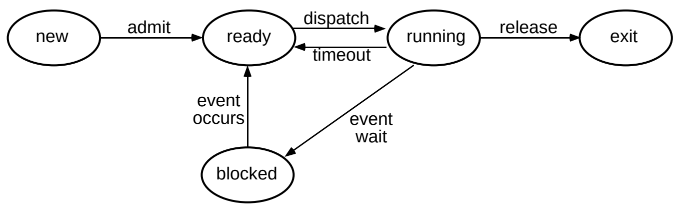
```
acquire(mutex);
while (i <0)
    wait(access);
i++;
release(mutex);

read database

acquire(mutex);
i– –;
if (i == 0)
    signal(access);
release(mutex);
```

**Writer:**

```
acquire(mutex);
while (i != 0)
   wait(access);
i– –;
release(mutex);

write database

acquire(mutex);
i = 0;
br'cast(access);
release(mutex);
```

- Notes:
  - "access" conveys right to access
  - If i > 0, i counts number of active readers
  - If i == 0, no one is accessing the data
  - If i < 0, there is an active writer

## CPU Scheduling



- The *CPU scheduler* (sometimes called the *dispatcher* or *short-term scheduler*):
  - Selects a process from the ready queue and lets it run on the CPU
    - Assumes all processes are in memory, and one of those is executing on the CPU
  - Crucial in multiprogramming environment
    - Goal is to maximize CPU utilization
- *Non-preemptive* scheduling — scheduler executes only when:
  - Process is terminated
  - Process switches from running to blocked

## Process Execution Behavior

- Assumptions:
  - One process per user
  - One thread per process
  - Processes are independent, and compete for resources (including the CPU)
- Processes run in CPU - I/O burst cycle:
  - Compute for a while (on CPU)
  - Do some I/O
  - Continue these two repeatedly
- Two types of processes:
  - CPU-bound — does mostly computation (long CPU burst), and very little I/O
  - I/O-bound — does mostly I/O, and very little computation (short CPU burst)

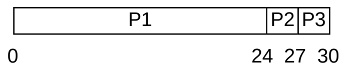
## First-Come-First-Served (FCFS)

- Other names:
  - First-In-First-Out (FIFO)
  - Run-Until-Done
- Policy:
  - Choose process from ready queue in the order of its arrival, and run that process non-preemptively
    - Early FCFS schedulers were overly non-preemptive: the process did not relinquish the CPU until it was finished, even when it was doing I/O
    - Now, non-preemptive means the scheduler chooses another process when the first one terminates or blocks
- Implement using FIFO queue (add to tail, take from head)
- Used in Nachos (as distributed)

## FCFS Example

- Example 1:

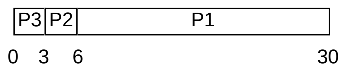
| **Process (Arrival Order)** | **P1** | **P2** | **P3** |
|---|---|---|---|
| **Burst Time** | 24 | 3 | 3 |
| **Arrival Time** | 0 | 0 | 0 |



average waiting time = (0 + 24 + 27) / 3 = 17

- Example 2:

| **Process (Arrival Order)** | **P3** | **P2** | **P1** |
|---|---|---|---|
| **Burst Time** | 3 | 3 | 24 |
| **Arrival Time** | 0 | 0 | 0 |



average waiting time = (0 + 3 + 6) / 3 = 3

## Scheduling in Nachos

- main( ) (in threads/main.cc)
  calls Initialize( ) (in threads/system.cc)
  - which starts scheduler, an instance of class Scheduler (defined in **threads/scheduler.h**, **scheduler.cc**)
- Interesting functions:
  - Mechanics of running a thread:
    - Scheduler::ReadyToRun( ) — puts a thread at the tail of the ready queue
    - Scheduler::FindNextToRun( ) — returns thread at the head of the ready queue
    - Scheduler::Run( ) — switches to thread
  - Scheduler is non-preemptive FCFS — chooses next process when:
    - Current thread terminates
    - Current thread calls Thread::Yield( ) to explicitly yield the CPU
    - Current thread calls Thread::Sleep( ) (to block (wait) on some event)

## Scheduling in Nachos (cont.)

```
Scheduler::Scheduler ( )
{
    readyList = new List;
}

void
Scheduler::ReadyToRun (Thread *thread)
{
   DEBUG('t',
      "Putting thread %s on ready list.\n",
      thread->getName());
   thread->setStatus(READY);
   readyList->Append((void *)thread);
}

Thread *
Scheduler::FindNextToRun ( )
{
    return (Thread *)readyList->Remove();
}
```

## Scheduling in Nachos (cont.)

```
void
Scheduler::Run (Thread *nextThread)
{
   Thread *oldThread = currentThread;

   oldThread->CheckOverflow();
   currentThread = nextThread;
   currentThread->setStatus(RUNNING);

   DEBUG('t', "Switching from thread \"%s\"
   to thread \"%s\"\n",oldThread->getName(),
      nextThread->getName());
   SWITCH(oldThread, nextThread);
   DEBUG('t', "Now in thread \"%s\"\n",
      currentThread->getName());

   if (threadToBeDestroyed != NULL) {
      delete threadToBeDestroyed;
      threadToBeDestroyed = NULL;
   }
}
```

## Manipulating Threads in Nachos (Review)

```
void
Thread::Fork(VoidFunctionPtr func, int arg)
{
   DEBUG('t',"Forking thread \"%s\" with
      func = 0x%x, arg = %d\n",
      name, (int) func, arg);

   StackAllocate(func, arg);

   IntStatus oldLevel = interrupt->
      SetLevel(IntOff);
   scheduler->ReadyToRun(this);
   (void) interrupt->SetLevel(oldLevel);
}
```

## Manipulating Threads in Nachos (cont.) (Review)

```
void
Thread::Yield ()
{
   Thread *nextThread;

   IntStatus oldLevel = interrupt->
      SetLevel(IntOff);

   ASSERT(this == currentThread);
   DEBUG('t', "Yielding thread \"%s\"\n",
      getName());

   nextThread = scheduler->
      FindNextToRun();
   if (nextThread != NULL) {
      scheduler->ReadyToRun(this);
      scheduler->Run(nextThread);
   }
   (void) interrupt->SetLevel(oldLevel);
}
```

## Manipulating Threads in Nachos (cont.) (Review)

```
void
Thread::Sleep ()
{
   Thread *nextThread;

   ASSERT(this == currentThread);
   ASSERT(interrupt->getLevel() == IntOff);
   DEBUG('t', "Sleeping thread \"%s\"\n",
      getName());

   status = BLOCKED;
   while ((nextThread = scheduler->
      FindNextToRun()) == NULL)
      interrupt->Idle();

   scheduler->Run(nextThread);
}
```

## Semaphores in Nachos (Review)

```
void
Semaphore::P()
{
   IntStatus oldLevel = interrupt->
      SetLevel(IntOff);    // disable interrupts

   while (value == 0) {        // sema not avail
      queue->                  // so go to sleep
          Append((void *)currentThread);
      currentThread->Sleep();
   }

   value--;                // semaphore available,
                           // consume its value

   (void) interrupt->    // re-enable interrupts
      SetLevel(oldLevel);
}
```

## Semaphores in Nachos (cont.) (Review)

```
void
Semaphore::V()
{
   Thread *thread;

   IntStatus oldLevel = interrupt->
      SetLevel(IntOff);

   thread = (Thread *)queue->Remove();
   if (thread != NULL) // make thread ready,
              // consuming the V immediately
      scheduler->ReadyToRun(thread);

   value++;

   (void) interrupt->SetLevel(oldLevel);
}
```

## CPU Scheduling Goals

- CPU scheduler must decide:
  - How long a process executes
  - In which order processes will execute
- User-oriented scheduling policy goals:
  - Minimize **average response time** (time from request received until response starts) while maximizing **number of interactive users** receiving adequate response
  - Minimize **turnaround time** (time from process start until completion)
    - Execution time plus waiting time
  - Minimize **variance of average response time**
    - Predictability is important
    - Process should always run in (roughly) same amount of time regardless of the load on the system

## CPU Scheduling Goals (cont.)

- System-oriented scheduling policy goals:
  - Maximize **throughput** (number of processes that complete in unit time)
  - Maximize **processor utilization** (percentage of time CPU is busy)
- Other (non-performance related) system-oriented scheduling policy goals:
  - *Fairness* — in the absence of guidance from the user or the OS, processes should be treated the same, and no process should suffer *starvation* (being infinitely denied service)
    - May have to be less fair in order to minimize average response time!
  - Balance resources — keep all resources of the system (CPU, memory, disk, I/O) busy
    - Favor processes that will underuse stressed resources

## FCFS Evaluation

- Non-preemptive
- Response time — slow if there is a large variance in process execution times
  - If one long process is followed by many short processes, short processes have to wait a long time
  - If one CPU-bound process is followed many I/O-bound processes, there's a "convoy effect"
    - Low CPU and I/O device utilization
- Throughput — not emphasized
- Fairness —penalizes short processes and I/O bound processes
- Starvation — not possible
- Overhead — minimal

## Preemptive vs. Non-Preemptive Scheduling

- *Non-preemptive* scheduling — scheduler executes only when:
  - Process is terminated
  - Process switches from running to blocked
- *Preemptive* scheduler — scheduler can execute at (almost) any time:
  - Executes at times above, also when:
    - Process is created
    - Blocked process becomes ready
    - A timer interrupt occurs
  - More overhead, but keeps long processes from monopolizing CPU
  - Must not preempt OS kernel while it's servicing a system call (e.g., reading a file) or otherwise in an inconsistent state
  - ✘ Can still leave data shared between user processes in an inconsistent state

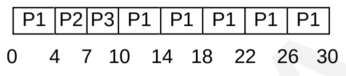
## Round-Robin

- Policy:
  - Define a fixed *time slice* (also called a *time quantum*)
  - Choose process from head of ready queue
  - Run that process for at most one time slice, and if it hasn't completed by then, add it to the tail of the ready queue
  - If that process terminates or blocks before its time slice is up, choose another process from the head of the ready queue, and run that process for at most one time slice…
- Implement using:
  - Hardware timer that interrupts at periodic intervals
  - FIFO ready queue (add to tail, take from head)

## Round-Robin Example

- Example 1:

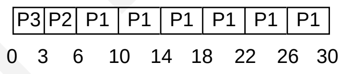
| Process (Arrival Order) | P1 | P2 | P3 |
|---|---|---|---|
| Burst Time | 24 | 3 | 3 |
| Arrival Time | 0 | 0 | 0 |



average waiting time = (4 + 7 + (10–4)) / 3 = 5.66

- Example 2:

| Process (Arrival Order) | P3 | P2 | P1 |
|---|---|---|---|
| Burst Time | 3 | 3 | 24 |
| Arrival Time | 0 | 0 | 0 |



average waiting time = (0 + 3 + 6) / 3 = 3

## Round-Robin Evaluation

- Preemptive (at end of time slice)
- Response time — good for short processes
  - Long processes may have to wait *n*q* time units for another time slice
    - *n* = number of other processes, *q* = length of time slice
- Throughput — depends on time slice
  - Too small — too many context switches
  - Too large — approximates FCFS
- Fairness — penalizes I/O-bound processes (may not use full time slice)
- Starvation — not possible
- Overhead — low

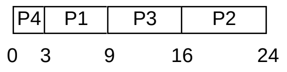
## Shortest-Job-First (SJF)

- Other names:
  - Shortest-Process-Next (SPN)
- Policy:
  - Choose the process that has the smallest next CPU burst, and run that process non-preemptively (until termination or blocking)
  - In case of a tie, FCFS is used to break the tie
- Difficulty: determining length of next CPU burst
  - Approximation — predict length, based on past performance of the process, and on past predictions

## SJF Example

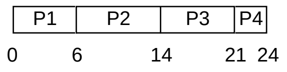
- SJF Example:

| Process (Arrival Order) | P1 | P2 | P3 | P4 |
|---|---|---|---|---|
| Burst Time | 6 | 8 | 7 | 3 |
| Arrival Time | 0 | 0 | 0 | 0 |



average waiting time = (0 + 3 + 9 + 16) / 4 = 7

- Same Example, FCFS Schedule:



average waiting time = (0 + 6 + 14 + 21) / 4 = 10.25

## SJF Evaluation

- Non-preemptive
- Response time — good for short processes
  - Long processes may have to wait until a large number of short processes finish
  - Provably *optimal* — minimizes average waiting time for a given set of processes
- Throughput — high
- Fairness — penalizes long processes
- Starvation — possible for long processes
- Overhead — can be high (recording and estimating CPU burst times)

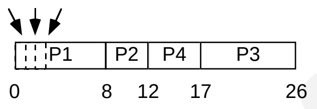
## Shortest-Remaining-Time (SRT)

- SRT is a preemptive version of SJF
- Policy:
  - Choose the process that has the smallest next CPU burst, and run that process preemptively…
    - (until termination or blocking, or
    - until a process enters the ready queue (either a new process or a previously blocked process))
  - At that point, choose another process to run if one has a smaller expected CPU burst than <u>what is left</u> of the current process' CPU burst

## SJF & SRT Example

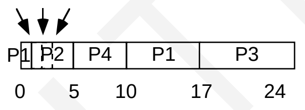
- SJF Example:

| Process (Arrival Order) | P1 | P2 | P3 | P4 |
|---|---|---|---|---|
| Burst Time | 8 | 4 | 9 | 5 |
| Arrival Time | 0 | 1 | 2 | 3 |



average waiting time = (0 + (8–1) + (12–3) + (17–2)) / 4 = 7.75

- Same Example, SRT Schedule:



average waiting time = ((0+(10–1) + (1–1) + (17–2) + (5–3)) / 4 = 6.5

## SRT Evaluation

- Preemptive (at arrival of process into ready queue)
- Response time — good
  - Provably *optimal* — minimizes average waiting time for a given set of processes
- Throughput — high
- Fairness — penalizes long processes
  - Note that long processes eventually become short processes
- Starvation — possible for long processes
- Overhead — can be high (recording and estimating CPU burst times)

## Priority Scheduling

- Policy:
  - Associate a priority with each process
    - Externally defined, based on importance, money, politics, etc.
    - Internally defined, based on memory requirements, file requirements, CPU requirements vs. I/O requirements, etc.
    - SJF is priority scheduling, where priority is inversely proportional to length of next CPU burst
  - Choose the process that has the highest priority, and run that process either:
    - preemptively, or
    - non-preemptively
- Evaluation
  - Starvation — possible for low-priority processes
    - Can avoid by *aging* processes: increase priority as they spend time in the system

## Multilevel Queue Scheduling

- Policy:
  - Use several ready queues, and associate a different priority with each queue
  - Choose the process from the occupied queue that has the highest priority, and run that process either:
    - preemptively, or
    - non-preemptively
  - Assign new processes permanently to a particular queue
    - Foreground, background
    - System, interactive, editing, computing
  - Each queue can have a different scheduling policy
    - Example: preemptive, using timer
      - 80% of CPU time to foreground, using RR
      - 20% of CPU time to background, using FCFS

## Multilevel Feedback Queue Scheduling

- Policy:
  - Use several ready queues, and associate a different priority with each queue
  - Choose the process from the occupied queue with the highest priority, and run that process either:
    - preemptively, or
    - non-preemptively
  - Each queue can have a different scheduling policy
  - Allow scheduler to move processes between queues
    - Start each process in a high-priority queue; as it finishes each CPU burst, move move it to a lower-priority queue
    - Aging — move older processes to higher-priority queues
    - Feedback = use the past to predict the future — favor jobs that haven't used the CPU much in the past — close to SRT!

