CS 370: OPERATING SYSTEMS

[COMPREHENSIVE REVIEW]

Computer Science
Colorado State University

Instructor: Louis-Noel Pouchet
Spring 2024

** Lecture slides created by: SHRIDEEP PALICKARA
Topics covered in this lecture

- Introduction on Operating Systems
- Processes
- Inter-Process Communications
- Threads
- Process Synchronization and Atomic Transactions
- CPU scheduling algorithms
- Deadlocks
- Memory management
- Virtual memory
- Virtualization
- File systems
Disclaimer and preparation for the final

- This slide set is meant to guide you in your preparation, but it does not mean other lecture slides omitted here are useless! Return to each lecture as needed to polish your understanding of concepts.

- Your final will be 2h duration, taken online via Canvas+respondus, and will close Monday May 6th at 11:59pm (end time to complete the exam). It will open Sunday May 5th at 00:01am. Official final date is Monday morning, but I open for a longer period of time to accommodate people with jobs, etc.

- All objectives listed for each module will be evaluated with at least one question, with a majority of points on checking you have achieved the learning objectives, and a minority of points on more advanced questions checking your full understanding of some specific concepts. These advanced questions will only cover lectures/content taught after the spring break.
Introduction on Operating Systems

Objectives:

- Summarize basic operating systems concepts
- Highlight key developments in the history of operating systems
A modern computer is a complex system

- Multiple processors
- Main memory and Disks
- Keyboard, Mouse and Displays
- Network interfaces
- I/O devices
Why do we need Operating Systems?

- If every programmer had to understand how all these components work?
  - Software development would be arduous

- Managing all components and using them optimally is a challenge
Computers are equipped with a layer of software

- Called the **Operating System**

- Functionality:
  - Provide user programs with a better, simpler, cleaner model of the computer
  - Manage resources efficiently
Where the operating system fits in

User interface Program

Web browser  E-mail reader  Music Player

User mode

Kernel mode

Operating System

Bare Hardware
Where the operating system fits in

- The OS runs on bare hardware in **kernel mode**
  - Complete access to all hardware
  - Can execute *any* instruction that the machine is capable of executing

- Provides the base for all software
  - Rest of the software runs in **user-mode**
    - Only a **subset** of machine instructions is available
The OS controls hardware and coordinates its use among various programs.
Kernel and user modes

- Everything running in kernel mode is part of the OS

- But some programs running outside it are part of it or at least closely associated with it
Operating systems tend to be huge, complex and long-lived

- **Source code of an OS like Linux or Windows?**
  - Order of 5 million lines of code (for kernel)
    - 50 lines page, 1000 pages/volume = 100 volumes

- **Application programs such as GUI, libraries and application software?**
  - 10-20 times that
Why do operating systems live for a long time?

- Hard to write and folks are loath to throw it out

- Typically **evolve** over long periods of time
  - Windows 95/98/Me is one OS
  - Windows NT/2000/XP/Vista/7/8 is another
  - System V, Solaris, BSD derived from original UNIX
  - Linux is a fresh code base
    - Closely modeled on UNIX and highly compatible with it
  - Apple OS X based on XNU (X is not Unix) which is based on the Mach microkernel and BSD’s POSIX API
An operating system performs two unrelated functions

- Providing application programmers a clean **abstract** set of resources
  - Instead of messy hardware ones

- **Managing** hardware resources
The OS as an extended machine

- The **architecture** of a computer includes:
  - Instruction set, memory organization, I/O, and bus structure

- The architecture of most computers at the machine language level:
  - Primitive and awkward to program especially for I/O
Main memory is generally the only large storage device the CPU deals with

- To execute a program, it must be mapped to absolute addresses and loaded into memory

- Execution involves accesses to instructions and data from memory
  - By generating absolute addresses

- When program terminates, memory space is reclaimed
Virtual memory allows processes not completely memory resident to execute

- Enables us to run programs that are larger than the actual physical memory
- Separates logical memory as viewed by user from physical memory
- Frees programmers from memory storage limitations
Program Construct:  
Asynchronous operation

- Events happen at unpredictable times AND in unpredictable order.
  - Interrupts from peripheral devices
  - For e.g. keystrokes and printer data
- To be correct, a program must work will all possible timings
- Timing errors are very hard to repeat
Program Construct: Concurrency

- Sharing resources in the same *time frame*
- Interleaved execution
- Major task of modern OS is *concurrency control*
- Bugs are hard to reproduce, and produce unexpected side effects
Concurrency occurs at the hardware level because devices operate at the same time

- Interrupt: **Electrical signal** generated by a peripheral device to set hardware flag on CPU
- Interrupt detection is part of instruction cycle
- If interrupt detected
  - **Save current** value of program counter
  - **Load new** value that is address of interrupt service routine or interrupt handler: device drivers
    - Drivers use signals (software) to notify processes
Signal is the software notification of an event

- Often a *response* of the OS to an interrupt
  - OS uses signals to notify processes of completed I/O operations or errors

- Signal generated when event that causes signal occurs
  - For example: keystroke and Ctrl-C

- A process catches a signal by executing handlers for the signal
Concurrency constructs: I/O operations

- Coordinate resources so that CPU is not idle
- Blocking I/O blocks the progress of a process
- Asynchronous I/O (dedicated) threads circumvent this problem
- Ex: Application monitors 2 network channels
  - If application is blocked waiting for input from one source, it cannot respond to input on 2nd channel
Concurrenty constructs: Processes & threads

- User can create multiple processes; `fork()` in UNIX
- Inter process communications
  - Common ancestor: pipes
  - No common ancestor: signals, semaphores, shared address spaces, or messages
- Multiple threads within process = concurrency
Trend: going multi-core for CPUs

- Driven by power / physics
- Problem: parallelism in the application?
- We merely see 16-core CPUs as HEDT in 2024

Grabbed from DoE Scidac
Multiprogramming organizes jobs so that the CPU always has one to execute

- A single program (generally) cannot keep CPU & I/O devices busy at all times
- A user frequently runs multiple programs
- When a job needs to **wait**, the CPU **switches** to another job.
- Utilizes resources (cpu, memory, peripheral devices) effectively.
Time sharing is a logical extension of the multiprogramming model

- CPU switches between jobs **frequently**, users can interact with programs
- Time shared OS allows many users to use computer simultaneously
- Each action in a time shared OS tends to be **short**
  - CPU time needed for each user is small
Processes

Objectives:

- Contrast programs and processes
- Explain the memory layout of processes
- Describe Process Control Blocks
- Explain the notion of Interrupts and Context Switches
- Describe process groups
A process is just an instance of an executing program

- Conceptually each process has its own **virtual CPU**
- In reality, the CPU switches back-and-forth from process to process
- Processes are not affected by the multiprogramming
  - Or *relative speeds* of different processes
An example scenario: 4 processes

Four Program Counters

4 processes in memory

A
B
C
D

A
B
C
D

4 processes in memory
Example scenario: 4 processes

- At any instant **only one** process executes
- **Viewed over a long time**, all processes have made **progress**
Programs and processes

- Programs are **passive**, processes are **active**

- The difference between a program and a process is subtle, but crucial
Key concepts

- Process is an **activity** of some kind; it has a
  - Program
  - Input and Output
  - State

- Single processor may be shared among several processes
  - **Scheduling algorithm** decides when to stop work on one, and start work on another
Key concepts

- Process is an **activity** of some kind; it has a
  - Program
  - Input and Output
  - State

- Single processor may be shared among several processes
  - **Scheduling algorithm** decides when to stop work on one, and start work on another
How a program becomes a process

- When a program is executed, the OS copies the program image into main memory
- Allocation of memory is not enough to make a program into a process
- Must have a process ID
- OS tracks IDs and process states to orchestrate system resources
A process in memory

- **Stack**: Function parameters, return addresses, and local variables
- **Heap**: Memory allocated dynamically during runtime
- **Data**: Global variables
- **Text**: Program code
Program in memory (I)

- Program image *appears* to occupy *contiguous* blocks of memory
- OS *maps* programs into non-contiguous blocks
Program in memory (II)

- Mapping divides the program into equal-sized pieces: pages
- OS loads pages into memory
- When processor references memory on page
  - OS looks up page in table, and loads into memory
Advantages of the mapping process

- Allows large logical address space for stack and heap
  - **No physical memory used** unless actually needed

- OS hides the mapping process
  - Programmer views program image as *logically contiguous*
  - Some pages may not reside in memory
Finite State Machine

- An initial state
- A set of possible input events
- A finite number of states
- Transitions between these states
- Actions
Process state transition diagram: When a process executes it changes state

- **new**
- **ready**
- **running**
- **waiting**
- **terminated**

Transition arrows:
- interrupt
- exit
- admitted
- scheduler dispatch
- I/O or event completion
- I/O or event wait
Each process is represented by a process control block (PCB)

- process state
- process number
- program counter
- registers
- memory limits
- list of open files

PCB is a repository for any information that varies from process to process.
An example of CPU switching between processes

Process A  Operating System  Process B

Save state into PCB$_{A}$

Reload state from PCB$_{B}$

Save state into PCB$_{B}$

Reload state from PCB$_{A}$

idle

idle

idle
Scheduling Queues

- Job Queue: Contains all processes
  - A newly created process enters here first

- Ready Queue
  - Processes residing in main memory
  - Ready and waiting to execute
  - Typically a linked list

- Device Queue
  - Processes waiting for a particular I/O device
Process scheduling

- Ready queue
- CPU
- I/O
- I/O Queue
- I/O request
- Time slice expired
- Fork a child
- Wait for an interrupt
- child executes
- interrupt occurs
Interrupts and Contexts

- Interrupt causes the OS to **change** CPU from its current task to run a kernel routine.

- Save current context so that **suspend** and **resume** are possible.

- Context is represented in the **PCB**:
  - Value of CPU registers
  - Process state
  - Memory management information
Context switch refers to switching from one process to another

1. **Save** state of current process

2. **Restore** state of a different process

- Context switch time is pure **overhead**
  - No useful work done while switching
Example: Process tree in Solaris

```
Sched
  pid=0
    init
      pid=1
        inetd
          telnet
            csh
              chrome
              emacs
        dtlogin
          Xsession
            sdt_shel
              csh
                ls
                cat
    pageout
      pid=2
    fsflush
      pid=3
```
Processes in \texttt{UNIX}

- \texttt{init}: Root parent process for all user processes

- Get a listing of processes with \texttt{ps} command
  - \texttt{ps}: List of all processes associated with user
  - \texttt{ps -a}: List of all processes associated with terminals
  - \texttt{ps -A}: List of all active processes
Resource sharing between a process and its subprocess

- Child process may obtain resources **directly from OS**

- Child may be **constrained** to a subset of parent’s resources
  - Prevents any process from overloading system

- Parent process also passes along initialization data to the child
  - Physical and logical resources
Parent/Child processes:

Execution possibilities

- Parent executes *concurrently* with children
- Parent *waits* until some or all of its children terminate
Parent/Child processes:
Address space possibilities

- Child is a **duplicate** of the parent
  - Same program and data as parent

- Child has a **new program** loaded into it
Process creation in UNIX

- Process created using `fork()`
  - `fork()` copies parent’s memory image
  - Includes copy of parent’s address space

- Parent and child continue execution **at instruction after** `fork()`
  - Child: Return code for `fork()` is 0
  - Parent: Return code for `fork()` is the **non-ZERO process-ID** of new child
`fork()` results in the creation of 2 distinct programs

Parent

```
...  
...  
id = fork()  
...  
```

Child

```
...  
...  
id = fork()  
...  
```

Child will execute from here

`id = xyz here`

`id = 0 here`
A parent can move itself from off the ready queue and await child’s termination

- Done using the `wait()` system call.
- When child process completes, parent process resumes

```
return value = Non-ZERO child PID
```

```
return value = ZERO
```
wait/waitpid allows caller to suspend execution till a child’s status is available

- Process status availability
  - Most commonly after termination
  - Also available if process is stopped

- `waitpid(pid, *stat_loc, options)`
  - `pid== -1` : any child
  - `pid > 0` : specific child
  - `pid == 0` : any child in the same process group
  - `pid < -1` : any child in process group `abs(pid)`
Process groups

- Process group is a *collection* of processes
- Each process has a **process group ID**
- Process group leader?
  - Process with `pid==pgid`
- `kill` treats negative `pid` as `pgid`
  - Sends signal to all constituent processes
Process Group IDs:
When a child is created with \texttt{fork()}

1. Inherit parent’s process group ID

2. Parent can change group ID of child by using \texttt{setpgid}

3. Child can give itself new process group ID
   - Set process group ID = its process ID
Process groups

- It can contain processes which are:
  1. Parent (and further ancestors)
  2. Siblings
  3. Children (and further descendants)

- A process can only send signals to members of its process group
Example: Process tree in Solaris
Windows has no concept of a process hierarchy

- The only hint of a hierarchy?
  - When a process is created, parent is given a special *token* (called *handle*)
    - Use this to *control* the child

- However, parent is free to *pass* this token to some other process
  - *Invalidates* hierarchy
Windows has no concept of a process hierarchy

- The only hint of a hierarchy?
  - When a process is created, parent is given a special *token* (called *handle*)
    - Use this to *control* the child

- However, parent is free to *pass* this token to some other process
  - *Invalidates* hierarchy
Process terminations

- **Normal exit (voluntary)**
  - E.g. successful compilation of a program

- **Error exit (voluntary)**
  - E.g. trying to compile a file that does not exist
Process terminations

- Fatal error (involuntary)
  - Program bug
    - Referencing non-existing memory, dividing by zero, etc

- Killed by another process (involuntary)
  - Execute system call telling OS to kill some other process
  - *Killer* must be authorized to do the *killing of the killee*
  - Unix: `kill`  Win32: `TerminateProcess`
Process terminations:
This can be either normal or abnormal

- OS deallocates the process resources
  - Cancel pending timers and signals
  - Release virtual memory resources and locks
  - Close any open files
- Updates statistics
  - Process status and resource usage
- Notifies parent in response to a wait()
On termination a UNIX process DOES NOT fully release resources until a parent execute a wait() for it

- When the parent is not waiting when the child terminates?
  - The process becomes a **zombie**

- Zombie is an *inactive* process
  - Still has an entry in the process table
  - But is already dead, so cannot be killed easily!! 😊

- Zombie processes often come from error in programming: not properly waiting on all children created, changing the parent while children still active, etc.
Zombies and termination

- When a process terminates, its orphaned children and are adopted by a special process
  - This special system process is init

- Some more about the special process init
  1. Has a pid of 1
  2. Periodically executes wait() for children
  3. Children without a parent are adopted by init
     - Zombie processes are adopted by init after killing their parent, then cleaned by the periodic wait()
Normal termination of processes

- Return from `main`
- Implicit return from `main`
  - Function **falls off the end**
- Call to `exit`, `_Exit` or `_exit`
Protection and Security

- Control access to system resources
  - Improve reliability

- Defend against use (misuse) by unauthorized or incompetent users

- Examples
  - Ensure process executes within its own space
  - Force processes to relinquish control of CPU
  - Device-control registers accessible only to the OS
Inter-Process Communications

Objectives:

- Explain inter-process communications based on Shared Memory
- Explain inter-process communications based on Pipes
- Explain inter-process communications based on message passing
- Contrast inter-process communications based on shared memory, pipes, and message passing
- Design programs that implement inter-process communications
Independent and Cooperating processes

- Independent: **CANNOT** affect or be affected by other processes
- Cooperating: **CAN** affect or be affected by other processes
Why have cooperating processes?

