# Operating System

The **Operating System** is a "special program" that allows the isolation of the hardware from the programs that run on the computer. It provides:

- Memory management
- Disk management
- Peripherals management (keyboard, mouse, graphics card)
- Management of users and programs, protecting the entire system
- ... everything so that the programmer doesn't have to do it

OS is a control program

- Controls execution of programs to prevent errors and improper use of the computer

OS is a resource manager

- Manages all resources
  - Process management
  - Memory management
  - File-system management
  - Mass-storage management
  - I/O management
- Decides between conflicting requests for efficient and fair resource use

Usually, a general purpose OS includes:

- Kernel (the OS part that is always running in the machine)
- Middleware frameworks (ease application development)
- System programs that help to manage the system

# What Do

Sharing resources among applications

- scheduling
- allocation

Making efficient use of limited resources

- improving utilization
- minimizing overhead
- improving throughput/goodput

Protecting applications from each other

- enforcement of boundaries

# Classification

Interface

- CLI (Command-line interface)
- Batch interface (commands are entered in files)
- GUI (Graphical User Interface)

Users

- Multi-user
- Mono-user

Tasks

- Single-task - can support only a single task at any time; tasks are handled sequentially (e.g., MS-DOS)
- Multi-task - executes more than one task at a time, by sharing resources

Real-time Operating Systems (RTOS)

- Operating systems that have strict time bounds and have a predictable behavior (e.g., VxWorks, VRTX, pSOS, LynxOS, ...)

Open/Closed Operating Systems:

- Proprietary
- Open systems
  - Have standardized programming interfaces and peripheral interconnects
- Open-source systems (GPL license)
- Operating systems made available in source-code format rather than just binary closed-source
- Started by Free Software Foundation (FSF), which has "copyleft" GNU Public License (GPL)
  - (GNU = GNU's Not Unix!)
- Examples include GNU/Linux and BSD UNIX (included in the core of Mac OS X), and many more

How does the hw know where the kernel is and how to load it?

Starting a computer and loading the kernel is called booting.

Most computers have a multistage boot

## Startup

Power up the computer (IBM - compatible PC)

- Test operation (power-up diagnostics) :
  - In the power-up (and at reset) the CPU executes a jump instruction to a pre-defined address where an initial boot loader located in nonvolatile firmware (BIOS or UEFI) is run;
    - BIOS = Basic Input Output System
    - UEFI = Unified Extensible Firmware Interface
  - The initial boot loader program executes tests for verifying the correct operation of the CPU (subset of instructions) and RAM (basic tests)
    - Power-on self-test, also known as POST
  - If OK , a second boot loader is started ...
- Sequence startup (boot) OS :
- A second boot loader is started, normally from the first sector of the boot device. This second boot loader may be able to start the OS, but normally it only knows the location of the remainder of the **bootstrap loader** program, which in turn locates the kernel, loads it into main memory, and starts its execution. .
- Common bootstrap loader for Linux, GRUB, allows selection of kernel from multiple disks, versions, kernel options

# Initialization

OS performs its boot "routines":

- hardware - Initialization of registers of the controllers , the system interrupts, the interrupt vector ;
- software - Initialization of data structures that represent the various system resources and algorithms that support the management of these resources .

Creation of the first cases :

- The system auxiliary processes
  - login , file system , network ...
- the interpreter commands
  - command line ( sh , bash, csh, command.com ) or windows

## Wait for interrupt

After booting, the OS is "quiet" waiting for work and ... there is a hardware interrupt

- OS performs a task that handles the interruption .

The OS is " quiet" and ... a program requests a service - through a software interrupt or trap:

- OS triggers a task that runs the service (this interruption)

# Protection provided by the OS

Dual Mode of Operation

- "User Mode" and "Kernel Mode"
- In "User Mode" only common instructions can be performed (e.g. arithmetic and logic). In "Kernel Mode" anything can be done.
- A mode bit contains the current mode of operation
- RTI: Return from interruption
- Kernel mode assumed on traps and on interrupts

I/O Protection

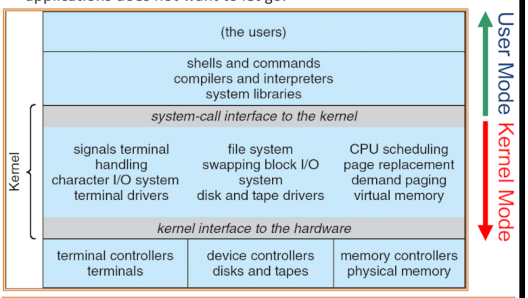
- All I/O Instructions are privileged.
  For doing I/O a trap into the operating system must be generated.

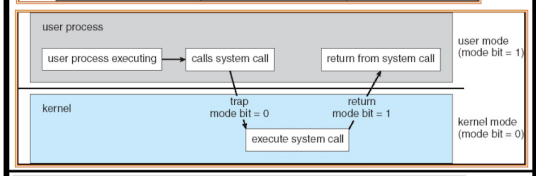
Memory Protection

- Each program can only access its memory

CPU Protection

- The CPU must remain in control of the Operating System, even if the applications does not want to let go.




# Input/Output versus CPU Usage

I/O devices and the CPU can execute concurrently

Each device controller oversees a particular device

Each device controller has a local buffer

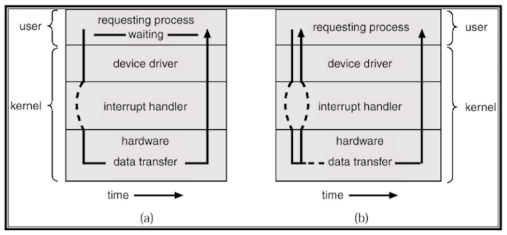
CPU moves data from/to main memory to/from local buffers

I/O is from the device to local buffer of controller

Device controller informs CPU that it has finished its operation by causing an interrupt

Synchronous (a) versus asynchronous I/O (b)

- In (a) the application blocks on the system call
- In (b) the application resumes execution after issuing the system call and later receives the answer




# Device Status Table

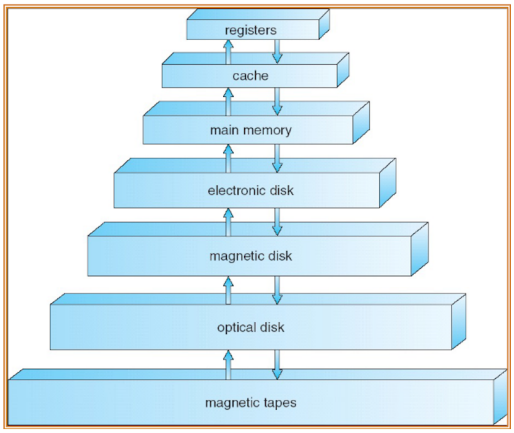
Each table entry indicates the device's type, address, and state (not functioning, idle, or busy). If the device is busy with a request, the type of request and other parameters will be stored in the table entry for that device.

If the kernel supports asynchronous I/O, it also keeps track of many I/O requests at the same time.

# Direct Memory Access (DMA)

- Used for high-speed I/O devices able to transmit information at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Implies the capability of arbitrating the system bus
- Only one interrupt is generated per block, rather than one interrupt per byte

# Storage Hierarchy




A great part of this hierarchy is controlled by the Operating System

- Cache Management Policies
- Buffer Management Policies
- Memory Management Policies

Principles of Locality:
- Temporal Locality
- Spatial Locality

| Level | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Name | registers | cache | main memory | disk storage |
| Typical size | < 1 KB | > 16 MB | > 16 GB | > 100 GB |
| Implementation technology | custom memory with multiple ports, CMOS | on-chip or off-chip CMOS SRAM | CMOS DRAM | magnetic disk |
| Access time (ns) | 0.25 – 0.5 | 0.5 – 25 | 80 – 250 | 5,000.000 |
| Bandwidth (MB/sec) | 20,000 – 100,000 | 5000 – 10,000 | 1000 – 5000 | 20 – 150 |
| Managed by | compiler | hardware | operating system | operating system |
| Backed by | cache | main memory | disk | CD or tape |

# Memory Protection

Each program must only be able to access its own memory

- Segmentation
- Virtual Memory

Segmentation is based on having a base pointer and a limit for each process running

- Any access made outside of its "space" generates a trap and normally leads to the process being killed

Virtual Memory is based on having a table that translates "virtual addresses" into real ones

- Normally, there is such a table for each process
- Each process sees all the address space
- An access made to an non existing page generates a trap and normally leads to the process being killed

# Multiprogramming

Multiprogramming: maintaining several jobs in memory, active at the same time, so that when the current program can no longer run (for example because it's blocked waiting for I/O), another is available to run

Optimizes resource utilization: CPU and I/O

# Multitasking

Multitasking is an extension of multiprogramming in which the execution of the active programs is time-sliced:

- Each program runs for a short period of time, then another is run.
- When a program is running and is forcibly replaced by another, typically with a higher priority, it is said to have been preempted, thus the term Preemptive Operating Systems

For implementing multitasking, a non-maskable interrupt must connected to a hardware timer

- Every few milliseconds (e.g. 100ms), the interrupt causes a task (or process) switch

# Process

Program = Binary Image (Executable) (passive entity)

Process = The living "image" of a program running

- It includes information such as :
  - A unique identifier; Program Counter; Owner; Security attributes;
  - I/O status information (e.g., open files)
  - Allocated Memory / Address Space

## The "Process Memory Image" in Unix

text → Where the code of the program goes
- Consists of the machine instructions that CPU executes
- Usually shareable (so that only a copy needs to be in memory, for frequently executed programs)
- Often read-only, to prevent modifications to instructions
- Called .text segment

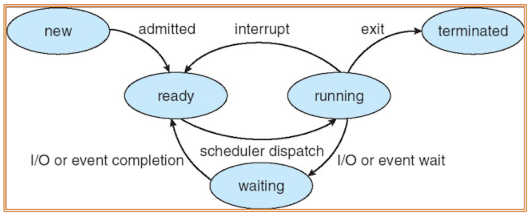
data → Where all variables are
- .data → global and static variables, initialized to non-zero
  - This is the initialized data segment
  - Contains global and static variables specifically initialized by the program
- .bss → global and static variables, non-initialized or initialized to 0
  - .bss aka Uninitialized data segment
  - Data in this segment is initialized by the kernel, to 0 or null before the program starts executing

heap → Where all dynamically allocated memory is set
- e.g., as a result of malloc()

stack → Where all automatic variables exist
- Variables that are automatically created and destroyed in methods
  - Including return address of functions, machine registers, etc.
- It grows down (higher to lower addresses)

**Note:** Contents of uninitialized data segment are not stored in the program file on disk, because the kernel sets them to 0 before the program starts running. Only text segment and the initialized data need to be saved in the program file.

## Process States




"**Waiting**" is also known as "**Blocked**"

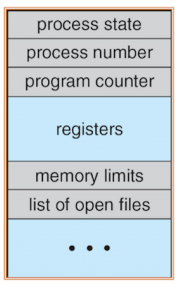
Processes in the Ready and Blocked states do not consume CPU!

This is very important!

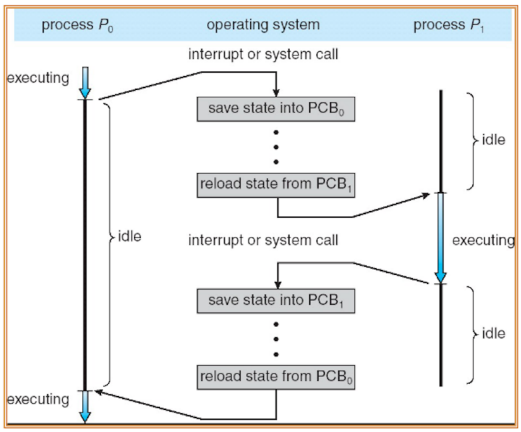
## Process Control Block (PCB)

One of the ***most*** important data structures of the OS
- Represents a process in the Operating System

The **PCB** includes:
- Process state
- Process identifier
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information
- List of open files
- ...




## Context Switch from Process to Process

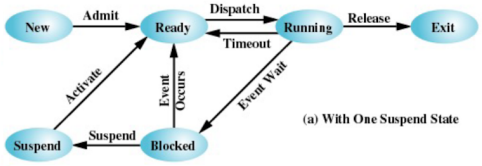


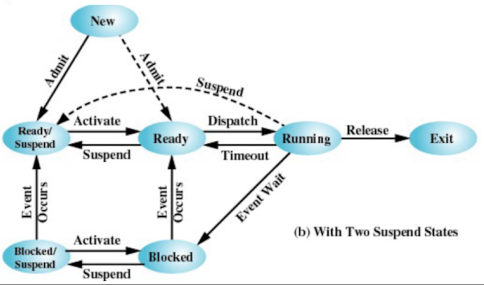

## Process Queues

- Processes migrate among the various queues...
- **Job queue**
  - Set of all processes in the system
- **Ready queue**
  - Set of all processes residing in main memory, ready and waiting to execute
- **Device queues**
  - Set of processes waiting for an I/O device (also known as blocked queues)

## Suspended States

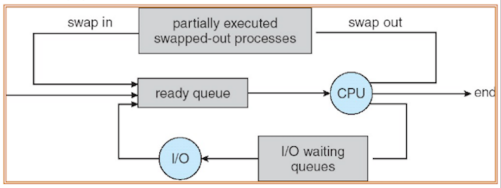
In some operating systems, additional states are introduced for describing processes that have been **swapped** to disk




## Scheduling with Suspended States

In systems with suspended states it is common to have:
- A long term (or medium term) scheduler
  - ... selects which processes should be brought from disk into the ready queue
- A short-term (or CPU) scheduler
  - ... selects which process should be executed next (from the ones in the ready-queue) and allocates CPU




The schedulers' names suggest the relative frequency with which their functions are executed
- The **long-term**, or admission scheduler, decides which jobs or processes are to be admitted to the ready queue
  - Sets the degree of concurrency to be supported at any one time
- The **mid-term** scheduler temporarily moves processes from main memory and places them on secondary memory and vice versa
- The **short-term** scheduler decides which of the ready, in-memory processes is to be executed

## Types of Processes

**I/O Bound**
- Spend more time doing I/O than computation (Lives mostly in the blocked queue)

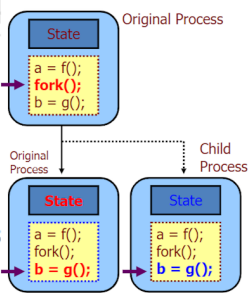
**CPU Bound**
- Spend more time doing computation than I/O (Live mostly in the ready queue)

For a good utilization of computer resources it's important to have a good mix of I/O bounded and CPU bounded processes: the long-term scheduler is responsible for this

## Process Model in UNIX

Process creation in Unix is based on spawning child processes which inherit all the characteristics of their fathers
- Variables, program counter, open files, etc.
- **Spawning a process is done using the fork() system call**

After forking, each process will be executing having different variables and different state.
- The Program Counter will be pointing to the next instruction
- Changing a variable in the child program does not affect its father (and vice-versa)




## Different Heritance Models...

The way resources are shared between parent and child processes varies widely among operating systems...

Resource sharing
- Parent and children share all resources
- Children share subset of parent's resources
- Parent and children share no resources

Execution
- Parent and children execute concurrently
- Parent waits until children terminate
- Children terminates if parent terminates (cascading termination)

## Copy-on-Write

Nowadays, the memory of a process is organized in **pages** of typically 4KB

After a **fork(),** the child process and the parent process share the same pages

The operating system detects when a page is being written either by the parent process or the child process. When that happens, a copy is made on demand, before the write is allowed to proceed.

This is called **copy-on-write**

Thus, changing a variable on a child process does not affect its parent and vice-versa

## Shared Address Space with the Kernel

For making system calls and traps fast, it is possible to jump directly to the kernel handler routine without remapping any memory
- The address space of each process is divided into two parts:
  - One that is specific of that process
  - One that corresponds to the kernel and is shared by all processes

How does it work?
- The user process does not have direct access to the kernel memory; it cannot read nor write that part of the address space
- Whenever a trap occurs, it enters in "kernel mode" and thus has access to the already mapped memory

## Process Termination in UNIX

A process is only truly **eliminated** by the operating system when its father calls **wait()/waitpid()** on it.
- This allows the parent to check things like the exit code of its sons

**Zombie Process**: One that has died and its parent has not acknowledged its death (by calling wait())
- Be careful with this if your are designing servers. They are eating up resources!!

**Orphan Process**: One whose original parent has died. In that case, its parent becomes init (process 1).
- Linux follows the same procedure. However, in current versions of the kernel not all orphans are adopted by init.

## Reasons for Process Termination

- Normal completion
- Time limit exceeded
- Memory unavailable
- Bounds violation
- Protection error
- I/O failure
- Invalid instruction
- Parent request
- Operating system intervention (e.g. on deadlock)
- Parent terminates so child processes terminate
- ...

# THREADS

Process switching is an extremely heavy operation
- It's necessary to remap address spaces
- It's necessary to establish new security contexts
- It's necessary to manage all information associated to processes

Thread
- It's a flow of execution inside a program
- The address space is the same (...changing a global variable in a thread affects all others)
- No context switching between threads
- Commutation among threads is very fast
- Communication among threads is easy to do and fast

They are very lightweight compared to processes
- Light context switches
- Fast to create and terminate
- Fast to synchronize
- Fast to communicate among threads
- Very useful in multiprocessor/multi-core architectures
- Economy or resources

Much easier to program than shared memory!
- Everything is already shared
- Be careful to synchronize accesses!

Threads share the same address space and resources! Changing one thing in one thread affects all others!