## CPU Scheduling in UNIX using Multilevel Feedback Queue Scheduling

- Policy:
  - Multiple queues, each with a priority value (low value = high priority):
    - Kernel processes have negative values
      - Includes processes performing system calls, that just finished their I/O and haven't yet returned to user mode
    - User processes (doing computation) have positive values
  - Choose the process from the occupied queue with the highest priority, and run that process preemptively, using a timer (time slice typically around 100ms)
    - Round-robin scheduling in each queue
  - Move processes between queues
    - Keep track of clock ticks (60/second)
    - Once per second, add clock ticks to priority value
    - Also change priority based on whether or not process has used more than it's "fair share" of CPU time (compared to others)

## Deadlock

- Consider this example:

```
Process A               Process B

printer–>wait( );       disk–>wait( );
disk->wait( );          printer->wait( );
print file              print file
printer->signal( );     disk->signal( );
disk->signal( );        printer->signal( );
```

- *Deadlock* occurs when two or more processes are each waiting for an event that will never occur, since it can only be generated by another process in that set
- Deadlock is one of the more difficult problems that OS designers face
  - As we examine various approaches to dealing with deadlock, notice the tradeoffs between how well the approach solves the problem, and its performance /OS overhead

## Deadlock (cont.)

- OS must distribute system *resources* among competing processes:

| | |
|---|---|
| ● CPU cycles | preemptable |
| ● Memory space | preemptable |
| ● Files | non-preemptable |
| ● I/O devices (printer) | non-preemptable |

- A request for a type of resource can be satisfied by any resource of that type
  - Use any 100 bytes in memory
  - Use either one of two identical printers
- Process *requests* resource(s), *uses* it/them, then *releases* it/them
  - We will assume here that the resource is *re-usable*; it is not *consumed*
  - Waits if resource is not currently available

## Deadlock Conditions

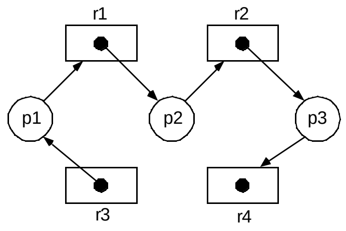
- These 4 conditions are **necessary** and **sufficient** for deadlock to occur:
  - **Mutual exclusion** — if one process holds a resource, other processes requesting that resource must wait until the process releases it (only one can use it at a time)
  - **Hold and wait** — processes are allowed to *hold* one (or more) resource and be *waiting* to acquire additional resources that are being held by other processes
  - **No preemption** — resources are released voluntarily; neither another process nor the OS can force a process to release a resource
  - **Circular wait** — there must exist a set of waiting processes such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, … Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held P0

## Resource-Allocation Graph

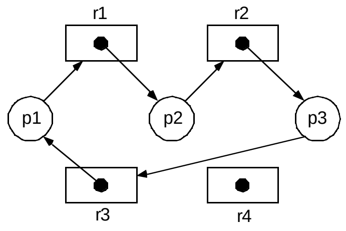
- The deadlock conditions can be modeled using a directed graph called a *resource-allocation graph* (RAG)
  - 2 kinds of nodes:
    - *Boxes* — represent resources
      - Instances of the resource are represented as dots within the box
    - *Circles* — represent threads / processes
  - 2 kinds of (directed) edges:
    - *Request edge* — from process to resource — indicates the process has requested the resource, and is waiting to acquire it
    - *Assignment edge* — from resource instance to process — indicates the process is holding the resource instance
  - When a request is made, a request edge is added
    - When request is fulfilled, the request edge is transformed into an assignment edge
    - When process releases the resource, the assignment edge is deleted

## Interpreting a RAG With Single Resource Instances

- If the graph does **not** contain a cycle, then **no** deadlock exists



- If the graph **does** contain a cycle, then a deadlock **does** exist



- With **single** resource instances, a **cycle** is a **necessary** and **sufficient** condition for deadlock

## Dealing with Deadlock

- *The Ostrich Approach* — stick your head in the sand and ignore the problem
- *Deadlock prevention* — prevent deadlock from occurring by eliminating one of the 4 deadlock conditions
- *Deadlock detection* algorithms — detect when deadlock has occurred
  - *Deadlock recovery* algorithms — break the deadlock
- *Deadlock avoidance* algorithms — consider resources currently available, resources allocated to each thread, and possible future requests, and only fulfill requests that will not lead to deadlock

## Deadlock Prevention

- Basic idea: ensure that one of the 4 conditions for deadlock can not hold
- **Mutual exclusion** — if one process holds a resource, other processes requesting that resource must wait until the process releases it (only one can use it at a time)
  - Hard to avoid mutual exclusion for non-sharable resources
    - Printer & other I/O devices
    - Files
  - However, many resources are sharable, so deadlock can be avoided for those resources
    - Read-only files
  - For printer, avoid mutual exclusion through spooling — then process won't have to wait on physical printer

## Deadlock Prevention (cont.)

- **Circular wait** — there must exist a set of waiting processes such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, … Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held P0
  - To avoid, impose a total order on all resources, and require process to request resource in that order
    - Order: disk drive, printer, CDROM
    - Process A requests disk drive, then printer
    - Process B requests disk drive, then printer
    - Process B does not request printer, then disk drive, which could lead to deadlock
  - Order should be in the logical sequence that the resources are usually acquired
    - Allow process to release all resources, and start request sequence over
    - Or force process to request total number of each resource in a single request

## Deadlock Prevention (cont.)

- **No preemption** — resources are released voluntarily; neither another process nor the OS can force a process to release a resource
  - To avoid, allow preemption
    - If process A requests resources that aren't available, see who holds those resources
      - If the holder (process B) is waiting on additional resources, preempt the resource requested by process A
      - Otherwise, process A has to wait
        - » While waiting, some of its current resources may be preempted
        - » Can only wake up when it acquires the new resources plus any preempted resources
    - If a process requests a resource that can not be allocated to it, **all** resources held by that process are preempted
      - Can only wake up when it can acquire all the requested resources
    - Only works for resources whose state can be saved/restored (memory, not printer)

## Deadlock Prevention (cont.)

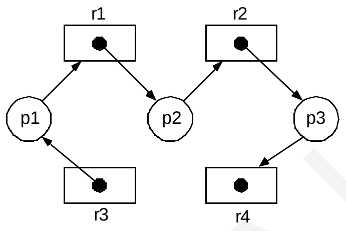
- **Hold and wait** — processes are allowed to *hold* one (or more) resource and be *waiting* to acquire additional resources that are being held by other processes
  - To avoid, ensure that whenever a process requests a resource, it doesn't hold any other resources
    - Request all resources (at once) at beginning of process execution
      - Process which loops forever?
    - Request all resources (at once) at any point in the program
    - To get a new resource, release all current resources, then try to acquire new one plus old ones all at once
  - Difficult to know what to request in advance
  - Wasteful; ties up resources and reduces resource utilization
  - Starvation is possible

## Resource-Allocation Graph (Review)

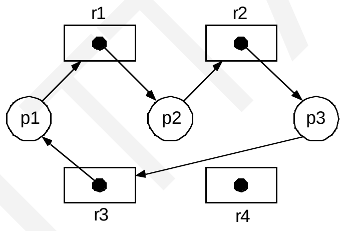
- The deadlock conditions can be modeled using a directed graph called a *resource-allocation graph* (RAG)
  - 2 kinds of nodes:
    - *Boxes* — represent resources
      - Instances of the resource are represented as dots within the box
    - *Circles* — represent threads / processes
  - 2 kinds of (directed) edges:
    - *Request edge* — from process to resource — indicates the process has requested the resource, and is waiting to acquire it
    - *Assignment edge* — from resource instance to process — indicates the process is holding the resource instance
  - When a request is made, a request edge is added
    - When request is fulfilled, the request edge is transformed into an assignment edge
    - When process releases the resource, the assignment edge is deleted

## Interpreting a RAG With Single Resource Instances (Review)

- If the graph does **not** contain a <u>cycle</u>, then **no** deadlock exists



- If the graph **does** contain a <u>cycle</u>, then a deadlock **does** exist



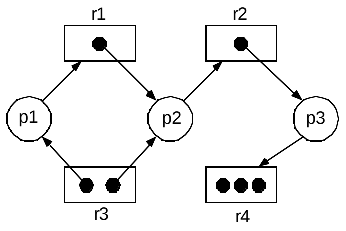
- With <u>**single**</u> resource instances, a <u>**cycle**</u> is a <u>**necessary**</u> and <u>**sufficient**</u> condition for deadlock

## Deadlock Detection (Single Resource of Each Type)

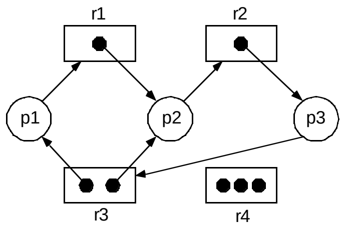
- If all resources have only a single instance, deadlock can be detected by searching the resource-allocation graph for cycles
  - Silberschatz defines a simpler graph, called the *wait-for* graph, and searches that graph instead
    - The wait-for graph is the resource-allocation graph, minus the resources
    - An edge from p1 to p2 means p1 is waiting for a resource that p2 holds (here we don't care which resource is involved)
- One simple algorithm:
  - Start at each node, and do a depth-first search from there
  - If a search ever comes back to a node it's already found, then it has found a cycle

## Interpreting a RAG With Multiple Resource Instances

- If the graph does **not** contain a <u>cycle</u>, then **no** deadlock exists



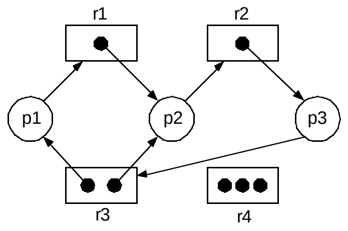
- If the graph **does** contain a <u>cycle</u>, then a deadlock **may** exist



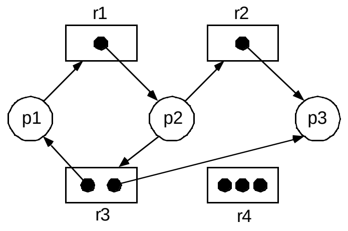
- With <u>**multiple**</u> resource instances, a <u>**cycle**</u> is a <u>**necessary**</u> (but not <u>**sufficient**</u>) condition for deadlock

## Interpreting a RAG With Multiple Resource Instances (cont.)

- If the graph **does** contain a <u>knot</u> (and a cycle), then a deadlock **does** exist



- If the graph **does not** contain a <u>knot</u>, then a deadlock **does not** exist



- With **<u>multiple</u>** resource instances, a **<u>knot</u>** is a **<u>sufficient</u>** condition for deadlock

## Deadlock Detection (Multiple Resources of Each Type)

- This algorithm (Coffman, 1971) uses the following data structures:

Existing Resources (E1, E2, E3, …, Em)

Available Resources (A1, A2, A3, …, Am)

Current Allocation

$$\begin{bmatrix} C11 & C12 & C13 & \cdots & C1m \\ C21 & C22 & C23 & \cdots & C2m \\ \vdots & \vdots & \vdots & & \vdots \\ Cn1 & Cn2 & Cn3 & \cdots & Cnm \end{bmatrix}$$

Request

$$\begin{bmatrix} R11 & R12 & R13 & \cdots & R1m \\ R21 & R22 & R23 & \cdots & R2m \\ \vdots & \vdots & \vdots & & \vdots \\ Rn1 & Rn2 & Rn3 & \cdots & Rnm \end{bmatrix}$$

- n processes, m types of resources
  - **Existing Resources** vector tells number of resources of each type that exist
  - **Available Resources** vector tells number of resources of each type that are available (unassigned to any process)
  - i-th row of **Current Allocation** matrix tells number of resources of each type allocated (assigned) to process i

## Deadlock Detection (Multiple Resources of Each Type) (cont.)

- Every resource is either allocated or available
  - Number of resources of type j that have been allocated to all processes, plus number of resources of type j that are available, should equal number of resources of type j in existence
- Processes may have unfulfilled requests
  - i-th row of **Request** matrix tells number of resources of each type process i has requested, but not yet received
- Notation: comparing vectors
  - If A and B are vectors, the relation $A \leq B$ means that each element of A is less than or equal to the corresponding element of B (i.e., $A \leq B$ iff $A_i \leq B_i$ for $0 \leq i \leq m$)
  - Furthermore, $A < B$ iff $A \leq B$ and $A \neq B$

## Deadlock Detection Algorithm (Multiple Resources of Each Type)

- Operation:
  - Every process is initially unmarked
  - As algorithm progresses, processes will be marked, which indicates they are able to complete, and thus are not deadlocked
  - When algorithm terminates, any unmarked processes are deadlocked
- Algorithm:
  1. Look for an unmarked process Pi for which the i-th row of the **Request** matrix is less than or equal to the **Available** vector
  2. If such a process is found, add the i-th row of the **Current** matrix to the **Available** vector, mark the process, and go back to step 1
  3. If no such process exists, the algorithm terminates

## Deadlock Detection Example (Multiple Resources of Each Type)

Existing Resources (4 2 3 1)

Available Resources (2 1 0 0)

Current Allocation

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request

$$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

resources = (tape drive plotter printer CDROM)

- Whose request can be fulfilled?
  - Process 1 — no — no CDROM available
  - Process 2 — no — no printer available
  - Process 3 — yes — give it the requested resources, and after it completes and releases those resources, A = (2 2 2 0)
  - Process 1 still can't run (no CDROM), but process 2 can run, giving A = (4 2 2 1)
  - Process 1 can run, giving A = (4 2 3 1)

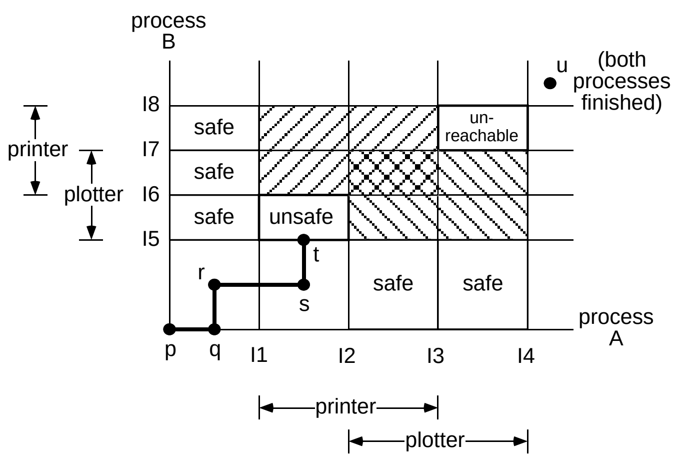
## After Deadlock Detection: Deadlock Recovery

- How often does deadlock detection run?
  - After every resource request?
  - Less often (e.g., every hour or so, or whenever resource utilization gets low)?
- What if OS detects a deadlock?
  - Terminate a process
    - All deadlocked processes
    - One process at a time until no deadlock
      - Which one?
      - One with most resources?
      - One with less cost?
        - CPU time used, needed in future
        - Resources used, needed
      - That's a choice similar to CPU scheduling
    - Is it acceptable to terminate process(es)?
      - May have performed a long computation
        - Not ideal, but OK to terminate it
      - Maybe have updated a file or done I/O
        - Can't just start it over again!