- Information sharing: shared files
- Computational speedup
  - Sub tasks for concurrency
- Modularity
- Convenience: Do multiple things in parallel
- Privilege separation
- Etc.
Cooperating processes need IPC to exchange data and information

- **Shared memory**
  - Establish memory region to be shared
  - Read and write to the shared region

- **Message passing**
  - Communications through message exchange
Contrasting the two IPC approaches

**Easier to implement**
- Best for **small** amounts of data
- **Kernel intervention** for communications

**Maximum speed**
- System calls to **establish** shared memory
Shared memory systems

- Shared memory resides in the address space of process creating it.
- Other processes must attach segment to their address space.
IPC: Use of the created shared memory

- Once shared memory is attached to the process’s address space
  - Routine memory accesses using * from shmat()
    - Write to it
      - `sprintf(shared_memory, “Hello”);`
    - Print string from memory
      - `printf(“*%\n”, shared_memory);`

- **RULE:** First attach, and then access
IPC Shared Memory:  
What to do when you are done

① **Detach** from the address space.  
- `shmdt()` : **SHared Memory DeTTach**  
- `shmdt(shared_memory)`

② **To remove** a shared memory segment  
- `shmctl()` : **SHared Memory ConTroL operation**  
  - Specify the segment ID to be removed  
  - Specify operation to be performed: **IPC_RMID**  
  - Pointer to the shared memory region
Message Passing: Communicate and synchronize actions without sharing the same address space

- Useful in distributed environments (e.g., Message Passing Interface)
- Two main operations
  - send(message)
  - receive(message)
- Message sizes can be:
  - Fixed: Easy
  - Variable: Little more effort
Communications between processes

- There needs to be a communication link
- Underlying physical implementation
  - Shared memory
  - Hardware bus
  - Network
Aspects to consider for IPC

① Communications
   - Direct or indirect

② Synchronization
   - Synchronous or asynchronous

③ Buffering
   - Automatic or explicit buffering
Naming allows processes to refer to each other

- Processes use each other’s identity to communicate
- Communications can be
  - Direct
  - Indirect
Direct Communications: Addressing

- **Symmetric addressing**
  - send(P, message)
  - receive(Q, message)

- **Asymmetric addressing**
  - send(P, message)
  - receive(id, message)
  - Variable id set to name of the sending process

Explicitly name recipient and sender of message

Only sender names recipient
Recipient does not
Direct Communications: Disadvantages

- **Limited modularity** of process definitions

- **Cascading effects** of changing the identifier of process
  - Examine all other process definitions
Indirect communications: Message sent and received from mailboxes (ports)

- Each **mailbox** has a unique identification & owner
  - POSIX message queues use **integers** to identify mailboxes

- Processes communicate **only** if they have **shared mailbox**
  - `send(A, message)`
  - `receive(A, message)`
Indirect communications

- **Processes** P1, P2 and P3 share mailbox A
  - P1 sends a message to A
  - P2, P3 execute a receive() from A

- **Possibilities? Allow ...**
  1. Link to be associated with at most 2 processes
  2. At most 1 process to execute receive() at a time
  3. System to arbitrarily select who gets message
Mailbox ownership: Owned by OS

- Mailbox has its own existence

- Mailbox is independent
  - Not attached to any process

- OS must allow processes to
  - Create mailbox
  - Send and receive *through* the mailbox
  - Delete mailbox
Message passing: Synchronization issues
Options for implementing primitives

- Blocking send
  - Block \textit{until} received by process or mailbox

- Nonblocking send
  - Send and \textit{promptly resume} other operations

- Blocking receive
  - Block \textit{until} message available

- Nonblocking receive
  - Retrieve \textit{valid} message or \textit{null}

- Producer-Consumer problem: Easy with blocking
Communicate and synchronize actions without sharing the same address space

- Useful in distributed environments (e.g., Message Passing Interface)

- Two main operations
  - `send(message)`
  - `receive(message)`

- Message sizes can be:
  - Fixed: Easy
  - Variable: Little more effort
Communications between processes

- There needs to be a communication link
- Underlying physical implementation
  - Shared memory
  - Hardware bus
  - Network
Aspects to consider for IPC

① **Communications**
- Direct or indirect

② **Synchronization**
- Synchronous or asynchronous

③ **Buffering**
- Automatic or explicit buffering
Naming allows processes to refer to each other

- Processes use each other’s identity to communicate
- Communications can be
  - Direct
  - Indirect
Direct communications

- Explicitly name recipient or sender
- Link is established automatically
  - Exactly one link between the 2 processes
- Addressing
  - Symmetric
  - Asymmetric
Direct Communications: Addressing

- **Symmetric addressing**
  - send(P, message)
  - receive(Q, message)

- **Asymmetric addressing**
  - send(P, message)
  - receive(id, message)
    - Variable id set to name of the sending process

Explicitly name recipient and sender of message

Only sender names recipient
Recipient does not
Direct Communications: Disadvantages

- **Limited modularity** of process definitions

- **Cascading effects** of changing the identifier of process
  - Examine all other process definitions
Indirect communications: Message sent and received from mailboxes (ports)

- Each **mailbox** has a unique identification & owner
  - POSIX message queues use integers to identify mailboxes

- Processes communicate *only* if they have **shared mailbox**
  - `send(A, message)`
  - `receive(A, message)`
Indirect communications: Link properties

- Link established only if both members share mailbox
- Link may be associated with more than two processes
Indirect communications

- Processes P1, P2 and P3 share mailbox A
  - P1 sends a message to A
  - P2, P3 execute a receive() from A

- Possibilities? Allow …
  1. Link to be associated with at most 2 processes
  2. At most 1 process to execute receive() at a time
  3. System to arbitrarily select who gets message
Mailbox ownership: Owned by OS

- Mailbox has its own existence

- Mailbox is **independent**
  - Not attached to any process

- OS must allow processes to
  - Create mailbox
  - Send and receive *through* the mailbox
  - Delete mailbox
Message passing: Synchronization issues
Options for implementing primitives

- Blocking send
  - Block *until* received by process or mailbox

- Nonblocking send
  - Send and *promptly resume* other operations

- Blocking receive
  - Block *until* message available

- Nonblocking receive
  - Retrieve *valid* message or *null*

- Producer-Consumer problem: Easy with blocking
Threads

Objectives:

- Explain differences between processes and threads
- Compare multithreading models
- Contrast differences between user and kernel threads
- Relate dominant threading libraries: POSIX, Win32, and Java
- Design threaded programs that can synchronize their actions
What are threads?

- Miniprocesses or lightweight processes
- Why would anyone want to have a kind of process within a process?
The main reason for using threads

- In many applications *multiple activities* are going on at once
  - Some of these may block from time to time

- Decompose application into multiple sequential threads
  - Running in *quasi-parallel*
Isn’t this precisely the argument for processes?

- Yes, but there is a new dimension …

- Threads have the ability to **share the address space** (and all of its data) among themselves

- For several applications
  - Processes (with their *separate* address spaces) don’t work
Threads are also lighter weight than processes

- **Faster** to create and destroy than processes
- In many systems thread creation is 10-100 times faster
- When number of threads needed changes dynamically and rapidly?
  - Lightweight property is very useful
Threads:
The performance argument

- When all threads are CPU bound all the time?
  - Additional threads would likely yield **no** performance gain

- But when there is substantial computing *and substantial I/O*
  - Having threads allows activities to **overlap**
  - Speeds up the application possibly
User-level threads: Overview

Kernel

Runtime System

Process table

Thread table

User space

Kernel space

Process

Thread

Runtime System

CS370: Operating Systems
Dept. Of Computer Science, Colorado State University
User threads are invisible to the kernel and have low overhead

- **Compete among themselves** for resources allocated to their encapsulating process
- Scheduled by a *thread runtime* system that is **part** of the process code
- Programs link to a special library
  - Each library function is enclosed by a **jacket**
  - Jacket function calls thread runtime to do thread management
    - Before (and possibly after) calling jacketed library function.
User level thread libraries: Managing blocking calls

- **Replace** potentially blocking calls with non-blocking ones
- If a call does not block, the runtime invokes it
- If the call *may block*
  1. Place thread on a list of *waiting* threads
  2. Add call to list of actions to *try later*
  3. Pick another thread to run
- ALL control is **invisible** to user and OS
Disadvantages of the user level threads model (1)

- Assumes that the runtime will eventually regain control, this is thwarted by:
  - CPU bound threads
  - Thread that rarely perform library calls …
    - Runtime can’t regain control to schedule other threads
- Programmer must avoid lockout situations
  - Force CPU-bound thread to yield control
Disadvantages of the user level threads model (2)

- Can only share processor resources allocated to encapsulating process
  - **Limits** available parallelism
Kernel-level threads: Overview
Kernel threads

- Kernel is aware of kernel-level threads as **schedulable entities**
  - Kernel maintains a thread table to keep track of all threads in the system

- **Compete systemwide** for processor resources
  - Can take advantage of multiple processors
Kernel threads:
Management costs

- **Scheduling** is *almost as expensive* as processes
  - **Synchronization** and data sharing *less expensive* than processes

- More expensive to manage than user-level threads
Hybrid thread models

- Write programs in terms of user-level threads
- Specify number of schedulable entities associated with process
  - Mapping at runtime to achieve parallelism
- Level of user-control over mapping
  - Implementation dependent
The Many-to-One threading model

User threads

Kernel thread
Many-to-One Model maps many user level threads to 1 kernel thread

- Thread management done by thread library in user-space

- What happens when one thread makes a *blocking system call*?
  - The entire process blocks!
Many-to-One Model maps many user level threads to 1 kernel thread

- Only 1 thread can access kernel at a time
  - Multiple threads *unable* to run in parallel on multi-processor/core system

- E.g.: Solaris Green threads, GNU Portable threads
The One-to-One threading model
One-to-One Model:
Maps each user thread to a kernel thread

- More **concurrency**
  - Another thread can continue to run, when a thread invokes a blocking system call

- Threads run in **parallel** on multiprocessors
One-to-One Model:
Maps each user thread to a kernel thread

- **Disadvantages:**
  - There is an *overhead* for kernel thread creation
    - Multiple user threads can degrade application performance
  - Uses more kernel threads so uses more resources

- **Supported by:**
  - Linux
  - Windows family: NT/XP/2000
  - Solaris 9 and up
Many-to-Many threading Model:
2-level is a variant of this
Many-to-Many model

- **Multiplex** many user-level threads on a smaller number of kernel threads

- Number of kernel threads may be specific to
  - Particular application
  - Particular machine

- Supported in
  - IRIX, HP-US, and Solaris (prior to version 9)
A comparison of the three models

<table>
<thead>
<tr>
<th></th>
<th>Many-to-one</th>
<th>One-to-One</th>
<th>Many-to-Many</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kernel Concurrency</strong></td>
<td>NO</td>
<td>YES if many threads</td>
<td>YES</td>
</tr>
<tr>
<td><strong>During blocking system call?</strong></td>
<td>Process Blocks</td>
<td>Process DOES NOT block if other threads</td>
<td>Process DOES NOT block</td>
</tr>
<tr>
<td><strong>Kernel thread creation</strong></td>
<td>Kernel thread already exists</td>
<td>Kernel thread creation overhead</td>
<td>Kernel threads available</td>
</tr>
<tr>
<td><strong>Caveat</strong></td>
<td>Use system calls (blocking) with care</td>
<td>Don’t create too many threads to not use too much resources</td>
<td></td>
</tr>
</tbody>
</table>
Thread libraries provide an API for creating and managing threads

<table>
<thead>
<tr>
<th>Library code and data structures</th>
<th>User level library</th>
<th>Kernel level library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reside in user space</td>
<td>Reside in kernel space</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread creation requires a system call?</th>
<th>User level library</th>
<th>Kernel level library</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>YES</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OS/Kernel support</th>
<th>User level library</th>
<th>Kernel level library</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>YES</td>
<td></td>
</tr>
</tbody>
</table>
Dominant thread libraries (1)

- POSIX pthreads
  - Extends POSIX standard (IEEE 1003.1c)
  - Provided as user- or kernel-level library
  - Solaris, Mac OS X, Linux, ...