# Multithreading Models

**User threads (N-to-1 model)**

- Threads are implemented in a library at user space.
- The kernel is completely unaware of the existence of threads

**Kernel threads (1-to-1 model)**

- Threads are implemented exclusively in kernel space
- The kernel does all the scheduling of threads

**The Many-to-Many Model**

- A number of kernel threads can map to a different number of user threads

**User Level Threads**

- ☺ Commutation among threads is fast since it does not imply a mode switch
- ☺ Thread creation and termination are fast
- ☺ Does not require any special support by the kernel, thus being available in any OS
- ☹ If one thread blocks all threads block (e.g. on I/O)
- ☹ Only one system call can happen at a time
- ☹ Scheduling is not fair
- ☹ One thread terminates all threads terminate

**Kernel Threads**

- ☹ Commutation among threads is slower since it implies going to the kernel
- ☹ Thread creation and termination is slower
- ☹ Requires that the kernel supports user visible threads
- ☺ If one thread blocks the others can continue
- ☺ Simultaneous system calls
- ☺ Fair scheduling
- ☺ Somewhat independent thread termination

## Thread Pools

Since most operating systems use a 1-to-1 model, it is important not to waste much time creating and destroying threads


- Some threads are previously created and BLOCKED waiting for work
- Whenever work appears, a managing thread wakes one thread in the pool and assigns it the work to be done
- After the thread completes the work, it returns to the pool waiting for more work

# Synchronization

Multiprogramming
- One CPU shared by several jobs or processes

Multiprocessing
- Several processors working together
  - Includes:
    - single computers with multiple cores
    - linked computing systems that share processing

## Parallel Processing

One form of multiprocessing
- When >=2 processors work together

Benefits
- Increase in reliability
  - If one processor fails, then the others can continue to operate and absorb the load – needs careful design!
- Faster Processing
  - Instructions can be executed in parallel

More flexibility => More complexity

| Parallelism Level | Process Assignments | Synchronization Required |
|---|---|---|
| Job level | Each job has its own processor and all processes and threads are run by that same processor. | No explicit synchronization required. |
| Process Level | Unrelated processes, regardless of job, are assigned to any available processor. | Moderate amount of synchronization required to track processes. |
| Thread Level | Threads are assigned to available processors. | High degree of synchronization required, often requiring explicit instructions from the programmer. |

# Multiprocessing configurations

## Master/slave

Asymmetric

One master processor controls slave processors

- The master is responsible for managing the entire system (files, devices, processors, etc.)

Well suited for environments where processing time is divided between backend and frontend processing

Disadvantages
- Poor reliability - if master fails, all fails
- Poor resource use (slaves wait for master to get work)
- Many interrupts as slaves interrupt master whenever they need OS intervention (e.g. I/O)

Advantages
- Simplicity

## Loosely Coupled

Each processor controls its own resources: files, memory, I/O devices

Not completely independent systems because processors communicate and cooperate with others

Jobs stick to their assigned processor
- Processor assignment to ensure well balanced use of resources

Not prone to catastrophic failures as the failure of one processors does not imply the fail of the all system

## Symmetric

More reliable than loosely coupled configurations, more effective using resources, good load balancing, degrades better in the event of a failures
- Prone to races and deadlocks
- Requires **synchronization!**

# CONCURRENCY & SYNCHRONIZATION

## Buying milk - unsynchronized

| Roomate A | | Roomate B |
|---|---|---|
| 3:00 | Arrive home: no milk | |
| 3:05 | Leave for store | |
| 3:10 | Arrive at store | Arrive home: no milk |
| 3:15 | Leave store | Leave for store |
| 3:20 | Arrive home, put milk away | Arrive at store |
| 3:25 | | Leave store |
| 3:30 | | Arrive home: too much milk! |

**Race condition:** The situation where several processes access and manipulate shared data concurrently. The result critically depends on the timing of these processes (on the order of execution of instructions in the multiple processes), which are "racing".

## Critical Regions

Processes within a critical region cannot be interleaved without threatening integrity

Synchronization can be achieved through lock-and-key arrangements

## Locking mechanisms

### Test-and-Set (TS)

Single, indivisible machine instruction

Introduced in IBM System 360/370 computers

TS semantics
- Saves the key in a storage location – the key is 0 (resource represented by the key is free) or 1 (it is busy)
- If the key is 0 changes it to 1 and returns free to the process that issued the instruction; the process is allowed to proceed
- If the key value is 1 (resource busy), the process awaits in cycle (!)

Can cause starvation
- A process might never get the resource if many are testing to enter the critical region!

Can cause busy waiting
- The process keeps testing the value!

### WAIT and SIGNAL

TS without busy waiting

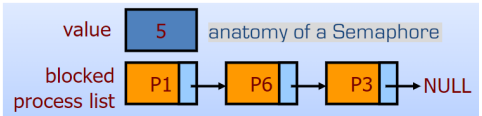
Semantics
- On a busy condition set the process to waiting status (WAIT)
- When a process exits the critical region, SIGNAL is activated and the OS moves the next process waiting to enter that critical region to the ready list
  - Removes busy waiting!

## Semaphores

A semaphore is a synchronization object
- Controlled access to a counter (a value)
- Two operations: `wait()` and `post()`
- Can also be used as a resource counter – to control access to finite resources!

`wait()`
- If the semaphore is positive, decrement it and continue
- If not, block the calling thread (process)

`post()`
- Increment the semaphore value
- If there was any (process) thread blocked due to the semaphore, unblock one of them.




imagine a warehouse and an office (multiple resources), and a stock to count...
- Warehouse = resource
- Stock book in office = resource
- 1 warehouse key = semaphore controlling access to one resource
- 1 office key = semaphore controlling access to one resource
- To count stocks:
  - In order to count stocks, an employee must access the office to get the stock book and to the warehouse; without both he cannot fulfill the job!
  - If two employees want to count stocks and one takes the key to the office, while the other takes the key to the warehouse, they become blocked by each other!

**Deadlock** !!!

When two or more processes are unable to make progress being blocked waiting for each other. All processes are in a waiting state.

**Livelock**

When two or more processes are alive and working but are unable to make progress.

**Starvation**

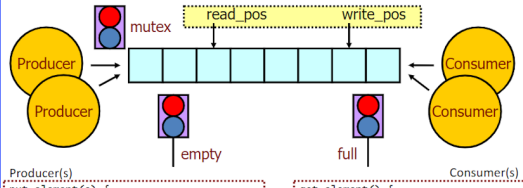
When a process is not being able to access resources that its needs to make progress

# Producer/Consumer

A producer puts elements on a finite buffer. If the buffer is full, it blocks until there's space.

The consumer retrieves elements. If the buffer is empty, it blocks until something comes along.

We will need three semaphores
- One to count the empty slots
- One to count the full slots
- One to provide for mutual exclusion to the shared buffer
- Note: if there was only 1 producer and 1 consumer, the mutex was not needed




```
put_element(e) {
  sem_wait(empty);
  sem_wait(mutex);
  buf[write_pos] = e;
  write_pos = (write_pos+1) % N;
  sem_post(mutex);
  sem_post(full);
}
```

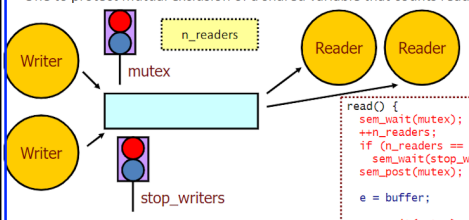
```
get_element() {
  sem_wait(full);
  sem_wait(mutex);
  e = buf[read_pos];
  read_pos = (read_pos+1) % N;
  sem_post(mutex);
  sem_post(empty);
  return e;
}
```

# Readers/Writers

Writer processes have to update shared data.

Reader processes have to check the values of the data. They should all be able to read at the same time.

We will need two semaphores:
- One to stop the writers and guarantying mutual exclusion when a writer is updating the data
- One to protect mutual exclusion of a shared variable that counts readers




```
write(e) {
  sem_wait(stop_writers);
  buffer = e;
  sem_post(stop_writers);
}
```

```
read() {
  sem_wait(mutex);
  ++n_readers;
  if (n_readers == 1)
    sem_wait(stop_writers);
  sem_post(mutex);

  e = buffer;

  sem_wait(mutex);
  --n_readers;
  if (n_readers == 0)
    sem_post(stop_writers);
  sem_post(mutex);
  return e;
}
```

The previous algorithm gives priority to readers
- Not always what you want to do

There is a different version that gives priority to writers in the original paper

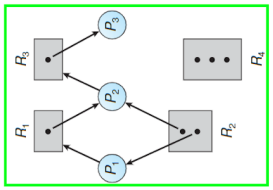
Both versions can suffer from starvation problems
- Priority to readers
  - Writers can starve
- Priority to writers
  - Readers can starve

## What is a deadlock?

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other.

## Deadlock arises if 4 conditions exist at the same time:

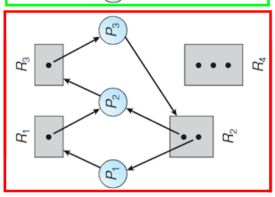
- Mutual exclusion
  - Only one process may use a resource at a time
- Hold-and-wait
  - A process holds a resource and is waiting for additional ones - a process requests all of its required resources at one time
- No-preemption
  - Resources cannot be preempted, i.e., can only be released voluntarily.
- Circular wait
  - A processes is waiting for a resource held by another process that in turn is waiting for a resource held by the first




## Resource-Allocation graph

Direct graph consisting of:

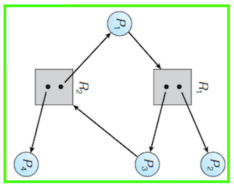
- Vertices
  - Active processes (circles) - P
  - Resources (rectangles) - R
    - Each dot represents an instance of the resource




Directed edges

- $P_x \rightarrow R_y$: Process $P_x$ requested an instance of resource $R_y$
- $R_y \rightarrow P_x$: An instance of $R_y$ has been allocated to $P_x$

It is used to describe the resource-allocation state at a specific time




Resource-allocation graph analysis

- If a graph contains no cycles ⇒ no deadlock.
- If a graph contains a cycle …
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

## Options for handling deadlocks in a OS:

- 1 - **Prevent** deadlocks to ensure that the systems never enters a deadlock state
- 2 - **Avoid** deadlocks to ensure that the systems never enters a deadlock state
- 3 - **Allow** a deadlock state, detect it and recover
- 4 - **Ignore** deadlocks
  - Is up to the programmers to handle deadlocks
  - Used in Linux and Windows

Prevention is accomplished by ensuring that at least one of the 4 necessary conditions for a deadlock does not happen:

- Mutual exclusion, hold-and-wait, no-preemption, circular wait

### Prevent Mutual exclusion

- Have sharable resources (e.g. read-only files)
  - However, not all resources can be sharable!

### Prevent Hold and Wait

- Guarantee that all the resources are requested and allocated before beginning execution
  - If all resources cannot be guaranteed release the ones allocated
- Or only request a resource after releasing the ones that are being hold
- Both options lead to low usage of resources and may lead to starvation of a process that uses different resources

### Prevent Circular Wait

- Prevented by defining a linear ordering of resource types

Possible problems

- Low device utilization
- Reduced system throughpu

### Prevent No-preemption

- If a process holding certain resources is denied a further request, that process must release its original resources.
- If a process requests a resource that is currently held by another process, the operating system may preempt the second process and require it to release its resources.

## How to avoid the occurrence of deadlocks?

- Require more information about the resources that are to be requested
- Approaches
  - Do not start a process if its demands might lead to a deadlock.
  - Do not grant an incremental resource request to a process if this allocation might lead to a deadlock.

Deadline avoidance algorithms dynamically examine the resource-allocation state to ensure that a circular-wait condition never happens

## Safe state

In a system with a a fixed number of processes and resources:

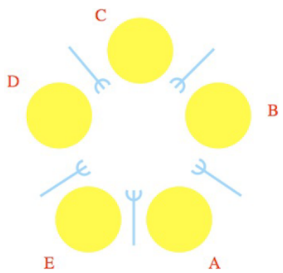
- The **state** of the system is the current allocation of resources to process.
- A **safe sequence** exists when all processes may access their needed resources without resulting in a deadlock
- A **safe state** is when there is at least one safe sequence of resource allocations that does not result in deadlock.
  - All processes can run to completion
  - The system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- Unsafe state is a state that is not safe (!).
  - An unsafe state **may** lead to a deadlock

### Resource-allocation-graph algorithm

- Algorithm used when there is only one instance of each resource type
- The resource-allocation graph used has an additional edge – **claim edge**
  - A claim edge indicates that a process $P_i$ may request a resource $R_j$ in the future
  - It is depicted by a dashed line

## DINING PHILOSOPHERS

- Five philosophers sitting around a table
- Each has a plate of spaghetti
- There are forks between each pair of plates
- Philosophers need two forks to eat




## can it be prevented?

Proposal 1:

- Instead of *take_fork((i+1) % N)*, see if it is available first
- If unavailable, **sleep** for a while and retry….

Still no good….

- what if everyone tries the left fork, waits, retries simultaneously, etc.
- Everyone can run, but no one makes progress: ***STARVATION***

Proposal 2:

- Before taking a fork, a philosopher locks a MUTEX
- The philosopher can then take two forks, with no interference
- When done eating, the forks are replaced and MUTEX is released
- Nope…. only one philosopher can eat at a time…

**Solution:**

```
void philosopher(int id) {
  while(TRUE) {
    think();
    if(id %2 == 0) { // even number: left, right
       take_fork(i);
       take_fork((i+1) % N);
    }
    else{   // odd number: right, left
       take_fork((i+1) % N);
       take_fork(i);
    }
    eat();
```

## Banker's Algorithm

- An algorithm that can be used in a banking system to allocate a limited amount of money to customers. The credit is given depending on the risk of the customer, so that the bank may continue to lend.

- Can be used with systems with multiple instances of each resource type
- A strategy of resource allocation denial
- Developed by Dijkstra
- When a process enters the system, it declares the maximum number of instances of each resource type that it may need
- When resources are requested, the system must determine if their allocation leaves the system in a safe state
  - If it does -> resources are allocated
  - It it does not -> the process must wait until other process releases enough resources
- Support data structures
  - **Resource vector** – total amount of each resource in system
  - **Available vector** – amount of each resource not allocated
  - **Claim matrix** – total amount of resources that may be required by each process (stated before the process begins executing)
  - **Allocation matrix** – current allocation of resources to processes

### Deadlock Avoidance Strategy:

(To assure that the system is always in a safe state)

- When a process makes a request for a set of resources, assume that the request is granted
- Update the system state accordingly and then determine if the result is a **SAFE STATE**.
  - If so, grant the request.
  - If not, block the process until it is safe to grant the request.

## How to detect deadlocks?

### Strategy

- Find processes whose resource requests can be satisfied
- Grant the resources and assume the process runs to completion and releases resources
  - This is an optimistic approach since the process may later request additional resources. However, what we want to detect is if a deadlock exists at this moment!
- Find another process and repeat
- The algorithm does not prevent deadlocks! It only determines if they exist at the time the algorithm runs.

### Uses 2 matrices and 1 vector

- Available
  - Vector that indicates the number of available resources of each type
- Allocation
  - Matrix that has the number of resources of each type currently allocated to each process
- Request
  - Matrix that has the current requests of each process

1. Mark each process that has a row in the Allocation Matrix (**A**) of all zeros.
   - **A process with no allocated resources does not participate in a deadlock.**
2. Initialize a temporary vector **W** (Work) to be equal to the Available Vector (that lists resources from **1** to **m**).
3. Find an index **i** such that process **i** is currently unmarked and the $i^{th}$ row of the Request Matrix **Q** is less than or equal to **W**.
   - That is, $Q_{ik} <= W_k$ **(k=1...m)**. I.e. the process **i** can run with the available resources!
   - If no such row is found terminate the algorithm.
4. If such a row is found mark process **i** and add the corresponding row of the allocation matrix to **W**.
   - That is: set $W_k = W_k + A_{ik}$ **(k=1...m)**. I.e. assume that on completion the process frees all the allocated resources.
   - Return to step 3.

A **deadlock exists** if and only if there are unmarked processes at the end of the algorithm. The set of unmarked rows corresponds to the set of deadlocked processes.

- These processes do not have sufficient resources available to fulfill their requests and cannot finish their execution

## Recovery from a deadlock

- Abort all deadlocked processes.
- Successively abort deadlocked processes until deadlock no longer exists.
- Successively preempt resources until deadlock no longer exists.

## CPU scheduling

- It is the basis of multiprogrammed OSs
- It allows one process to use the processor while all others are kept on hold
- It maximizes the CPU utilization by putting on hold all processes that are waiting for some resource (e.g. I/O)
- CPU scheduling aims to make the system efficient, fast and fair
- On OSs that support kernel-level threads, it is threads that are being scheduled, not processes

## CPU scheduler

*mais sobre os vários tipos - pág. 2.5

- When a CPU becomes idle the OS must select one process from the **ready queue** to be executed
  - The selection is done by the short-term scheduler
  - It selects a process from the ones in memory that are ready to execute
- When more processes are submitted than can be executed immediately (e.g., in a batch system) they are spooled to a mass-storage device (e.g., disk).
  - The long-term scheduler/job scheduler selects processes from the waiting list and loads them in memory for execution
- Swapping enables the removal of a process from memory to reduce the level of multiprogramming. The process can be later reintroduced and continue execution
  - The swapping is done by the medium-term scheduler

## Dispatcher

- The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler
  - Switches context
  - Switches to user mode
  - Restarts the program in the proper location
- The dispatcher is invoked during every process switch
  - The time it takes for the dispatcher to stop one process and start another is known as the **dispatch latency**

- **Non-Preemptive Scheduling**
  Once the CPU is allocated to a process, the process keeps the CPU until it terminates or blocks. A process blocks by switching to a waiting state, such as in I/O operations or OS service requests.
  - (e.g., in Windows 3.x).
- **Preemptive Scheduling**
  The operating system may decide and is able to take the CPU away from a running process
  - (e.g., in WindowsXP, Linux, Mac OS X)

## Scheduling Criteria

- Maximum CPU utilization
- Maximum throughput
- Minimum turnaround time
- Minimum waiting time
- Minimum response time

- **CPU utilization**
  - Keep the CPU as busy as possible
- **Throughput**
  - Number of processes that complete their execution per time unit
- **Turnaround time**
  - Amount of time to complete a particular process (from submission to completion)
- **Waiting time**
  - Amount of time a process has been waiting in the ready queue
- **Response time**
  - Amount of time it takes from when a request was submitted until the first response is produced

## Scheduling Algorithms

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

The process that requests the CPU first gets it first

Simple to manage with a FIFO queue

Average waiting time is often long

This is a non-preemptive algorithm!