## After Deadlock Detection: Deadlock Recovery (cont.)

- Any less drastic alternatives?
  - Preempt resources
    - One at a time until no deadlock
    - Which "victim"?
      - Again, based on cost, similar to CPU scheduling
    - Is rollback possible?
      - *Preempt* resources — take them away
      - *Rollback* — "roll" the process back to some safe state, and restart it from there
        - OS must *checkpoint* the process frequently — write its state to a file
      - Could roll back to beginning, or just enough to break the deadlock
        - This second time through, it has to wait for the resource
        - Has to keep multiple checkpoint files, which adds a lot of overhead
    - Avoid starvation
      - May happen if decision is based on same cost factors each time
      - Don't keep preempting same process (i.e., set some limit)

## Deadlock Avoidance — Motivation



- Example to motivate a D.A. algorithm:
  - state p — neither process running
  - state q — scheduler ran A
  - state r — scheduler ran B
  - state s — scheduler ran A, A requested and received printer
  - state t — schedule ran B, B just requested and received plotter

## Deadlock Avoidance — Motivation (cont.)

- Look at shaded areas:
  - The one shaded “\\\” represents both processes using printer at same time — this is not allowed by mutual exclusion
  - Other (“///”) is similar, involving plotter
- Look at box marked “unsafe”
  - If OS enters this box, it will eventually deadlock because it will have to enter a shaded (illegal mutual exclusion) region
    - All paths must proceed up or right (why?)
  - Box is unsafe — should not be entered!
    - From state t, must avoid the unsafe area by going to the right (up to I4) (blocking B)
- At state t, the OS must decide whether or not to grant B’s request
  - A good choice will avoid deadlock!
  - Need to know resource needs in advance

## Deadlock Avoidance — Safe and Unsafe States

| | Has | Max |
|---|---|---|
| **A** | 3 | 9 |
| **B** | 2 | 4 |
| **C** | 2 | 7 |

free: 3
(a)

| | Has | Max |
|---|---|---|
| **A** | 3 | 9 |
| **B** | 4 | 4 |
| **C** | 2 | 7 |

free: 1
(b)

| | Has | Max |
|---|---|---|
| **A** | 3 | 9 |
| **B** | 0 | — |
| **C** | 2 | 7 |

free: 5
(c)

| | Has | Max |
|---|---|---|
| **A** | 3 | 9 |
| **B** | 0 | — |
| **C** | 7 | 7 |

free: 0
(d)

| | Has | Max |
|---|---|---|
| **A** | 3 | 9 |
| **B** | 0 | — |
| **C** | 0 | — |

free: 7
(e)

| | Has | Max |
|---|---|---|
| **A** | 9 | 9 |
| **B** | 0 | — |
| **C** | 0 | — |

free: 1
(f)

| | Has | Max |
|---|---|---|
| **A** | 0 | — |
| **B** | 0 | — |
| **C** | 0 | — |

free: 10
(g)

- State (a) is *safe*, meaning there **<u>exists</u>** a sequence of allocations that allows **<u>all</u>** processes to complete:
  - B runs, asks for 2 more resources, 1 free
    - B finishes, releases its resources, 5 free
  - C runs, asks for 5 more resources, 0 free
    - C finishes, releases its resources, 7 free
  - A runs, gets 6 more, everyone done…

## Deadlock Avoidance — Safe and Unsafe States (cont.)

| | Has | Max |
|---|---|---|
| **A** | 3 | 9 |
| **B** | 2 | 4 |
| **C** | 2 | 7 |

free: 3
(a)

| | Has | Max |
|---|---|---|
| **A** | 4 | 9 |
| **B** | 2 | 4 |
| **C** | 2 | 7 |

free: 2
(b)

| | Has | Max |
|---|---|---|
| **A** | 4 | 9 |
| **B** | 4 | 4 |
| **C** | 2 | 7 |

free: 0
(c)

| | Has | Max |
|---|---|---|
| **A** | 4 | 9 |
| **B** | 0 | — |
| **C** | 2 | 7 |

free: 4
(d)

- Suppose we start in state (a), and reach state (b) by giving A another resource
  - B runs, asks for 2 more resources, 0 free
    - B finishes, releases its resources, 4 free
  - C can’t run — might want 5 resources
    - Same for A
- State (b) is unsafe, meaning that from there, deadlock **<u>may</u>** eventually occur
    - State (b) is <u>not</u> a deadlocked state — the system can still run for a bit
    - Deadlock <u>may not</u> occur — A might release one of its resources before asking for more, which allows C to complete

## The Banker's Algorithm for Single Resources (Dijkstra, 1965)

| | Has | Max |
|---|---|---|
| **A** | 0 | 6 |
| **B** | 0 | 5 |
| **C** | 0 | 4 |
| **D** | 0 | 7 |

free: 10
(a)

| | Has | Max |
|---|---|---|
| **A** | 1 | 6 |
| **B** | 1 | 5 |
| **C** | 2 | 4 |
| **D** | 4 | 7 |

free: 2
(b)

| | Has | Max |
|---|---|---|
| **A** | 1 | 6 |
| **B** | 2 | 5 |
| **C** | 2 | 4 |
| **D** | 4 | 7 |

free: 1
(c)

- A banker has granted lines of credit to customers A, B, C, and D (unit is $1000)
  - She knows it's not likely they will all need their maximum credit at the same time, so she keeps only 10 units of cash on hand
  - At some point in time, the bank is in state (b) above, which is *safe*
    - Can let C finish, have 4 units available
    - Then let B or D finish, etc.
  - But… if banker gives B one more unit (state (c) above), state would be *unsafe* — if everyone asks for maximum credit, **no** requests can be fulfilled

## The Banker's Algorithm for Single Resources (cont.)

- Resource-request algorithm:
  - The banker considers each request as it occurs, determining whether or not fulfilling it leads to a safe state
    - If it does, the request is granted
    - Otherwise, it is postponed until later
- Safety algorithm:
  - To determine if a state is safe, the banker checks to see if she has enough resources to satisfy some customer
    - If so, she assumes those loans will be repaid (i.e., the process will use those resources, finish, and release all of its resources), and she checks to see if she has enough resources to satisfy another customer, etc.
  - If all loans can eventually be repaid, the state is safe and the initial request can be granted

## Evaluation of Deadlock Avoidance Using the Banker's Algorithm

- Advantages:
  - No need to preempt resources and rollback state (as in deadlock detection & recovery)
  - Less restrictive than deadlock prevention
- Disadvantages:
  - Maximum resource requirement for each process must be stated in advance
  - Processes being considered must be independent (i.e., unconstrained by synchronization requirements)
  - There must be a fixed number of resources (i.e., can't add resources, resources can't break) and processes (i.e., can't add or delete processes)
  - Huge overhead — must use the algorithm every time a resource is requested

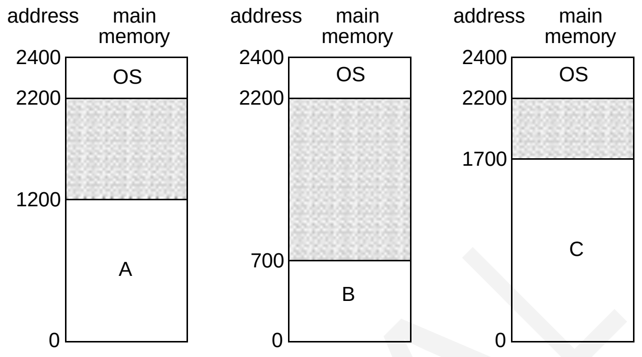
## Evaluating the Approaches to Dealing with Deadlock

- *The Ostrich Approach* — ignoring the problem
  - Good solution if deadlock isn't frequent
- *Deadlock prevention* — eliminating one of the 4 deadlock conditions
  - May be overly restrictive
- *Deadlock detection and recovery* — detect when deadlock has occurred, then break the deadlock
  - Tradeoff between frequency of detection and performance / overhead added
- *Deadlock avoidance* — only fulfilling requests that will not lead to deadlock
  - Need too much a priori information, not very dynamic (can't add processes or resources), huge overhead

## Classifying Information Stored in Memory

- By role in program:
  - Program instructions (unchangeable)
  - Constants: (unchangeable)
    - pi, maxnum, strings used by printf/scanf
  - Variables: (changeable)
    - Locals, globals, function parameters, dynamic storage (from malloc or new)
    - Initialized or uninitialized
- By protection status:
  - Readable and writable: variables
  - Read-only: code, constants
  - Important for sharing data and/or code
- Addresses vs. data:
  - Must modify addresses if program is moved (relocation, garbage collection)

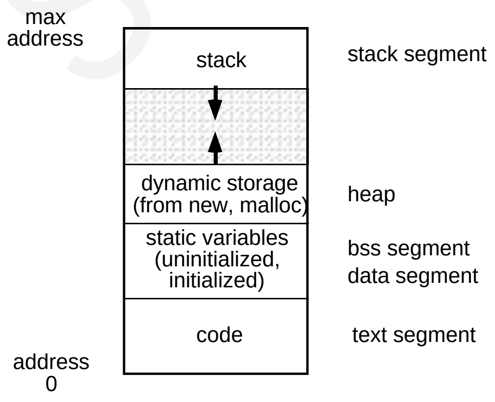
## Memory Management in a Uniprogrammed System



- OS gets a fixed segment of memory (usually highest memory)
- One process executes at a time in a single memory segment
  - Process is always loaded at address 0
  - Compiler and linker generate physical addresses
  - Maximum address = memory size – OS size

## Classifying Information Stored in Memory (cont.)

- Binding time (when is space allocated?):
  - Static: before program starts running
    - Program code, static global variables (initialized and uninitialized)
  - Dynamic: as program runs
    - Procedure stack, dynamic storage (space allocated by malloc or new)
- UNIX view of a process's memory (uniprogramming ***only*** for now):



## Segments of a Process

- Process' memory is divided into logical *segments* (text, data, bss, heap, stack)
  - Some are read-only, others read-write
  - Some are known at compile time, others grow dynamically as program runs
- Who assigns memory to segments?
  - *Compiler* and *assembler* generate an *object file* (containing code and data segments) from each *source file*
  - *Linker* combines all the object files for a program into a single executable object file, which is complete and self-sufficient
  - *Loader* (part of OS) loads an executable object file into memory at location(s) determined by the operating system
  - *Program* (as it runs) uses new and malloc to dynamically allocate memory, gets space on stack during function calls

## Linking

- Functions of a *linker*:
  - Combine all files and libraries of a program
  - Regroup all the segments from each file together (one big data segment, etc.)
  - Adjust addresses to match regrouping
  - Result is an executable program
- Contents of object files:
  - File header — size and starting address (in memory) of each segment
  - Segments for code and initialized data
  - Symbol table (symbols, addresses)
  - External symbols (symbols, location)
  - Relocation information (symbols, location)
  - Debugging information
  - For UNIX details, type "man a.out"

## Why is Linking Difficult?

- When assembler assembles a file, it may find *external references* — symbols it doesn't know about (e.g., printf, scanf)
  - Compiler just puts in an address of 0 when producing the object code
  - Compiler records external symbols and their location (in object file) in a *cross-reference list*, and stores that list in the object file
  - Linker must *resolve* those external references as it links the files together
- Compiler doesn't know where program will go in memory (if multiprogramming, always 0 for uniprogramming)
  - Compiler just assumes program starts at 0
  - Compiler records *relocation information* (location of addresses to be adjusted later), and stores it in the object file

## Loading

- The *loader* loads the completed program into memory where it can be executed
  - Loads code and initialized data segments into memory at specified location
  - Leaves space for uninitialized data (bss)
  - Returns value of start address to operating system
- Alternatives in loading (*next 2 lectures*…)
  - *Absolute loader* — loads executable file at fixed location
  - *Relocatable loader* — loads the program at an arbitrary memory location specified by OS (needed for multiprogramming, not for uniprogramming)
    - Assembler and linker assume program will start at location 0
    - When program is loaded, loader modifies all addresses by adding the real start location to those addresses

## Running the Program — Static Memory Allocation

- Compiling, linking, and loading is sufficient for static memory
  - Code, constants, static variables
- In other cases, static allocation is not sufficient:
  - Need dynamic storage — programmer may not know how much memory will be needed when program runs
    - Use malloc or new to get what's necessary when it's necessary
    - For complex data structures (e.g., trees), allocate space for nodes on demand
  - OS doesn't know in advance which procedures will be called (would be wasteful to allocate space for every variable in every procedure in advance)
  - OS must be able to handle recursive procedures

## Running the Program — Dynamic Memory Allocation

- Dynamic memory requires two fundamental operations:
  - Allocate dynamic storage
  - Free memory when it's no longer needed
  - Methods vary for stack and heap
- Two basic methods of allocation:
  - Stack (hierarchical)
    - Good when allocation and freeing are somewhat predictable
    - Typically used:
      - to pass parameters to procedures
      - for allocating space for local variables inside a procedure
      - for tree traversal, expression evaluation, parsing, etc.
    - Use stack operations: **push** and **pop**
    - Keeps all free space together in a structured organization
    - Simple and efficient, but restricted

## Running the Program — Dynamic Memory Allocation (cont.)

- Two basic methods of allocation:
  - Heap
    - Used when allocation and freeing are not predictable
    - Typically used:
      - for arbitrary list structures, complex data organizations, etc.
    - Use **new** or **malloc** to allocate space, use **delete** or **free** to release space
    - System memory consists of allocated areas and free areas (holes)
    - Problem: eventually end up with many small holes, each too small to be useful
      - This is called *fragmentation*, and it leads to wasted memory
      - Fragmentation wasn't a problem with stack allocation, since we always add/delete from top of stack
      - Solution goal: reuse the space in the holes in such a way as to keep the number of holes small, and their size large
    - Compared to stack: more general, less efficient, more difficult to implement

## Topics in Memory Management

- Uniprogrammed operating systems
  - Assembling, linking, loading
  - Static memory allocation
  - Dynamic memory allocation
    - Stacks, heaps
    - Managing the free list, memory reclamation
- Multiprogrammed operating systems
  - Includes most of the above topics
  - Static relocation
  - Dynamic relocation
    - Virtual vs. physical address
    - Partitioning (and compaction)
    - Segmentation
    - Paging
  - Swapping
  - Demand paging

## Managing the Free List

- Heap-based dynamic memory allocation techniques typically maintain a *free list*, which keeps track of all the holes
- Algorithms to manage the free list:
  - Best fit
    - Keep linked list of free blocks
    - Search the whole list at each allocation
    - Choose the hole that comes the closest to matching the request size
      - Any unused space becomes a new (smaller) hole
    - When freeing memory, combine adjacent holes
    - Any way to do this efficiently?
  - First fit
    - Scan the list for the first hole that is large enough, choose that hole
    - Otherwise, same as best fit
  - Which is better? Why??