- Win32 thread library
  - Kernel-level library
Dominant thread libraries (2)

- Java threading API
  - Implemented using **thread library on host system**
    - On Windows: Threads use Win32 API
    - UNIX/Linux: Uses pthreads
Process Synchronizations and Atomic Transactions

Objectives:

- Formulate the critical section problem
- Dissect a software solution to the critical section problem (case study: Peterson's solution)
- Explain Synchronization hardware and Instruction Set Architecture support for concurrency primitives.
- Assess classic problems in synchronization: bounded buffers, readers-writers, dining philosophers.
- Explain serializability of transactions
- Assess locking protocols
- Explain checkpointing and rollback recovery in transactional systems
A look at the producer consumer problem

```java
while (true) {
    while (counter == BUFFER_SIZE) {
        ; /*do nothing */
    }
    buffer[in] = nextProduced
    in = (in +1)%BUFFER_SIZE;
    counter++;
}

while (true) {
    while (counter == 0) {
        ; /*do nothing */
    }
    nextConsumed = buffer[out]
    out = (out +1)% BUFFER_SIZE;
    counter--;
}
```
Implementation of ++/-- in machine language

```
counter++
  register1 = counter
  register1 = register1 + 1
  counter   = register1

counter--
  register2 = counter
  register2 = register2 - 1
  counter   = register2
```
Lower-level statements may be interleaved in any order

Producer execute:  register1 = counter
Producer execute:  register1 = register1 + 1
Producer execute:  counter = register1

Consumer execute:  register2 = counter
Consumer execute:  register2 = register2 - 1
Consumer execute:  counter = register2
Lower-level statements may be interleaved in any order

**Producer** execute: register1 = counter

**Consumer** execute: register2 = counter

**Producer** execute: register1 = register1 + 1

**Consumer** execute: register2 = register2 - 1

**Producer** execute: counter = register1

**Consumer** execute: counter = register2

The order of statements *within* each high-level statement is *preserved*. 
Lower-level statements may be interleaved in any order (counter = 5)

Producer execute: register1 = counter
{register1 = 5}

Producer execute: register1 = register1 + 1
{register1 = 6}

Consumer execute: register2 = counter
{register2 = 5}

Consumer execute: register2 = register2 - 1
{register2 = 4}

Producer execute: counter = register1
{counter = 6}

Consumer execute: counter = register2
{counter = 4}

Counter has incorrect state of 4
Lower-level statements may be interleaved in any order (counter = 5)

Producer execute: register1 = counter  \{register1 = 5\}
Producer execute: register1 = register1 + 1  \{register1 = 6\}
Consumer execute: register2 = counter  \{register2 = 5\}
Consumer execute: register2 = register2 - 1  \{register2 = 4\}
Consumer execute: counter = register2  \{counter = 4\}
Producer execute: counter = register1  \{counter = 6\}

Counter has \textit{incorrect} state of 6
Race condition

- Several processes access and manipulate data **concurrently**

- **Outcome** of execution **depends** on
  - Particular **order** in which accesses takes place

- Debugging programs with race conditions?
  - Painful!
  - Program runs fine most of the time, but once in a rare while something weird and unexpected happens
The kernel is subject to several possible race conditions

- E.g.: Kernel maintains list of all open files
  - 2 processes open files simultaneously
  - Separate updates to kernel list may result in a race condition

- Other kernel data structures
  - Memory allocation
  - Process lists
  - Interrupt handling
Critical-Section

- System of $n$ processes \{P_0, P_1, \ldots, P_{n-1}\}
- Each process has a segment of code (critical section) where it:
  - Changes common variables, updates a table, etc
- No two processes can execute in their critical sections at the same time
The Critical-Section problem

- Design a protocol that processes can use to cooperate

- Each process must request permission to enter its critical section
  - The entry section
General structure of a participating process

do {
    entry section
    critical section
    exit section
    remainder section
} while (TRUE);
Requirements for a solution to the critical section problem

1. Mutual exclusion
2. Progress
3. Bounded wait

- PROCESS SPEED
  - Each process operates at \textit{non-zero} speed
  - Make no assumption about the \textit{relative speed} of the \( n \) processes
Mutual Exclusion

- Only one process can execute in its critical section

- When a process executes in its critical section
  - No other process is allowed to execute in its critical section
Mutual Exclusion: Depiction

Process A

A enters critical section

Process B

B attempts to enter critical section

B blocked

B enters critical section

B exits critical section

T1 T2 T3 T4

A exits critical section
Progress

- {C1} If No process is executing in its critical section, and …
- {C2} Some processes wish to enter their critical sections

- Decision on who gets to enter the critical section
  - Is made by processes that are NOT executing in their remainder section
  - Selection cannot be postponed indefinitely
Bounded waiting

- *After* a process has made a **request** to enter its critical section
  - **AND** *before* this request is granted

- **Limit number** of times other processes are allowed to enter their critical sections
Approaches to handling critical sections in the OS

- **Nonpreemptive kernel**
  - If a process runs in kernel mode: no preemption
  - **Free** from race conditions on kernel data structures

- **Preemptive kernels**
  - Must ensure shared kernel data is free from race conditions
  - **Difficult** on SMP (Symmetric Multi Processor) architectures
    - 2 processes may run simultaneously on different processors
Kernels: Why preempt?

- Suitable for real-time
  - A real-time process may preempt a kernel process

- More **responsive**
  - *Less risk* that kernel mode process will run arbitrarily long
Peterson’s Solution

- Software solution to the critical section problem
  - Restricted to two processes

- No guarantees on modern architectures
  - Machine language instructions such as load and store implemented differently

- Good algorithmic description
  - Shows how to address the 3 requirements
Peterson’s Solution: The components

- Restricted to two processes in this example (but generalizable to n)
  - $P_i$ and $P_j$

- **Share** two data items
  - `int turn`
    - Indicates whose *turn* it is to enter the critical section
  - `boolean flag[2]`
    - Whether process *is ready* to enter the critical section
Peterson’s solution: Structure of process $P_i$

```c
    do {
        flag[0] = TRUE;
        turn = 1;
        while (flag[0] && turn==1) {;
            critical section
            flag[0] = FALSE;
        remainder section
        } while (TRUE);
```
Peterson’s solution: Structure of process $P_i$

do {
    flag[1] = TRUE;
    turn = 0;
    while (flag[0] && turn==0) {;}

    critical section

    flag[0] = FALSE;

    remainder section

} while (TRUE);
Peterson’s solution: Mutual exclusion

- $P_i$ enters critical section only if
  $flag[j] == false \ OR \ turn == i$

- If both processes try to execute in critical section at the same time
  - $flag[0] == flag[1] == true$
  - **But** turn can be 0 or 1, not BOTH

- If $P_j$ entered critical section
  - $flag[j] == true \ AND \ turn == j$
  - Will persist as long as $P_j$ is in the critical section

```java
while (flag[j] == true && turn==j) {};
```
Peterson’s Solution:
Progress and Bounded wait

- $P_i$ can be stuck only if $\text{flag}[j] == \text{true}$ AND $\text{turn} == j$
  - If $P_j$ is not ready: $\text{flag}[j] == \text{false}$, and $P_i$ can enter
  - Once $P_j$ exits: it resets $\text{flag}[j]$ to $\text{false}$

- If $P_j$ resets $\text{flag}[j]$ to $\text{true}$
  - Must set $\text{turn} = i$;

- $P_i$ will enter critical section (progress) after at most one entry by $P_j$ (bounded wait)
Solving the critical section problem using locks

\[
do \{ \\
\text{acquire lock} \\
\text{critical section} \\
\text{release lock} \\
\text{remainder section} \\
\} \text{ while (TRUE);} \\
\]
Possible assists for solving critical section problem (1/2)

- **Uniprocessor environment**
  - *Prevent interrupts* from occurring when shared variable is being modified
    - *No unexpected modifications!*

- **Multiprocessor environment**
  - Disabling interrupts is *time consuming*
    - Message passed to ALL processors
Possible assists for solving critical section problem (2/2)

- Special **atomic** hardware instructions
  - Swap content of two words
  - Modify word
Swap ()

```c
void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```
Swap: Shared variable LOCK is initialized to false

```
do {
    key = TRUE;
    while (key == TRUE) {
        Swap(&lock, &key)
    }
    critical section
    lock = FALSE;
    remainder section
} while (TRUE);
```

Cannot enter critical section UNLESS lock == FALSE

lock is a SHARED variable
key is a LOCAL variable
TestAndSet()
TestAndSet: Shared boolean variable lock initialized to false

```c
do {
    while (TestAndSet(&lock)) {;}
    critical section
    lock = FALSE;
    remainder section
} while (TRUE);
```

To break out:
Return value of TestAndSet should be FALSE

If two TestAndSet() are executed simultaneously, they will be executed sequentially in some arbitrary order
Entering and leaving critical regions using TestAndSet and Swap (Exchange)

enter_region:
  TSL REGISTER, LOCK
  CMP REGISTER, #0
  JNE enter_region
  RET

leave_region:
  MOVE LOCK, #0
  RET

enter_region:
  MOVE REGISTER, #1
  XCHNG REGISTER,LOCK
  CMP REGISTER, #0
  JNE enter_region
  RET

leave_region:
  MOVE LOCK, #0
  RET

All Intel x86 CPUs have the XCHG instruction for low-level synchronization
Semaphores

- Semaphore S is an integer variable

- Once initialized, accessed through **atomic** operations
  - wait()
  - signal()
Defining the semaphore

typedef struct {
    int value;
    struct process *sleeping_list;
} semaphore;
The *wait()* operation to eliminate busy waiting

```c
wait(semaphore *S){
    S->value--; 
    if (S->value < 0) {
        add process to S->sleeping_list;
        block();
    }
}
```

If `value < 0` then `abs(value)` is the number of waiting processes.

`block()` suspends the process that invokes it.
The `signal()` operation to eliminate busy waiting

```c
signal(semaphore *S) {
    S->value++;

    if (S->value <= 0) {
        remove a process P from S->sleeping_list;
        wakeup(P);
    }
}
```

`wakeup(P)` resumes the execution of process P
Deadlocks and Starvation: Implementation of semaphore with a waiting queue

**PROCESS P0**

```c
wait(S);
wait(Q);

signal(S);
signal(Q);
```

**PROCESS P1**

```c
wait(Q);
wait(S);

signal(Q);
signal(S);
```

**Say:** P0 executes `wait(S)` and then P1 executes `wait(Q)`

**P0 must wait till P1 executes** `signal(Q)`

**P1 must wait till P0 executes** `signal(S)`

**Cannot be executed so deadlock**
Semaphores and atomic operations

- Once a semaphore action has started
  - **No other process** can access the semaphore UNTIL
    - Operation has **completed** or **process has blocked**

- Atomic operations
  - Group of related operations
  - Performed without interruptions
    - Or not at all
The bounded buffer problem

- **Binary semaphore (mutex)**
  - Provides mutual exclusion for accesses to buffer pool
  - Initialized to 1

- **Counting semaphores**
  - **empty**: Number of empty slots available to produce
    - Initialized to $n$
  - **full**: Number of filled slots available to consume
    - Initialized to 0
Some other things to bear in mind

- Producer and consumer must be **ready** before they **attempt to enter** critical section

- Producer readiness?
  - When a slot is available **to add** produced item
    - `wait(empty); empty` is initialized to `n`

- Consumer readiness?
  - When a **producer has added** new item to the buffer
    - `wait(full); full` initialized to `0`
The Producer

```java
do {
    produce item nextp
    wait(empty);
    wait(mutex);
    add nextp to buffer
    signal(mutex);
    signal(full);
    remainder section
} while (TRUE);
```

- `wait till slot available` (wait until a slot is available)
- `Only producer OR consumer can be in critical section`
- `Allow producer OR consumer to (re)enter critical section`
- `signal consumer that a slot is available`
The Consumer

do {
    wait (full);
    wait (mutex);
    remove item from buffer
    (nextc)
    signal (mutex);
    signal (empty);
    consume nextc
}
while (TRUE);

wait till slot available for consumption
Only producer OR consumer can be in critical section
Allow producer OR consumer to (re)enter critical section

signal producer that a slot is available to add
The Readers-Writers problem

- A database is **shared** among several concurrent processes

- Two types of processes
  - Readers
  - Writers
Readers-Writers: Potential for adverse effects

- If **two readers** access shared data simultaneously?
  - No problems

- If a **writer and some other reader** (or writer) access shared data simultaneously?
  - Chaos
Writers must have exclusive access to shared database while writing

- **FIRST readers-writers problem:**
  - No reader should wait for other readers to finish; simply because a writer is waiting
    - Writers may starve

- **SECOND readers-writers problem:**
  - If a writer is ready it performs its write ASAP
    - Readers may starve
Solution to the FIRST readers-writers problem

- **Variable** `int readcount`
  - Tracks how many readers are reading object

- **Semaphore** `mutex {1}`
  - Ensure mutual exclusion when `readcount` is accessed

- **Semaphore** `wrt {1}`
  1. Mutual exclusion for the writers
  2. First (last) reader that enters (exits) critical section
     - Not used by readers, when other readers are in their critical section
The Writer: When a writer signals either a waiting writer or the readers resume

```
do {
    wait(wrt);
    writing is performed
    signal(wrt);
} while (TRUE);
```

When:
- writer in critical section
- and if n readers waiting
- 1 reader is queued on wrt
- (n-1) readers queued on mutex
The Reader process

\[
\text{do } \{
\text{wait}(\text{mutex});
\text{readcount}++; \\
\text{if} \ (\text{readcount} == 1) \{ \\
\text{wait}(\text{wrt}); \\
\} \\
\text{signal}(\text{mutex});
\}
\text{while} \ (\text{TRUE});
\]

\[
\text{wait}(\text{mutex});
\text{readcount}--; \\
\text{if} \ (\text{readcount} == 0) \{ \\
\text{signal}(\text{wrt}); \\
\} \\
\text{signal}(\text{mutex});
\]

**mutex for mutual exclusion to readcount**

When:
- writer in critical section
- and if n readers waiting
- 1 is queued on wrt
- (n-1) queued on mutex
Dining Philosopher’s Problem: the situation
The Problem

1. Philosopher tries to *pick up two closest* \{LR\} chopsticks

2. Pick up only 1 *chopstick at a time*
   - Cannot pick up a chopstick being used

3. Eat only when you have *both* chopsticks

4. When done; *put down both* the chopsticks
Why is the problem important?

- Represents allocation of several resources
  - AMONG several processes

- Can this be done so that it is:
  - Deadlock free
  - Starvation free
Dining philosophers: Simple solution

- Each chopstick is a semaphore
  - Grab by executing `wait()`
  - Release by executing `signal()`

- Shared data
  - `semaphore chopstick[5];`
  - All elements are initialized to 1
What if all philosophers get hungry and grab the same \{L/R\} chopstick?

do {
    wait(\texttt{chopstick}[i]);
    wait(\texttt{chopstick}[(i+1)\%5]);
    
    //eat

    signal(\texttt{chopstick}[i]);
    signal(\texttt{chopstick}[(i+1)\%5]);
    
    //think

    } while (TRUE);

Deadlock:
If all processes access chopstick with same hand

We will look at solution with monitors
Dining-Philosophers Using Monitors
Deadlock-free

```c
enum {THINKING, HUNGRY, EATING} state[5];

state[i] = EATING only if
  - state[(i+4)%5] != EATING &&
  - state[(i+1)%5] != EATING

condition self[5]
  - Delay self when HUNGRY but unable to get chopsticks
```
The `pickup()` and `putdown()` operations