- A process keeps the CPU until it terminates or requests I/O.

FCFS performs better for long processes, than for short ones

- ***Convoy effect***
  - Short processes arriving after long processes are penalized, since they must wait until the long process leaves the CPU

I/O bound vs. CPU bound problem

- FCFS tends to favor processor-bound over I/O-bound processes

### Shortest-Job-First (SJF)

Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

- How do you determine that? Can you guess the future?
- Also known as Shortest-Job-Next (SJN).

Two schemes:

- Non-preemptive: once CPU given to the process it cannot be preempted until completes its CPU burst. This scheme is known as Shortest-Process-Next (SPN).
- Preemptive: if a new process **arrives** with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time First (SRTF) or SRT.

## Priority Scheduling

A priority number is associated with each process (can be anything measurable)

The CPU is allocated to the process with the highest priority:

- Preemptive
- Non-preemptive

Equal priority is scheduled in FCFS order

- If two processes have the same priority, the one closer to the head of the ready-queue is chosen.

SJF can be seen as priority scheduling where priority is the *predicted next CPU burst time*

### Problem: Starvation

- Low priority processes may never execute

### Solution -> Aging

- As time progresses, the priority of the process increases

## Round-Robin (RR)

Is designed especially for time-sharing systems

Similar to FCFS with added preemption

- Reduces the penalty that short jobs suffer with FCFS by using preemption based on a clock.

Each process gets a small unit of CPU time (***time quantum***), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

- New processes are added to the **tail** of the ready queue
- If, at a given instant, a process depletes its time quantum (and still has service time) and if at that instant a new process arrives to the system, the new process is placed in the queue in front of the process that just executed (but behind other processes that are already in the queue)

If there are $n$ processes in the ready queue and the time quantum is $q$, then each process gets $1/n$ of the CPU time in chunks of at most $q$ time units at once.

- No process waits more than $(n-1)q$ time units.
- If the process CPU burst is less than $q$, the process will by itself release the CPU voluntarily (e.g., I/O request, process end)

**Performance**

- $q$ large ⇒ FCFS
- $q$ small ⇒ $q$ must be large with respect to context switch, otherwise overhead is too high. If so, having N processes is like having a machine running at 1/N of the speed.

## Highest Response Ratio Next (HRRN)

Choose next process with the highest ratio (R):

$$R = \frac{\text{time spent waiting} + \text{expected service time}}{\text{expected service time}}$$

- This algorithm accounts for the age of the process.
- Short jobs are favored (smaller denominator)
  - ex: for a waiting of 1 => $\frac{1+3}{3} > \frac{1+4}{4}$ => short job is favored
- Aging without service (waiting time) increases R.
- If Rs are equal, then uses FCFS
- Non-preemptive!

## Multilevel Queue Scheduling

Used when processes are easily classified into different groups (e.g., foreground vs. background)

Each process is assigned to one specific queue based on its specificities, and each queue has its own scheduling algorithm

There is an additional scheduling among queues.

Ready queue is partitioned into separate queues. E.g.

- Foreground (interactive)
- Background (batch)

Each queue has its own scheduling algorithm:

- Foreground – RR
- Background – FCFS

Scheduling must be done between the queues.

- Fixed priority scheduling: high priority queues must be empty for the lower to execute its processes (i.e., serve all processes from foreground; then from background). Possibility of starvation.
- Time slice: each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR; 20% to background in FCFS

## Multilevel Feedback Queue Scheduling

Normally, when using Multilevel Queue Scheduling, processes remain in the queue they were assigned when entering the system.

- Low scheduling overhead

With **Multilevel Feedback Queue Scheduling**, **processes can move between queues!**

- Aging can be implemented this way

Only when the high priority queues are empty will the lower have their processes executed

Normally (not always!), a process that arrives for a higher priority queue will preempt a process from a lower priority queue

The Multilevel-feedback-queue scheduler is defined by the following parameters:

- number of queues
- scheduling algorithms for each queue
- method used to determine when to upgrade a process
- method used to determine when to demote a process
- method used to determine which queue a process will enter when that process needs service (when it first enters)

Many configurations are possible! It can be configured to address specific needs.

Simple version of a Multilevel Feedback approach

- Sometimes there is no indication of the length of the processes and then SPN, SRT and HRRN cannot be used.
- A way to establish preferences is to penalize jobs that have been running longer
  - It focus on the time spent until now (that is known) instead of focusing in the time remaining to execute (most of the times unknown).
- A process is demoted to a lower priority queue whenever it is preempted.

| | Selection Function | Decision Mode | Throughput | Response Time | Overhead | Effect on Processes | Starvation |
|---|---|---|---|---|---|---|---|
| FCFS | $\max[w]$ | Nonpreemptive | Not emphasized | May be high, especially if there is a large variance in process execution times | Minimum | Penalizes short processes; penalizes I/O bound processes | No |
| Round Robin | constant | Preemptive (at time quantum) | May be low if quantum is too small | Provides good response time for short processes | Minimum | Fair treatment | No |
| SPN | $\min[s]$ | Nonpreemptive | High | Provides good response time for short processes | Can be high | Penalizes long processes | Possible |
| SRT | $\min[s-e]$ | Preemptive (at arrival) | High | Provides good response time | Can be high | Penalizes long processes | Possible |
| HRRN | $\max\left(\frac{w+s}{s}\right)$ | Nonpreemptive | High | Provides good response time | Can be high | Good balance | No |
| Feedback | (see text) | Preemptive (at time quantum) | Not emphasized | Not emphasized | Can be high | May favor I/O bound processes | Possible |

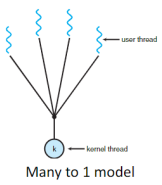
## Thread scheduling

Support for threads may be provided in user level - **User threads** or by the kernel – **Kernel threads**.

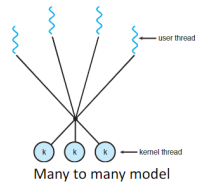
Only kernel threads are managed directly by the OS.

User threads are managed without kernel support.

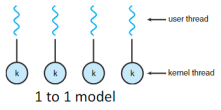
A relationship must exist between user threads and kernel threads what leads to different models:



Many to 1 model



Many to many model



1 to 1 model

In OSs that support threads, it is kernel level threads that are scheduled – not processes

**User-level threads** are managed by a library and the OS is unaware of them

- To run user-level threads they must be mapped to a kernel-level thread, what can be accomplished indirectly by using a lightweight process (LWP)

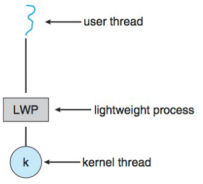
## Lightweight Process (LWP)

- Communication between kernel and the thread library may be done through an intermediate data structure - LWP.
- A LWP is seen as Virtual Processor from the thread library side
  - The library schedules its tasks into this virtual processor

An application may require more than one LWP to run efficiently

- Example:
  - An I/O intensive application may require more than one LWP to deal with each concurrent blocking system call. If it does not have the necessary number of LWP it will have to wait until one is freed.

- Each LWP is attached to a kernel thread.
- It is the kernel level thread that is scheduled (to the CPU) by the OS.




## Contention Scopes

Threads can be scheduled using different contention scopes:

Process contention scope (PCS)

- The contention scope is the process
- Scheduling is made among threads belonging to the same process

System contention scope (SCS)

- The contention scope is the system
- Scheduling is made among all threads in the system

In systems using one-to-one model the scheduling only uses SCS, not PCS. Other models use both.

In many-to-one and many-to-many models, the thread library schedules **user-level threads** to an available LWP (not directly to the CPU)

- (The LWP may or may not be actually running in a CPU – depends on the kernel-thread being scheduled).

Competition for the CPU is made between threads belonging to the same process

⬇

Uses Process Contention Scope

Kernel schedules **kernel-level threads** to a physical CPU

Competition for the physical CPU is made between all system kernel-level threads

⬇

Uses System Contention Scope

## AMDAHL'S LAW

This law identifies potential performance gains from adding additional computing cores

$$speedup(f,n) = \frac{1}{(1-f)+\frac{f}{n}}$$

to an application that has both serial (nonparallel) and parallel components.

S = portion of the application to perform serially

N = Number of processing cores

Example

- S = 25% and N = 2 → Speedup = 1 / (0,25 + (1-0,25)/2) = 1,6X
- S = 25% and N = 4 → Speedup = 1 / (0,25 + (1-0,25)/4) = 2,28X

Interesting Fact:

- As N tends to infinity, Speedup tends to 1/S
  - For an app with S=0,4 the **max Speedup is 2,5X** independently of the number of cores

## Multiprocessing

If a system has multiple processors, Load-Sharing becomes possible

Scheduling in a system with multiple processors is more complex and there is no one best solution

Our focus case:

- Multiprocessor scheduling for homogenous systems, i.e., systems in which the processors are all identical

## Multiple-Processor Scheduling

Asymmetric multiprocessing

- A single processor – the master server – has all scheduling decisions, I/O processing and other system activities.
- Other processors execute only user code.
- Simple approach as it reduces the needs for data sharing.

Symmetric multiprocessing (SMP)

- Each processor is self-scheduling.
  - All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.
- Must avoid scheduling the same process more than once simultaneously – Critical section

## Processor Affinity

Data most recently accessed by a process populates the cache for the processor.

- successive memory accesses by the process are often satisfied in cache memory.

What happens if the process migrates to another processor?

- The contents of cache memory must be invalidated for the first processor and the cache for the second processor must be repopulated.
- HIGH COST → most SMP systems avoid migration and keep a process running on the same processor → Processor affinity

Processor affinity

- a process has an affinity for a specific processor
- Soft affinity – the OS tries to keep a process in the same processor but does not guarantee it
- Hard affinity - it is possible for a process, through system calls, to specify a subset of processors on which it may run.

## NUMA and CPU Scheduling

Non-Uniform Memory Access (NUMA)

- CPUs on a board can access the memory on that board faster than they can access memory on other boards in the system

If OS's CPU scheduler works together with memory placement algorithms, the process with affinity to a specific processor is allocated memory in that CPU board.

## Load Balancing

Keeps the workload evenly distributed across all processors in an SMP system.

- Necessary only on systems where each processor has its own private queue of eligible processes to execute. (common case in today's OSs)
- With a common run queue, load balancing is often unnecessary, because once a processor becomes idle, it immediately extracts a runnable process from the common run queue.

**Two load balancing approaches:**

- **Push migration**
  a specific task periodically checks the load on each processor and, if it finds an imbalance, evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors.
- **Pull migration**
  occurs when an idle processor pulls a waiting task from a busy processor.

- EDF does not require processes to be periodic
- The only requirement is that, when a process becomes runnable, it announces its deadline to the scheduler.

## Multicore Processors scheduling

Traditionally, SMP systems provided multiple physical processors. Today, multiple processor cores are placed on the same physical chip – **multicore processors**.

- This cores appear as a separate physical processor to the OS.

Multicore processors may complicate scheduling

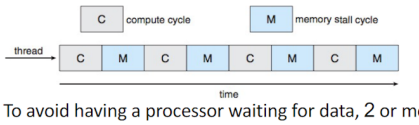
- May have multithreaded cores
- Common caches between different cores
  - As the number of cores per chip increases, a need to minimize access to off-chip memory takes precedence over a desire to maximize processor utilization.

Multicore scheduling levels

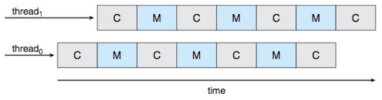
- Multithreaded multicore processors require two scheduling levels
- **First level**
  - Operating System chooses which software thread to run on each hardware thread (logical processor).
    - Any algorithm for CPU scheduling already discussed will do the job
- **Second level**
  - Each core decides which hardware thread to run
    - e.g., Round-robin, use of urgency values assigned to each hw thread

Why multithreaded processor cores?

- When a processor accesses memory, it can spend a significant amount of time waiting for the data to become available (e.g., cache miss) – **memory stalls**.




- To avoid having a processor waiting for data, 2 or more hardware threads are assigned to each core.
- If one hardware thread stalls while waiting for memory, the core can switch to another thread.




- From an OS perspective, each hardware thread appears as a logical processor that is available to run a software thread.
- On a dual-threaded, dual-core system, four logical processors are presented to the OS.

## Real-Time CPU Scheduling

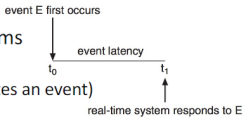
**Soft real-time** systems provide no guarantee as to when a critical real-time process will be scheduled.

- They guarantee only that the process will be given preference over noncritical processes.

**Hard real-time** systems have stricter requirements.

- A task must be serviced by its deadline
  - Service after the deadline has expired is the same as no service at all.

## Event Latency




Events in real-time systems

- SW (e.g., a timer expires)
- HW (e.g., a sensor generates an event)

Event latency

- Time between the occurrence of the event and its service

Types of latency that affect real-time systems:

- **Interrupt latency**: time between when the interrupt is received by the CPU to the start of its service routine.
- Must have strict bounds;
- One important factor is the amount of time interrupts are disabled in the kernel while its data structures are being updated.

**Dispatch latency** – amount of time required for the scheduling dispatcher to stop one process and start another

Conflict phase

- Preempt processes running
- Releases resources from low-priority processes that are needed by high priority ones

Preemptive kernels improve dispatch latency

- Dispatch occurs more frequently and does not depend on processes blocks

## Priority-Based Scheduling

A real-time system must respond immediately to a real-time process as soon as it requires CPU

- Applications are generally less concerned with raw speed than with the start and completion deadline of their execution.

To guarantee it, priority-based scheduling algorithms with preemption must be supported by the OS

- Processes executing must be preempted if a higher priority process is ready to run
  (Priority scheduling algorithms were already studied)

However, a preemptive, priority-based scheduling only guarantees **soft real-time** systems!

**Hard real-time** systems also have deadline requirements, implying additional scheduling features.

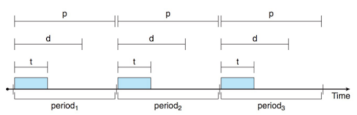
## Real-time processes

- **Aperiodic** task – has a strict deadline to finish and/or to start
- **Periodic** task – must execute once in a period *T* or exactly *T* units apart

Case of periodic processes/tasks

- Have a sequence of times that is known in advance (e.g., checking and processing data from a sensor)
- Require the CPU at constant intervals (periods) (p), a fixed processing time (t) and a deadline (d). The rate is 1/p .

$$0 \le t \le d \le p$$




- Schedulers can assign different priorities according to each process requirements

## Admission control

To address their specific requirements, real-time processes may have to announce their deadlines to the scheduler using an admission control algorithm.

Based on the information received from the processes, the scheduler:

- admits the process, guaranteeing that it will complete on time;
- rejects the process as it cannot guarantee that it can be serviced by its deadline.

## Rate-Monotonic Scheduling

Schedules periodic tasks by using static priority policy with preemption.

- Upon entering the system, each periodic task is assigned a priority inversely based on its period → priorities are static!
  - The shorter the period, the higher the priority; the longer the period, the lower the priority.
  - The aim is to assign a higher priority to tasks that require the CPU more often.
- It assumes that the processing time of a periodic process is the same for each CPU burst.
- A running lower-priority process will be preempted if a higher-priority process becomes ready to run.

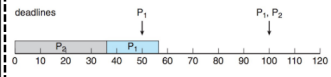
P1: period = 50; processing time = 20

P2: period = 100; processing time = 35

The deadline for each process requires that it completes its CPU burst by the start of its next period.

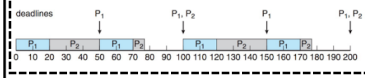
Predicted CPU utilizations= (20/50+35/100)=0,75 ← Apparently, it is possible to schedule both processes and fulfill deadlines

Without rate-monotonic P2 could get higher priority:




Deadline not fulfilled

With rate monotonic, P1 gets higher priority because it has a lower period → both deadlines are fulfilled even if P2 is preempted




Deadline fulfilled

RM Scheduling is optimal between algorithms using static priorities

- If it can be scheduled with static priority, RM will do it

But it can fail...

- In the case where P1 and P2 CPU utilization is:
  - (25/50)+(35/80)=0,94 → Less than 100% CPU
- It is not always possible to fully maximize CPU resources!

## Earliest-Deadline-First (EDF)

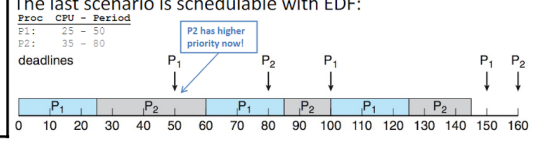
Assigns priorities according to deadlines → priorities are dynamic.