## Reclaiming Dynamic Memory

- When can memory be freed?
  - Whenever programmer says to
  - Any way to do so automatically?
    - Difficult if that item is shared (i.e., if there are multiple pointers to it)
- Potential problems in reclamation
  - Dangling pointers — have to make sure that <u>everyone</u> is finished using it
  - Memory leak — must not "lose" memory by forgetting to free it when appropriate
- Implementing automatic reclamation:
  - Reference counts
    - Used by file systems
    - OS keeps track of number of outstanding pointers to each memory item
    - When count goes to zero, free the memory

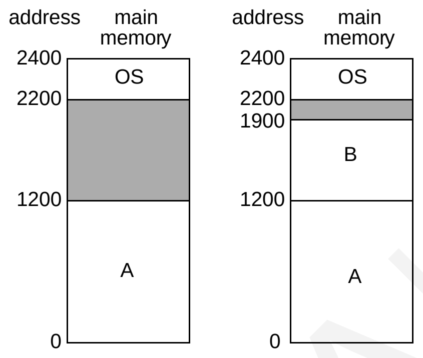
## Reclaiming Dynamic Memory (cont.)

- Implementing automatic reclamation:
  - Garbage collection
    - Used in LISP
    - Storage isn't explicitly freed by a free operation; programmer just deletes the pointers and doesn't worry about what it's pointing at
    - When OS needs more storage space, it recursively searches through all the active pointers and reclaims memory that no one is using
    - Makes life easier for application programmer, but is difficult to program the garbage collector
    - Often expensive — may use 20% of CPU time in systems that use it
      - May spend as much as 50% of time allocating and automatically freeing memory

## Multiprogramming — Goals in Sharing the Memory Space

- *Transparency*:
  - Multiple processes must coexist in memory
  - No process should be aware that the memory is shared
  - Each process should execute regardless of where it is located in memory
- *Safety*:
  - Processes must not be able to corrupt each other, or the OS
  - *Protection* mechanisms are used to enforce safety
- *Efficiency*:
  - The performance of the CPU and memory should not degrade very much as a result of sharing

## Static Relocation



- Put the OS in the highest memory
- Compiler and linker assume each process starts at address 0
- At load time, the OS:
  - Allocates the process a segment of memory in which it fits completely
  - Adjusts the addresses in the processes to reflect its assigned location in memory

## Static vs. Dynamic Relocation

- Problems with static relocation:
  - Safety — not satisfied — one process can access / corrupt another's memory, can even corrupt OS's memory
  - Processes can not change size (why…?)
  - Processes can not move after beginning to run (why would they want to?)
  - Used by MS-DOS, Windows, Mac OS
- An alternative: dynamic relocation
  - The basic idea is to change each memory address dynamically <u>as the process runs</u>
  - This translation is done by hardware — between the CPU and the memory is a *memory management unit* (MMU) (also called a *translation unit* ) that converts virtual addresses to physical addresses
    - This translation happens for every memory reference the process makes

## Dynamic Relocation

- There are now two different views of the address space:
  - The *physical address space* — seen only by the OS — is as large as there is physical memory on the machine
  - The *virtual (logical) address space* —seen by the process — can be as large as the instruction set architecture allows
    - For now, we'll assume it's much smaller than the physical address space
  - Multiple processes share the physical memory, but each can see only its own virtual address space
- The OS and hardware must now manage two different addresses:
  - *Virtual address* — seen by the process
  - *Physical address* — address in physical memory (seen by OS)

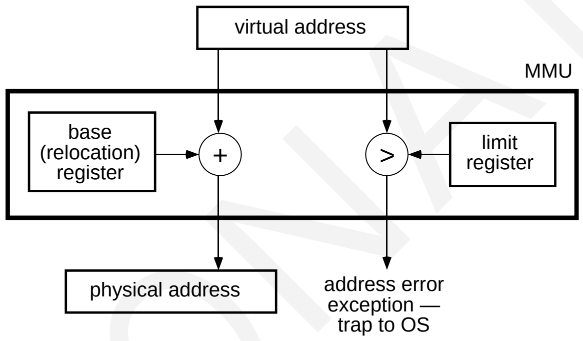
## Topics in Memory Management (Review)

- Uniprogrammed operating systems
  - Assembling, linking, loading
  - Static memory allocation
  - Dynamic memory allocation
    - Stacks, heaps
    - Managing the free list, memory reclamation
- Multiprogrammed operating systems
  - Includes most of the above topics
  - Static relocation
  - Dynamic relocation
    - Virtual vs. physical address
    - Partitioning (and compaction)
    - Segmentation
    - Paging
  - Swapping
  - Demand paging

## Static vs. Dynamic Relocation (Review)

- Problems with static relocation:
  - Safety — not satisfied — one process can access / corrupt another's memory, can even corrupt OS's memory
  - Processes can not change size (why…?)
  - Processes can not move after beginning to run (why would they want to?)
  - Used by MS-DOS, Windows, Mac OS
- An alternative: dynamic relocation
  - The basic idea is to change each memory address dynamically as the process runs
  - This translation is done by hardware — between the CPU and the memory is a *memory management unit* (MMU) (also called a *translation unit*) that converts virtual addresses to physical addresses
    - This translation happens for every memory reference the process makes

## Implementing Dynamic Relocation


virtual address
MMU
base (relocation) register
+
>
limit register
physical address
address error exception — trap to OS

- MMU protects address space, and translates virtual addresses
  - *Base register* holds lowest virtual address of process, *limit register* holds highest
  - Translation:
    physical address = virtual address + base
  - Protection:
    if virtual address > limit, then trap to the OS with an address exception

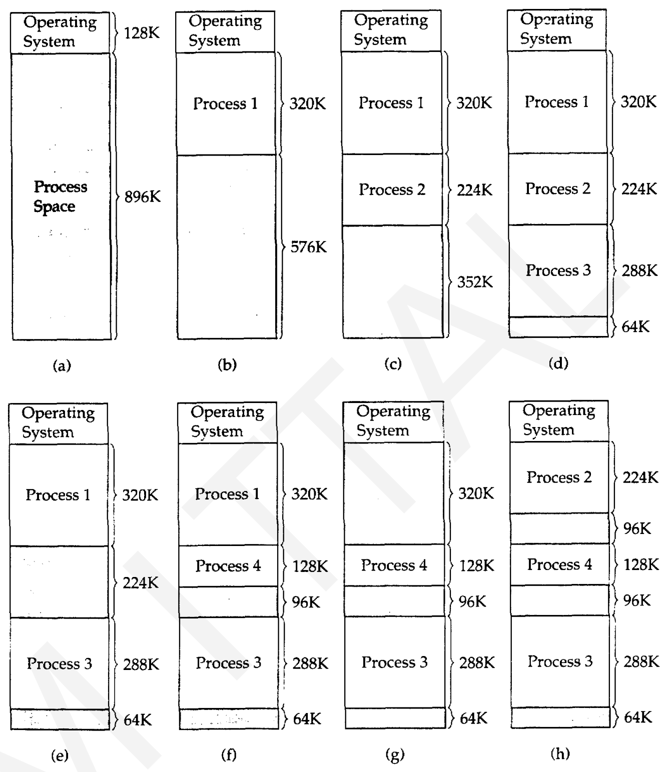
## Dynamic Relocation — OS vs. User Programs

- User programs (processes) address their own virtual memory
  - Run in relocation mode — indicated by a bit in the PSW — and in user mode
    - User programs can not change the relocation mode
- OS directly addresses physical memory
  - OS runs with relocation turned off, and in kernel mode
- When user program makes a system call:
  - CPU atomically goes into kernel mode, turns off relocation, traps to trap handler
  - OS trap handler accesses physical memory and does whatever is necessary to service the system call
  - CPU atomically turns on relocation, goes into user mode, returns to user program

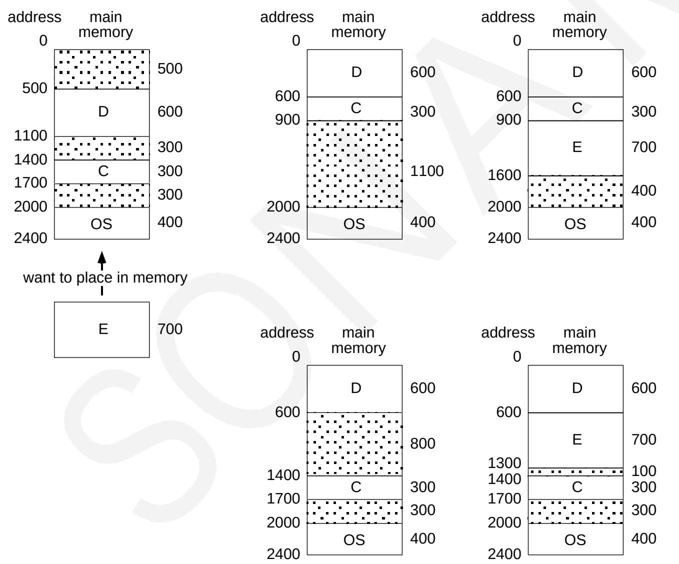
## Dynamic Relocation and Partitioning

- Physical memory is divided into *partitions*
  - A process is loaded into a free partition (a “hole” in the memory space)
- Fixed-size partitions:
  - Memory is divided into a predetermined number of fixed-size partitions
    - Partitions may be either of equal size, or of different (although fixed) sizes
  - Use first-fit, best-fit, etc. as discussed for dynamic allocation of heaps
  - Number of partitions limits the *degree of multiprogramming* — number of active processes
- Dynamic (variable-size) partitions:
  - When a process gets brought into memory, it is allocated a partition of exactly the right size

## Effect of Dynamic Relocation with Dynamic Partitioning



## Compaction



- Evaluation:
  - Memory moved =
  - Space created =

## Swapping (Medium-Term Scheduling)

- If there isn’t room enough in memory for all processes, some processes can be swapped out to make room
  - OS *swaps a process out* by storing its complete state to disk
  - OS can reclaim space used (not really…) by ready or blocked processes
- When process becomes active again, OS must *swap* it back *in* (into memory)
  - With static relocation, the process must be replaced in the same location
  - With dynamic relocation, OS can place the process in any free partition (must update the relocation and limit registers)
- Swapping and dynamic relocation make it easy to increase the size of a process and to compact memory (although slow!)

## UNIX Process Model (From Lecture 06)

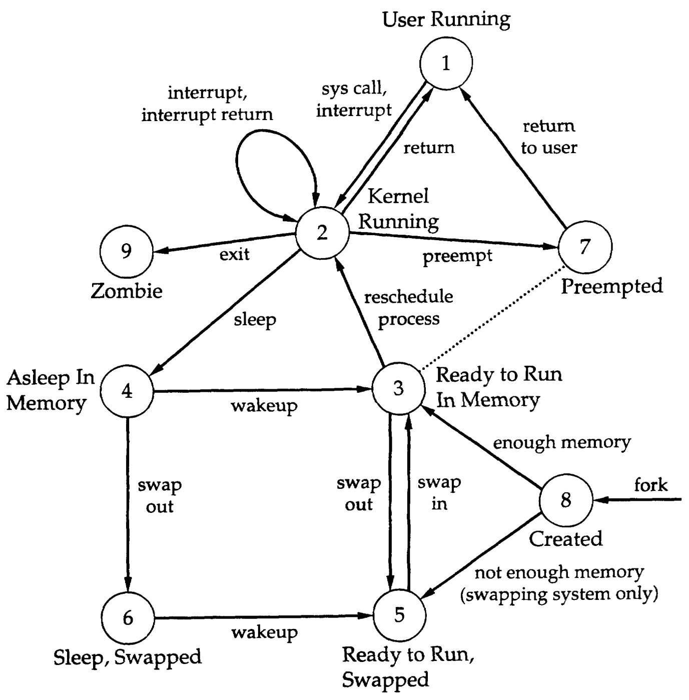


**FIGURE 3.16 UNIX process state transition diagram [BACH86]**

Figure from *Operating Systems*, 2nd edition, Stallings, Prentice Hall, 1995

Original diagram from *The Design of the UNIX Operating System*, M. Bach, Prentice Hall, 1986

## Evaluation of Dynamic Relocation

- Advantages:
  - OS can easily move a process
  - OS can allow processes to grow
  - Hardware changes are minimal, but fairly fast and efficient

  ➥Transparency, safety, and efficiency are all satisfied, although there is some small overhead to dynamic relocation

- Disadvantages:
  - Compared to static relocation, memory addressing is slower due to translation
  - Memory allocation is complex (partitions, holes, fragmentation, etc.)
  - If process grows, OS may have to move it
  - Process limited to physical memory size
  - Not possible to share code or data between processes

## Evaluation of Dynamic Relocation

- Advantages:
  - OS can easily move a process
  - OS can allow processes to grow
  - Hardware changes are minimal, but fairly fast and efficient
  - ➥Transparency, safety, and efficiency are all satisfied; overhead is small
- Disadvantages:
  - Addresses must be translated
  - Memory allocation is complex (partitions, holes, fragmentation, etc.)
  - If process grows, OS may have to move it
  - Process limited to physical memory size
  - Process needs contiguous memory space
  - Not possible to share code or data between processes