```c
pickup(int i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING) {
        self[i].wait();
    }
}

putdown(int i) {
    state[i] = THINKING;
    test((i+4)%5);
    test((i+1)%5);
}
```

Suspend self if unable to acquire chopstick

Check to see if person on left or right can use the chopstick
test(int i) {
    if (state[(i+4)%5] != EATING &&
        state[i] == HUNGRY &&
        state[(i+1)%5] != EATING) {
        state[i] = EATING;
        self[i].signal();
    }
}
Atomic transactions

- Mutual exclusion of critical sections ensures their atomic execution
  - As one *uninterruptible unit*

- Also important to ensure, that critical section forms a *single logical unit of work*
  - Either work is performed in its entirety or not at all
  - E.g. transfer of funds
    - Credit one account and debit the other
Transaction

- Collection of operations performing a **single logical function**

- Preservation of **atomicity**
  - Despite the possibility of failures
Transaction rollbacks

- An aborted transaction may have \textit{modified} data

- State of accessed data must be \textit{restored}
  - \textit{To what it was} before transaction started executing
Log-based recovery to ensure atomicity:
Rely on stable storage

- Record info describing all modifications made by transaction to various accessed data.

- Each log record describes a single write
  - Transaction name
  - Data item name
  - Old value
  - New value

- Other log records exist to record significant events
  - Start of transaction, commit, abort etc
Rationale for checkpointing

- When failure occurs we consult the log for undoing or redoing

- But if done naively, we need to search entire log!
  - Time consuming
  - Recovery takes longer
    - Though no harm done by redoing (idempotency)
Concurrent atomic transactions

- Since each transaction is atomic
  - Executed serially in some arbitrary order
    - Serializability
  - Maintained by executing each transaction within a critical section
    - Too restrictive

- Allow transactions to **overlap** while maintaining serializability
  - Concurrency control algorithms
Serializability

- Serial schedule: Each transaction executes atomically

\[ n! \] schedules for \( n \) independent transactions
Non-serial schedule:
Allow two transactions to overlap

- Does not imply incorrect execution
  - Define the notion of conflicting operations

- $O_i$ and $O_j$ conflict if they access same data item
  - AND at least one of them is a write operation

- If $O_i$ and $O_j$ do not conflict; we can swap their order
  - To create a new schedule
Concurrent serializable schedule

Serial Schedule

T0
read(A)
write(A)
read(B)
write(B)

T1
read(A)
write(A)
read(B)
write(B)

T0
read(A)
write(A)
read(B)
write(B)

T1
read(A)
write(A)
read(B)
write(B)
Conflict serializability

- If schedule $S$ can be **transformed** into a serial schedule $S'$
  - By a series of swaps of non-conflicting operations
CPU Scheduling Algorithms

Objectives:

- Assess scheduling criteria including fairness and time quanta.
- Explain and contrast different approaches to scheduling: preemptive and non-preemptive.
- Explain and assess scheduling algorithms: FCFS, shortest jobs, priority, round-robin, multilevel feedback queues, and the Linux completely fair scheduler.
- Understand how CPU scheduling algorithms function on multiprocessors.
CPU scheduling takes places under the following circumstances

1. I/O or wait
2. interrupt
3. I/O or event completion
4. exit

No scheduling choice \{1,4\} Non preemptive
Nonpreemptive or cooperative scheduling

- Process **keeps** CPU *until it relinquishes* it when:
  1. It terminates
  2. It switches to the waiting state

- Sometimes the *only* method on certain hardware platforms
  - E.g. when they don’t have a hardware timer

- Used by initial versions of OS
  - Windows: Windows 3.x
  - Mac OS
Preemptive scheduling

- Pick a process and let it run for a maximum of some fixed time
- If it is still running at the end of time interval?
  - Suspend it ..
- Pick another process to run
Preemptive scheduling: Requirements

- A **clock interrupt** at the end of the time interval to give control of CPU back to the scheduler

- If no hardware timer is available?
  - Nonpreemptive scheduling is the only option
Preemptive scheduling incurs some costs:
Affects the design of the OS

- System call processing
  - Kernel may be changing kernel data structure (I/O queue)

- Process preempted in the middle AND
  - Kernel needs to read/modify same structure?

- **SOLUTION**: Before context switch
  - Wait for system call to complete OR
  - I/O blocking to occur
Preemptive scheduling incurs some costs:

**Interrupt processing**

- Interrupts can occur at **any** time
  - Cannot always be ignored by kernel
    - Consequences: Inputs lost or outputs overwritten

- Guard code affected by interrupts from simultaneous use:
  - Disable interrupts during entry
  - Enable interrupts at exit
  - **CAVEAT**: Should not be done often, and critical section must contain few instructions
The dispatcher is invoked during every process switch

- **Gives control** of CPU to process selected by the scheduler

- Operations performed:
  - Switch context
  - Switch to user mode
  - Restart program at the right location

- Dispatch latency
  - Time to stop one process and start another
Scheduling Algorithms: Goals

Throughput
Turnaround time
CPU Utilization
Batch Systems

Fairness
Policy Enforcement
Balance
All Systems

Meeting deadlines
Predictability
Real-time systems

Response time
Proportionality
Interactive Systems

Interactive Systems

Fairness
Policy Enforcement
Balance
Throughput
Turnaround time
CPU Utilization

Meeting deadlines
Predictability

Interactive Systems

Response time
Proportionality

Throughput
Turnaround time
CPU Utilization
Batch Systems

Meeting deadlines
Predictability
Real-time systems
CPU Utilization

- Difference between elapsed time and idle time
- Average over a period of time
  - Meaningful only within a context
Scheduling Criteria: Choice of scheduling algorithm may favor one over another

- **CPU Utilization**: Keep CPU as busy as possible? For example:
  - 40% for lightly loaded system
  - 90% for heavily loaded system

- **Throughput**: Number of completed processes per time unit? For example:
  - Long processes: 1/hour
  - Short processes: 10/second
Scheduling Criteria: Choice of scheduling algorithm may favor one over another

- Turnaround time
  - $t_{completion} - t_{submission}$

- **Waiting** time
  - Total time spent waiting in the ready queue

- Response time
  - Time to start responding
  - $t_{first\_response} - t_{submission}$
  - Generally *limited* by speed of output device
Scheduling Algorithms

- **Decides** which process in the ready queue is allocated the CPU
- Could be preemptive or nonpreemptive
- Optimize **measure** of interest
- We will use **Gantt charts** to illustrate **schedules**
  - Bar chart with start and finish times for processes
First-Come, First-Served Scheduling (FCFS)

- Process requesting CPU first, gets it first
- Managed with a FIFO queue
  - When process **enters** ready queue?
    - PCB is tacked to the **tail** of the queue
  - When CPU is **free**?
    - It is allocated to process at the **head** of the queue
- Simple to write and understand
Average waiting times in FCFS

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>24</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
</tbody>
</table>

Wait time for P1: $(0 + 24 + 27)/3 = 17$

Wait time for P2: $(6 + 0 + 3)/3 = 3$
Disadvantages of the FCFS scheme (1)

- Once a process gets the CPU, it keeps it
  - Till it terminates or does I/O
  - Unsuitable for time-sharing systems

- Average waiting time is generally not minimal
  - Varies substantially if CPU burst times vary greatly
Disadvantages of the FCFS scheme (2)

- Poor performance in certain situations
  - 1 CPU-bound process and many I/O-bound processes
  - **Convoy effect**: Smaller processes wait for the one big process to get off the CPU
Shortest Job First (SJF) scheduling algorithm

- When CPU is available it is assigned to process with **smallest CPU burst**

- Moving a short process before a long process?
  - Reduction in waiting time for short process **GREATER THAN** Increase in waiting time for long process

- Gives us **minimum average waiting time** for a set of processes that arrived *simultaneously*
  - Provably Optimal
Depiction of SJF in action

Process | Burst Time
---|---
P1 | 6
P2 | 8
P3 | 7
P4 | 3

Wait time = \((3 + 16 + 9 + 0)/4 = 7\)
SJF is optimal ONLY when ALL the jobs are available simultaneously

- Consider 5 processes A, B, C, D and E
  - Run times are: 2, 4, 1, 1, 1
  - Arrival times are: 0, 0, 3, 3, 3

- SJF will run jobs: A, B, C, D and E
  - Average wait time: \((0 + 2 + 3 + 4 + 5)/5 = 2.8\)
  - But if you run B, C, D, E and A?
    - Average wait time: \((7 + 0 + 1 + 2 +3)/5 = 2.6!\)
Preemptive SJF

- A new process arrives in the ready queue
- If it is shorter than the currently executing process
  - Preemptive SJF will preempt the current process

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival</th>
<th>Burst</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>P4</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Wait time = \[
\frac{(10-1) + (1-1) + (17-2) + (5-3)}{4}
\]

= \frac{26}{4} = 6.5
Use of SJF in long term schedulers

- Length of the process time limit
  - Used as CPU burst estimate

- Motivate users to accurately estimate time limit
  - Lower value will give faster response times
  - Too low a value?
    - Time limit exceeded error
    - Requires resubmission!
The SJF algorithm and short term schedulers

- **No way to know** the length of the next CPU burst
- So try to **predict** it
- Processes scheduled **based on predicted** CPU bursts
Priority Scheduling

- **Priority** associated with each process
- CPU allocated to process with **highest** priority
- Can be preemptive or nonpreemptive
  - If preemptive: Preempt CPU from a lower priority process when a higher one is ready
### Depiction of priority scheduling in action

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>P5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Here: Lower number means higher priority

Wait time = \((6 + 0 + 16 + 18 + 1)/5 = 8.2\)
How priorities are set

- Internally defined priorities based on:
  - Measured quantities
  - Time limits, memory requirements, # of open files, ratio (averages) of I/O to CPU burst

- External priorities
  - Criteria outside the purview of the OS
  - Importance of process, $ paid for usage, politics, etc.
Issue with priority scheduling

- Can leave lower priority processes waiting indefinitely

- Perhaps apocryphal tale:
  - MIT’s IBM 7094 shutdown (1973) found processes from 1967!
Coping with issues in priority scheduling: Aging

- **Gradually increase priority** of processes that wait for a long time

- **Example:**
  - Process with priority of 127 and increments every 15 minutes
  - Process priority becomes 0 in no more than 32 hours
Round-Robin Scheduling

- Similar to FCFS scheduling
  - **Preemption** to enable switch between processes

- Ready queue is implemented as **FIFO**
  - Process Entry: PCB at *tail* of queue
  - Process chosen: From *head* of the queue

- CPU scheduler goes around ready queue
  - Allocates CPU to each process one after the other
    - CPU-bound up to a maximum of 1 **quantum**
Round Robin: Choosing the quantum

- Context switch is **time consuming**
  - Saving and loading registers and memory maps
  - Updating tables
  - Flushing and reloading memory cache

- What if quantum is 4 ms and context switch overhead is 1 ms?
  - 20% of CPU time thrown away in administrative overhead
Round Robin: Improving efficiency by increasing quantum

- Let's say quantum is 100 ms and context-switch is 1 ms
  - Now wasted time is only 1%

- But what if 50 concurrent requests come in?
  - Each with widely varying CPU requirements
  - 1<sup>st</sup> one starts immediately, 2<sup>nd</sup> one 100 ms later, …
  - The last one may have to wait for 5 seconds!
  - A shorter quantum would have given them better service
If quantum is set longer than mean CPU burst?

- Preemption will not happen very often
- Most processes will perform a blocking operation before quantum runs out
- Switches happens only when process blocks and cannot continue
Quantum: Summarizing the possibilities

- Too short?
  - Too many context switches
  - Lowers CPU efficiency

- Too long?
  - Poor responses to interactive requests
Deadlocks

Objectives:
- Explain deadlock characterization
- Contrast and explain schemes for deadlock prevention
- Evaluate approaches to deadlock avoidance
- Understand recovery from deadlocks
System model

- **Finite** number of resources
  - Distributed among *competing processes*

- Resources are *partitioned* into different **types**
  - Each *type* has a number of identical instances
  - Resource type examples:
    - Memory space, files, I/O devices
A process must utilize resources in a sequence

- **Request**
  - Requesting resource must *wait until it can acquire* resource
  - `request()`, `open()`, `allocate()`

- **Use**
  - Operate on the resource

- **Release**
  - `release()`, `close()`, `free()`
For kernel managed resources, the OS maintains a system resource table

- Is the resource free?
  - Record process that the resource is allocated to

- Is the resource allocated?
  - Add to queue of processes waiting for resource

- For resources not managed by the OS
  - Use `wait()` and `signal()` on semaphores
Preemptable resources

- Can be taken away from process owning it with no ill effects

- Example: Memory
  - Process B’s memory can be taken away and given to process A
    - Swap B from memory, write contents to backing store, swap A in and let it use the memory
Non-preemptable resources

- Cannot be taken away from a process without causing the process to fail
- If a process has started to burn a CD
  - Taking the CD-recorder away from it and giving it to another process?
    - Garbled CD
    - CD recorders are not preemptable at an arbitrary moment
- In general, **deadlocks involve non-preemptable resources**
Some notes on deadlocks

- The OS typically does not provide deadlock prevention facilities

- Programmers are responsible for designing deadlock free programs
Deadlock: Formal Definition

- A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

- Because all processes are waiting, none of them can cause events to wake any other member of the set
  - Processes continue to wait forever
Deadlocks: Necessary Conditions (I)

- **Mutual Exclusion**
  - At least one resource held in *nonsharable* mode
  - When a resource is being used
    - Another requesting process must wait for its release

- **Hold-and-wait**
  - A process must hold one resource
  - Wait to acquire additional resources
    - Which are currently held by other processes
Deadlocks:
Necessary Conditions (ⅠⅠ)

- **No preemption**
  - Resources cannot be preempted
  - Only voluntary release by process holding it

- **Circular wait**
  - A set of \{P_0, P_1, ..., P_n\} waiting processes must exist
    - \(P_0 \rightarrow P_1; P_1 \rightarrow P_2, ..., P_n \rightarrow P_0\)
  - Implies hold-and-wait
Methods for handling deadlocks

- Use protocol to **prevent** or **avoid** deadlocks
  - Ensure system never enters a deadlocked state

- Allow system to enter deadlocked state; BUT
  - **Detect** it and **recover**

- Ignore problem, pretend that deadlocks never occur
When is ignoring the problem viable?

- When they occur infrequently (once per year)
  - Ignoring is the *cheaper* solution
  - Prevention, avoidance, detection and recovery
    - Need to run constantly
Four strategies for dealing with deadlocks

- Ignore the problem
  - May be if you ignore it, it will ignore you

- Detection and Recovery
  - Let deadlocks occur, detect them, and take action

- Deadlock avoidance
  - By careful resource allocation

- Deadlock prevention
  - By structurally negating one of the four required conditions
Deadlock Prevention

- Ensure that **one** of the necessary conditions for deadlocks cannot occur
  1. Mutual exclusion
  2. Hold and wait
  3. No preemption
  4. Circular wait
Mutual exclusion must hold for non-sharable resources, but ...

- Sharable resources do not require mutually exclusive access
  - *Cannot be involved* in a deadlock

- A process never needs to wait for sharable resource
  - Read-only files

- Some resources are *intrinsically nonsharable*
  - So denying mutual exclusion often not possible
Deadlock Prevention: Ensure hold-and-wait never occurs in the system [Strategy 1]

- Process must request and be allocated all its resources before execution
  - Resource requests must precede other system calls

- E.g. copy data from DVD drive, sort file & print
  - Printer needed only at the end
  - BUT process will hold printer for the entire execution
Deadlock Prevention: Ensure hold-and-wait never occurs in the system [Strategy 2]

- Allow a process to request resources only when it has none
  - *Release* all resources, *before requesting* additional ones

- E.g. copy data from DVD drive, sort file & print
  - First request DVD and disk file
    - Copy and release resources
  - Then request file and printer
Disadvantages of protocols doing hold-and-wait

- **Low resource utilization**
  - Resources are allocated but unused for long durations

- **Starvation**
  - If a process needs several popular resources
    - Popular resource might always be *allocated to some other* process
Deadlock Prevention: Eliminate the preemption constraint [1/2]

- {C1} If a process is holding some resources
- {C2} Process requests another resource
  - Cannot be immediately allocated

- All resources currently held by process is preempted
  - Preempted resources added to list of resources process is waiting for
Deadlock Prevention: Eliminate the preemption constraint [2/2]

- Process requests resources that are not currently available