- The earlier the deadline, the higher the priority
- The later the deadline, the lower the priority.

When a process becomes runnable, it must announce its deadline requirements to the system.

Priorities may have to be adjusted to reflect the deadline of a new runnable process.

The last scenario is schedulable with EDF:




- Theoretically, EDF is optimal, achieving 100% of CPU use.
- In practice it is impossible to achieve 100% due to the cost of context switching.

## Proportional Share Scheduling

Proportional share schedulers operate by allocating T shares among all applications.

- An application can receive N shares of time, thus ensuring that the application will have N/T of the total processor time.
- Requires **Admission-control** to prevent impossible scheduling scenarios (a client requesting more shares than the ones available) and to guarantee that an application receives its allocated shares of time.

## Memory Management

In unipromgramming systems, main memory is divided between the OS memory and the memory used by the program being executed

In multiprogramming systems, multiple programs exist, each with its own memory, which must be dynamically managed

- Multiprogramming improves both the CPU utilization and the response speed to users; It implies that multiple processes must coexist in memory

Instructions executed by the CPU can only address the CPU registers or the main memory;

- Any instruction executed or data used must be in one of these direct-access memory
- If data is not in main memory, or in a register, it must be moved there to complete the execution of the instruction

Memory management determines what is in memory and when
- To optimize CPU utilization and response to users

Memory-management schemes depends heavily on the hardware available

Requirements of a memory management system

- Relocation
  - Programs may be moved in memory (e.g., after being swapped) and must continue working, i.e., memory references in the code must enable access to the correct physical addresses
- Protection
  - Each process memory should be protected against interference from other processes whether accidental or intentional - Usually done by HW
- Sharing
  - Any protection mechanism must have the flexibility to allow several processes to access the same portion of main memory (e.g., shared memory)
- Logical organization
  - Memory used by programs may have different modules with different characteristics (e.g., some modules may be read only, other writable);
- Physical organization
  - Memory used by a program may be in different locations (e.g., main memory, secondary memory)

For proper system operation, we must protect the operating system from access by user processes, as well as protect user processes from each other. This protection is provided by the hardware for performance reasons.

## Loading a program

The first step in the creation of an active process is to load a program into main memory

How does the loader deal with addresses in the program?

### 3 approaches to loading:

Absolute loading

- If final places in memory are known,

all references may use **absolute addresses**

- May be done by the programmer or at

compile/assembly time

- Must recompile code if starting location

changes or inserts/deletions are made in the module

  - To avoid it, **symbolic addresses** can be used by the programmer, which are converted to specific addresses by the assembler or compiler

Relocatable loading

- Binding memory references to specific addresses, before loading, only allows the use of specific main memory addresses;
- By using **relative addresses**, the assembled/compiled program may be located anywhere in the main memory, starting at a specific point defined at initial load time

Dynamic run-time loading

- Relocatable loading does not enable changes in the location of a process during execution (e.g. due to a swap), if the start of the program is defined at initial load time
- Dynamic run-time loading defers the absolute address calculation until needed at run time, to enable the process to be moved during its execution. To assure that this function does not degrade performance, it must be done by hardware rather than software.

## Linking and Loading

When more than one module is needed, linking is necessary:

- Compiled or assembled modules in object-code form are linked. Library routines may be incorporated into the program or referenced as shared code that must be supplied by the operating system at run time.

## Dynamic Loading

Until now we were considering that the whole program with its data were in physical memory, for the process to execute.

By using dynamic loading, a process does not have to load the entire program and all its data to physical memory

- Enables a better memory-space utilization
- A routine is not loaded until it is called
- An unused routine is never loaded

Useful when large amounts of code are needed to handle infrequently occurring cases

## Dynamic Linking

Dynamically linked libraries are system libraries that are linked to user programs when the programs are run

Dynamic linking postpones linking until execution time

- Without this facility, each program on a system must include a copy of all the routines referenced by the program (even routines from system libraries) in the executable image, wasting both disk space and main memory – **static linking**.

Uses a small piece of code (**stub**) that is included in the image for each library-routine reference. The stub contains information on how to locate the appropriate memory-resident library routine.

- Stub replaces itself with the address of the routine, and executes the routine

Dynamic linking is particularly useful for libraries.

This system is also known as shared libraries.

## Address binding

Addresses may be represented differently along the different steps prior to execution. Each binding is a mapping from one address space to another.

Address binding of instructions and data to **memory physical** (absolute) **addresses** can happen at different times:

- **Programming time**: if all physical addresses are specified by the programmer in the program itself; only works if memory location are known a priori; must recompile code if starting location changes.
- **Compile/assembly time**: symbolic addresses in the program are converted to physical (absolute) addresses by the compiler/assembler; **absolute code** is generated; only works if memory location are known a priori; must recompile code if starting location changes.
- **Load time**: the compiler must generate **relocatable code** if final memory location is not known at compile time - final binding to physical addresses is delayed until load time.
- **Execution time**: binding to memory addresses must be delayed until run time if the process can be moved during its execution, from one memory place to another; the loaded program retains relative addresses that are converted to physical addresses as needed.

## Logical and physical addresses

**Logical (virtual) address**

- Is a reference to memory independently of the current assignment of data to memory – a translation is needed to access physical memory
- Also referred to as virtual address
- Is generated by the CPU

**Physical address**

- Addresses seen by the memory unit - they correspond to addresses in physical memory
- Is an actual location in main memory
- Is the one seen by the memory unit

Compile-time and load-time address binding methods results in identical logical and physical addresses; execution-time address bindings result in different ones

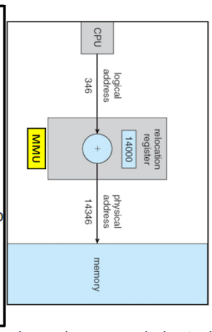
The set of all logical addresses generated by a program is named **logical/virtual address space**; the set of correspondent physical addresses is named **physical address space**.

## Memory Management Unit

The run-time mapping from virtual to physical addresses is done by hardware in a device called **Memory Management Unit (MMU)**

The programmer does not know where the program will be placed in memory when it is executed.

While the program is executing, it may be swapped to disk and returned to main memory at a different location (relocated).




Memory references must be translated in the code to actual physical memory address

- Several methods can be used in the mapping from virtual addresses to physical addresses;
- A simple method uses a generalization of the base-register scheme. In this context the register is known as **relocation register.**

## Swapping

A process must be in memory to be executed;

A process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution;

Swapping allows the use of a total physical address space bigger than the real physical system memory and increases multiprogramming. When the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see if the process is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process.

Example:

| | |
|---|---|
| Memory to swap (in bits): | 1 Mb |
| Disk transfer rate | 5Mb/sec |
| Time it takes to swap out: | 1/5 sec = 200 msec |
| Average disk latency: | 8 msec |
| Total time to 'swap-out' + 'swap-in': | 416 msec |

**Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped**

## Contiguous Memory Allocation

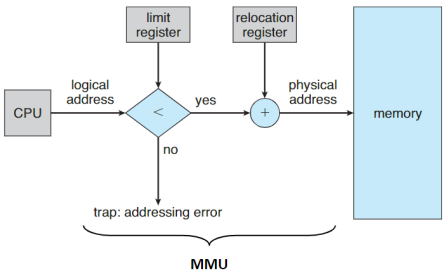
The main memory must accommodate both the OS and the user processes

Contiguous memory allocation is one method of allocating the available memory

- Each process is contained in a single section of memory that is contiguous to the section containing the next process

Two partitions are usually allocated:

- OS partition
- User processes partition (where all user processes are located)




**MMU**

Memory must be allocated to processes that are created

- Hole – block of available memory; holes of various size are scattered throughout memory.
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- Operating system maintains information about:
  **a)** allocated partitions **b)** free partitions (holes)

**External Fragmentation**: When there is no block large enough to accommodate a process, although overall there may be enough memory

**Internal Fragmentation**: When memory is allocated in fixed-sized chunks, leading to some internal memory not being used

## Memory Management techniques

### Fixed Partitioning

**Equal-size partitions**

- Any process whose size is less than or equal to the partition size can be loaded into an available partition.
- If all partitions are full, the operating system can swap a process out of a partition
- If the program is too big to fit a partition, the programmer must design it to allow that only parts are in main memory at any one time (using overlays).
- Main memory use is inefficient. Any program, no matter how small, occupies an entire partition: **internal fragmentation.**

**Unequal-size partitions**

- Can assign each process to the smallest partition within which it will fit
- Processes are assigned in such a way as to minimize wasted memory within a partition (minimizes **internal fragmentation**)
- Placement algorithm:
  - Assign each process the smallest partition within which it fits
    - Needs one queue for each partition to manage swap-outs
    - Minimizes internal fragmentation
    - Has a common queue for all processes or one queue per partition
    - Not optimum at a system level as there may not be programs with some specific partition sizes
  - When a new program is loaded a swap-out of other may occur
- Advantages:
  - Relatively simple to implement and require minimal OS software and processing overhead.
  - Any process with **size** <= **partition size** can be loaded into an available partition
- Disadvantages:
  - The number of partitions specified at system generation time limits the number of active (not suspended) processes in the system.
  - Because partition sizes are preset at system generation time, the use of main memory is inefficient. All programs, despite their size, occupy an entire partition -> **internal fragmentation.**
  - A program may be too big to fit into a partition

# Dynamic Partitioning

Partitions are of variable length and number.

Process is allocated exactly as much memory as required.

Eventually get holes in the memory. This is called **external fragmentation**.

Must use **compaction** to shift processes so they are contiguous, and all free memory is in one block.

- Time consuming task!

Placement algorithms

- Operating system must decide which free block to allocate to a process, to avoid the need of compaction.
- **First-fit**: Allocate the *first* hole that is big enough
  - Fast: can stop searching after finding the first adequate block
  - External fragmentation
- **Best-fit**: Allocate the *smallest* hole that is big enough
  - Must search entire list, unless ordered by size
  - Produces the smallest leftover hole: External fragmentation
- **Worst-fit**: Allocate the *largest* hole
  - Must also search entire list
  - Produces the largest leftover hole
- **Next-fit:** Scans memory from the location of the last placement and chooses the next available block that is large enough

# Buddy System

Normally used to allocate memory in the kernel

In a buddy system blocks are available in $2^K$ sizes with a minimum of $2^L$ and a maximum of $2^U$

- Generally, $2^U$ is the size of the entire memory available for allocation
- L changes with the specific system

If a request of size s is made and

$2^{U-1} < s <= 2^U$

Then, entire block is allocated.

- Otherwise, block is split into two equal buddies.
- Process continues until smallest block greater than or equal to s is generated

The buddy system maintains a a list of holes (unallocated blocks) of each size $2^i$

When a $2^i$ block becomes free, it may be coalesced with a corresponding buddy to produce a block of size $2^{i+1}$

It's a very fast system!

Compromise between internal and external fragmentation.

# Segmentation

Memory-management scheme that supports user view of memory

- Programmers do not usually think of memory as a linear array of bytes
- A programmer sees memory as methods, procedures, functions and data structures with no care of their addresses in memory

A program is a collection of segments. A segment is a logical unit of data.

A program has several segments, each with a name (a number in real implementations) and a length;

Different segments may have different lengths;

There is a maximum segment length;

Logical addresses consist of two parts: <s,d>

- A segment number (s)
- A segment displacement/offset (d)

| segment | offset |
|---|---|

Since segments are not equal, segmentation is similar to dynamic partitioning.

- Segmentation does not avoid external fragmentation and may need compaction.

## Segmentation Hardware

Implements the translation from user addresses to physical addresses

Uses a segment table where each entry has a segment base (starting physical address) and a segment limit (length of the segment)

# Paging

As happens with segmentation, paging permits the physical space of a process to be non-contiguous.

To achieve this, physical main memory is divided in small fixed-size blocks (frames) and processes are divided in blocks of the same size (pages).

External fragmentation is avoided.

Paging is used in most OSs.

Internal fragmentation only affects the last page of a process.

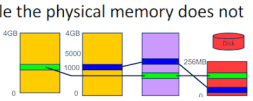
It also solves the problem of fitting the memory blocks onto a backing store whenever swapping-out is necessary.

Basic Method

- Divide physical memory into fixed-sized blocks called **frames** (typically 4KB)
- Divide logical memory into blocks of same size called **pages** (typically 4KB)
- Keep track of all free frames
- To run ***n*** pages from a program, ***n*** free frames have to be found;
  - (With no virtual memory, the *n* frames are the total size of the process – more on this later)
- The program is then loaded to the free frames from the file system or from a backing store
  - The backing store uses blocks of the same size as frames and pages
- Set up a **Page Table** to translate logical to physical addresses

Using paging, the logical and physical address spaces are separated, which enables that a process in a 64 bit machine can have a logical 64 bit address, while the physical memory does not necessarily have that space ($2^{64}$)

processes page table physical memory




# Address Translation

Each address (**m** bits) generated by CPU is divided in two:

- **Page number (*p*)** – used as an index into a *page table* which contains the **base address** of each page in physical memory
- **Page offset (d)** – displacement within the page; combined with the **base address** defines the physical memory address that is sent to the memory unit

## Example

32 bit system with 20 bits for page number and 12 bits for page offset

| Page Number: 20 bits | Page Offset: 12 bits |
|---|---|

- Logical address space = $2^{32}$ = 4294967296 = 4GB
- Page size = $2^{12}$ = 4096 = 4kB
- Number of pages = $2^{20}$

## Logical to physical mapping

3 decimal digit system with 1 digit for page number and 2 digits for page offset

| Page Number: 1 decimal digit | Page Offset: 2 decimal digits |
|---|---|

Maximum of 10 pages: 0..9

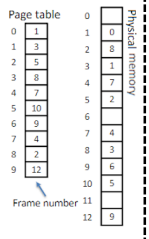
Page size of 100 bytes: 0..99

To which physical address does the logical address $205_{10}$ correspond?

page = $2_{10}$; offset = $5_{10}$

Page 2 => Frame 5

Physical address = $505_{10}$




## Hardware support to access a page table

Page table is kept in main memory

- **Page-table base register (PTBR)** points to the page table
  - Changing page tables in context-switch only requires the change of this register (stored in the process control block)
- **Page-table length register (PTLR)** indicates the size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for accessing the page table entry and one for the needed data/instruction!
  - Intolerable overhead!

The two memory access problem can be solved by the use of a special fast-lookup hardware cache called Translation Look-Aside Buffer (TLB)

- Small number of entries... 32-1024
- Separate data and instructions TLB may be used
- All entries of the table are evaluated in parallel.
  - If the page number is found the frame number is returned and used to access memory.
  - If the page number is not in the TLB a TLB miss is generated and a memory reference to the page table in memory must be made.

## Protection

Memory protection is implemented by associating protection bits with each frame, normally kept in the page table.

- E.g.: one bit can define a page to be read-write or read-only.
- Attempts to violate the settings generate a HW trap to the OS
- Finer levels of protection are possible.

Valid-invalid bit attached to each entry in the page table:

- "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
- "invalid" indicates that the page is not in the process' logical address space

Example

- System using 14-bit addresses ($2^{14}$=16384)
- Program using only 0-10468 logical addresses
- There is internal fragmentation in page 5 (page 5 extends to 12287 but the process only uses addresses till 10468)
- Only addresses beyond page 5 are invalid

Normally a program does not use all its available address range

So, there is no need to have a page-table with entries for all pages in the logical address range, as the space would be unused, and the memory wasted

A Page-table length register can be used to indicate the size of the table

- All logical addresses that exceed the PLTB are returned as invalid

## Shared pages

Shared code

- Paging allows sharing common non-modifying code (saves memory space!)
- Only one copy of read-only (reentrant) code shared among processes
  - Heavily used programs benefit the most (e.g., editors, run-time libraries, etc.)
- Each user table maps the shared pages on the same physical copy of the program (i.e., in the same frames).

Private code and data

- Each process keeps a separate individual copy of the code and data
- The pages for the private code and data can appear anywhere in the physical address space

Shared pages may also be used by OS to implement shared memory IPC

Example

- Text editor used by 3 users
- Each user process shares a 3 page editor program
- Data for each process is different
- Only one copy of the editor is kept in physical memory saving memory

# Structure of Page Tables

## Hierarchical Paging

Divide the page table in smaller pieces.

- The entire page table can itself be paged.

- The entire page table is divided in pages
- A root table (page directory) indicates if the page is valid (and where it is) or invalid (if it does not contain valid pages)
- Saves memory by not allocating invalid regions of memory, not used by a program
- Many pages of the logical address space of a program are not really used (as the program uses less than the maximum addressable space)
  - E.g., not all available heap space is used!
- If a single level is used many PTE (Page Table Entries) are marked as invalid

A logical address (on 32-bit machine with 4kB=$2^{12}$ bytes page size) is divided into:

- a page number consisting of 20 bits.
- a page offset consisting of 12 bits.

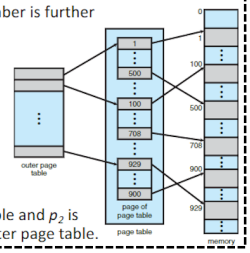
Since the page table is paged, the page number is further divided into:

- a 10-bit outer page number ($p_1$).
- a 10-bit inner page number ($p_2$).

Thus, a logical address is as follows:

| outer page number | inner page number | page offset |
|---|---|---|
| $p_1$ | $p_2$ | $d$ |
| 10 | 10 | 12 |

Where $p_1$ is an index into the outer page table and $p_2$ is the displacement within the page of the outer page table.




## Hashed Page Tables

Common in address spaces > 32 bits

The virtual page number is hashed into a page table. This page table contains a linked list of elements hashing to the same location.

Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

A single page table entry can store the mapping for multiple physical page frames.

Another way to optimize the page table, reducing its size.

## Inverted Page Tables

One hash table for all processes

One page table in the system with one entry for each real page of memory (sorted by physical address)

Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.

**Decreases memory** needed to store each page table (it has only one page table and one entry for each page of physical memory), but **increases time** needed to search the table when a page reference occurs

- Use hash table to limit the search to one (or at most a few) page-table entries

# Paging + Segmentation

Paging:

- Transparent to the programmer
- Eliminates external fragmentation

Segmentation

- Modularity
- Simplifies handling of growing data structures if dynamic segment sizes are allowed
- By using multiple segments, it allows programs to be altered and recompiled independently, without requiring the entire set of programs to be relinked and reloaded

A combined paging and segmentation system:

- Breaks address space into segments, at the discretion of the programmer
- Breaks each segment into a number of fixed-size pages
- For the programmer, a logical address still consists of a segment number and a segment offset
- For the system, the segment offset is viewed as a page number and page offset for a page within the specified segment

## Address Translation

Associated with each process there is a segment table and a number of page tables, one per process segment.

For each running process, a register holds the starting address of the segment table.

# Address Translation Comparison

| | Advantages | Disadvantages |
|---|---|---|
| **Segmentation** | Fast context switching: segment mapping maintained by CPU | External fragmentation |
| **Paging (single-level page)** | No external fragmentation, fast easy allocation | Large table (virtual memory size) Internal fragmentation |
| **Paged segmentation** | Table size ~ # of pages in virtual memory, fast easy allocation | Multiple memory references per page access |
| **Two-level pages** | | |
| **Inverted Table** | Table size ~ # of pages in physical memory | Hash function more complex |

# Virtual Memory

All instructions being executed must be in main memory

- If all instructions must be in physical memory, then the physical memory is the limit of the size of the program
- However:
  - Many programs have code rarely used (e.g., error handlers, specific features)
  - Not all allocated memory in data structures such as arrays or lists is actually used
  - Not all code is needed at the same time

Virtual Memory is a technique to allow the execution of processes not completely in memory.

- Allows programs to be larger than the available physical memory

With Virtual Memory:

- A program can be larger than the physical memory available
- More programs can be used at the same time, as each program uses less physical space
- Less I/O is needed to load or swap programs
- The programmer task is easier as it does not have to worry about the physical memory limits
- Allows the separation between logical memory and physical memory
- Files and memory can be shared by two or more processes through page sharing

As seen before, the separation between the physical memory and the logical address space, allows for

- More "logical memory" than what physically exists
- Easy sharing of memory between processes
- Efficient/Fast process creation (copy-on-write)

# Fetch Policy

**How do we bring pages from disk to main memory?**

- **Demand paging** only brings pages into main memory when a reference is made to a location on the page. When a process is restarted after a swap-out, every page is only brought in after each page fault generated.
  - Less I/O needed ☺
  - Less memory needed ☺
  - Faster response ☺
  - More users ☺
  - Many page faults when process first started ☹
- **Pre-paging** brings in more pages than needed. Some OSs keep a working set containing all the pages in use before the process was suspended and they bring them to memory before restarting the process.
  - Efficient if pages reside contiguously on disk ☺
  - Uses up memory ☹

# Demand paging

Demand paging allows that a program only loads the needed pages into physical memory

- Is similar to a paging system with swapping, but in this case not all process is swapped in/out
- It uses a pager (in contrast to a swapper that manipulates the entire process) to "swap" individual pages as they are needed

Pages in memory and in disk must be distinguished

- With each page table entry a *valid/invalid* bit is associated
  valid ⇒ legal AND in-memory
  invalid ⇒ not legal OR not-in-memory
  (not legal = page is not mapped in the address space of the process)
- Initially, *valid–invalid* bit is set to **i** on all entries

During address translation, if valid/invalid bit in page table entry is **i** ⇒ page fault

## Steps on Handling a Page Fault

1 - Memory reference (instruction with memory access or instruction fetch)

2 - An invalid page causes a trap to the OS. The OS checks internal tables to determine whether the reference was a valid or invalid (not legal) memory access

- Invalid access terminates the process
- If the reference is valid but the page was not loaded to memory, it is now loaded

4 - Find a free frame and read page from disk to memory

5 - Alter page table

6 - Restart instruction that was interrupted by the trap.

requires solving two issues:
- Page replacement algorithm (which page to replace?)
- Frame Allocation algorithm (how many frames to allocated to each process?)

## I/O Interlock

Sometimes, pages must be locked in memory

- E.g., When I/O is done to/from the memory by a device using DMA (Direct-Memory Access)
  - If the page is replaced during the I/O, the device would write/read to/from a wrong page

A locked page cannot be replaced

- Uses a "lock bit" associated to each frame

*1 bottom right of page 15

# Copy-on-Write

Virtual memory permits fast process creation (forking) by using **Copy-on-Write**.

- Copy-on-Write allows both parent and child processes to initially *share* the same pages in memory, bypassing the initial need for demand paging and minimizing the pages to be allocated for the new process.
  - Only pages that can be modified need be marked as copy-on-write. Pages that cannot be modified (e.g., executable code) can be shared by the parent and child.
- Only when a process modifies a shared page (write), the page is copied
  - Sometimes the pages are never modified (e.g call of exec() after a fork() )
- Copy-on-Write allows more efficient process creation as only modified pages are copied
- Free pages are allocated from a pool of zeroed-out pages

# Memory Mapped Files

Using Paging and Virtual Memory also allows for **Memory-Mapped Files**

- Memory-mapped files allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory.
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than **read()**/**write()** system calls.
- Also allows several processes to map the same file allowing the pages in memory to be shared.

When a page fault occurs and there are no free frames on the physical memory (all memory is in use) the OS must decide what to do:

- Swap out entirely another process, reducing the level of multi-programming (i.e., less different programs in memory)
  - May be useful in certain situations (e.g. when trashing occurs)
- Replace a page in memory for the new one read from the disk – Page Replacement

Some considerations:

- Use a *dirty* (*modify*) bit in the page or frame to reduce overhead of page transfers – it indicates that the page or frame have been modified - **only modified pages are written back to disk** (**swap-out**).
- Page removed should be the page least likely to be referenced in the near future.
- Try to predict the future behavior based on the past behavior.

If a **page fault** occurs in an executing process:

1. Find the location of the desired page on disk
2. Find a free frame to load the desired page:
   - If there is a free frame, use it.
   - If there is no free frame, use a page replacement algorithm to select a victim frame
   - If victim is dirty write it back to disk
3. Read the desired page into the (newly) free frame; update the page and frame tables
4. Restart the process

If there are no free frames and the victim frame is dirty, two page transfers are needed, doubling the page-fault servicing time.

**Goal: get the lowest page-fault rate**

# Page Replacement Algorithms

## 1- FIFO Page Replacement

Each page is time-stamped according to when it was brought into memory.

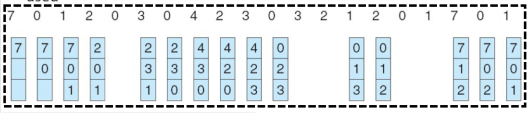
Page that has been in memory the longest is replaced.
Implemented with a simple queue

- New pages in memory go to the tail
- When needed, replace the page in the head of queue

Pros: Simplest algorithm to implement

Cons:

- The page replace victim may be one heavily used or one that was rarely used




## 2- Optimal Replacement

Replaces the page that will not be used for the longest period of time – needs to know the future reference string!

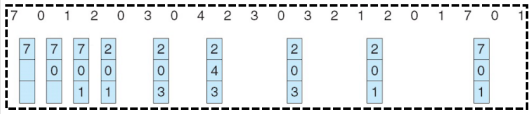
Is a Conceptual Model

- Mainly used for comparison studies
  - To measure how good real algorithms are

Pros: Best algorithm

Cons:

- Very Hard/Impossible to Implement - requires knowledge of the future!




## 3- LRU – Least Recently Used Algorithm

Replaces the page that has not been referenced for the longest time.

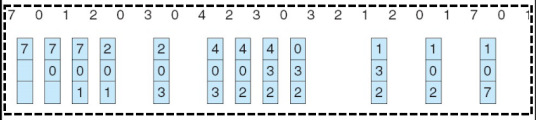
- By the principle of temporal locality, this should be the page least likely to be referenced soon.
- Each page could be tagged with the time of last reference.

Pros:

- Accounts for the past use of the pages, providing better performance than FIFO

Cons:

- Requires HW support to be efficient (to update last page reference time by sw, for every memory access, requires a great deal of overhead).




## 4- Additional-Reference-Bits Algorithm (LRU approximation)

While not providing full HW support to LRU, many systems provide a reference bit (use bit).

- The reference bit is set by HW per page whenever a page is accessed (r or w)

Allows to implement aging of pages (i.e. approximate the LRU)
Additional use info may be used if at regular intervals the OS saves the reference bits.

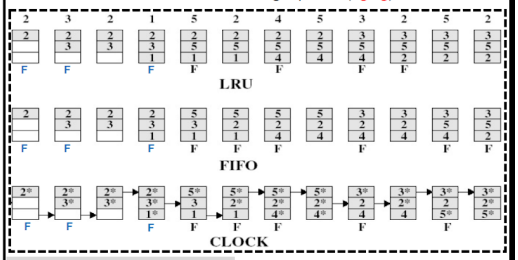
- E.g.
  - To keep history of last 8 periods, the reference bit for each page may be periodically recorded by the OS in a byte (for example each 100ms). The last reference value from each period, for each page entry, is inserted in the high order bit and the others are shifted right. The page with a lowest value is the LRU (more than one can have the same value)

## 5- Clock Replacement Algorithm (LRU approximation)

If only 1 bit is used the algorithm is called second-chance or clock algorithm

Algorithm

- When a page is first loaded into a frame in memory, set use bit =1
- Whenever a page is referenced (after the reference that generated the page fault), set use bit =1
- A pointer moves through pages, in a circular way, to indicate the next potential replacement victim.
- When it is time to replace a page, the first frame encountered with the use bit set to 0 is replaced
  - When a page is replaced, the pointer is set to the next frame after the one just updated
- During the search for replacement, each use bit found at 1 is set to 0 and has a second chance of not being replaced (aging)




# Frame allocation

How to allocate the fixed number of frames available to existing processes?

Lower number of frames allocated per process => higher number of page faults => slower execution

The minimum number of frames per process depends on the specific computer architecture

- E.g., If the instruction set has an instruction that can reference memory in 2 different pages, each process should have at least 2 frames allocated

The maximum number of frames depends on the available physical memory.

## Allocation algorithms

Equal allocation

- Each process gets the same number of frames
  # frames allocated = #available frames / # existing processes
  - Leftovers can be used as a free-frame buffer pool

However, a process may not need all the frames ... Different processes have different needs

Proportional allocation

- Allocate available memory to each process according to its size

Global replacement

- A process can select a replacement frame from the set of all frames (even from frames allocated to other processes)

Local replacement

- A process can only select a replacement frame from its own allocated frames

# Trashing

A high paging activity, with a high number of successive faults and replaces, where the time spent paging is longer than the time spent executing is called **trashing**.

- If a process does not have "enough" pages, the page-fault rate is very high, leading to a low CPU utilization!
- Operating system detects the low CPU use and thinks that it needs to increase the degree of multiprogramming to compensate
- Another process is added to the system
- More processes will have faults => High I/O => Low CPU use

To avoid trashing

- A local replacement algorithm may limit the effects
  - A process that starts trashing cannot "steal" pages from other processes, therefore limiting the trashing effects to other processes
  - However, the average time for servicing any page increases as one process keeps trying to swap pages

To prevent trashing a process must have as many pages as it needs

- One approach: working-set strategy

# Trashing Locality

## Locality

- Set of pages that are actively used together
- A process migrates from working with "one locality" to another
  - E.g., when a function is called a new locality is defined
- Localities actually overlap
- **If enough pages are allocated to accommodate the current process locality, it only faults until all that pages are in memory. After, no more faults occur until the locality is changed!**

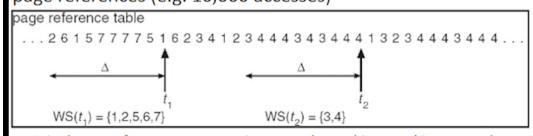
## Why does trashing occurs?

- Σ size of locality > total memory size

## Working-set

The working-set model is based on locality

Working-set of a process $p_i$ : set of pages in the most recent Δ page references (e.g. 10,000 accesses)




- It is the set of pages a process is currently working on. (Captures the current locality!)
- Working-set size of a process $p_i$ ($WSS_i$) is the number of pages in its set
- The accuracy of the working set depends on choosing the right Δ
  - if Δ too small ⇒ will not encompass entire locality
  - if Δ too large ⇒ will encompass several localities
  - if Δ = ∞ ⇒ will encompass entire program

## Working-set model use

***m*** = total number of frames available

Process **i** needs **$WSS_i$** pages

***D*** = Σ***WSS***$_i$ ⇒ total demand for frames in the system

if *D* > *m* ⇒ Thrashing

Policy:

- if *D* > m, then suspend one of the processes
  - Reduce multiprogramming
- If there are enough extra frames, then initiate another process

=> Keeps multiprocessing high and trashing low

# Storage Management

Main memory is usually too small to accommodate all programs and data permanently

⬇

Secondary storage options are needed

The file system provides mechanisms to enable storage and access to both programs and data

OS I/O subsystem must provide a simple interface to access all the different devices and optimize the I/O for maximum concurrency

## Mass-Storage Structure

# Disk scheduling

## Disk's structure

- Disks are addressed as large 1-dimension arrays of *logical blocks*, where the logical block is the smallest unit of transfer.
- Each array of logical blocks is mapped into the sectors of the disk sequentially.
  - Sector 0 is the first sector of the first track on the outermost cylinder.
  - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

## Disk scheduling – Performance parameters

- To read or write, the disk head must be positioned at the desired track and at the beginning of the desired sector
- Seek time
  - Time it takes to position the head at the desired track.
  - Typical seek time = 5-10ms
- Rotational delay or rotational latency
  - Time its takes for the beginning of the sector to reach the head.
  - Average rotational delay: 5ms (5400 rpm disk) - 3ms (10000 rpm)
- Access time
  - Ta = (seek time) + (rotational delay)
  - The time it takes to get in position to read or write.

- Transfer time (T)
  - Time to transfer the data once the head is in position
  - Data transfer occurs as the sector moves under the head

$$T = \frac{size}{r \times tracksize} \quad \text{(seconds)}$$

**size** = number of bytes to transfer
**r** = rotational speed (rotations per second)
**tracksize** = track size in bytes

- Average Total Access Time ($T_t$)
  - $T_t$ = (Seek time) + (rotational delay) + (transfer time)

$$T_t = T_{seek} + \frac{1}{2 \cdot r} + \frac{size}{r \cdot tracksize} \quad \text{(seconds)}$$

  - $T_{seek}$ = Average seek time (s)
  - r = Rotational Speed (RPS – rotations per second)
  - tracksize = Track size (bytes)
  - size = Size to transfer (bytes)

Consider a disk with average seek time of 10 ms, rotation speed of 10,000 rpm, 512-byte sectors with 320 sectors per track. Suppose we want to read a file with 2560 sectors for a total of 1310720 bytes.

What is the **Total Transfer Time?**

- Assuming **continuous allocation of the file**
- First, we assume that the file is stored contiguously on disk and occupies all the sectors in 8 adjacent tracks (8 tracks x 320 sectors/track = 2560 sectors).
  - This corresponds to a sequential storage of the file.
- 10000 rpm = 166.(6) rps; 1/2r = 1/(2x166,(6) )= 3 ms
- T1 = Time to read first track:
  - T1 = 10 (seek time in ms) + 3 (rotational delay) + 6 (read 320 sectors) = 19 ms
  - Next reads the seek time will be 0.
  - Each of the 7 successive tracks are read in (3 + 6) ms
    - If it must deal with rotational delay in each track; otherwise, it would be just 6ms
- Read a complete track: size/(r x tracksize) (320 x 512) / (166.(6) x 320 x 512)
- Total time = 19 + 7 x 9 = 82 ms = 0,082 seconds
- Assuming **scattered allocation of the file**
- Now, assume the file is spread over the disk and the accesses to the sectors are distributed randomly.
- 1 track = 6ms; 1 sector = 6/320
- For each sector we have:
  - T sector = 10 (seek) + 3 (rotational delay) + 0.01875 (read 1 sector)= 13.01875 ms
- Total time = 2560 x 13.01875 = 33328 ms = 33.328 seconds
- In this case it is 406 times slower!!!

## Disk performance – some observations

- Seek time is the main reason for differences in performance
  - Seek time ≈ seek distance
- The order in which the sectors are read from the disk has a tremendous effect on I/O performance.
- For a single disk request there will be a number of I/O requests.
- If requests are selected randomly, we will get the worst possible performance!

## Disk I/O operations also include several queuing delays

When a process needs I/O from a disk it issues a system call to the OS:

- If the device (disk drive) and I/O channel are available, the request is served
  - If the disk drive is available, it is assigned to the process
  - If the I/O channel used by the disk is shared between devices, additional wait time may be required
- If it is busy, the request is put in a queue
  - When the current request is completed, the OS must choose the next request to service

## Disk scheduling ALGORITHMS!

Example to compare disk scheduling algorithms:

- At the same time, different processes are reading from a disk. The requested blocks by reception time, are located in the following **cylinders**:
  - 98, 183, 37, 122, 14, 124, 65, 67
- The requests are put in a requests queue
- The disk head is initially at cylinder 53

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

- Is the simplest algorithm (FIFO)
- Is fair (every request is served and by the order received)
  - 1st: move from 53 to 98 … (…)
  - 2nd: move from 98 to 183 … Total head movement = 640 cylinders!