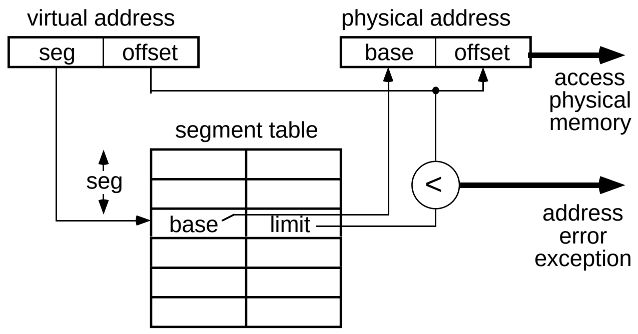
## Segmentation

- Basic idea — using the programmer's view of the program, divide the process into separate *segments* in memory
  - Each segment has a distinct purpose:
    - Example: code, static data, heap, stack
      - Maybe even a separate code and stack segment for each function
    - Segments may be of different sizes
    - Stack and heap don't conflict
  - The whole process is still loaded into memory, but the segments that make up the process do ***not*** have to be loaded contiguously into memory
    - Space within a segment is contiguous
- Each segment has *protection bits*
  - Read-only segment (code)
  - Read-write segments (data, heap, stack)
  - Allows processes to share code and data

## Segment Addresses

- Virtual (logical) address consists of:
  - Segment number
  - Offset from beginning of that segment
  - Both are generated by the assembler
- What is stored in the instruction?
  - Simple method:
    - Top bits of address specify segment
    - Bottom bits of address specify offset
  - Implicit segment specification:
    - Segment is selected implicitly by the instruction being executed (code vs. data)
    - Examples: PDP-11, Intel 386/486
  - Explicit segment specification:
    - Instruction prefix can request that a specific segment be used
    - Example: Intel 386/486

## Implementing Segments



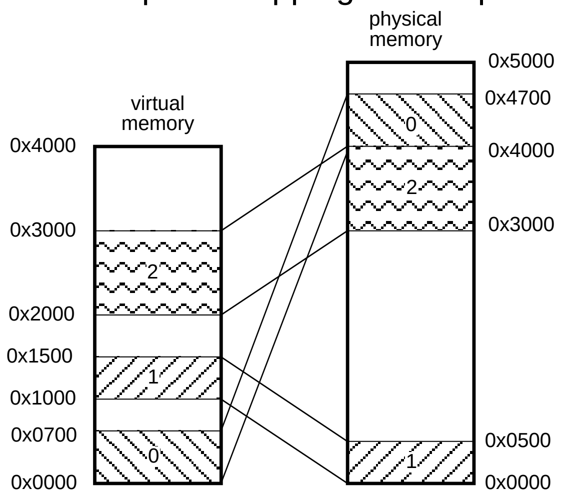
- A *segment table* keeps track of every segment in a particular process
  - Each entry contains base and limit
  - Also contains protection information (sharing allowed, read vs. read/write)
- Additional hardware support required:
  - Multiple base and limit registers, or
  - Segment table base pointer (points to table in memory)

## Segmentation Example

- 2 bits for segment number, 12 bit offset
- Part of segment table:

| segment | base | limit | R W |
|---|---|---|---|
| 0 | 0x4000 | 0x6FF | 1 0 |
| 1 | 0x0000 | 0x4FF | 1 1 |
| 2 | 0x3000 | 0xFFF | 1 1 |
| 4 | | | 0 0 |

- Address space mapping for that part:



## Managing Segments

- When a process is created:
  - Allocate space in virtual memory for all of the process's segments
  - Create a (mostly empty) segment table, and store it in the process's PCB
- When a context switch occurs:
  - Save the OS's segment table in the old process's PCB
  - Load OS's segment table from new process's PCB, allocating space in physical memory if first time process runs
- If there's no space in physical memory:
  - Compact memory (move segments, update bases) to make contiguous space
  - Swap one or more segments out to disk
    - To run that process again, swap *all* of its segments back into memory

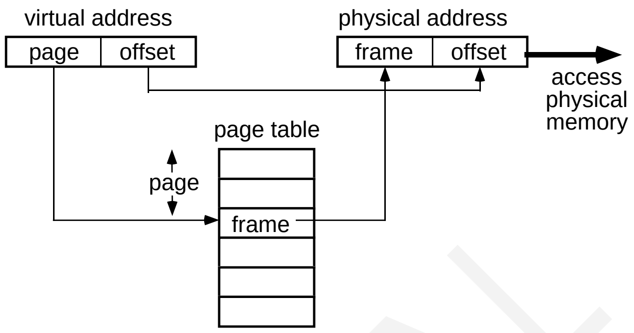
## Managing Segments (cont.)

- To enlarge a segment:
  - If space above the segment is free, OS can just update the segment's limit and use some of that space
  - Move this segment to a larger free space
  - Swap the segment above this one to disk
  - Swap this segment to disk, and bring it back into a larger free space
- Advantages of segmentation:
  - Segments don't have to be contiguous
  - Segments can be swapped independently
  - Segments allow sharing
- Disadvantages of segmentation:
  - Complex memory allocation (first-fit, etc.)
  - External fragmentation

## Paging

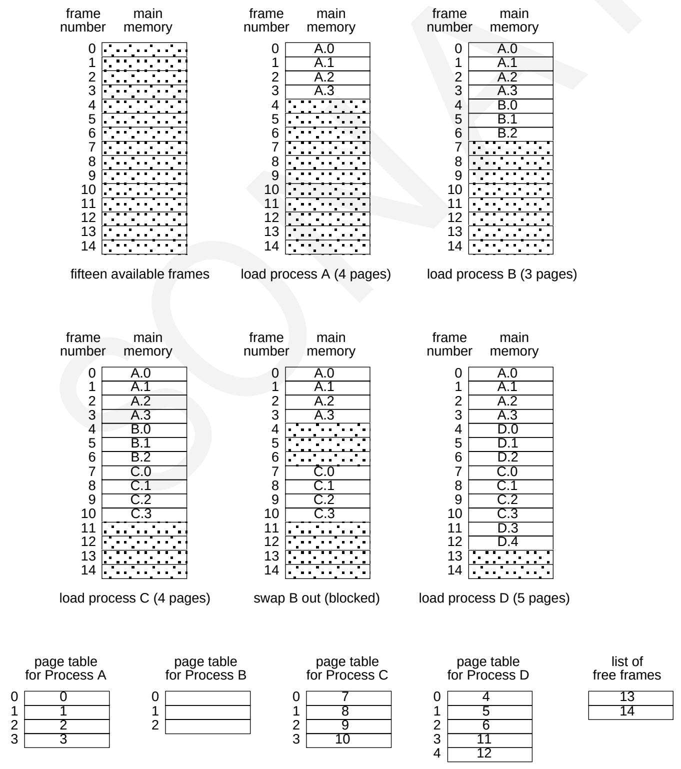
- Compared to segmentation, paging:
  - Makes allocation and swapping easier
  - No external fragmentation
- Each process is divided into a number of small, fixed-size partitions called *pages*
  - Physical memory is divided into a large number of small, fixed-size partitions called *frames*
  - Pages have nothing to do with segments
  - Page size = frame size
    - Usually 512 bytes to 16K bytes
  - The whole process is still loaded into memory, but the pages of a process do ***not*** have to be loaded into a contiguous set of frames
  - Virtual address consists of page number and offset from beginning of that page

## Implementing Paging



- A *page table* keeps track of every page in a particular process
  - Each entry contains the corresponding frame in main (physical) memory
  - Can add protection bits, but not as useful
- Additional hardware support required is slightly less than for segmentation
  - No need to keep track of, and compare to, limit. Why not?

## Paging Example



page table for Process A

| | |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |

page table for Process B

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | |

page table for Process C

| | |
|---|---|
| 0 | 7 |
| 1 | 8 |
| 2 | 9 |
| 3 | 10 |

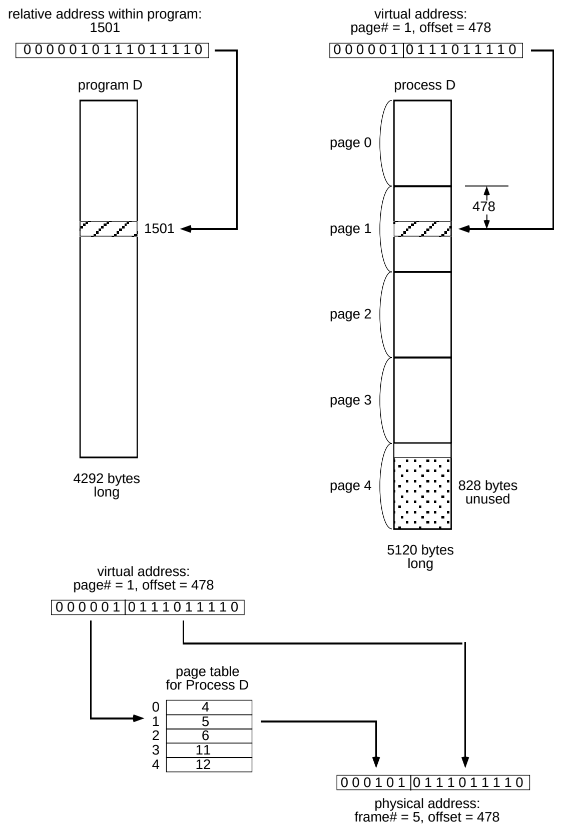
page table for Process D

| | |
|---|---|
| 0 | 4 |
| 1 | 5 |
| 2 | 6 |
| 3 | 11 |
| 4 | 12 |

list of free frames

| |
|---|
| 13 |
| 14 |

## Paging Example (cont.)



## Managing Pages and Frames

- OS usually keeps track of free frames in memory using a bit map
  - A bit map is just an array of bits
    - 1 means the frame is free
    - 0 means the frame is allocated to a page
  - To find a free frame, look for the first 1 bit in the bit map
    - Most modern instruction sets have an instruction that returns the offset of the first 1 bit in a register
- Page table base pointer (special register) points to page table of active process
  - Saved/restored as part of context switch
  - Page table also contains:
    - Valid bit — indicates page is in the process's virtual address space
    - Other bits for demand paging (discussed next time)

## Evaluation of Paging

- Advantages:
  - Easy to allocate memory — keep a list of available frames, and simple grab first one that's free
  - Easy to swap — pages, frames, and often disk blocks as well, all are same size
  - One frame is just as good as another!
- Disadvantages:
  - Page tables are fairly large
    - Most page tables are too big to fit in registers, so they must live in physical memory
    - This table lookup adds an extra memory reference for every address translation
  - Internal fragmentation
    - Always get a whole page, even for 1 byte
    - Larger pages makes the problem worse
    - Average = 1/2 page per process

## Address Translation, Revisited

- A modern microprocessor has, say, a 32 bit virtual address space ($2^{32}$ = 4 GB)
  - If page size is 1k ($2^{10}$), that means all the page tables combined could have over $2^{22}$ (approximately 4 million) page entries, each at least a couple of bytes long
  - Problem: if the main memory is only, say, 16 Mbytes, storing these page table there presents a problem!
    - Solution: store page tables in virtual memory, bring in pieces as necessary
  - New problem: memory access time may be doubled since the page tables are now subject to paging
    - (one access to get info from page table, plus one access to get data from memory)
    - New solution: use a special cache (called a Translation Lookaside Buffer (TLB)) to cache page table entries

## Translation Lookaside Buffer (TLB)

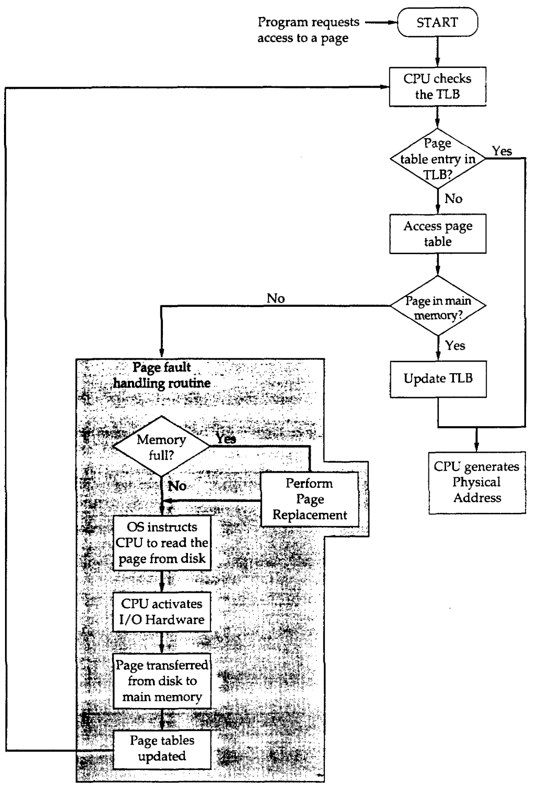

Program requests access to a page
START
CPU checks the TLB
Page table entry in TLB?
Yes
No
Access page table
Page in main memory?
No
Yes
Update TLB
CPU generates Physical Address
Page fault handling routine
Memory full?
Yes
No
Perform Page Replacement
OS instructs CPU to read the page from disk
CPU activates I/O Hardware
Page transferred from disk to main memory
Page tables updated

FIGURE 5.16 Operation of paging and translation lookaside buffer (TLB) [FURH87]

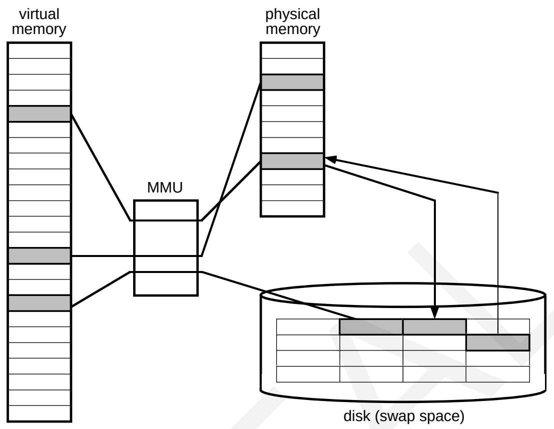
## Paging and Segmentation

- Use two levels of mapping:
  - Process is divided into variable-size segments
    - Segments are logical divisions as before
  - Each segment is divided into many small fixed-size pages
    - Pages are easy for OS to manage
    - Eliminates external fragmentation
  - Virtual address = segment, page, offset
  - One segment table per process, one page table per segment
- Sharing at two levels: segment, page
  - Share frame by having same frame reference in two page tables
  - Share segment by having same base in two segment tables
  - Still need protection bits (sharing, r/o, r/w)

## Memory Management So Far

- An application's view of memory is its virtual address space
  - OS's view of memory is the physical address space
  - A MMU (hardware) is used to implement segmentation, paging, or a combination of the two, by providing address translation
- Limitation until now — ***all*** segments / pages of a process must be in main (physical) memory for it to run
  - When process isn't running, the ***entire process*** can be *swapped out* to disk
- Insight — at a given time, we probably only need to access some small subset of process's virtual memory
  - Load pages / segments on demand