  - If resources allocated to another waiting process
    - Preempt resources from the second process and assign it to the first one

- Often applied when resource state can be saved and restored
  - CPU registers and memory space
  - Unsuitable for tape drives
Deadlock Prevention: Eliminating Circular wait

- Impose **total ordering** of all resource types
  - Assign each resource type a unique number
  - One-to-one function $F : R \rightarrow N$
    
    \[
    F(\text{tape drive}) = 1; \\
    F(\text{printer}) = 12
    \]

1. Request resources in **increasing order**
2. If several instances of a resource type needed?
   - Single request for all them must be issued
Deadlock Prevention: **Summary**

- Prevent deadlocks by **restraining** how requests are made.
  - Ensure at least 1 of the 4 conditions cannot occur

- **Side effects:**
  - Low device utilization
  - Reduced system throughput
Deadlock avoidance

- Require *additional* information about how resources are to be requested

- Knowledge about sequence of requests and releases for processes
  - Allows us to decide if resource allocation *could cause a future deadlock*
  - Process P: Tape drive, then printer
  - Process Q: Printer, then tape drive
Deadlock avoidance:
Handling resource requests

- For each resource request:
  - Decide whether or not process should wait
    - To avoid possible future deadlock

- Predicated on:
  1. Currently available resources
  2. Currently allocated resources
  3. Future requests and releases of each process
Avoidance algorithms differ in the amount and type of information needed.

- **Resource allocation state**
  - Number of available and allocated resources
  - Maximum demands of processes

- **Dynamically examine** resource allocation state
  - Ensure circular-wait cannot exist

- **Simplest model:**
  - Declare maximum number of resources for each type
  - Use information to avoid deadlock
Safe sequence

- **Sequence** of processes \(<P_1, P_2, ..., P_n>\) for the current allocation state

- Resource requests made by \(P_i\) can be satisfied by:
  - Currently available resources
  - Resources held by \(P_j\) where \(j < i\)
    - If needed resources not available, \(P_i\) can wait
  - In general, when \(P_i\) terminates, \(P_{i+1}\) can obtain its needed resources

- If no such sequence exists: system state is **unsafe**
Safe states and deadlocks

- A system is safe ONLY IF there is a safe sequence

- A safe state is not a deadlocked state
  - Deadlocked state is an unsafe state
  - Not all unsafe states are deadlocks
Unsafe states

- A unsafe state *may lead* to deadlock
- **Behavior** of processes controls unsafe states
- Cannot prevent processes from requesting resources such that deadlocks occur
Banker’s Algorithm

- Designed by Dijkstra in 1965

- Modeled on a small-town banker
  - Customers have been extended lines of credit
  - Not ALL customers will need their maximum credit immediately

- Customers make loan requests from time to time
Crux of the Banker’s Algorithm

- Consider each request as it occurs
  - See if granting it is safe

- If safe: grant it; If unsafe: postpone

- For safety banker checks if he/she has **enough** to satisfy some customer
  - If so, that customer’s loans are assumed to be repaid
  - Customer closest to limit is checked next
  - **If all loans can be repaid; state is safe: loan approved**
Banker’s Algorithm: Managing the customers.
Banker has only reserved 10 units instead of 22

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Has</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Max</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>Free: 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAFE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Has</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Max</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>Free: 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAFE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Has</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Max</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>Free: 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNSAFE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Delay all requests except C

A customer may not need the entire credit line. But the banker cannot count on this behavior.

There is ONLY ONE resource: Credit
Banker’s algorithm: Crux

- Declare **maximum** number of resource instances needed
  - Cannot exceed resource thresholds

- Determine if resource allocations leave system in a safe state
Bankers Algorithm: Resource-request

\[ \text{Request}_i \leq \text{Need}_i \]

- If true, go to next step.
- If false, error: Exceeded claim.

\[ \text{Request}_i \leq \text{Available} \]

- If true, update:
  \[ \text{Available} = \text{Available} - \text{Request}_i \]
  \[ \text{Allocation}_i = \text{Allocation}_i + \text{Request}_i \]
  \[ \text{Need}_i = \text{Need}_i - \text{Request}_i \]

- If false, wait for availability.

L29.258
Bankers Algorithm: Safety

Initialize \( \text{Work} = \text{Available} \)

Find \( i \) such that:
\[
\text{Finish}[i] = \text{false} \quad \&\& \quad \text{Need}_i \leq \text{Work}
\]

\[
\text{Work} = \text{Work} + \text{Allocation}_i
\]
\[
\text{Finish}[i] = \text{true}
\]

for all \( i \)
\[
\text{if} \ (\text{Finish}[i] = \text{true})
\]

Safe state

YES

Unsafe state

NO
Recovery from deadlock

- Automated or manual

- OPTIONS
  - Break the circular wait: **Abort** processes
  - **Preempt** resources from deadlocked process(es)
Breaking circular wait: Process termination

- Abort **all** deadlocked processes

- Abort processes **one at a time**
  - After each termination, check if deadlock persists

- **Reclaim all resources** allocated to terminated process
Deadlock recovery: Resource preemption

For a set of deadlocked processes

Preempt resources from some process

Give resources to some other process

DONE

Deadlock broken

Deadlock persists
Resource preemption: Issues

- Selecting a victim
  - Which resource and process
  - Order of preemption to minimize cost

- Starvation
  - Process can be selected for preemption *finite* number of times
Livelocks

- Polling (busy waits) used to enter critical section or access a resource
  - Typically used for a short time when overhead for suspension is considered greater

- In a livelock two processes need each other’s resource
  - Both run and make no progress, but neither process blocks
  - Use CPU quantum over and over without making progress
Livelocks do occur

- If `fork` fails because process table is full
  - Wait for some time and try again

- But there could be a collection of processes each trying to do the same thing
Objectives:

- Understand address binding and address spaces
- Explain contiguous memory allocations: including their advantages and disadvantages
- Analyze the key constructs underpinning paging systems including hardware support, shared pages, and structure of page tables
- Explain memory protection in paging environments
- Explain segmentation based approaches to memory management alongside settings in which they are particularly applicable
Memory Management: Why?

- Main objective of system is to execute programs
- Programs and data must be in memory (at least partially) during execution
- To improve CPU utilization and response times
  - Several processes need to be memory resident
  - Memory needs to be shared
Memory Unit

- Sees only a **stream** of memory addresses

- Oblivious to
  - *How* these addresses are generated
    - Instruction counter, indexing, indirection, etc.
  - *What* they are for
    - Instructions or data
Why processes must be memory resident

- Storage that the CPU can access **directly**
  1. Registers in the processor
  2. Main memory

- Machine instructions take memory addresses as arguments
  - None operate on disk addresses

- Any instructions in execution **plus** needed data
  - Must be in memory
Processes and memory

- To execute, a program needs to be placed inside a process

- Process executes
  - Access instructions and data from memory

- Process terminates
  - Memory reclaimed and declared available
Binding is a mapping from one address space to the next

- Processes can reside in **any part** of the physical memory
  - First address of process need not be x0000

- Addresses in source program are **symbolic**

- Compiler binds symbolic addresses to **relocatable** addresses

- Loader binds relocatable addresses to **absolute** addresses
Binding can be done at …

- **Compile time**
  - Known that the process will reside at location $R$
    - If location changes: recompile
  - MS-DOS .COM programs were bound this way

- **Load time**
  - Based on compiler generated relocatable code
Binding can be done at ...  

**Execution-time**

- Process can be moved around during execution
  - Binding *delayed* until run time
  - Special hardware needed
  - Supported by most OS
Partitioning of memory

- Main memory needs to **accommodate** the OS and user processes

- Divided into two partitions
  - Resident OS
    - Usually low memory
  - User processes
Memory Mapping and Protection

- When CPU scheduler selects a process for execution
  - Relocation and limit registers reloaded as part of context switch

- Every address generated by the CPU
  - Checked against the relocation/limit registers
Memory Mapping and Protection

E.g.: relocation=100040 and limit=74600

TRAP to OS: Addressing ERROR
Address spaces

- **Logical**
  - Addresses generated by the program running on CPU

- **Physical**
  - Addresses seen by the memory unit

- **Logical address space**
  - Set of logical addresses generated by program

- **Physical address space**
  - Set of physical addresses corresponding to the logical address space
Generation of physical and logical addresses

- Compile-time and load-time
  - *Identical* logical and physical addresses

- Execution time
  - Logical addresses *differ* from physical addresses
  - Logical address referred to as *virtual* address

- Runtime mapping performed in hardware
  - Memory management unit (*MMU*)
Memory management unit

- Mapping converts logical to physical addresses

- User program *never sees* real physical address
  - Create pointer to location
  - Store in memory, manipulate and compare

- When used as a **memory address** (load/store)
  - Relocated to physical memory
Dynamic Storage Allocation Problem

- Satisfying a request of size $n$ from the set of available spaces
  - First fit
  - Best fit
  - Worst fit
First fit

- Scan list of segments until you find a memory-hole that is big enough

- Hole is broken up into two pieces
  - One for the process
  - The other is unused memory
Best Fit

- Scan the entire list from beginning to the end
- Pick the smallest memory-hole that is adequate to host the process
Comparing Best Fit and First Fit

- Best fit is **slower** than first fit

- Surprisingly, it also results in more **wasted memory** than first fit
  - Tends to fill up memory with tiny, useless holes
Worst fit

- How about going to the other extreme?
  - Always take the largest available memory-hole
  - Perhaps, the new memory-hole would be useful

- Simulations have shown that worst fit is not a good idea either
Contiguous Memory Allocation: Fragmentation

- As processes are loaded/removed from memory
  - Free memory space is broken into small pieces

- **External fragmentation**
  - Enough space to satisfy request; BUT
  - Available spaces are *not contiguous*
Fragmentation: Example

Process P₅ cannot be loaded because memory space is fragmented
Fragmentation can be internal as well

- Memory allocated to process may be *slightly larger* than requested

- **Internal fragmentation**
  - Unused memory is internal to blocks
Compaction: Solution to external fragmentation

- **Shuffle** memory contents
  - Place free memory into large block

- Not possible if relocation is static
  - Load time

- Approach involves moving:
  1. Processes towards one end
  2. Gaps towards the other end
Compaction: Example
Overview of how mapping of logical and physical addresses is performed

CPU

Virtual address

Memory Management Unit (MMU)

Translation Lookaside Buffer (TLB)

Physical Memory

Physical address

MMU may access Physical Memory to perform translations
{PageTable may be stored there}
The Paging memory management scheme

- Physical address space of process can be **non-contiguous**

- Solves problem of fitting variable-sized memory chunks to backing store
  - Backing store has fragmentation problem
    - Compaction is impossible
Basic method for implementing pages

- Break memory into **fixed-sized** blocks
  - Physical memory: **frames**
  - Logical memory: **pages**

- Backing store is also divided the same way
Paging Hardware: Paging is a form of dynamic relocation.
Paging: Logical and Physical Memory

Logical Memory

Page 0
Page 1
Page 2
Page 3

Page Table

0 1
1 4
2 3
3 7

Physical Memory

0 Page 0
1 Page 2
2 Page 1
3 Page 3
4
5
6
7
Page size

- Usually a **power of 2**
  - 512 bytes – 16 MB

- Size of logical address: $2^m$

- Page size: $2^n$

Page number | Page offset
---|---
$m - n$ | $n$

$m$ bits

Logical address
Paging and Fragmentation

- **No external fragmentation**
  - Free frame available for allocation to other processes

- **Internal fragmentation possible**
  - Last frame may not be full
  - If process size is independent of page size
    - Internal fragmentation = $\frac{1}{2}$ page per process
Paging: User program views memory as a single space

- Program is **scattered** throughout memory
- User view and physical memory **reconciled** by
  - Address-translation hardware
- Process has **no way** of addressing memory outside of its page table
OS manages the physical memory

- Maintains **frame-table;** one entry per frame
  - Free or allocated?
  - If allocated: Which page of which process

- Maintains a page table for **each process**
  - Used by CPU dispatcher to define hardware page table when process is CPU-bound
    - Paging increases context switching time
The purpose of the page table is to map virtual pages onto physical frames

- Think of the page table as a function
  - Takes virtual page number as an argument
  - Produces physical frame number as result

- Virtual page field in virtual address replaced by frame field
  - Physical memory address
Two major issues facing page tables

- Can be **extremely large**
  - With a 4 KB page size, a 32-bit address space has 1 million pages
  - Also, each process has its own page table

- The **mapping must be fast**
  - Virtual-to-physical mapping must be done on *every memory reference*
  - Page table lookup should not be a bottleneck
Translation look-aside buffer
Small, fast-lookup hardware cache

- Number of TLB entries is small (64 ~ 1024)
  - Contains few page-table entries

- Each entry of the TLB consists of 2 parts
  - A key and a value

- When the associative memory is presented with an item
  - Item is compared with all keys *simultaneously*
The purpose of the page table is to map virtual pages onto page frames

- Think of the page table as a **function**
  - Takes virtual page number as an argument
  - Produces physical frame number as result

- Virtual page field in virtual address replaced by frame field
  - Physical memory address
Paging Hardware with a TLB

Logical Address: \( p \) and \( d \)
Physical Address: \( f \) and \( d \)

Page Table: \( f \)
TLB: \( f \) and \( d \)

TLB Miss

CPU

Page number
Page offset

Frame \( f \)

\( f000...000 \)
\( f111...111 \)
Protection bits are associated with each frame

- Kept in the page table

- Bits can indicate
  - Read-write, read-only, execute
  - Illegal accesses can be trapped by the OS

- Valid-invalid bit
  - Indicates if page is in the process’s logical address space
Protection Bits: Page size=2K;
Logical address space = 16K

Program restricted to 0 – 10468

<table>
<thead>
<tr>
<th>Logical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page 0</td>
</tr>
<tr>
<td>Page 1</td>
</tr>
<tr>
<td>Page 2</td>
</tr>
<tr>
<td>Page 3</td>
</tr>
<tr>
<td>Page 4</td>
</tr>
<tr>
<td>Page 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame Number</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page 0</td>
</tr>
<tr>
<td>Page 1</td>
</tr>
<tr>
<td>Page 2</td>
</tr>
<tr>
<td>Page 3</td>
</tr>
<tr>
<td>Page 4</td>
</tr>
<tr>
<td>Page 5</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Page n</td>
</tr>
</tbody>
</table>

10K = 10240
A computer program or subroutine is called **reentrant** if:

- It can be *interrupted* in the middle of its execution and
- Then safely called again ("re-entered") *before* its previous invocations complete execution
Reentrant Code

- **Non-self-modifying**
  - Does not change during execution

- **Two or more processes can:**
  1. Execute same code at same time
  2. Will have different data

- **Each process has:**
  - Copy of registers and data storage to hold the data
Shared Pages

- System with \( N \) users
  - Each user runs a text editing program

- Text editing program
  - 150 KB of code
  - 50 KB of data space

- 40 users
  - Without sharing: 8000 KB space needed
  - With sharing: \( 150 + 40 \times 50 = 2150 \) KB needed
Shared Paging

Process $P_1$
- ed 1
- ed 2
- ed 3
- Data 1

Process $P_2$
- ed 1
- ed 2
- ed 3
- Data 2

Process $P_3$
- ed 1
- ed 2
- ed 3
- Data 3

Page Tables

Physical Memory
- Data 1
- Data 3
- ed 1
- ed 2
- ed 3
- Data 2
- Page n...
Shared Paging

- Other heavily used programs can be shared
  - Compilers, runtime libraries, database systems, etc.

- To be shareable:
  1. Code must be *reentrant*
  2. The *OS must enforce read-only* nature of the shared code
Overheads in paging:
Page table and internal fragmentation

- Average process size = \( s \)
- Page size = \( p \)
- Size of each page entry = \( e \)
- Pages per process = \( s/p \)
  - \( se/p \): Total page table space

- Total Overhead = \( se/p + p/2 \)
Typical use of the page table

- Process refers to addresses through pages’ *virtual* address