### Shortest-Service-Time-First (SSTF)

- Service request with minimum seek time from current head position
- May cause starvation if new requests appear that are closer to the head!
  - 1st: move from 53 to 65 … (…)
  - 2nd: move from 65 to 67 … Total head movement = 236

### SCAN (elevator algorithm)

- Disk arm starts at one end of the disk and moves toward the other end. At the end the head movement is reversed. Requests are serviced along the way.
  - Start at 53 … Go to 0 and reverse … Total head movement
  - Service 37 … Service 65 = 236 cylinders!
  - Service 14 … (…)

### Circular SCAN (C-SCAN)

- Same as SCAN but only services requests in one direction. Reaching the end of the disk, the head travels to the start
  - Provides a more uniform wait time than SCAN.

### LOOK and Circular LOOK (C-LOOK)

- "Smarter" version of SCAN and C-SCAN. Head only travels until it goes as far as the last request on the queue
  - In C-LOOK it then goes to the location of the first request (not to the other extreme).
- LOOK serves requests in both directions
- C-LOOK Serves requests in only one direction.

### Selecting a Disk-Scheduling Algorithm !!!

- SSTF is more common and better than FCFS.
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk as they avoid starvation.
- The computation of an optimal order of request service may not justify the savings over SSTF or SCAN.
- Performance depends on the number and types of requests.
- Requests for disk service can be influenced by the file-allocation method.
- The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary.
- Either SSTF or C-LOOK is a reasonable choice for the default algorithm.

### Disk-Scheduling and SSDs

- Disk scheduling focus on minimizing head movement in magnetic disk drives
- SSDs do not contain moving disk heads and so use a common FCFS policy
- In SSDs, normally the time required to service reads is uniform but, because of the properties of flash memory, write service time is not uniform.
  - Some SSD schedulers have exploited this property and merge adjacent **write** requests, servicing all **read** requests in FCFS order.

# RAID

Disks are currently a bottleneck to performance

Due to their mechanics, they are also prone to failures

As disks got smaller and cheaper, solutions that used multiple disks became economically feasible

Advantages of using multiple disks in parallel:

- Better speed when accessing data
- Better capacity at lower cost (multiple small cheaper disks **vs.** big high-performance disk)
- Increased reliability – redundant data can be stored on multiple disks; a failure in one disk does not lead to data loss

Disadvantages:

- The use of multiple devices increases the probability of failure

## Disk array

- Set of multiple disks that are organized to provide for better performance and reliability
  - Performance is obtained by using data striping
  - Reliability is obtained with redundancy
- The array of disks is seen by the OS as a single logical drive
- A single I/O request may produce multiple accesses to different disks, depending on the localization of the data blocks requested

## RAID (Redundant Array of Independent Disks)

- Disk organization technique that is used to address performance and reliability issues
  - Has an array of disks that operate independently and in parallel
- Implements striping and redundancy in different combinations (specific levels exist)
- In the past RAID was also used for economic reasons (used small cheap disks as a cheaper solution than big high-performance disks) and was known as Redundant Array of Inexpensive Disks.

## Availability

**Availability** - Percentage of time a system is available for performing the operations people expect it to perform (e.g. a disk actually serving I/O requests). Is related to MTBF (Mean Time Between Failures) and to MTTR (Mean Time to Recovery)

$$Availability = \frac{MTBF}{MTBF + MTTR}$$

### Serial Availability

$$Availability = Availability_A \times Availability_B$$

The availability of the whole is always lower than the availability of the individual parts!!!

### Parallel Availability

$$Availability = 1 - (1 - Availability_A) \times (1 - Availability_B)$$

(1 minus the probability of both not being available)

The availability of the whole is always (largely) superior to the availability of the individual parts!!!

# RAID Levels

| Category | Level | Description | I/O Request Rate (Read/Write) | Data Transfer Rate (Read/Write) | Typical Application |
|---|---|---|---|---|---|
| Striping | 0 | Nonredundant | Large strips: Excellent | Small strips: Excellent | Applications requiring high performance for noncritical data |
| Mirroring | 1 | Mirrored | Good/Fair | Fair/Fair | System drives; critical files |
| Parallel access | 2 | Redundant via Hamming code | Poor | Excellent | |
| | 3 | Bit-interleaved parity | Poor | Excellent | Large I/O request size applications, such as imaging, CAD |
| Independent access | 4 | Block-interleaved parity | Excellent/Fair | Fair/Poor | |
| | 5 | Block-interleaved distributed parity | Excellent/Fair | Fair/Poor | High request rate, read-intensive, data lookup |
| | 6 | Block-interleaved dual distributed parity | Excellent/Poor | Fair/Poor | Applications requiring extremely high availablity |

### RAID 0 - Non-redundant striping

- Does not include redundancy or provide data protection
- Simply strips blocks across different disks
- Best write performance of all RAID levels
- Lower cost - each disk is 100% used as there is no redundancy
- Can be implemented with any number of disks
- Availability is bad!
  - Adding more disks makes it worse.
- Requires a minimum of 2 drives

### RAID 1 – Disk mirroring

- Disk information is duplicated
- Does not use striping
- Allows parallel reading from different blocks but implies duplicated writes (the duplicated writes do not penalize time as are done in parallel)
  - Excellent read performance (twice the I/O)
- 50% of disk space is redundant -> High cost
- Very commonly used!
- Requires a minimum of 2 drives

### RAID 2 – Redundancy through Hamming Code

- Uses error correction codes
- Typically Hamming Code to enable the correction of 1-bit errors and the detection of 2-bit errors
- Simple parity code cannot correct errors and can detect only an odd number of bits in error
- Hamming codes can detect up to two-bit errors or correct one-bit errors without detection of uncorrected errors.
- Uses bit-level striping
- The reading implies using all disks, the ones that have the data and the auxiliary parity disks
- The writing of one block implies changing all disks

### RAID 3 – Bit-interleaved parity

- Like RAID 2, uses bit-level striping
- Disk controller can tell us which disk has failed. Thus, it is possible to use the other drives to reconstruct the data – a detection code like the one used in RAID 2 is not needed.
- Stripe parity is generated on writes and recorded on the parity disk; parity is checked on reads.
- RAID Level 3 requires a minimum of 3 drives to implement.
- Requires only 1 redundant disk (to save parity data), independently of the number of data disks -> low cost
- Redundancy
  - The failure of one disk still permits the availability of data
  - Lost info can be rebuilt using the other working disks
  - If more than one disk fails information cannot be recovered
- Performance
  - As data is divided in strips very high data transfer rates are possible
  - Any read involves parallel transfer of data from all data disks
  - Good for sequential reads (e.g., multimedia) but not for transaction processing as only one I/O operation can be executed at a time.

### RAID 4 – Block-interleaved parity

- The striping unit is a disk block
- Improves performance for reads, since for reading a block is only necessary to access one disk. Reading of different blocks is made in parallel.
- Writing implies a *read-modify-write* cycle with only one data disk and the parity disk being affected
  - Contention in the parity disk – only one write is possible at a time as the parity disk (only 1) must be updated
- Multiple reads at the same time are possible
- Requires a minimum of 3 drives

### RAID 5 – Distributed block-interleaved parity

- Improves RAID 4 by distributing the parity blocks by all disks instead of a dedicated one.
- Enables simultaneous writes and reads
- Only some writes are less efficient than RAID 1 due to the read-modify-write cycle
- The additional cost corresponds to 1 additional disk (a parity disk that that is distributed by all others)
- Permits recovery from the failure of 1 disk
- Best cost/performance for transaction-oriented networks; Very high performance, very high data protection; Can also be optimized for large, sequential requests.
- Very commonly used
- Requires a minimum of 3 drives

### RAID 6 – Dual redundancy

- An extension of RAID-5 : similar read and worst write performance
- Permits recovery from the simultaneous failure of 2 disks
- Requires 2 parity disks, each using a different parity generator algorithm
- Writing of a block implies a read-modify-write cycle with 6 disk accesses (3 reads and 3 writes, corresponding to the data disk and the 2 parity disks) - substantial write penalty when compared to RAID 5.
- Used in high-availability scenarios
- Requires a minimum of 4 drives

### RAID Levels – conclusions

- RAID 0, RAID 1 and RAID 5 are extremely common.
- If redundancy is not needed RAID 0 is the one with best performance at a lower price
- RAID level 1 is popular for applications that require high reliability with fast recovery.
- RAID 2 and RAID 4 are worst than RAID 3 and RAID 5 respectively
- RAID 3 is a good option to the transfer of big contiguous blocks of data
- RAID 5 is the more balanced choice
- RAID 6 is adequate to high availability scenarios

- Beware: many modern motherboards claim to support RAID-0, RAID-1 and/or RAID-5. In most cases, this is done in software, not hardware!
  - Performance wise, it's completely different!

- Sometimes RAID 0, RAID 1 and RAID 5 are combined:
  - RAID 0+1: Striping + Mirroring
  - RAID 1+0: Mirroring + Stripping
  - RAID 5+0: Block-Interleaved Distributed Parity + Striping
  - RAID 0+1+5: Striping + Mirroring + Block-Interleaved Distributed Parity

- E.g.: RAID 0+1 versus RAID 1+0
  - RAID 0 provides performance
  - RAID 1 provides reliability

# File System Structure

File systems provide efficient and convenient access to the disk (or other secondary storage) allowing data to be stored, located and retrieved easily.

The more common secondary storage is the disk.

The file system:

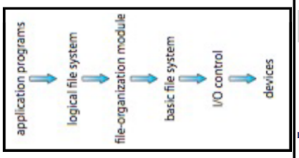
- Resides permanently on secondary storage.
- Provides a user interface to storage and mapping logical to physical.
- Provides an efficient and easy way to access the storage.

The file system structure provides a logical storage unit

To improve I/O efficiency, I/O transfers are performed in blocks, each block containing one or more disk sectors (if using a disk).

File systems are generally composed of different layers, each serving/using the adjacent ones.

The different layers reduce complexity but add overhead and decrease performance.




## Layered File System

- Application programs
  - Request I/O
- Logical File System
  - Manages metadata information (file system structures excluding data itself)
  - Manages the directory structure
  - Maintains file structure via file control blocks
    - File control blocks contain info about the file
  - Translates filename into a file number, file handle, location
- File-organization module
  - Translates **logical** block addresses to **physical**
  - Tracks unallocated blocks
- Basic File System
  - Issues generic commands to the **device driver** to read and write blocks of data
  - Manages **memory buffers** and **caches** that hold file system, directory and data blocks
- I/O control
  - **Device drivers** and **interrupt handlers** that permit the data transfer between main memory and the secondary storage system
  - Receives high level commands and translates them to specific device instructions (low level hardware commands)
- Devices
  - Secondary storage

# Sectors and Blocks

## Disk sector

- Smallest individual reference to disk
  - 512 bytes to 4096 bytes (4kB)

## File System Block

- Minimum unit of storage that the file system uses
  - 512 bytes, 1024 bytes (1kB), 2048 bytes (2kB), ..., 32678 bytes (32kB)
- Files smaller than a block size will have internal fragmentation
- Bigger blocks mean that OSs can address bigger disks with the same address size, but more space is wasted when files are not multiple of a block size

# File System Implementation

Several **on-disk** and **in-memory** structures are used to implement a file system. While they may be different in different OSs, the principles are similar.

On disk:

- Boot Control block (per volume)
  - Contains information on how an OS can boot from that volume. May be empty if no OS exists in the volume. Also known as "boot block" or "partition boot sector".
  - Usually the first block of volume.
- Volume Control block
  - Has volume or partition details such as the number, size of and free blocks. Also known as "super block" or "master file table"
- Directory structure (per file system)
  - Organizes the files
- FCB (File Control Block) (per file)
  - Contains details of the file

In memory:

- Mount table
  - Has information about each mounted volume
- Directory structure cache
  - Holds directory information of recently accessed directories
- System-wide open-file table
  - Has a copy of the FCB of each open file, among other information
- Per process open-file table
  - Has a pointer to the appropriate entry in the system-wide open-file table, among other information
- Buffers
  - Holds blocks that are being read or written to disk (or other secondary storage)

Creation of new files:

- Files are created by allocating a new FCB or selecting one free.
- In Unix, both files and directories are treated as files. In Windows they have different calls and are treated differently.
- To create directories, the file-organization module maps the directory I/O into disk-blocks

Open a file:

- The system call open() passes the filename to the logical file system
- A search is made in the open-wide open-file table to see if any other process is using the file.
  - If it is, a per-process open-file table entry is created pointing to the system-wide one.
  - If it is not, the file name is searched in the directory structure and, when found, the FCB is copied to the system-wide table. A pointer to that table is created in the per-process open file table.
- The per-process open-file table points to the respective entry in the system-wide open-file table, keeps a pointer to the current location of the file (for read and write operations of the file by the process) – File pointer, and saves the access mode in which the file was open.

Read a file:

- open() returns a pointer to the file entry in the per-process open-file table.
- All file operations are then performed via this pointer.
- This pointer is called file descriptor in Unix and file handle in Windows.

Close a file:

- Removes the per-process table entry
- Decrements the system-wide table entry that has a counter indicating how many processes have the file open
  - If the counter reaches 0 any updated metadata is copied to disk and the entry is removed

## Partitions and Mounting

- Partitions contain a file system ("cooked" partition) or none ("raw" partitions).
- Raw disks are used when no file system is needed (e.g., for databases, Unix swap files).
- Boot information can be stored in a separate partition with its own format (at boot time the OS is not loaded and cannot interpret the format).
- The boot loader loaded is executed and has information on the file systems used and where to find the kernel and start its execution.
- Dual-booted systems rely on the boot-loader to select and start one of the existing OSs.
- The root partition contains the OS kernel and is mounted at boot time. Other volumes can be mounted at boot time or later.
- A disk can have multiple partitions, each containing a different type of file system and a different operating system.

# Directory Implementation

Directories contain information (attributes, location, ownership, etc.) about files.

- Directories provide a mapping between file names and the files themselves.
- Many times, they are Hierarchical and Tree-Structured (but not necessarily)

Implementation of directories

- As a linear list
  - Linear list of filenames with pointers to the data blocks
  - Finding a file requires linear search
- Using a hash table
  - Complements the linear list with a hash table decreasing the access time

## Virtual File Systems

- Modern OSs concurrently support multiple types of file systems.
- How can it be achieved?
  - Different directory and file routines for each file system – too complex.
  - Use object-oriented techniques to simplify, organize and modularize the implementation – most used approach!
    - Allows different file systems to be implemented within the same structure
    - Even supports the use of network file systems
    - Uses a Virtual File System layer
- Three layers:
  - File system interface – open(), read(), write(), close()
  - Virtual file system (VFS)
    - Separate operations from implementation
    - Unique identification of each file across the network
  - Specific implementations of each file system type
- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is defined by the VFS interface, rather than by any specific type of file system.

# File allocation Methods

Let's assume that we want to create a new file…

How are blocks allocated to the file?

- Space must be used effectively
- Files must be accessed quickly

### Contiguous allocation

- Requires that each file occupies a set of contiguous blocks on the disk
- Is defined by a disk address and a length – in block units (block address and length in number of blocks)

Pros:

- Simple to implement
- Random access to any part of the file (only necessary to know where it begins)

Cons:

- Waste of space due to **external fragmentation** – compaction is an option but the performance penalty is high
- Dynamic storage-allocation problem – where to allocate given N free spaces?
- Files cannot grow – estimate the size of a new file Is not always simple

### Linked allocation

- Each file is a linked list of disk blocks placed anywhere in the secondary storage
- Directory contains a pointer to the first and last file blocks
- Each block contains a pointer to the next block
- Pros:
  - Simple – needs only the start address
  - Free-space management system – no waste of space
  - No external fragmentation
- Cons:
  - No random access (to read a specific block of the file, the blocks must be followed till the block is found, starting at the first)
  - No accommodation of the principle of locality (data blocks are spread all over the disk)
  - Space waste by the pointers (e.g. 4bytes in a 512-byte block)
  - Reliability (a pointer can be lost)
- An important variation => FAT / FAT32

### FAT/FAT32 (variation of Linked allocation)

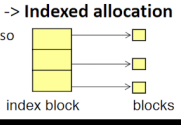
- Has a File Allocation Table (FAT) in the beginning of the disk
- The table has one entry for each disk block, indexed by block number
- Each FAT entry points to the next block that a file is occupying
- Last block contains a special EOF (end-of-file) value
- An unused block contains value 0
- Each directory entry contains the name and attributes of a certain file, as well as the number of its first block
- Directories are implemented as "special files"
- If FAT is not cached it implies many disk head seeks (read FAT, read block, read FAT, read block,…)

### Indexed allocation

- In **linked allocation** pointers are scattered all over the disk what results in:
  - Without a FAT it does not support efficient random access
  - Poor support for locality!