## Demand Paging (Virtual Memory)



- At a given time, a virtual memory page will be stored either:
  - In a frame in physical memory
  - On disk (*backing store*, or *swap space*)
- A process can run with only part of its virtual address space in main memory
  - Provide illusion of almost infinite memory

## Starting a New Process

- Processes are started with 0 or more of their virtual pages in physical memory, and the rest on the disk
- *Page selection* — ***when*** are new pages brought into physical memory?
  - Prepaging — pre-load enough to get started: code, static data, one stack page (DEC ULTRIX)
  - Demand paging — start with 0 pages, load each page on demand (when a page fault occurs) (most common approach)
    - Disadvantage: many (slow) page faults when program starts running
- Demand paging works due to the principle of *locality of reference*
  - Knuth estimated that 90% of a program's time is spent in 10% of the code

## Page Faults

- An attempts to access a page that's not in physical memory causes a *page fault*
  - Page table must include a *present* bit (sometimes called *valid* bit) for each page
  - An attempt to access a page without the present bit set results in a *page fault*, an *exception* which causes a *trap* to the OS
  - When a page fault occurs:
    - OS must *page in* the page — bring it from disk into a free frame in physical memory
    - OS must update page table & present bit
    - Faulting process continues execution
- Unlike interrupts, a page fault can occur any time there's a memory reference
  - Even in the middle of an instruction! (how? and why not with interrupts??)
  - However, handling the page fault must be invisible to the process that caused it

## Handling Page Faults

- The page fault handler must be able to recover enough of the machine state (at the time of the fault) to continue executing the program
- The PC is usually incremented at the beginning of the instruction cycle
  - If OS / hardware doesn't do anything special, faulting process will execute the next instruction (skipping faulting one)
- With hardware support:
  - Test for faults before executing instruction (IBM 370)
  - Instruction completion — continue where you left off (Intel 386…)
  - Restart instruction, undoing (if necessary) whatever the instruction has already done (PDP-11, MIPS R3000, DEC Alpha, most modern architectures)

## Performance of Demand Paging

- Effective access time for demand-paged memory can be computed as:

  eacc = (1–p)(macc) + (p)(pfault)

  where:

  p = probability that page fault will occur

  macc = memory access time

  pfault = time needed to service page fault
- With typical numbers:

  eacc = (1–p)(100) + (p)(25,000,000)
  = 100 + (p)(24,999,800)
  - If p is 1 in 1000,
    eacc = 25,099 ns (250 times slower!)
  - To keep overhead under 10%,
    110 > 100 + (p)(24,999,800)
    - p must be less than 0.0000004
    - Less than 1 in 2,5000,000 memory accesses must page fault!

## Page Replacement

- When the OS needs a frame to allocate to a process, and all frames are busy, it must evict (copy to backing store) a page from its frame to make room in memory
  - Reduce overhead by having CPU set a *modified / dirty* bit to indicate that a page has been modified
    - Only copy data back to disk for dirty pages
    - For non-dirty pages, just update the page table to refer to copy on disk
- Which page to we choose to replace? Some page replacement policies:
  - Random
    - Pick any page to evict
  - FIFO
    - Evict the page that has been in memory the longest (use a queue to keep track)
    - Idea is to give all pages "fair" (equal) use of memory

## Page Replacement Policy

- When OS needs a frame to use, and all are busy, which page does it evict?
  - Random
    - Pick any page to evict
  - FIFO
    - Evict the page that has been in memory the longest (use a queue to keep track)
  - Optimal (Minimal)
    - Evict the page that will be referenced the farthest into the future
      - Requires knowledge of future
    - Cannot really be implemented
      - Useful for evaluating other policies
  - Least-Recently-Used (LRU)
    - Use the past to predict the future
    - Evict the page that has been unreferenced for the longest period of time

## Page Reference Example

- Assumptions: 4 pages, 3 frames
  Page references: ABCABDADBCB

| FIFO | A | B | C | A | B | D | A | D | B | C | B |
|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 1 | | | | | | | | | | | |
| frame 2 | | | | | | | | | | | |
| frame 3 | | | | | | | | | | | |

| Optimal | A | B | C | A | B | D | A | D | B | C | B |
|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 1 | | | | | | | | | | | |
| frame 2 | | | | | | | | | | | |
| frame 3 | | | | | | | | | | | |

| LRU | A | B | C | A | B | D | A | D | B | C | B |
|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 1 | | | | | | | | | | | |
| frame 2 | | | | | | | | | | | |
| frame 3 | | | | | | | | | | | |

## Implementing LRU

- A perfect implementation would be something like this:
  - Associate a clock register with every page in physical memory
  - Update the clock value at every access
  - During replacement, scan through all the pages and find the one with the lowest value in its clock register
  - What's wrong with all this?
- Simple approximations:
  - FIFO
  - Not-recently-used (NRU)
    - Use an R (reference) bit, and set it whenever a page is referenced
    - Clear the R bit periodically, such as every clock interrupt
    - Choose any page with a clear R bit to evict

## Implementing LRU (cont.)

- Clock / Second Chance Algorithm
  - Use an R (reference) bit as before
  - On a page fault, circle around the "clock" of all pages in the user memory pool
    - Start after the page examined last time
    - If the R bit for the page is set, clear it
    - If the R bit for the page is clear, replace that page and set the bit
  - Questions:
    - Can it loop forever?
    - What does it mean if the "hand" is moving slowly? …if the hand is moving quickly?
- Least Frequently Used (LFU) / N-th Chance Algorithm
  - Don't evict a page until hand has swept by N times
  - Use an R bit and a counter
  - How is N chosen? Large or small?

## Frame Allocation

- How many frames does each process get (M frames, N processes)?
  - At least 2 frames (one for instruction, one for memory operand), maybe more…
  - Maximum is number in physical memory
- Allocation algorithms:
  - Equal allocation
    - Each gets M / N frames
  - Proportional allocation
    - Number depends on size and priority
- Which pool of frames is used for replacement?
  - Local replacement
    - Process can only reuse its own frames
  - Global replacement
    - Process can reuse any frame (even if used by another process)

## Thrashing

- Consider what happens when memory gets overcommitted:
  - After each process runs, before it gets a chance to run again, all of its pages may get paged out
  - The next time that process runs, the OS will spend a ***lot*** of time page faulting, and bringing the pages back in
    - All the time it's spending on paging is time that it's not getting useful work done
    - With demand paging, we wanted a very large virtual memory that would be as fast as physical memory, but instead we're getting one that's as slow as the disk!
- This wasted activity due to frequent paging is called *thrashing*
  - Analogy — student taking too many courses, with too much work due

## Working Sets

- Thrashing occurs when the sum of all processes' requirement is greater than physical memory
    - Solution — use local page frame replacement, don't let processes compete
      - Doesn't help, as an individual process can still thrash
    - Solution — only give a process the number of frames that it "needs"
      - Change number of frames allocated to each process over time
      - If total need is too high, pick a process and suspend it
- *Working set* (Denning, 1968) — the collection of pages that a process is working with, and which must be resident in main memory, to avoid thrashing
  - Always keep working set in memory
  - Other pages can be discarded as necessary

## Rules for "The Paging Game"

1. Each player gets several million *things*.
2. Things are kept in *crates* that hold 1024 things each. Things in the same crate are called crate-mates.
3. Crates are stored either in the *workshop* or the *warehouse*. The workshop is almost always too small to hold all the crates.
4. There is only one workshop but there may be several warehouses. Everybody shares them.
5. Each thing has its own *thing number*.
6. What you do with a thing is to *zark* it. Everybody takes turns zarking.
7. You can only zark your things, not anybody else's.
8. Things can only be zarked when they are in the workshop.
9. Only the *Thing King* knows whether a thing is in the workshop or in a warehouse.
10. The longer a thing goes without being zarked, the *grubbier* it is said to become.

## Rules for “The Paging Game” (cont.)

11. The way you get things is to ask the Thing King.

12. The way you zark a thing is to give its thing number. If you give the number of a thing that happens to be in a workshop it gets zarked right away. If it is in a warehouse, the Thing King packs the crate containing your thing back into the workshop. If there is no room in the workshop, he first finds the grubbiest crate in the workshop, whether it be yours or somebody else’s, and packs it off with all its crate-mates to a warehouse. In its place he puts the crate containing your thing. Your thing then gets zarked and you never know that it wasn’t in the workshop all along.

13. Each player’s stock of things have the same numbers as everybody else’s. The Thing King always knows who owns what thing and whose turn it is, so you can’t ever accidentally zark somebody else’s thing even if it has the same thing number as one of yours.

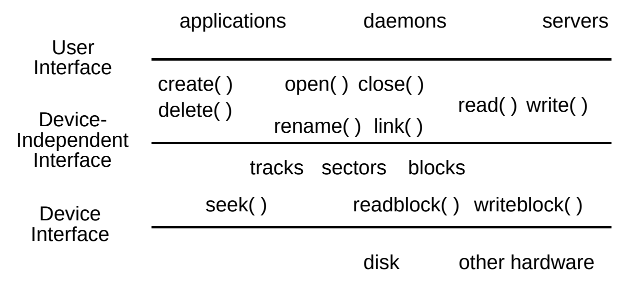
## Notes on “The Paging Game”

1. Traditionally, the Thing King sits at a large, segmented table and is attended to by pages (the so-called “table pages”) whose job it is to help the king remember where all the things are and who they belong to.
2. One consequence of Rule 13 is that everybody’s thing numbers will be similar from game to game, regardless of the number of players.
3. The Thing King has a few things of his own, some of which move back and forth between workshop and warehouse just like anybody else’s, but some of which are just too heavy to move out of the workshop.
4. With the given set of rules, oft-zarked things tend to get kept mostly in the workshop while little-zarked things stay mostly in a warehouse. This is efficient stock control.

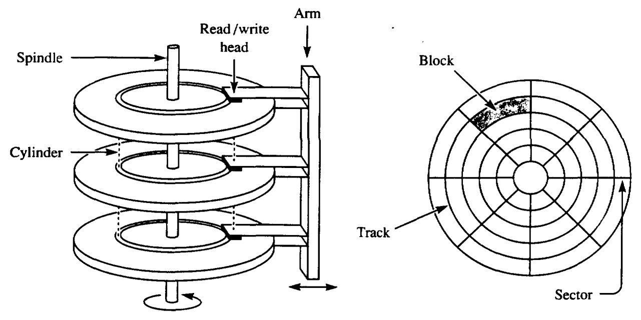
Long Live the Thing King!

## File System Abstraction

- Levels of abstraction:



- The hardware underneath:



**(a)** A hard disk drive. **(b)** A single disk.

Diagram from *Computer Science*, Volume 2, J. Stanley Warford, Heath, 1991.

## File System Issues

- Important to the user:
  - Persistence — data stays around between power cycles and crashes
  - Ease of use — can easily find, examine, modify, etc. data
  - Efficiency — uses disk space well
  - Speed — can get to data quickly
  - Protection — others can't corrupt (or sometimes even see) my data
- OS provides:
  - File system with directories and naming — allows user to specify directories and names instead of location on disk
  - Disk management — keeps track of where files are located on the disk, accesses those files quickly
  - Protection — no unauthorized access

## User Interface to the File System

- A *file* is a logical unit of storage:
  - A series of records (IBM mainframes)
  - A series of bytes (UNIX, most PCs)
  - A resource fork and data fork (Macintosh)
    - Resource fork — labels, messages, etc.
    - Data fork — code and data
- What is stored in a file?
  - C++ source code, object files, executable files, shell scripts, PostScript…
  - Macintosh OS explicitly supports file types — TEXT, PICT, etc.
  - Windows uses file naming conventions — ".exe" and ".com" for executables, etc.
  - UNIX looks at contents to determine type:
    - Shell scripts — start with "#"
    - PostScript — starts with "%!PS-Adobe…"
    - Executables — starts with *magic number*

## File Operations

- **Create**(*name*)
  - Constructs a *file descriptor* on disk to represent the newly created file
    - Adds an entry to the *directory* to associate *name* with that file descriptor
  - Allocates disk space for the file
    - Adds disk location to file descriptor
- *fileId* = **Open**(*name*, *mode*)
  - Allocates a unique identifier called the *file ID* (identifier) (returned to the user)
  - Sets the mode (r, w, rw) to control concurrent access to the file
- **Close**(*fileId*)
- **Delete**(*fileId*)
  - Deletes the file's file descriptor from the disk, and removes it from the directory

## Common File Access Patterns

- Sequential access
  - Data is processed in order, one byte at a time, always going forward
  - Most accesses are of this form
  - Example: compiler reading a source file
- Direct / random access
  - Can access any byte in the file directly, without accessing any of its predecessors
  - Example: accessing database record 12
- Keyed access
  - Can access a byte based on a *key* value
  - Example: database search, dictionary
  - OS does not support keyed access
    - User program must determine the address from the key, then use random access (provided by the OS) into the file

## File Operations (cont.)

- **Read**(*fileId*, *from*, *size*, *bufAddress*)
  - Random access read
  - Reads *size* bytes from file *fileId*, starting at position *from*, into the buffer specified by *bufAddress*

```
for (pos=from, i=0 ; i < size ; i++)
  *bufAddress[i] = file[pos++];
```

- **Read**(*fileId*, *size*, *bufAddress*)
  - Sequential access read
  - Reads *size* bytes from file *fileId*, starting at the current file position *fp*, into the buffer specified by *bufAddress*, and then increments *fp* by *size*

```
 for (pos=fp, i=0 ; i < size ; i++)
  *bufAddress[i] = file[pos++];
fp += size;
```

- **Write** — similar to **Read**

## Directories and Naming

- Directories of named files
  - User and OS must have some way to refer to files stored on the disk
  - OS wants to use numbers (index into an array of file descriptors) (efficient, etc.)
  - User wants to use textual names (readable, mnemonic, etc.)
  - OS uses a *directory* to keep track of names and corresponding file indices
- Simple naming
  - One name space for the entire disk
    - Every name must be unique
  - Implementation:
    - Store directory on disk
    - Directory contains <name, index> pairs
  - Used by early mainframes, early Macintosh OS, and MS DOS

## Directories and Naming (cont.)

- User-based naming
  - One name space for each user
    - Every name in that user's directory must be unique, but two different users can use the same name for a file in their directory
  - Used by TOPS-10 (DEC mainframe from the early 1980s)
- Multilevel naming
  - Tree-structured name space
  - Implementation:
    - Store directories on disk, just like files
    - Each directory contains <name, index> pairs in no particular order
      - The file pointed to by a directory can be another directory
        - » Names have "/" separating levels
      - Resulting structure is a tree of directories
  - Used by UNIX
    - More on UNIX disk structures next time…

## Disk Hardware

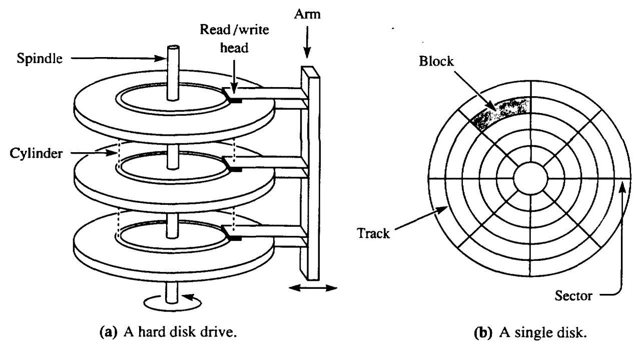


Diagram from *Computer Science*, Volume 2, J. Stanley Warford, Heath, 1991.