- Process has page table
- Table has entries for pages that process uses
  - One slot for each page
    - Irrespective of whether it is valid or not
- Page table sorted by virtual addresses
Paging Hardware: Paging is a form of dynamic relocation

- **Logical Address**
  - Page number
  - Page offset

- **Physical Address**
  - Frame $f$
  - Page Table
  - Page number $p$
  - Page offset $d$
  - Frame number $f$
  - Physical address $f_{000...000}$ or $f_{111...111}$
Hierarchical Paging

- **Logical address spaces:** $2^{32} \sim 2^{64}$

- **Page size:** $4\text{KB} = 2^2 \times 2^{10} = 2^{12}$

- **Number of page table entries?**
  - Logical address space size/page size
  - $2^{32}/2^{12} = 2^{20} \approx 1 \text{ million entries}$

- **Page table entry = 4 bytes**
  - Page table for process = $2^{20} \times 4 = 4 \text{ MB}$
Issues with large page tables

- Cannot allocate page table *contiguously* in memory

- Solution:
  - Divide the page table into smaller pieces
  - Page the page-table
Two-level Paging

<table>
<thead>
<tr>
<th>Page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>12</td>
</tr>
</tbody>
</table>

32-bit logical address
## Two-level Paging

<table>
<thead>
<tr>
<th>Outer Page</th>
<th>Inner Page</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

32-bit logical address
Address translation in two-level paging

$p_1$ $p_2$ $d$

Track pages of page-table

Outer page table

Page of page table

Actual Physical address

Physical memory frame
x86-64

- Intel: IA-64 Itanium
  - Not much traction

- AMD: x86-64
  - Intel adopted AMD’s x86-64 architecture

- 64-bit address space: $2^{64}$ (16 exabytes)

- Currently x86-64 provides
  - 48-bit virtual address
  - Page sizes: 4 KB, 2 MB, and 1 GB
  - 4-level paging hierarchy
ARM architectures

- iPhone and Android systems use this

- 32-bit ARM
  - 4 KB and 16 KB pages
  - 1 MB and 16 MB pages

There are two levels for TLBs:
- A separate TLB for data
- Another for instructions
In our discussions so far ...

- Virtual memory is **one-dimensional**
  - Logical addresses go from 0 to some max value

- Many problems can benefit from having two or more **separate** virtual address spaces
One dimensional address space with growing tables

Program has an exceptional number of variables

Address space being used

Address space allocated to the constant table
One dimensional address space with growing tables

Program has an exceptional number of variables

Symbol table has BUMPED INTO the source text table

Address space allocated to the constant table

Address space being used

Free

Symbol Table
Source text
Constant table
Parse tree
Call stack
Segmentation

- Logical address space is a collection of segments
- Segments have name and length
- Addresses specify
  - Segment name
  - Offset within the segment
- Tuple: `<segment-number, offset>`
Segmentation Hardware

CPU → Logical Address

Segment Table:
- Limit
- Base

Physical Address

TRAP: Addressing Error

< YES NO
Rationale for Paging and Segmentation

- Get a large linear address space **without** having to buy more physical memory
  - PAGING

- Allow programs and data to be broken up into **logically independent** address spaces
  - Aids Sharing AND Protection
    - Segmentation
## Comparison of Paging and Segmentation

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>How many linear address spaces are there?</td>
<td>1</td>
<td>Many</td>
</tr>
<tr>
<td>Can total address space exceed physical memory</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Can procedures and data be distinguished and protected separately?</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Can fluctuating table sizes be accommodated?</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>
Comparison of Paging and Segmentation

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Should the programmer be aware the the technique is being used?</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Is sharing of procedures between users facilitated?</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Why was this technique invented?</td>
<td>To get a large linear address space without having to buy more physical memory</td>
<td>To allow programs and data to be broken up into logically independent address spaces and to allow sharing and protection</td>
</tr>
</tbody>
</table>
Segmentation with Paging

- **Multics**: Each program can have up to 256K independent segments
  - Each with 64K 36-bit words

- **Intel Pentium**
  - 16K independent segments
  - Each segment has $10^9$ 32-bit words
  - Few programs need more than 1000 segments, but many programs need large segments
Virtual Memory

Objectives:

- Explain demand paging and page faults
- Contrast page replacement algorithms and explain Belady's anomaly
- Justify the rationale for stack algorithms
- Explain frame allocations
- Synthesize the concepts of thrashing and working sets
How we got here …

Contiguous Memory → External Fragmentation → Pure Paging → Single Address space → Segmentation

Low Degree of Multiprogramming

Virtual Memory
Logical view of a process in memory

- **Stack**
  - Function parameters, return addresses, and local variables

- **Heap**
  - Memory allocated dynamically during runtime

- **Data**
  - Global variables

- **Text**
  - Program code
Logical view of a process in memory

Requires actual physical space ONLY IF heap or stack grows
Sparse address spaces

- Virtual address spaces with holes

- Harnessed by
  - Heap or stack segments
  - Dynamically linked libraries
Loading an executable program into memory

- What if we load the entire program?
  - We may not need the entire program

- Load pages only when they are needed
  - Demand Paging
Differences between the swapper and pager

- **Swapper**
  - Swaps the *entire program* into memory

- **Pager**
  - Lazy swapper
  - Never swap a page into memory *unless* it is actually *needed*
Swapping: Temporarily moving a process out of memory into a backing store

![Diagram showing the concept of swapping in an operating system with processes P1 and P2, where Swap out and Swap in are indicated.]
Pager swapping pages in and out of physical memory
Demand Paging: Basic concepts

- **Guess** pages to be utilized by process
  - Before the process will be swapped out

- **Avoid** reading unused pages
  - Better physical memory utilization
  - Reduced I/O
    - Lower swap times
Distinguishing between pages in memory and those on disk

- **Valid-Invalid bits**
  - Associated with entries in the page table

- **Valid**
  - Page is both legal and in memory

- **Invalid**
  1. Page is *not in logical address space* of process
  2. Valid BUT currently *on disk*
Distinguishing between pages in memory and those on disk

Logical Memory

Page Table

0
1
2
3
4
5
6
7

A
B
C
D
E
F
G
H

0 1 2 3 4 5 6 7 8 9 10 11 12 13

Page Table

0
1
2
3
4
5
6
7

Logical Memory

Physical Memory

Backing Store

A
B
C
D
E
F
G
H

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

CS370: Operating Systems
Dept. Of Computer Science, Colorado State University
Handling page faults

1. Reference
2. Trap to the OS
3. Locate page on backing store
4. Bring in missing page
5. Reset page table
6. Restart instruction

load M

Reference: I

Free Frame

PAGE TABLE

Locate page on backing store

FREE FRAME

Physicall Memory

Backing Store

CS370: Operating Systems
Dept. Of Computer Science, Colorado State University
Pure demand paging

- Never bring a page into memory unless it is required
- Execute process with no pages in memory
  - First instruction of process will fault for the page
- Page fault to load page into memory and execute
Potential problems with pure demand paging

- Multiple page faults per instruction execution
  - One fault for instruction
  - Many faults for data

- Multiple page faults per instruction are rare
  - Locality of reference
Hardware requirements to support demand paging

- Page Table

- Secondary memory
  - Section of disk known as *swap space* is used
Restarting instructions after a page fault

- Page faults occur at memory reference

- Use PCB to save state of the interrupted process

- Restart process in exactly the same place
  - Desired page is now in memory and accessible
Effective access times

- **Without** page faults, effective access times are equal to memory access times
  - 200 nanoseconds approximately

- **With** page faults
  - Account for fault servicing with disk I/O
Calculating the effective access times with demand paging

\( p \) : probability of a page fault

\( ma \) : memory access time

Effective access time =

\[(1-p) \times ma + p \times \text{page-fault-time}\]
## Components of page-fault servicing

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service interrupt</td>
<td></td>
<td>1~100 µS</td>
</tr>
<tr>
<td>Read in the page</td>
<td></td>
<td>Latency: 3 mS, Seek: 5 mS</td>
</tr>
<tr>
<td>Restart process</td>
<td></td>
<td>1~100 µS</td>
</tr>
</tbody>
</table>
Page replacement

- If no frame is free
  - Find one that is not currently being used
    - Use it
Freeing a physical memory frame

- Write frame contents to swap space
- Change page table of process
  - To reflect that page is no longer in memory
- Freed frame can now hold some other page
Servicing a page fault

Retrieve page from disk

Free frame available?

Select **victim** frame

Write victim frame to disk

Use it

YES

NO

Done using a page replacement algorithm
Page replacement is central to demand paging.

1. Reference
2. Trap to the OS
3. Locate page on backing store
4. Bring in missing page
5. Reset page table
6. Restart instruction

**Operating System**

**PAGE TABLE**

**PHYSICAL MEMORY**

**BACKING STORE**

Load M

**Free Frame**
Page replacement algorithms:

- What are we looking for?
  - Low page-fault rates

- How do we evaluate them?
  - Run algorithm on a string of memory references
    - Reference string
  - Compute number of page faults
FIFO page replacement algorithm:
Out with the old; in with the new

- When a page must be replaced
  - Replace the **oldest** one

- OS maintains list of all pages currently in memory
  - Page at head of the list: Oldest one
  - Page at the tail: Recent arrival

- During a page fault
  - Page at the head is removed
  - New page added to the tail
### FIFO example: 3 memory frames

<table>
<thead>
<tr>
<th>Reference String</th>
<th>Youngest</th>
<th>Oldest</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 1 2 0 3 0 4 2 3 0 0 1 2 2 2 7 0 1</td>
<td>7 0 1 2 2 3 0 4 2 3 0 0 1 2 2 2 7 0 1</td>
<td>7 0 0 1 2 3 0 4 2 2 2 3 0 0 0 1 2 7</td>
</tr>
</tbody>
</table>

No page fault
How we got here …

- Contiguous Memory
- External Fragmentation
- Pure Paging
- Low Degree of Multiprogramming

- Demand Paging
- Page Faults
- Page replacement algorithms
- Page Bufferring
- Working Sets
- Frame Allocation

CS370: Operating Systems
Dept. Of Computer Science, Colorado State University
Intuitively the greater the number of memory frames, the lower the faults

- Surprisingly this is not always the case
- In 1969 Belady, Nelson and Shedler discovered counter example* in FIFO
  - FIFO caused more faults with 4 frames than 3
- This strange situation is now called Belady’s anomaly

* An anomaly in space-time characteristics of certain programs running in a paging machine. Belady, Nelson and Shedler.
Belady’s anomaly: FIFO

Same reference string, different frames

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>4</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Youngest</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Oldest</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Numbers in this color: No page fault

9 page faults with 3 frames

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
<th>4</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Youngest</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Oldest</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10 page faults with 4 frames
Belady’s anomaly

- Led to a whole theory on paging algorithms and properties
- Stack algorithms
The Model

- There is an array \( M \)
  - Keeps track of the state of memory

- \( M \) has as many elements as pages of virtual memory

- Divided into two parts
  - Top part: \( m \) entries \{Pages currently in memory\}
  - Bottom part: \( n-m \) entries
    - Pages that were referenced BUT paged out
The model

Reference String

Tracking the state of the array $M$ over time

Page fault
Properties of the model

- When a page is referenced
  - Move to the top entry of M

- If the referenced page is already in M
  - All pages above it moved down one position
  - Pages below it are not moved

- Transition from within box to outside of it
  - Page eviction from main memory
The model

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

The model is a grid representing a sequence of numbers.
The optimal page replacement algorithm

- The best possible algorithm
- Easy to describe but **impossible to implement**
- **Crux:**
  - Put off unpleasant stuff for as long as possible
- **Idea:** evict “Furthest-in-the-future”
The optimal page replacement algorithm description

- When a page fault occurs some set of pages are in memory
- One of these pages will be referenced next
  - Other pages may be not be referenced until 10, 100 or 1000 instructions later

- **Label** each page with the number of instructions to be executed *before* it will be referenced
  - Page with the highest label should be removed
The Least Recently Used (LRU) page replacement algorithm

- Approximation of the optimal algorithm

- Observation
  - Pages used heavily in the last few instructions
    - Probably will be used heavily in the next few
  - Pages that have not been used
    - Will probably remain unused for a long time

- When a page fault occurs?
  - **Throw out** page that has been *unused the longest*
LRU example: 3 memory frames

|      | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| Recent| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| Least Used | 7 | 0 | 1 | 2 | 2 | 3 | 0 | 4 | 2 | 2 | 0 | 3 | 3 | 1 | 2 | 0 | 1 | 7 |   |   |   |

Reference String
Using Logical clocks to implement LRU

- Each page table entry has a **time-of-use** field
  - Entry updated when page is referenced
    - Contents of clock register are copied

- Replace the page with the smallest value
  - Time increases monotonically
    - **Overflows** must be accounted for

- Requires **search of page table** to find LRU page
Stack based approach

- Keep stack of page numbers
- When page is referenced
  - Move to the top of the stack
- Implemented as a doubly linked list
- No search done for replacement
  - Bottom of the stack is the LRU page
Problems with clock/stack based approaches to LRU replacements

- Inconceivable without hardware support
  - Few systems provide requisite support for true LRU implementations

- Updates of clock fields or stack needed at every memory reference

- If we use interrupts and do software updates of data structures things would be very slow
  - Would slow down every memory reference
    - At least 10 times slower
### Summary of Page Replacement Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>Not implementable, but useful as a benchmark</td>
</tr>
<tr>
<td>NRU (Not Recently Used)</td>
<td>Very crude approximation of LRU</td>
</tr>
<tr>
<td>FIFO (First-In, First-Out)</td>
<td>Might throw out important pages</td>
</tr>
<tr>
<td>Second chance</td>
<td>Big improvement over FIFO</td>
</tr>
<tr>
<td>Clock</td>
<td>Realistic</td>
</tr>
<tr>
<td>LRU (Least Recently Used)</td>
<td>Excellent, but difficult to implement</td>
</tr>
<tr>
<td>NFU (Not Frequently Used)</td>
<td>Fairly crude approximate to LRU</td>
</tr>
<tr>
<td>Aging</td>
<td>Efficient algorithm that approximates LRU well</td>
</tr>
</tbody>
</table>
Page Buffering: Being proactive

- Maintain a list of **modified** pages
- When the paging device is **idle**
  - Write modified pages to disk
- Implications
  - If a page is selected for replacement, *increase likelihood* of that page being clean
Page Buffering: Reuse what you can

- Keep pool of free frames as before
  - BUT remember which pages they held

- Frame contents are not modified when page is written to disk

- If page needs to come back in?
  - Reuse the same frame if it was not used to hold some other page
Buffering and applications

- Applications often understand their memory/disk usage better than the OS
  - Provide their own buffering schemes

- If both the OS and the application were to buffer
  - Twice the I/O is being utilized for a given I/O
Frame allocation: How do you divvy up free memory among processes?

Frame size = 1 MB; Total Size = 128 MB

35 MB for the OS

93 MB for others

128 MB

2 processes at $T_0$
How are frames allocated?

With demand paging all 93 frames would be in the free frame pool
Constraints on frame allocation

- **Max**: Total number of frames in the system
  - Available physical memory

- **Min**: Need to allocate at least a minimum number of frames
  - Defined by the architecture of the underlying system
Minimum number of frames

- As you decrease the number of frames for a process
  - Page fault increases
  - Execution time increases too

- Defined by the **architecture**
  - In some cases instructions and operands (indirect references) straddle page boundaries
    - With 2 operands at least 6 frames needed
Global vs Local Allocation

- **Global replacement**
  - One process can *take* a memory frame from another process

- **Local replacement**
  - Process can only choose from the set of frames that was allocated to it
Local vs Global replacement:
Based on how often a page is referenced

<table>
<thead>
<tr>
<th>Processes A, B and C</th>
<th>Usage Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>10</td>
</tr>
<tr>
<td>A2</td>
<td>7</td>
</tr>
<tr>
<td>A3</td>
<td>5</td>
</tr>
<tr>
<td>A4</td>
<td>3</td>
</tr>
<tr>
<td>B1</td>
<td>9</td>
</tr>
<tr>
<td>B2</td>
<td>4</td>
</tr>
<tr>
<td>B3</td>
<td>2</td>
</tr>
<tr>