**Solution:** Bring together all pointers! -> **Indexed allocation**

- One special block called Index Block does so
  - inode (indexed node) in Unix!
- Each file has its own Index block, which is an array of disk-block addresses




### Indexed allocation (2)

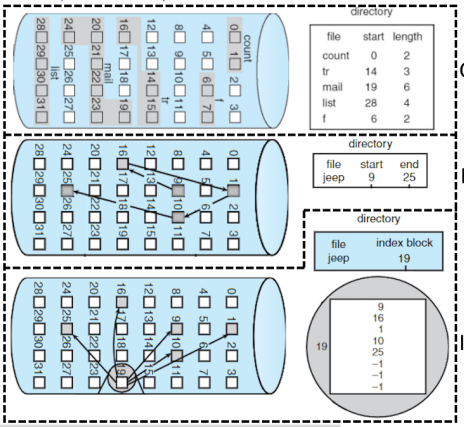
- The directory contains the address of the index block
- To find block *i* the $i^{th}$ pointer entry in the index block is used
- Pros:
  - Supports direct access (random access)
  - No external fragmentation
- Cons:
  - Space wasted by the index block (e.g., A file occupying 2 blocks must have an additional just for the index block!)
  - No accommodation of the principle of locality (data blocks are spread all over the disk)
- To minimize the space overhead, the size of the index block should be as small as possible
  - Some files may not have sufficient space to hold all pointers needed!

Example:

- Consider an index block of 4KB
- Consider that each entry of index is a 32 bit address representing a disk block
- Each disk block is also of 4KB
  - The number of pointers in the index block is:
    - 4KB (index block) / 4bytes (address) = 1024
  - Thus, the maximum file size is:
    - 1024 x 4KB = **4MB** <= Largely insufficient!

Unix solution: **inodes (**indexed **nodes)**

- There is one inode per file.
- The inode object holds all the information about a named file except its name and the actual data contents of the file
- Items contained in an inode object include owner, group, permissions, timestamp, etc.
- Uses combined multi-level indexing
- 10 direct pointers (for the first 40KB of the file – uses 4KB disk blocks)
- 1 single indirect pointer (for the next 4MB of the file)
- 1 double indirect pointer (for the next 4GB of the file)
- 1 triple indirect pointer (for the next 4TB of the file)
- Directories implementation
- Directories are just normal files marked with a "directory" attribute.
- The content of a "directory file" consists in a mapping of the file names it contains to its corresponding inodes.
- Sub-directories point to the corresponding "sub-directory file" inode.
- The parent directory points to the corresponding "parent-directory file" inode.
- The "super-block" on the partition contains the inode of the root directory




# Free-Space Management

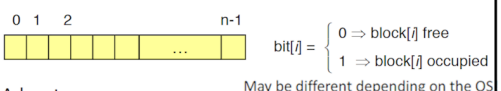
Let's assume that we want to create a new file…

How do I know which blocks are available? How do I reuse space from deleted files?

To keep track of free space the OS maintains free-space "list" (may not be a list) that records of all free blocks

### Bit Vector

- The free-space "list" is implemented as a bit vector where each block is represented by a bit.




May be different depending on the OS

- Advantages:
  - Simple to implement
  - Fast to search the first available block
- Disadvantages:
  - Becomes inefficient unless it can be kept in memory at all times
  - (1TB disk with 4KB blocks => 256 MB bitmap!)

### Linked List of Free Blocks

- Linked list of free blocks, keeping a pointer to the first one
- In general, it's not efficient:
  - To traverse the list it's necessary to read several blocks
- Optimizations
  - **Grouping:** store the addresses of N free blocks in the first entry of the list
  - **Counting:** allocation and releasing of blocks is normally done in groups; thus, store the addresses of contiguous blocks in the list – each entry consists of address + number of contiguous free block

# Efficiency and Performance

### Efficiency

- The efficiency of disk usage depends on:
  - Disk allocation and directory algorithms
  - Types of data kept in the file directory entry
    - (e.g., Maintaining a last access/write data implies added overhead as the directory entry must be rewritten)
  - Size of pointers used to access data
    - (e.g., 64 bits requires more space that using 32 bits – more disk space used)

### Performance

- Independently of the algorithms used, performance can be further improved by using other techniques
  - disk cache – use main memory to keep frequently used blocks
  - free-behind / read-ahead – techniques to optimize sequential access
  - improve PC performance by dedicating section of memory as virtual disk, or RAM disk
- Once blocks are read from the disk cache (or other secondary storage) to the main memory, the OS may cache them:
  - **Buffer cache** – blocks are kept under the assumption that will be used again soon
  - **Page cache** – uses virtual memory techniques to cache file data rather than file blocks
    - File accesses use virtual memory accesses rather than the file system, enhancing the performance
  - **Unified virtual memory** – some systems use page caching to cache both process pages and file data
    - E.g., Solaris, Linux, Windows
- Page cache and buffer cache
  - Page cache caches pages rather than disk blocks using virtual memory techniques.
  - Memory-mapped I/O uses a page cache
  - Routine I/O through the file system uses the buffer (disk) cache

**Double caching**

- Requires caching file system data twice
- Wastes memory and I/O cycles due to the extra data movement within system memory

### Unified Buffer cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O.
- Used in some versions of Unix.

# Recovery

Data loss or data inconsistency, due to system failure, must be avoided

Different methods are used to deal with file system corruption

### Consistency Checking

- Whatever the cause of corruption in data/OS structures, the file system must be able to detect the problems and then correct them.
- Consistency checker - compares data in directory structure with data blocks on disk and tries to fix inconsistencies (E.g.: fsck in Unix; chkdsk in MS_DOS)

### Log-Structured (or journaling) File Systems

- The file system records each update as a transaction
- All transactions are written sequentially to a log file
- A transaction is committed once it is written to the log; at this time, the system call can return to the user process
  - However, some of the file system structures may not yet be updated
    - The transactions in the log are asynchronously written to the file system
    - When all required changes are done, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Used in most operating systems today:
  - NTFS, Veritas, EXT3, XFS, ReiserFS

### Backup and restore

- Backups of a file system can be done using specific system programs
  - E.g., Backup of a magnetic disk to another secondary storage (such as floppy disk, magnetic tape, other magnetic disk, optical, cloud)
- Different backups may be done
  - Full-backups
  - Incremental backups
- In case of a failure the recovery consists in restoring the data saved in the backup

# SECURITY

Ensures the authentication of system users

Protects the integrity of data and code

Protects the physical resources available (e.g., CPU, memory, disks, networking)

Prevents unauthorized access, destruction or alteration of data

Considers the external environment where the system operates

A system is secure if resources are used and accessed as intended under all circumstances

- Unachievable
- Security breaches must be minimized

Security breaches can be intentional (malicious) or accidental

Some terms used:

- Intruders – those who attempt to breach security
- Threat – a potential security violation (e.g., existing vulnerability)
- Attack - attempt to break security

Security violation categories

- Breach of confidentiality
  - Unauthorized reading of data
- Breach of integrity
  - Unauthorized modification of data
- Breach of availability
  - Unauthorized destruction of data
- Theft of service
  - Unauthorized use of resources
- Denial of service (DOS)
  - Prevention of legitimate use of the system

Security violation methods used by attackers

- Masquerading (breach authentication)
  - The breach of authentication results in an incorrect identification of the user by the system; the user can gain access or escalate privileges that should not have.
- Replay attack
  - Repeats a valid transmission of data; can also involve message modification
- Man-in-the-middle attack
  - Intruder sits in data flow, masquerading as sender to receiver and vice-versa
- Session hijacking
  - Intercept an already-established session to bypass authentication

Security must occur at four levels to be effective:

- **Physical**
  - Data centers, servers, connected terminals must be physically secured against entry by intruders
- **Human**
  - Give user authorizations to system access carefully
  - Avoid social engineering - phishing, dumpster diving
- **Operating System**
  - Protection mechanisms, debugging
- **Network**
  - Intercepted communications, interruption, DOS

## Program Threats

Threats caused by the use of programs

### Trojan Horse

- Code segment that misuses its environment
  - When executed by an user, grants the rights from that user to the user that created the program
- Login spoofing (variation of a Trojan horse)
  - Emulates a login program – steals user/password from a user that thinks that is logging into the system
- Spyware (variation of a Trojan horse)
  - Comes with other software (either freeware, shareware or even commercial software)
  - Normally download ads, open pop-up windows or capture information of user system and return it to a central server
  - Some create covert channels in which unknown communication occurs without user knowledge
  - Up to 90% of spam is delivered by spyware-infected systems (theft of service)

### Trap Door

- Hole left by a program or system designer to allow him to access the program system
  - Specific user identifier or password that circumvents normal security procedures
- A trap door can be included in a compiler that generates code with a specific trap door (the source code of the program, if inspected, would not contain any problem!)
- To detect a trap door an inspection of all the source code of all the components of a system must be made!!

### Logic Bomb

- Program that initiates a security incident under certain circumstances.

### Stack and Buffer Overflow

- Most common way for an attacker out of the system gain unauthorized access to the target system
- An authorized user may also use it to escalate privileges
- Exploits a bug in a program (overflow of either the stack or memory buffers)
  - Failure of programmer to check bounds on inputs, arguments
- Procedure:
  - Overflow an input field, command-line argument or input buffer until it writes into the stack
  - Overwrite the current return address on the stack with the return address of the exploit code. When routine returns from call, returns to hacked address.
  - The following overflow data is the exploit code to be called from the stack

### Viruses

- Code fragment embedded in legitimate program
- Self-replicating, designed to infect other programs/computers
- Very specific to CPU architecture, operating system, applications
- Usually via email or as a macro
- File / parasitic
  - Appends itself to a file and its execution does not prevent the execution of the host program.
- Boot
  - Infects the boot sector of the system and executes before the OS. Also known as memory viruses as they do not appear in the file system.
- Macro
  - Written in high level language. Executed by a program that runs that type of macros (e.g. spreadsheet)
- Source code
  - Looks for source code and includes the virus in that code.
- Polymorphic
  - Changes every time it is installed to avoid having a virus signature
- Encrypted
  - Is encrypted to avoid detection. The virus first decrypts and then executes.
- Stealth
  - Avoids detection by changing parts of the system that could detect them.
- Tunneling
  - Install itself in the interrupt-handler chain to avoid detection.
- Multipartite
  - Infects multiple parts of the system.
- Armored
  - Coded to be hard to understand and therefore be harder to detect and avoid.

### Rootkits

- Program(s) that enable(s) administrator-level access to a computer or computer network. Their main aim is to be hidden and remain undetected.
- Five kinds of rootkits by place where they hide:
  - a) Firmware rootkit
    - Hidden in the BIOS
  - b) Hypervisor rootkit
    - Runs the OS in a virtual machine under its control.
  - c) Kernel rootkit
    - Hides as a loadable kernel module
  - d) Library rootkit
  - e) Application rootkit

### Some defenses possible

- Anti-virus
- Code-signing
  - Only run unmodified code form reliable software vendors
- Sandboxing
  - Limits execution to a limited range of virtual addresses enforced at run time.
- Interpretation
  - Run untrusted applets interpretively and not let them control the hw (e.g. interpret Java applets in a browser). Each instruction can be analyzed before being executed.

## User Authentication

The OS protection system depends on the ability to identify the programs and processes currently executing, which in turn depends on the ability to identify each user of the system.

- User authentication is crucial to identify an user correctly, as protection systems depend on user ID

User identity is often established through one or more of:

- user knowledge of something (e.g., user identifier and password)
- user possession of something (e.g., a card or key)
- user specific attribute (e.g., fingerprint, retina pattern)

## Passwords

Most common approach used

Different passwords may be required for different access rights (for convenience this does not happen often)

Password vulnerabilities

- If simple, can be guessed (or brute forced)
- Accidental exposure (e.g. password written in a paper, shoulder surfing – intruder sees the user typing the password)
- Sniffing (can be done by another user with access to the network)
- Trojan horse keystroke logger
- Transference from one user to other (password borrows!!)

Passwords must be kept secret – how can the system try to accomplish it?

- Demand frequent change of passwords
- Keep history of old passwords to avoid repeats
- Check for ease of guessing
- Log all invalid access attempts (but not the passwords themselves)
- Use one-time passwords
  - Use a function based on a seed to compute a password – the system presents one part of the password and the user responds based in a function
  - Hardware device / calculator to generate the password
- Use biometrics
  - Use some physical attribute (fingerprint, hand scan)
- Use multi-factor authentication

How to keep the password secret within the computer?

- Only store encrypted password, never decrypted
- Use algorithm easy to compute but difficult to invert
- Traditional Unix systems keep user account information, including one-way encrypted passwords, in a text file called '/etc/passwd'. As the file is readable it can pose security risks.

## Implementing Security Defenses

Defense in depth is most common security theory – uses multiple layers of security

Implement a Security policy - describes what is being secured and how to implement the security

Vulnerability assessment compares real state of system / network compared to security policy

Intrusion detection endeavors to detect attempted or successful intrusions

- Signature-based detection spots intrusions based on known bad patterns
- Anomaly detection spots differences from normal behavior
  - Can detect zero-day attacks (i.e. detect previously unknown methods of intrusion)
  - False-positives and false-negatives are a problem

Virus protection

- Searching all programs or programs at execution for known virus patterns
- Or run in sandbox so can't damage system
- Avoid opening any e-mail attachments from unknown users
- Practice safe computing – avoid sources of infection, download from only "good" sites, etc.

Auditing, accounting, and logging of all or specific system or network activities

- May decrease system performance

A firewall is placed between trusted and untrusted hosts

- The firewall limits network access between these two security domains
- However, it can be tunneled or spoofed
  - Tunneling allows disallowed protocol to travel within allowed protocol (i.e., telnet inside of HTTP)
  - Firewall rules typically based on host name or IP address which can be spoofed

Personal firewall is software layer on given host

- Can monitor / limit traffic to and from the host

Application proxy firewall understands protocols used by applications and can control them (i.e., SMTP)

System-call firewall monitors all important system calls and apply rules to them (i.e., decides if a program can execute a specific system call)

# PROTECTION

Security is easier to achieve if there is a clear model of what is to be protected and who is allowed to do what.

Protection refers to mechanisms for controlling the access of programs, processes or users to available system resources

In one protection model, computer consists of a collection of objects, hardware or software

Each object has a unique name and can be accessed through a well-defined set of operations

Protection problem

- ensure that each object is accessed correctly and only by those processes that are allowed to do so

## Principles of Protection

Guiding principle in designing protection of an OS – principle of least privilege

- Programs, users and systems should be given just enough privileges to perform their tasks
- Limits damage if entity has a bug or gets abused
- The privileges can be static (during the life of a system or process)
- Or dynamic (changed by process as needed) – domain switching, privilege escalation
- A similar concept is applied to data access - "Need to know"
  - If the process does not have to know it is not given permission to access data

"Grain" aspects to consider

- Rough-grained privilege management is easier and simpler - least privilege is now done in large chunks
  - For example, traditional Unix processes either have the privileges of the associated user or of the root
- Fine-grained management is more complex and implies more overhead but is more protective
  - File ACL (Access Control Lists), RBAC (Role-Based Access Control)

## Domain Structure

To facilitate protection, a process operates within a protection domain which specifies the resources it may access. The ability to execute an operation on an object is an access right.

Access-right = *<object-name, rights-set>*

Domain = set of access-rights

A domain can be realized in different ways such as user, process or procedure

### Access Matrix

The domain model of protection can be viewed as a matrix

- Rows represent domains
- Columns represent objects

| domain \ object | $F_1$ | $F_2$ | $F_3$ | printer |
|---|---|---|---|---|
| $D_1$ | read | | read | |
| $D_2$ | | | | print |
| $D_3$ | | read | execute | |
| $D_4$ | read write | | read write | |

# Virtualization

Inside a virtual machine, guest operating systems run in an environment that seems to them as native hardware, creating the illusion that each environment has its own private computer.

- The environment behaves as if it was native hardware but also protects, manages and limits the virtual machines.

Virtualization enables a single machine to simultaneously run multiple OSs or different instances of the same OS.

Virtualization creates a virtual system on which operating systems and applications can run.