- Arm can move in and out
  - Read / write head can access a ring of data as the disk rotates
- Disk consists of one or more *platters*
  - Each platter is divided into rings of data, called *tracks*, and each track is divided into *sectors*
  - One particular platter, track, and sector is called a *block*

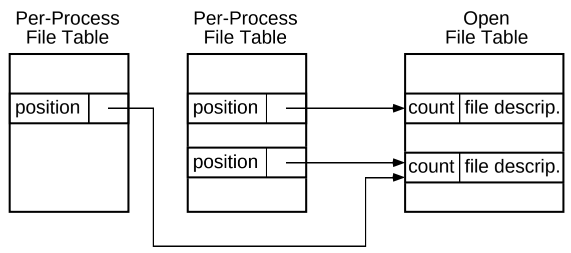
## Disk Hardware (cont.)

- Typical disk today:
  - Total capacity = 4.2GB
  - 9 platters (18 surfaces)
  - 4096 tracks per surface
  - 32-64 sectors per track
  - 1K bytes per sector
- Trends in disk technology
  - Disks are getting smaller, for similar capacity
    - Faster data transfer, lighter weight
  - Disk are storing data more densely
    - Faster data transfer
    - Density improving faster than mechanical limitations (seek time, rotational delay)
  - Disks are getting cheaper (factor of 2 per year since 1991)

## Data Structures for Files

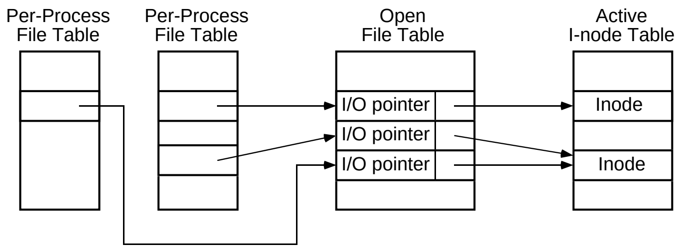
- Every file is described by a *file descriptor*, which may contain (varies with OS):
  - Type
  - Access permissions — read, write, etc.
  - Link count — number of directories that contain this file
  - Owner, group
  - Size
  - Access times — when created, last accessed, last modified
  - Blocks where file is located on disk
- Not included:
  - Name of file

## OS Data Structures for Files



- *Open file table* (one, belongs to OS)
  - Lists all open files
  - Each entry contains:
    - A *file descriptor*
    - Open count — number of processes that have the file open
- *Per-process file table* (many)
  - List all open files for that process
  - Each entry contains:
    - Pointer to entry in open file table
    - Current position (offset) in file

## UNIX Data Structures for Files



- *Active Inode table* (one, belongs to OS)
  - Lists all active *inodes* (file descriptors)
- *Open file table* (one, belongs to OS)
  - Each entry contains:
    - Pointer to entry in active inode table
    - Current position (offset) in file
- *Per-process file table* (many)
  - Each entry contains:
    - Pointer to entry in open file table

## Disk Data Structures for Files

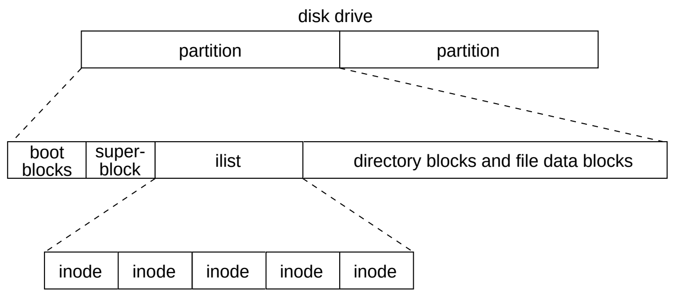
- The file descriptor information must also be stored on the disk, for persistence
  - Includes all the basic information listed on the previous slides
  - All *inodes* (file descriptors) are stored in a fixed-size array on the disk called the *ilist*
    - The size of the ilist array is determined when the disk is initialized
    - The index of a file descriptor in the array is called its *inode number*, or *inumber*
- File descriptors are stored:
  - Originally, together on the inner (or outer) track
  - Then, together on the middle track (why?)
  - Now: there are small file descriptor are spread out across the disk, so as to be closer to the file data

## UNIX File System

- A file descriptor (*inode*) represents a file
  - All *inodes* are stored on the disk in a fixed-size array called the *ilist*
    - The size of the ilist array is determined when the disk is initialized
    - The index of a file descriptor in the array is called its *inode number*, or *inumber*
  - Inodes for active files are also cached in memory in the *active inode table*
- A UNIX disk may be divided into *partitions*, each of which contains:
  - Blocks for storing directories and files
  - Blocks for storing the ilist
    - Inodes corresponding to files
    - Some special inodes
      - Boot block — code for booting the system
      - Super block — size of disk, number of free blocks, list of free blocks, size of ilist, number of free inodes in ilist, etc.

## UNIX File System (cont.)

- High-level view:



- Low-level view:

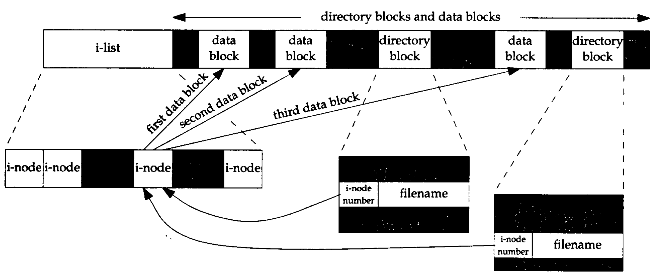
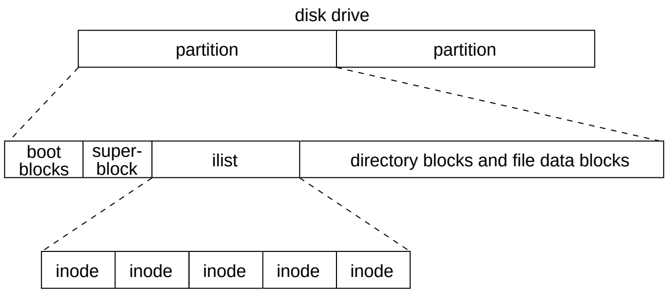


Diagram from *Advanced Programming in the UNIX Environment*, W. Richard Stevens, Addison Wesley, 1992.

## UNIX File System (Review)

- High-level view:



- Low-level view:

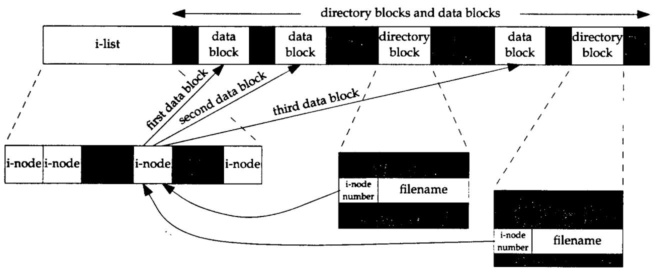
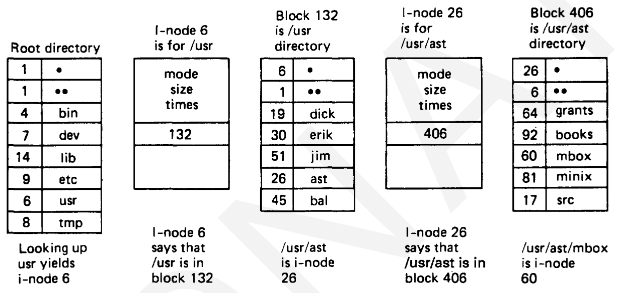


Diagram from *Advanced Programming in the UNIX Environment*, W. Richard Stevens, Addison Wesley, 1992.

## Working with Directories

- Searching a directory in UNIX:
  - If filename begins with " / ", start at root of the file system tree (inode 2)
  - If filename begins with " ~ ", start at the user's home directory
  - If filename begins with any other character, start at current working directory
- Working directories
  - A file name can be given as the full *pathname*, separating levels by " / "
  - UNIX also keeps track of the inode number of current working directory for each process; we don't have to use full names
- A UNIX directory has two special entries
  - " . " refers to the directory itself
  - " .. " refers to the parent directory

## Working with Directories (Lookup)



Root directory

| | |
|---|---|
| 1 | • |
| 1 | •• |
| 4 | bin |
| 7 | dev |
| 14 | lib |
| 9 | etc |
| 6 | usr |
| 8 | tmp |

Looking up usr yields i-node 6

I-node 6 is for /usr

| |
|---|
| mode size times |
| 132 |
| |

I-node 6 says that /usr is in block 132

Block 132 is /usr directory

| | |
|---|---|
| 6 | • |
| 1 | •• |
| 19 | dick |
| 30 | erik |
| 51 | jim |
| 26 | ast |
| 45 | bal |

/usr/ast is i-node 26

I-node 26 is for /usr/ast

| |
|---|
| mode size times |
| 406 |
| |

I-node 26 says that /usr/ast is in block 406

Block 406 is /usr/ast directory

| | |
|---|---|
| 26 | • |
| 6 | •• |
| 64 | grants |
| 92 | books |
| 60 | mbox |
| 81 | minix |
| 17 | src |

/usr/ast/mbox is i-node 60

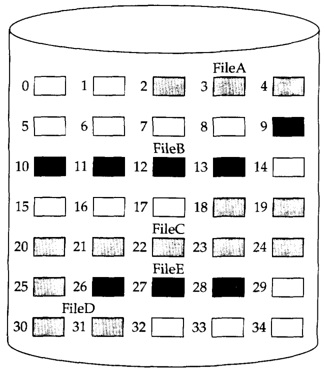
**Fig. 4-16.** The steps in looking up */usr/ast/mbox*.

- A directory is a table of entries:
  - 2 bytes — inumber
  - 14 bytes — file name (improved in BSD 4.2 and later)
- Search to find the file begins with either root, or the current working directory
  - Inode 2 points to the root directory (" / ")
  - Example above shows lookup of /usr/ast/mbox

## Working with Directories (Links)

- UNIX supports links — two directories containing the same file
  - Example: aos/nachos & os/nachos
- Hard links (" ln *target_file directory* ")
  - Specified directory refers to the target file
    - Both directories point to same inode
  - Link count in inode is used to ensure that the file is deleted only when the last directory entry referring to it is removed
- Soft / symbolic links (" ln –s *target_file directory* ")
  - Adds a pointer to the target file (or target directory) from the specified directory
    - Special bit is set in inode, and the file just contains the name of the file it's linked to
    - View symbolic links with "ls –F" and "ls –l"
  - Can link across disk drives

## Organization of Files (Contiguous Allocation)



File Allocation Table

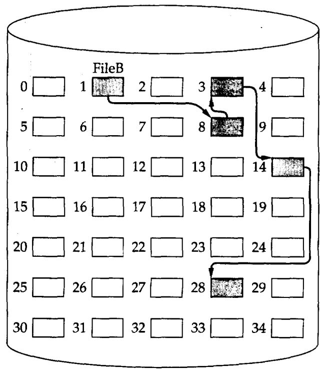
| File Name | Start Block | Length |
|---|---|---|
| FileA | 2 | 3 |
| FileB | 9 | 5 |
| FileC | 18 | 8 |
| FileD | 30 | 2 |
| FileE | 26 | 3 |

**FIGURE 11.7 Contiguous file allocation**

Diagram from *Operating Systems*, William Stallings, Prentice Hall, 1995.

- OS keeps an ordered list of free blocks
  - Allocates contiguous groups of blocks when it creates a file
  - File descriptor must store start block and length of file
- Used in IBM 370, some write-only disks

## Organization of Files (Linked / Chained Allocation)



File Allocation Table

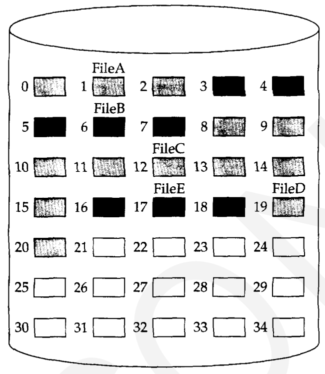
| File Name | Start Block | Length |
|---|---|---|
| ... | ... | ... |
| FileB | 1 | 5 |
| ... | ... | ... |

**FIGURE 11.9 Chained allocation**

Diagram from *Operating Systems*, William Stallings, Prentice Hall, 1995.

- OS keeps an ordered list of free blocks
  - File descriptor stores pointer to first block
  - Each block stores pointer to next block
- Used in DEC TOPS-10, Xerox Alto

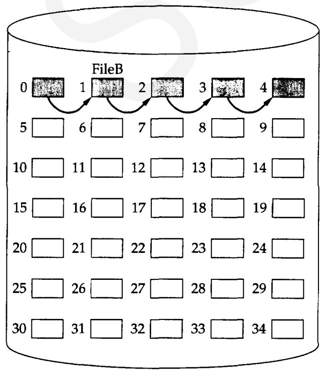
## Organization of Files (Compaction for Contiguous and Linked Allocation)



File Allocation Table

| File Name | Start Block | Length |
|---|---|---|
| FileA | 0 | 3 |
| FileB | 3 | 5 |
| FileC | 8 | 8 |
| FileD | 19 | 2 |
| FileE | 16 | 3 |

**FIGURE 11.8 Contiguous file allocation (after compaction)**



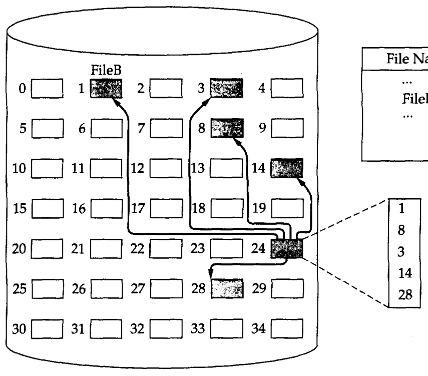
File Allocation Table

| File Name | Start Block | Length |
|---|---|---|
| ... | ... | ... |
| FileB | 0 | 5 |
| ... | ... | ... |

**FIGURE 11.10 Chained allocation (after consolidation)**

Diagrams from *Operating Systems*, William Stallings, Prentice Hall, 1995.