<td>B4</td>
<td>6</td>
</tr>
<tr>
<td>C1</td>
<td>3</td>
</tr>
<tr>
<td>C2</td>
<td>5</td>
</tr>
<tr>
<td>C3</td>
<td>6</td>
</tr>
</tbody>
</table>

Local Replacement

<table>
<thead>
<tr>
<th>Processes A, B and C</th>
<th>Usage Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>10</td>
</tr>
<tr>
<td>A2</td>
<td>7</td>
</tr>
<tr>
<td>A3</td>
<td>5</td>
</tr>
<tr>
<td>A4</td>
<td>3</td>
</tr>
<tr>
<td>A5</td>
<td>9</td>
</tr>
<tr>
<td>B1</td>
<td>4</td>
</tr>
<tr>
<td>B2</td>
<td>2</td>
</tr>
<tr>
<td>B4</td>
<td>6</td>
</tr>
<tr>
<td>C1</td>
<td>3</td>
</tr>
<tr>
<td>C2</td>
<td>5</td>
</tr>
<tr>
<td>C3</td>
<td>6</td>
</tr>
</tbody>
</table>

Global Replacement

<table>
<thead>
<tr>
<th>Processes A, B and C</th>
<th>Usage Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>10</td>
</tr>
<tr>
<td>A2</td>
<td>7</td>
</tr>
<tr>
<td>A3</td>
<td>5</td>
</tr>
<tr>
<td>A4</td>
<td>3</td>
</tr>
<tr>
<td>A5</td>
<td>9</td>
</tr>
<tr>
<td>B1</td>
<td>4</td>
</tr>
<tr>
<td>B2</td>
<td>2</td>
</tr>
<tr>
<td>B4</td>
<td>6</td>
</tr>
<tr>
<td>C1</td>
<td>3</td>
</tr>
<tr>
<td>C2</td>
<td>5</td>
</tr>
<tr>
<td>C3</td>
<td>6</td>
</tr>
</tbody>
</table>
## Global vs Local Replacement

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of frames allocated to process</td>
<td>Fixed</td>
<td>Varies dynamically</td>
</tr>
<tr>
<td>Can process control its own fault rate?</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Can it use free frames that are available?</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Increases system throughput?</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>
Locality of References

- During any phase of execution a process references a relatively small **fraction** of its pages.

- Set of pages that a process is currently using
  - **Working set**

- Working set **evolves** during process execution.
Implications of the working set

- If the entire working set is in memory
  - Process will execute without causing many faults
    - Until it moves to another phase of execution

- If the available memory is too small to hold the working set?
  1. Process will cause many faults
  2. Run very slowly
Characterizing the effect of multiprogramming on thrashing

![Graph showing CPU Utilization vs. Degree of Multiprogramming with a peak and thrashing point](image)
Mitigating the effects of thrashing

- Using a local page replacement algorithm
  - One process thrashing does not cause cascading thrashing among other processes
  - BUT if a process is thrashing
    - Average service time for a page fault increases

- Best approach
  1. Track a process’ working set
  2. Make sure the working set is in memory before you let it run
Virtualization

Objectives:

- Explain Virtual Machine Monitors (VMMs)
- Justify the Popek and Goldberg requirements for virtualization
- Explain how Virtualization works in the x86 architecture
- Compare Type-1 and Type-2 Hypervisors
Firms often have multiple, dedicated servers: e-mail, FTP, e-commerce, web, etc.

- **Load**: May be one machine cannot handle all that load
- **Reliability**: Management does not trust the OS to run 24 x 7 without failures
- By putting one server on a separate computer, if one of the server crashes?
  - At least the other ones are not affected
- If someone breaks into the web server, at least sensitive e-mails are still protected
  - **Sandboxing**
But …

- While this approach achieves isolation and fault tolerance
  - This solution is expensive and hard to manage because so many machines are also involved

- Other reasons for having separate machines?
  - Organizations depend on more than one OS for their daily operations
    - Web server on Linux, mail server on Windows, e-commerce server on OS X, other services on various flavors of UNIX
What to do?

- A possible (and popular) solution is to use virtual machine technology

- This sounds very hip and modern
  - But the idea is old … dating back to the 1960s
  - Even so, the way we use it today is definitely new
Main idea

- **VMM** (Virtual Machine Monitor) creates the *illusion* of multiple (virtual) machines on the same physical hardware
  - VMM is also known as a **hypervisor**
    - We will look at type 1 hypervisors (bare metal) and type 2 hypervisors (use services and abstractions offered by an underlying OS)

- **Virtualization** allows a single computer to host multiple virtual machines
  - Each potentially running a different OS
Failure in one virtual machines does not bring down any others

- Different servers run on different virtual machines
  - Maintains **partial-failure** model at a lower cost with easier maintainability

- Also, we can run different OS on the same hardware
  - Benefit from virtual machine isolation in the face of attacks
  - Plus enjoy other good stuff: savings, real estate, etc.
  - Convenient for complex software stack with precise system dependencies
    - Think core libraries
Why virtualization works

- Service outages are due not to faulty hardware, but due to poor software, emphatically including OSes
  - Ill-designed, unreliable, buggy, and poorly configured software
- Migration to another machine may be easier
Why virtualization works

- The only software running in the **highest privilege** is the hypervisor
- Hypervisor has 2 orders of magnitude fewer lines of code than a full operating system
  - Has 2 orders of magnitude fewer bugs
- A hypervisor is simpler than an OS because it **does only one thing**
  - Emulate copies of the bare metal (most commonly the Intel x86 architecture)
Advantages to running software in VMs besides strong isolation

- Few physical machines
  - Saves money on hardware and electricity
  - Takes up less rack space

- For companies such as Amazon or Microsoft
  - Reducing physical demands on data centers represents huge cost savings
  - Companies frequently locate their data centers in the middle of nowhere
    - Just to be close to hydroelectric dams (and cheap energy)
Hypervisors should score well on

- **Safety**
  - Hypervisor should have full control of the virtualized resources

- **Fidelity**
  - Behavior of program on a virtual machine should be identical to the same program running on bare hardware

- **Efficiency**
  - Much of the code in the virtual machine should run *without intervention* from the hypervisor
Safety

- Consider each instruction in turn in an interpreter (such as Bochs) and perform exactly what is needed
  - May execute some instructions (INC) as is, but other instructions must be simulated

- We cannot allow the guest OS to disable interrupts for the entire machine or modify page-table mappings
  - Trick is to make the guest OS believe that it has

- Interpreter may be safe, even hi-fi, but performance is abysmal
  - So, VMMs try to execute most code directly
Virtualization has long been a problem on x86
- Defects in 386 carried forward into new CPUs for 20 years in the name of backward compatibility

Every CPU with kernel mode and user mode has instructions that behave differently
- Depending on whether it is executed in kernel/user mode
  - Sensitive instructions
  - Some instructions cause a trap when executed in user-mode
  - Privileged instructions

A machine is virtualizable only if sensitive instructions are a subset of privileged instructions
If you do something in user mode that you should not:
- The hardware should trap!
- IBM/370 had this property, Intel’s 386 did not

Several sensitive 386 instructions were ignored if executed in user mode:
- Or executed with a different behavior
- E.g. POPF instruction replaces flags register which changes the bit that enables/disables interrupts
  - In user-mode this bit was simply not changed

Also, some instructions could read sensitive state in user mode without causing a trap.
Full virtualization

- Trap all instructions
- Fully simulate entire computer
- Trade-off: High overhead
- Benefit: Can virtualize any OS
Paravirtualization

- Never aims to present a virtual machine that looks just like the actual underlying hardware
- Present **machine-line software interface** that explicitly exposes that it is a virtualized environment
  - Offers a set of **hypercalls** that allow the guest to send explicit requests to the hypervisor
    - Similar to how a system call offers kernel services to applications
- **Drawback:** Guest OS has to be aware of the virtual machine API
Paravirtualization [2/2]

- Guests use hypercalls for privileged, sensitive operations like updating page tables
  - But they do it in cooperation with the hypervisor
  - Overall system can be simpler and faster

- Paravirtualization was offered by IBM since 1972

- Idea was revived by Denali (2002) and Xen (2003) hypervisors
Terms

- **Guest Operating System**
  - The OS running on top of the hypervisor

- **Host Operating System**
  - For a type 2 hypervisor: the OS that runs on the hardware

- **Safe executions**
  - Execute the machine’s instruction set in a safe manner
  - Guest OSes may change or mess up its own page tables … but not those of others
Type 1 hypervisor

- Only program running in the most privileged mode
- Support multiple copies of the actual hardware
  - Virtual machines
  - Similar to processes a normal OS would run
Location of Type-1 hypervisor

Hardware
(CPU, disk, network, interrupts, etc)

Type 1 hypervisor

Control Domain

Linux

Excel
Word

Windows

Emacs
Type 2 hypervisor

- Also referred to a **hosted hypervisor**
- Relies on a host OS, say Windows or Linux, to allocate and schedule resources
- Still pretends to be a full computer with a CPU and other devices
Type 2: Running Guest OS

- When it starts for the first time, acts like a newly booted computer
  - Expects to find a DVD, USB drive or CD-ROM containing an OS
    - The drive could be a virtual device
    - Store the image as an ISO file on the hard drive and have hypervisor pretend its reading from proper DVD drive
- Hypervisor installs the OS to its virtual disk (just a file) by running installation that it found on DVD
- Once guest OS is installed on virtual disk, it can be booted and run
Location of Type-2 hypervisor

- **Guest OS Processes**
  - Example: Windows

- **Host OS Process**
  - Example: Linux

- **Type 2 hypervisor**

- **Host OS**
  - Example: Linux

- **Hardware**
  - (CPU, disk, network, interrupts, etc.)
### Examples of hypervisors [Partial List]

<table>
<thead>
<tr>
<th>Virtualization Method</th>
<th>Type 1 hypervisor</th>
<th>Type 2 hypervisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtualization without hardware support</td>
<td>ESX Server 1.0</td>
<td>VMware workstation 1.0</td>
</tr>
<tr>
<td>Paravirtualization</td>
<td>Xen 1.0</td>
<td></td>
</tr>
<tr>
<td>Virtualization with hardware support</td>
<td>vSphere, Xen, Hyper-V</td>
<td>VMware Fusion, KVM, Parallels</td>
</tr>
<tr>
<td>Process Virtualization</td>
<td></td>
<td>WINE</td>
</tr>
</tbody>
</table>
Type-1 hypervisors

- Virtual machine runs as a user-process in user mode
  - Not allowed to execute sensitive instructions (in the Popek-Goldberg sense)

- But the virtual machine runs a **Guest OS that thinks** it is in kernel mode (although, of course, it is not)
  - **Virtual kernel mode**

- The virtual machine also runs user processes, which think they are in the user mode
  - And really are in user mode
Modes

- **Hardware**
  - **Type 1 hypervisor**
    - Trap on privileged instruction
  - **Guest Operating System**
  - **Virtual kernel mode**
  - **Virtual user mode**
  - **User mode**
  - **Kernel Mode**

- **User processes**
Execution of kernel model instructions

- What if the Guest OS executes an instruction that is allowed only when the CPU is really in kernel mode?
  - On CPUs without VT (Intel: Virtualization Technology)?
    - Instruction fails and the OS crashes
  - On CPUs with VT?
    - A trap to the hypervisor does occur
      - Hypervisor can inspect instruction to see if it was issued:
        - By Guest OS: Arrange for the instruction to be carried out
        - By user-process in that VM: Emulate what hardware would do when confronted with sensitive instruction executed in user-mode
What if the guest is running and an interrupt arrives from an external device?

- Type 2 hypervisor depends on host’s device drivers to handle to the interrupt
- So, the hypervisor **reconfigures hardware** to run the host OS system code
  - When the device driver runs, it finds everything just as it expected it to be
- Hypervisor behaves just like teenagers throwing a party when parents are away
  - It’s OK to rearrange furniture completely, as long as they put it back as they found it before parents get home
Why do hypervisors work even on unvirtualizable hardware?

- Sensitive instructions in the guest kernel replaced by calls to procedures that **emulate** these instructions

- No sensitive instructions issued by the guest OS are ever executed directly by true hardware
  - Turned into calls to the hypervisor, which emulates them
Cost of virtualization

- We expect CPUs with VT would greatly outperform software techniques
- Trap-and-emulate approach used by VT hardware generates a lot of traps … and these are expensive
  - Ruin CPU caches, TLBs, and branch predictions
- In contrast, when sensitive instructions are replaced by calls to hypervisor procedures
  - None of this context-switching overhead is incurred
True virtualization & paravirtualization

- **True virtualization**
  - Unmodified Windows
  - Trap due to sensitive instruction

- **Paravirtualization**
  - Modified Linux
  - Trap due to hypervisor call

- **Type 1 hypervisor**
- **Microkernel**
- **Hardware**
x86 privilege level architecture without virtualization

- **Ring 3**: User Apps
- **Ring 2**: Direct execution of User and OS Requests
- **Ring 1**: 
- **Ring 0**: OS

**Host Computer System Hardware**
Full Virtualization: Binary translation approach to x86 virtualization

- Direct execution of User and OS Requests
- Binary Translation of OS Requests
Paravirtualization approach to x86 virtualization

Host Computer System Hardware

Virtualization Layer

Paravirtualized Guest OS

User Apps

Ring 0

Ring 1

Ring 2

Ring 3

Direct execution of User and OS Requests

“Hypercalls” to the Virtualization Layer replace non-virtualizable OS instructions

Paravirtualized Guest OS

User Apps

Host Computer System Hardware

CS370: Operating Systems
Dept. Of Computer Science, Colorado State University
Hardware assisted virtualization

- **Root Mode Privilege Levels**
  - Ring 0
  - Ring 1
  - Ring 2
  - Ring 3

- **Host Computer System Hardware**
  - Direct execution of User and OS Requests
  - OS Requests trap to VMM without Binary Translation or Paravirtualization

- **VMM**
  - **Guest OS**
  - **User Apps**
Contrasting the virtualization approaches

<table>
<thead>
<tr>
<th>Technique</th>
<th>Full virtualization with Binary Translation</th>
<th>Hardware Assisted Virtualization</th>
<th>OS Assisted Virtualization/Para virtualization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technique</td>
<td>Binary Translation and Direct Execution</td>
<td>Exit to Root Mode on privileged instructions</td>
<td>Hypercalls</td>
</tr>
<tr>
<td>Guest Modification/Compatibility</td>
<td>Unmodified Guest OS</td>
<td>Unmodified Guest OS</td>
<td>GuestOS codified to issue Hypercalls so it can’t run on native hardware.</td>
</tr>
<tr>
<td>Compatibility</td>
<td>Excellent compatibility</td>
<td>Excellent compatibility</td>
<td>Compatibility is lacking</td>
</tr>
</tbody>
</table>

- **Full virtualization with Binary Translation**
  - Technique: Binary Translation and Direct Execution
  - Guest OS: Unmodified Guest OS
  - Excellent compatibility

- **Hardware Assisted Virtualization**
  - Technique: Exit to Root Mode on privileged instructions
  - Guest OS: Unmodified Guest OS
  - Excellent compatibility

- **OS Assisted Virtualization/Para virtualization**
  - Technique: Hypercalls
  - Guest OS: GuestOS codified to issue Hypercalls so it can’t run on native hardware.
  - Compatibility is lacking
Installing application software

- VMs offer a solution to a problem that has long plagued users (especially open source)
  - How to install application programs

- Applications are dependent on numerous other applications and libraries
  - Which themselves depend on a host of software packages

- Plus there are dependencies on particular versions of compilers, scripting languages, OS etc.
With VMs ...

- Developer can carefully **construct** a virtual machine
  - Load it with required OS, compiler, libraries, and application code
  - **Freeze the entire unit** … ready to run

- Only the software developer has to understand the dependencies
Licensing Issues

- Some software is licensed on a per-CPU basis
  - Especially, software for companies
  - When they buy a program they have the right to run it on just one CPU
    - What is a CPU anyway?
    - Can we run multiple VMs all running on the same physical hardware?

- Problem is even worse, when companies have licenses for \( N \) machines running the software
  - VMs come and go on demand
File Systems

Objectives:

- Summarize file system structure
- Contrast contiguous allocation vs indexed allocations
- Explain the Unix File System
- Explain and contrast Windows File Systems: the File Allocation table and NTFS
Files are an abstraction mechanism

- Provide a way to store information and read it back later

- Do this in a way that shields the user from
  - How and where information is stored on disk
  - How disks really work
Files can be structured in many ways:
Unstructured sequence of bytes

- The OS does not know or care what is in the file
  - Maximum **flexibility**
- OS does not help, but does not get in the way either
- Meaning is imposed by programs
- Most OS support this
Mounting file systems

- Many systems have more than one disk
  - How do you handle them?

- **S1:** Keep self contained file system on each disk
  - And keep them separate

- **S2:** Allow one disk to be mounted in another disk’s file tree
Mounting file systems