## Organization of Files (Indexed Allocation)



File Allocation Table

| File Name | Index Block |
|---|---|
| ... | ... |
| FileB | 24 |
| ... | ... |

**FIGURE 11.11 Indexed allocation with block portions**

Diagram from *Operating Systems*, William Stallings, Prentice Hall, 1995.

- OS keeps a list of free blocks
  - OS allocates an array (called the index block) to hold pointers to all the blocks used by the file
  - Allocates blocks only on demand
  - File descriptor points to this array
- Used in DEC VMS, Nachos

## Organization of Files (Multilevel Indexed Allocation)

- Used in UNIX (numbers below are for traditional UNIX, BSD UNIX 4.1)
- Each inode (file descriptor) contains 13 *block pointers*
  - First 10 pointers point to data blocks (each 512 bytes long) of a file
    - If the file is bigger than 10 blocks (5,120 bytes), the 11th pointer points to a *single indirect block*, which contains 128 pointers to 128 more data blocks (can support files up to 70,656 bytes)
      - If the file is bigger than that, the 12th pointer points to a *double indirect block*, which contains 128 pointers to 128 more single indirect blocks (can support files up to 8,459,264 bytes)
        - » If the file is bigger than that, the 13th pointer points to a *triple indirect block*, which contains 128 pointers to 128 more double indirect blocks
  - Max file size is 1,082,201,087 bytes

## Organization of Files (Multilevel Indexed Allocation) (cont.)

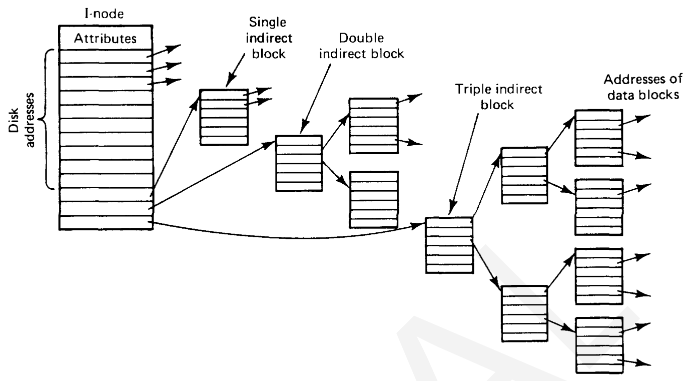


Diagram from *Modern Operating Systems*, Andrew Tanenbaum, Prentice Hall, 1992.

- BSD UNIX 4.2, 4.3:
  - Maximum block size is 4096 bytes
  - Inode contains 14 block pointers
    - 12 to data
    - 13 to single indirect block containing 1024 pointers, 14 to triple indirect block…
  - Max file size is $2^{32}$ bytes

## Disk Hardware

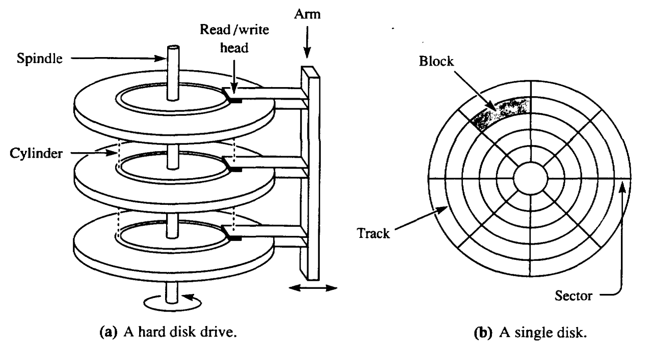


Diagram from *Computer Science*, Volume 2, J. Stanley Warford, Heath, 1991.

- *Seek time* — time required to position heads over the track / cylinder
  - Typically 10 ms to cross entire disk
- *Rotational delay* — time required for sector to rotate underneath the head
  - 120 rotations / second = 8 ms / rotation

## Disk Access Times

- From last lecture (typical numbers):
  - 32-64 sectors per track
  - 1K bytes per sector
- *Data transfer rate* is number of bytes rotating under the head per second
  - 1 KB / sector * 32 sectors / rotation * 120 rotations / second = 4 MB / s
- Disk I/O time = seek + rotational delay + transfer
  - If head is at a random place on the disk
    - Avg. seek time is 5 ms
    - Avg. rotational delay is 4 ms
    - Data transfer rate for a 1KB is 0.25 ms
    - I/O time = 9.25 ms for 1KB
    - ↪Real transfer rate is roughly 100 KB / s
  - In contrast, memory access may be 20 MB / s (200 times faster)

## Selecting the Sector Size

- The read / write head needs to synchronize with the head as it rotates
  - Need 100-1000 bits between each sector to measure how fast disk is spinning
- If sector size is 1 byte
  - Only 1% of disk holds useful data
  - 1/1000 transfer rate as before = 100 B / s
- If sector size is 1 KB
  - 90% of disk holds useful data
  - Transfer rate is 100 KB / s
- If sector size is 1 MB
  - Almost all of disk holds useful data
  - Transfer rate is 4 MB / s (full disk transfer rate — seek and rotational latency usually won't matter anymore)

## Evolution of UNIX Disk Management

- In traditional UNIX, and Berkeley BSD 3.0 UNIX
  - Disk lock size was 512 bytes
- In Berkeley BSD 4.0 UNIX:
  - Block size was changed to 1024 bytes
  - More or less doubled performance
    - Each block access fetched twice as much data, so there was less disk seek overhead
    - More files could use only the direct block of the inode, which saved further space
  - When file system was first created
    - Free list was ordered, and they got transfer rates up to 175 KB / s
    - After a few weeks, data and free blocks got so randomized that the transfer rate went down to 30 KB / s
    - This was less than 4% of the maximum transfer rate!

## Evolution of UNIX Disk Management (cont.)

- What about making the blocks bigger?
  - Causes internal fragmentation
  - Most files are small, maybe one block
- Some measurements from a file system at UC Berkeley:

| Organization | Space used | Waste |
|---|---|---|
| Data only | 775.2 | 0% |
| +inodes, 512B block | 828.7 | 6.9% |
| +inodes, 1KB block | 866.5 | 11.8% |
| +inodes, 2KB block | 948.5 | 22.4% |
| +inodes, 4KB block | 1128.3 | 45.6% |

- The presence of small files kills the performance for large files!
  - Want big blocks to reduce the seek overhead for big files
  - But… big blocks increase fragmentation for small files

## Evolution of UNIX Disk Management (cont.)

- In Berkeley BSD 4.2 UNIX:
  - See “A Fast File System for UNIX” on class home page for details
  - Introduced concept of a *cylinder group*
    - A *cylinder* is the set of corresponding tracks on all the disk surfaces
      - For a given head position, it’s just as easy to access one track in the cylinder as it is to access any other
      - A cylinder group is a set of adjacent cylinders
    - Each cylinder group has a copy of super block, bit map of free blocks, ilist, and blocks for storing directories and files
    - The OS tries to put related information together into the same cylinder group
      - Try to put all inodes in a directory in the same cylinder group
      - Try to put blocks for one file contiguously in the same cylinder group
        - » Bitmap of free blocks makes this easy
      - For long files, redirect each megabyte to a new cylinder group

## Evolution of UNIX Disk Management (cont.)

- In Berkeley BSD 4.2 UNIX: (cont.)
  - Block size was changed to 4096 bytes
    - Reduced fragmentation as follows:
      - Each disk block can be used in its entirety, or can be broken up into 2, 4, or 8 *fragments*
      - For most of the blocks in the file, use the full block
      - For the last block in the file, use as small a fragment as possible
      - Can get as many as 8 very small files in one disk block
    - This change resulted in
      - Only as much fragmentation as a 1KB block size (w/ 4 fragments)
      - Data transfer rates that were 47% of the maximum rate
  - Other improvements:
    - Bit map instead of unordered free list (easier to keep files contiguous)
    - Variable length file names, symbolic links
    - File locking, disk quotas

## Improving Performance with Good Block Management

- OS usually keeps track of free blocks on the disk using a *bit map*
  - A bit map is just an array of bits
    - 1 means the block is free,
    - 0 means the block is allocated to a file
  - For a 1.2 GB drive, there are about 307,000 4KB blocks, so a bit map takes up 38.4 KB (usually kept in memory)
- Try to allocate the next block of the file close to the previous block
  - Works well if disk isn’t full
  - If disk is full, this is doesn’t work well
    - Solution — keep some space (about 10% of the disk) in reserve, and don’t tell users; never let disk get more than 90% full
    - With multiple platters / surfaces, there are many possibilities (one surface is as good as another), so the block can usually be allocated close to the previous one

## Improving Performance Using a Disk Cache

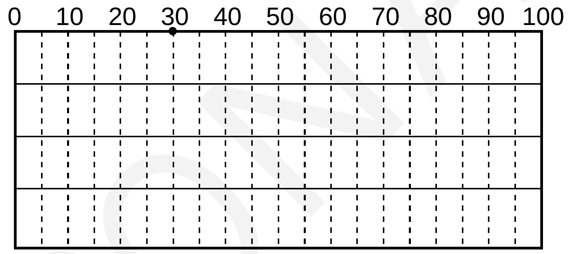
- Have OS (not hardware) manage a *disk block cache* to improve performance
  - Use part of main memory as a cache
  - When OS reads a file from disk, it copies those blocks into the cache
  - Before OS reads a file from disk, it first checks the cache to see if any of the blocks are there (if so, uses cached copy)
- Replacement policies for the blocks:
  - Same options as paging
    - FIFO, LRU using clock / second chance
  - Easy to implement exact LRU
    - OS just records time along with everything else it has to update when a block is read
  - But — sequential access degrades LRU
    - Solution: free-behind policy for large sequentially-accessed files — as block is read, remove previous one from cache

## Improving Performance with Disk Head Scheduling

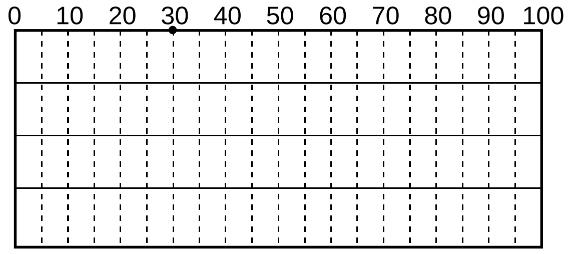
- Permute the order of the disk requests
  - From the order that they arrive in
  - Into an order that reduces the *distance* of seeks
- Examples:
  - Head just moved from lower-numbered track to get to track 30
  - Request queue: 61, 40, 18, 78
- Algorithms:
  - First-come first-served (FCFS)
  - Shortest Seek Time First (SSTF)
  - SCAN (0 to 100, 100 to 0, …)
  - C-SCAN (0 to 100, 0 to 100, …)

## Disk Head Scheduling (cont.)

- FCFS (used in Nachos)
  - Handle in order of arrival

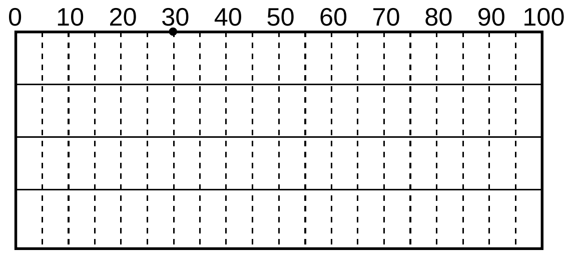


- SSTF
  - Select request that requires the smallest seek from current track

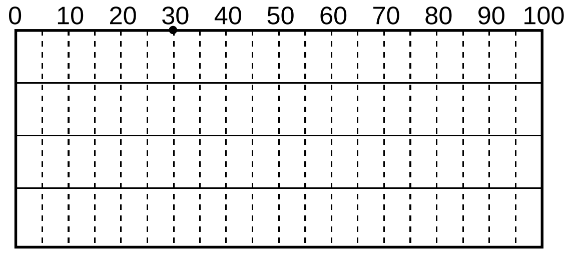


## Disk Head Scheduling (cont.)

- SCAN (Elevator algorithm)
  - Move the head 0 to 100, 100 to 0, picking up requests as it goes

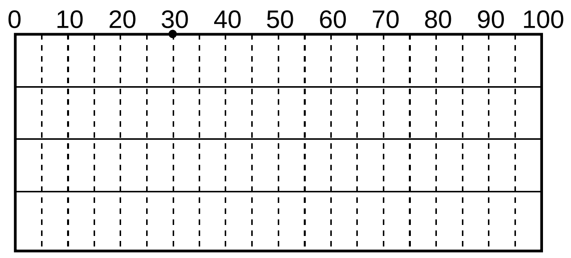


- LOOK (variation on SCAN)
  - Don’t go to end unless necessary

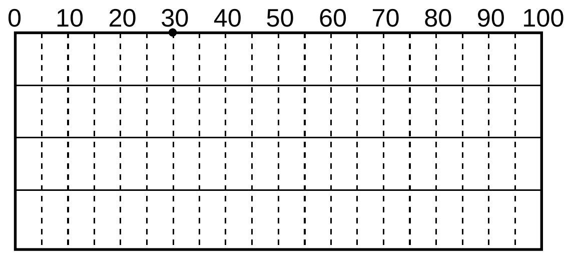


## Disk Head Scheduling (cont.)

- C-SCAN (Circular SCAN)
  - Move the head 0 to 100, picking up requests as it goes, then big seek to 0



- C-LOOK (variation on C-SCAN)
  - Don't go to end unless necessary



## Improving Disk Performance

- Keep some structures in memory
  - Active inodes, file tables
- Efficient free space management
  - Bitmaps
- Careful allocation of disk blocks
  - Contiguous allocation where possible
  - Direct / indirect blocks
  - Good choice of block size
  - Cylinder groups
  - Keep some disk space in reserve
- Disk management
  - Cache of disk blocks
  - Disk scheduling

## Disk Management

- Disk formatting
  - Physical formatting — dividing disk into sectors: header, data area, trailer
  - Most disks are preformatted, although special utilities can reformat them
  - After formatting, must partition the disk, then write the data structures for the file system (logical formatting)
- *Boot block* contains the "bootstrap" program for the computer
  - System also contains a ROM with a bootstrap loader that loads this program
- Disk system should ignore bad blocks
  - When disk is formatted, a scan detects bad blocks and tells disk system not to assign those blocks to files
  - Disk may also do this as disk is used

## Disk Management (cont.)

- Swap space management
  - Swap space in normal file system
  - Swap space in separate partition
    - One big file — don't need whole file system, directories, etc.
    - Only need manager to allocate/deallocate blocks (optimized for speed)
- Disk reliability
  - Data normally assumed to be persistent
  - Disk striping — data broken into blocks, successive blocks stored on separate drives
  - Mirroring — keep a "shadow" or "mirror" copy of the entire disk
  - Stable storage — data is never lost during an update — maintain two physical blocks for each logical block, and both must be same for a write to be successful