```
cp D:/x /a/d/x
H is default

cp /b/x /a/d/x
```
Checks performed during mounting

- OS **verifies** that the device contains a valid file system

- Read device directory
  - Make sure that the format is an expected one

- Windows mounting
  - Each device in a separate name space
  - {Letter followed by a colon e.g. \texttt{G:}}
There are many levels that comprise a file system
I/O Control consists of device drivers

- Transfers information *between main memory and disk*

- Receives *high-level* commands
  - Retrieve block 123, etc

- Outputs low-level, hardware-specific instructions
  - Used by the hardware controller
  - Writes bit patterns into specific locations of the I/O controller
There are many levels that comprise a file system
Basic file system issues commands to the device driver

- Read and write physical blocks on disk
  - E.g. Drive 1, cylinder 73, sector 10

- Manages **buffers and caches**
  1. To hold file system, directory and data blocks
  2. Improves performance
There are many levels that comprise a file system
File organization module

- Knows about files
  - Logical and physical blocks

- **Translate** logical addresses to physical ones
  - Needed for every block

- Includes a **free space manager**
  - Tracks unallocated blocks and allocates as needed
There are many levels that comprise a file system

- Application Programs
  - Logical File System
    - File Organization Module
      - Basic File System
        - I/O Control
          - Devices
The logical file system

- Manages **metadata** information
  - Metadata is *data describing the data*

- Maintains file structure via **file control blocks**
  - Info about the file
    - Ownership and permissions
    - Location of file contents
  - **inode** in UNIX file systems
Several file systems are in use

- CD-ROMs written in ISO 9660 format
  - Designed by CD manufacturers
- UNIX
  - Unix file system (UFS)
  - Berkley Fast File System (FFS)
- Windows: FAT, FAT32 and NTFS
- Linux
  - Supports 40 different file systems
  - Extended file system: ext2, ext3 and ext4
On-disk structures used to implement a file system

1. **Boot control block**
   - Information needed to boot an OS from that volume

2. **Volume control block**: Volume information
   - Number of blocks in the partition
   - Size of the blocks
   - Free-block count/pointers
   - Free file-control-block count/pointers
   - UFS: super-block
   - Windows: Master file table
On-disk structures used to implement a file system (2)

- Directory structure to organize files
  - One per file system

- Per file file-control-block
  - Contains details about individual files
In memory structures used to improve performance via caching

- **Mount table**
  - Information about each mounted volume

- Directory structure **cache**
  - Holds information about recently accessed directories

- System-wide **open file** table
  - File control block of each open file

- **Buffers** to hold file-system blocks
  - To read and write to storage
Creation of a new file

- **Allocate** a file-control block (FCB)

- **Read** appropriate directory into memory
  - Directory is just a file in UNIX
    - Special **type** field

- **Update** directory with new file name and FCB

- **Write** directory back to disk
Directory implementation:
Hash table

- Linear list and a hash table is maintained
- Key computed from file name
  - Hash table value returns pointer to entry in linear list
- Things to consider
  1. Account for collisions in the hash space
  2. Need to rehash the hash table when the number of entries exceed the limit
Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk
  - If file is of size $n$ blocks and starts at location $b$
    - Occupies blocks $b, b+1, \ldots, b+n-1$

- Disk head movements
  - None for moving from block $b$ to $(b+1)$
  - Only when moving to a different track
Sequential and direct access in contiguous allocations

- Sequential accesses
  - Remember *disk address* of the last referenced block
  - When needed, read the next block

- **Direct access** to block \( i \) of file that starts at block \( b \)
  \[ b + i \]
Contiguous allocations suffer from external fragmentation

- Free space is broken up into chunks
  - Space is **fragmented** into holes

- Largest continuous chunk may be insufficient for meeting request

- **Compaction** is very slow on large disks
  - Needs several hours
Linked Allocation: Each file is a linked list of disk blocks

File A

- File block 0: 4
- File block 1: 7
- File block 2: 2
- File block 3: 10
- File block 4: 12

File B

- File block 0: 6
- File block 1: 3
- File block 2: 11
- File block 3: 14

Pointer to next block
Linked List Allocations:
Advantages

- **Every** disk block can be used
  - No space is lost in external fragmentation

- Sufficient for directory entry to merely store *disk address of first block*
  - Rest can be found starting there
Linked List Allocation:
Disadvantages

- Used effectively only for sequential accesses
  - Extremely slow random access

- Space in each block set aside for pointers
  - Each file requires slightly more space

- Reliability
  - What if a pointer is lost or damaged?
Linked list allocation: Take pointers from disk block and put in table

Table tracks **EVERY** disk block in the system
Linked list allocation using an index

- **Entire** disk block is available for data
- Random access is much easier
  - Chain must still be followed
    - But this chain could be cached in memory
- **MS-DOS and OS/2** operating systems
  - Use such a file allocation table (FAT)
CS 370: Operating Systems

[File Systems]

Computer Science
Colorado State University

Instructor: Louis-Noel Pouchet
Spring 2024

** Lecture slides created by: Shrideep Pallickara
inode

- **Fixed-length** data structure
  - One per file

- Contains information about
  - File **attributes**
    - Size, owner, creation/modification time etc.
  - **Disk addresses**
    - File blocks that comprise file
The inode is used to encapsulate information about a large number of file blocks.

For e.g.

- Block size = 8 KB, and file size = 8 GB
- There would be a million file-blocks
  - Inode will store info about the **pointers to these blocks**
- Inode allows us to access info for all these blocks
  - Storing pointers to these file blocks also takes up storage
Managing information about data blocks in the inode

- First few data blocks of the file stored in the inode
- If the file is large: **Indirect** pointer
  - To a block of pointers that point to additional data blocks
- If the file is larger: **Double indirect** pointer
  - Pointer to a block of indirect pointers
- If the file is huge: **Triple indirect** pointer
  - Pointer to a block of double-indirect pointers
Schematic structure of the inode

- File Attributes:
  - Size (bytes)
  - Owner UID/GID
  - Relevant times
  - Link and Block counts
  - Permissions

- Direct pointers to first few file blocks
- Single indirect pointer
- Double indirect pointer
- Triple indirect pointer

Address of disk block

Pointers to next file blocks
i-Node: How the pointers to the file blocks are organized

- Single indirect block
- Double indirect block
- Triple indirect block

Attributes
Disk Layout in traditional UNIX systems

An integral number of inodes fit in a single data block
Super Block describes the state of the file system

- Total size of partition
- Block size and number of disk blocks
- Number of inodes
- List of free blocks
- Inode number of the root directory

- Destruction of super block?
  - Will render file system unreadable
A linear array of inodes follows the data block

- Inodes are numbered from 1 to some max
- Each inode is identified by its inode number
  - Inode number contains info needed to locate inode on the disk
  - Users think of files as filenames
  - UNIX thinks of files in terms of inodes
UNIX directory structure

- Contains only file names and the corresponding inode numbers

- Use `ls -i` to retrieve inode numbers of the files in the directory
Directory entry, inode and data block for a simple file

inode 12345

File name

dir_entry

Block 23567

Fragment of the text in the file
Two hard links to the same file

Directory entry in /dirA

- i-node: 12345
- File name: name1

inode 12345

Directory entry in /dirB

- i-node: 12345
- File name: name2

Block 23567

Fragment of the text in the file
File with a symbolic link

Directory entry in /dirA

i-node 12345

File name name1

inode 12345

Block 23567

Fragment of the text in the file

Directory entry in /dirB

i-node 13579

File name name2

inode 13579

"/dirA/name1"

Block 15213
Limitations of a file system based on inodes

- File **must fit** in a single disk partition
- Partition size and number of files are **fixed** when system is set up
Memory mapped files

- open(), read(), write()
  - Requires system calls and disk access

- Allow part of the virtual address space to be logically associated with the file
  - Memory mapping
Memory-mapping maps a disk block to a page (or pages) in memory

- Manipulate files through memory
  - Multiple processes may map file concurrently
    - Enables data sharing
  - Since JVM 1.4, Java supports memory-mapped files
    - FileChannel

- Writes to files in memory are not necessarily immediate