A virtual machine abstracts the hardware of a single computer into several different execution environments

Several components

- **Host** – underlying hardware system
- **Virtual machine manager** (**VMM**) or **hypervisor** – creates and runs virtual machines by providing an interface that is ***identical*** to the host
  - (Except in the case of paravirtualization)
- **Guest** – process provided with virtual copy of the host
  - Usually, an operating system

A single physical machine can run multiple operating systems concurrently, each in its own virtual machine

# Implementations

Type 0 hypervisors

- Hardware-based solutions that provide support for virtual machine creation and management via firmware
  - IBM LPARs and Oracle LDOMs are examples

Type 1 hypervisors

- Operating-system-like software built to provide virtualization
  - Including VMware ESX, Joyent SmartOS, and Citrix XenServer
- Also includes general-purpose operating systems that provide standard functions as well as VMM functions
  - Including Microsoft Windows Server with HyperV and RedHat Linux with KVM

Type 2 hypervisors

- Applications that run on standard operating systems but provide VMM features to guest operating systems
  - Including VMware Workstation and Fusion, Parallels Desktop, and Oracle VirtualBox

# Other variations

Paravirtualization

- The guest OS is modified to work in cooperation with the VMM to optimize performance

Programming-environment virtualization

- VMMs do not virtualize real hw but create an optimized virtual system
  - Used by Oracle Java and Microsoft.Net

Emulators

- Allow applications written for one hardware environment to run on a very different hardware environment, such as a different type of CPU

Application containment

- Not virtualization at all but rather provides virtualization-like features by segregating applications from the OS, making them more secure, manageable
  - Including Oracle Solaris Zones, BSD Jails, and IBM AIX WPARs

# Benefits and features

Sharing of Resources

- Cost efficiency in using hardware
- Cost efficiency in using space
- Cost efficiency in using energy

Convenience and Maintainability

- Run multiple OS on the same machine
- Simplified "centralized" machine management
- Faster deployments using disk images and snapshots
- A VM can be suspended, resumed, cloned

Simpler High-Availability Solutions

- Simpler redundant deployment scenarios
- Ability to move machines while running between servers to optimize workload

Security

- Host system protected from VMs, VMs protected from each other

# Disadvantages

Sharing of physical machines impacts…

- Performance Isolation (CPU, IO)
- Failure isolation between virtual machines
- A crash of the host machine can affect many different VMs

Higher entrance cost

Security

# Building blocks

It is generally difficult to provide an exact duplicate of an underlying machine

- Especially if only dual-mode operation is available on CPU
  - user and kernel modes
- Is getting easier over time as CPU features support VMM
- Most VMMs implement a virtual CPU (vCPU) to represent state of CPU per guest, as guest believes it to be
  - When guest context is switched onto CPU by VMM, the information from vCPU is loaded and stored

## Some virtualization techniques

### Trap-and-Emulate

- Virtual machine guests execute in user mode
- Only host Kernel runs in kernel mode
  - Not safe to let guest kernel run in kernel mode too!
- A VM needs two modes – virtual user mode and virtual kernel mode
  - Both of which run in real user mode
- How can actions such as system calls, interrupts or execution of privileged instructions, be executed by the virtual kernel?
- Actions in guest that usually cause switch to kernel mode must cause switch to virtual kernel mode
  - When the kernel in the guest attempts to execute a privileged instruction, that causes an error (because the system is in user mode) and causes a trap to the VMM in the real machine.
  - The VMM gains control and executes (or "emulates") the action that was attempted by the guest kernel on behalf of the guest. It then returns control to the virtual machine -> trap-and-emulate
- User mode code in guest runs natively on the hw with the same performance as the native system
- Kernel mode code runs slower due to trap-and-emulate
- Especially a problem when multiple guests running, each needing trap-and-emulate
- Hardware support in CPUs and extra modes provided by the CPU minimize the problem

### Binary translation

- If guest vCPU is in user mode, guest can run instructions natively
- If guest vCPU is in kernel mode (guest believes it is in kernel mode)
  - VMM examines every instruction the guest is about to execute by reading a few instructions ahead of program counter
  - Non-special-instructions run natively
  - Special instructions translated into new set of instructions that perform equivalent task
- Basics are simple, but implementation very complex!
- Implemented by translation of code within VMM
- Code reads native instructions dynamically from guest, on demand, generates native binary code that executes in place of original code
- Performance of this method would be poor without optimizations
  - Products like VMware use caching
  - Translate once, and when guest executes code containing special instruction cached translation used instead of translating again
  - Testing showed booting Windows XP as guest caused 950,000 translations, at 3 microseconds each, or 3 second (5 %) slowdown over native

### Hardware assistance

- All virtualization needs some HW support
- More support -> more feature rich, stable, performance of guests
- Intel added new VT-x instructions in 2005 and AMD the AMD-V instructions in 2006
  - CPUs with these instructions remove need for binary translation
  - Generally, define two more CPU modes – "guest" and "host"
  - VMM can enable host mode, define characteristics of each guest VM, and then switch to guest mode, passing control to guest OSs
  - In guest mode, guest OS thinks it is running natively, and only sees devices that were defined by the VMM for that guest
  - Access to a virtualized device causes a trap to the VMM (that takes control of the interaction)
  - VCPUs (Virtual CPUs) are provided to save CPU context switches of guests

## Virtualization and OS components

How do VMMs provide core OS functions?

### CPU Scheduling

- Virtualization sw presents one or more virtual CPUs to each VM running;
- The VMM has its own threads and guest threads (from the VMs) to run
- If there are enough physical CPUs, the VMM can allocate one to each virtual CPU requested by each guest
  - In this case, each guest acts like a native OS on a native CPU
- When not enough physical CPUs are available, the VMM schedules its own threads (for guest management, I/O management, etc.) and the guest threads, on available physical CPUs
- Even if the scheduler algorithm used by the VMM provides fairness between guest VMs (e.g., allocating physical CPUs in the same proportion of the allocated guest CPUs) it may result in poor user response
- Examples:
  - An OS that tries to give a limited time slice to each process, when virtualized, cannot guarantee that time slice as the VMM scheduling may deeply affect this time
  - Real time OSs cannot guarantee their timings
  - As each guest OS believes that it own the CPU, receiving all the CPU cycles available, the fact that it does not may affect some of its functions, such as the time-of-day clock. To correct this, VMMs supply applications to install in each type of guest OS permitted. This applications correct clock drift and supply other functions such as virtual device management

### Memory Management

- Memory is one of the major keys to performance
- More users of memory add more pressure on memory use
- VMMS usually overcommit memory – the total memory configured in guests usually exceeds the available physical memory
- Before memory optimization can occur, VMMs establish how much real memory each guest will get based on their initial needs, system load, etc.
- To reclaim memory from the guests, the VMM uses different mechanisms:
  - It uses a nested page table that re-translates the guest page table to the real page table; this additional indirection level can be used to optimize the guest's use of memory, without its knowledge
  - Another method to reduce memory pressure is to determine if a given page has been loaded more than once

### I/O

- VMs do not always represent hardware exactly, when I/O is concerned
- Virtualization provides specific virtualized devices to guest OSs
- I/O devices may be dedicated to guests, allowing each guest direct access to those devices, greatly improving performance
- Shared access to devices implies that the VMM must provide protection in its use. VMMs become part of every I/O operation, checking it for correctness and routing data to and from devices and guests
- In networking VMMs act as virtual switch to route network packets between the guest addresses; VMMs may also provide bridging (direct access) or NAT to guests

### Storage Management

- Both boot disk and general data access need be provided by VMM
- Need to support potentially dozens of guests per VMM (so standard disk partitioning not sufficient)
- Type 1 hypervisors – storage guest root disks and config information within file system provided by VMM as a disk image
- Type 2 hypervisors – store as files in file system provided by host OS
  - To create a new guest, it is enough to duplicate the file
  - By moving the file to another system, the guest is moved
- Physical-to-virtual (P-to-V) convert native disk blocks into VMM format
- Virtual-to-physical (V-to-P) convert from virtual format to native or disk format
- VMM also needs to provide access to network attached storage (just networking) and other disk images, disk partitions, disks, etc.

Pages may be referenced according to the table. (i.e. pag. 5.3 means bottom left corner of page 5)

| | | |
|---|---|---|
| 1 | 4 | 7 |
| 2 | 5 | 8 |
| 3 | 6 | 9 |

## Table of Contents



*1

Page Fault Rate ($p$)

- $0 \le p \le 1.0$
- if $p = 0$, no page faults
- if $p = 1$, every reference is a fault

Memory access time ($ma$)

- Normally ranges from 10-200 ns

**Effective Access Time (EAT)**

EAT = $(1 - p) \times ma + p \times$ '*page fault overhead*'

[swap out + swap in + OS overhead]

# Deadlocks

**claim / max M = alloc M + requests M !!!**

Consider the following scenario where 5 processes (P1 .. P5) are using 4 different types of system resources (A..D). Next tables present the resources allocated, the maximum number of resources needed and the resources available.

Allocation:

| | A | B | C | D |
|---|---|---|---|---|
| P1 | 0 | 0 | 1 | 2 |
| P2 | 1 | 0 | 0 | 0 |
| P3 | 1 | 3 | 5 | 4 |
| P4 | 0 | 6 | 3 | 2 |
| P5 | 0 | 0 | 1 | 4 |

Max:

| | A | B | C | D |
|---|---|---|---|---|
| P1 | 0 | 0 | 1 | 2 |
| P2 | 1 | 7 | 5 | 0 |
| P3 | 2 | 3 | 5 | 6 |
| P4 | 0 | 6 | 5 | 2 |
| P5 | 0 | 6 | 5 | 6 |

Available:

| A | B | C | D |
|---|---|---|---|
| 1 | 5 | 2 | 0 |

a) Is the system in a safe state? Use the Banker's algorithm to prove.

P1 já está em condições de ser completado. Com os 2 recursos D ganhos e 1 recurso A, P3 pode ser completado. (...) P2, P4, P5. Dado que todos os processos conseguiram terminar, o sistema está num safe state.

b) If P2 requests the resources [0,4,2,0] should it be allowed?

Completando P1, P3, P4 e P5 anteriormente o available vector tem os valores [3, 20, 18, 16]. Neste cenário, P2 ainda precisa de [1, 7, 5, 0] + [0, 4, 2, 0] - [1, 0, 0, 0] = [0, 11, 7, 0]. Dado que nenhum dos valores ultrapassa o AV, o pedido deve ser permitido.

Consider the following scenario where 4 processes are using 4 different types of system resources.

Allocation Matrix A:

| | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| P1 | 0 | 0 | 1 | 0 |
| P2 | 0 | 0 | 1 | 1 |
| P3 | 2 | 0 | 0 | 1 |
| P4 | 0 | 1 | 2 | 0 |

Requests Matrix Q:

| | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| P1 | 1 | 0 | 2 | 0 |
| P2 | 2 | 0 | 0 | 1 |
| P3 | 1 | 0 | 1 | 1 |
| P4 | 2 | 1 | 0 | 0 |

Vector Available

| R1 | R2 | R3 | R4 |
|---|---|---|---|
| 2 | 1 | 0 | 0 |

Are any of the processes in a deadlock? Apply the deadlock detection algorithm to answer.

Sim, P4 e P1 podem ser completados, mas P2 e P3 necessitam ambos de 2 R4, sendo que cada um tem 1 R4 alocado.

# Scheduling

Notes to exercises

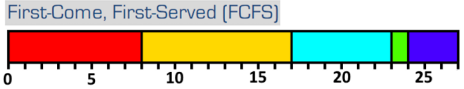
- If two processes arrive at the same time, the one that first enters the ready-queue is the one that is in the higher line in the table where the processes characteristics are defined;
- When a new process arrives and, at the same time, the process that is running leaves the processor, it is the process that was running that will go to the end of the ready-queue (i.e., the process that just arrived gets first into the ready-queue);
- If two processes have the same characteristics, the one closer to the head of the ready-queue is chosen.

Consider 5 processes (P1...P5) with the following behaviour:

| Process | Arrival time | Processing time (ms) (CPU burst) | Priority |
|---|---|---|---|
| P1 | 0 | 8 | 4 |
| P2 | 2 | 6 | 1 |
| P3 | 2 | 1 | 2 |
| P4 | 1 | 9 | 2 |
| P5 | 3 | 3 | 3 |

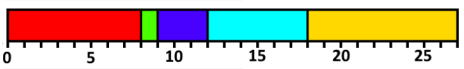
- **B.1** – Assume that the highest priority is **1** and use a Gantt to depict the scheduling of the processes using the following algorithms:
- **B.2** – For each algorithm create a table with the following values:
  (a) *Turnaround Time*; (b) *Waiting Time*; (c) *Finish Time*

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

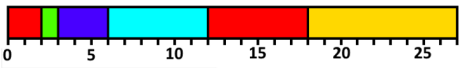



| | | | | | |
|---|---|---|---|---|---|
| waiting | 0 | 15 | 21 | 7 | 21 |
| turnaround | 8 | 21 | 22 | 16 | 24 |
| finish | 8 | 23 | 24 | 17 | 27 |

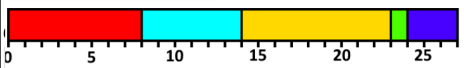
## Shortest-Job-First (SJF) Non-preemptive



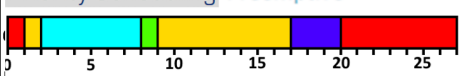

## Shortest-Job-First (SJF) Preemptive




## Priority Scheduling Non-preemptive

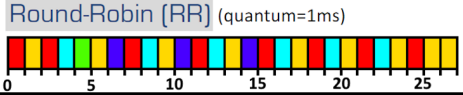



## Priority Scheduling Preemptive




| | | | | | |
|---|---|---|---|---|---|
| waiting | 19 | 0 | 6 | 7 | 14 |
| turnaround | 27 | 6 | 7 | 16 | 17 |
| finish | 27 | 8 | 9 | 17 | 27 |

## Round-Robin (RR) (quantum=1ms)



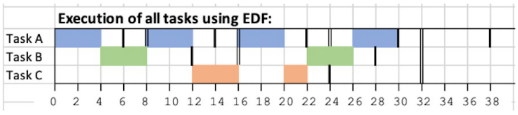

The table below describes three periodic tasks.

| | C[1] | D[2] | T[3] |
|---|---|---|---|
| Task A | 4 | 6 | 8 |
| Task B | 4 | 12 | 16 |
| Task C | 6 | 24 | 32 |

- Construct a timing diagram for the execution of the tasks that spans from time t=0 to time LCM (Task A, Task B, Task C)[4]. Create the diagram by simulating the execution of the tasks using EDF scheduling. The first instance of each task arrives at time 0.

[1] C = Worst case execution/computing time
[2] D = Deadline (within the period)
[3] T = Period
[4] Shortest repeating cycle = least common multiple (LCM) of periods

LCM(8, 16, 32) = 32




| | Task A | Task B | Task C |
|---|---|---|---|
| Finish time of last run | 30 | 26 | 22 |
| Number of runs | 4 | 2 | 1 |

# Memory

Consider the following free memory partitions (by order):

- 100KB, 500KB, 200KB, 300KB and 600KB

The following requests to place programs in memory are made (by order of the request) :

- 212KB, 417KB, 112KB, 426KB

What is the result of executing Best-Fit, First-Fit and Worst-Fit algorithm? And which makes the most efficient use of memory?

| | 100KB | 500KB | 200KB | 300KB | 600KB |
|---|---|---|---|---|---|
| Best-Fit | | 417KB | 112KB | 212KB | 426KB |
| First-Fit | | 212KB | 112KB | | 417KB |
| Worst-Fit | | 417KB | | 112KB | 212KB |

Best-fit é o mais eficiente.

Consider a logical address space of 64 pages of 1024 bytes each, mapped onto a physical memory of 32 frames.

a) How many bits are there in the logical address?
Logical address: 16 bits $2^6 = 64$ ; $2^{10} = 1024$

| Page number = $2^6$ | Page offset = $2^{10}$ |
|---|---|

b) How many bits does the physical address have?
Physical address: 15 bits 32 frames = $2^5$ frames $2^5 = 32$ ; $2^{10} = 1024$

| Frame number= $2^5$ | Frame offset = $2^{10}$ |
|---|---|

Consider a computer with 256 MBytes of physical memory having a virtual memory of 2 GBytes through paging. The number of pages occupies the first 20 more significant bits of the logical address.

a) What is the maximum number of pages?
- $2^{20}$

b) What is the size of each page?
- 2KB 256MB / 2^20 bits = (2^28 / 2^17) Bytes = 2^11 Bytes

c) If a PTE (Page Table Entry) occupies 8 bytes, which will be the maximum size of the Table of Pages?
- Maximum size = 8MB 2^20 bits * 8 Bytes = 2^23 Bytes

i. What if it was an inverted page table?
If limited by the size of physical memory =128kB x 8 bytes

d) How many pages will the Table of Pages occupy?
- 4096 2^23 / 2^11

Consider a computer system with a 32-bit logical address and 4-KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?

a) A conventional single-level page table?
$2^{20}$ entries (1 048 576) 4KB -> 2^12B -> 32 - 12 = 20

b) An inverted page table?
$2^{17}$ entries (131 072) 512MB / 4KB = 2^17

Consider a paging system with the page table stored in memory. Each memory reference takes 200 nanoseconds.

a) How long does a paged memory reference take? (do not consider a TLB)
400 nanoseconds mem. ref. to see the table (200) + mem. ref. for the needed instruction (200)

b) If we add TLB, and 75 percent of all page-table references are found in the TLB, what is the effective memory reference time? (Assume that finding a page-table entry in the TLB takes zero time, if the entry is there.)
Effective access time = 250 nanoseconds.
EAT = 0,75 * (0 + 200) + 0,25 * 400 = 250ns

- A machine has a simple pagination system, 32 bits logic addresses, a page table size with 1024 entries, 128 frames of memory, and uses a LRU page replacement algorithm.
- Consider that initially all frames are empty with the exception of the frame that contains the code.
- How many *page-faults* does the following code generate?

```
char A[2*1024*1024]; // table in address 0x02800000 (start of a page)
for (int i=0; i<2*1024*1024; i++)
    A[i]=0;
```

Page-faults generated = 1 page size is 2^(32 - 10 = 22), and the size of A is 2^21. therefore, only 1 page needed.

A computer has a virtual address with 32 bits and uses segmentation+paging. There are 16 equal segments, and each page has 4KB. Each PTE (Page Table Entry) has 4 bytes.

a) What is the maximum virtual address space of a process
b) How many bits are there in the logical address? How many for each Segment, Page and Offset?
c) What is the size of each Page Table?
d) How many pages does each Page Table occupy?

segs: 16 -> 2^4
off: 4KB -> 2^12
pags: 32 - 4 - 12

4KBytes = $2^{12}$ bytes

a) $2^{32}$ = 4GB
b) Segments: 4 bits; Pages: 16 bits; Offset: 12 bits
c) $2^{18}$ bytes 2^16 * 2^2 = 2^18
d) $2^6$ 2^18 / 2^12 = 2^6

Considering the segment table below what are the physical addresses for the following logical addresses?

a) Seg. 0, offset 430
b) Seg. 1, offset 10
c) Seg. 2, offset 500
d) Seg. 3, offset 400

| Segment | Base | Length |
|---|---|---|
| 0 | 219 | 600 |
| 1 | 2300 | 14 |
| 2 | 90 | 100 |
| 3 | 1327 | 580 |

219 + 430 = 549 / 2310 / -410 (illegal reference => trap to OS) / 1727

Assume we have a demand-paged memory. It takes 8 ms to service a page fault if an empty page is available or the replaced page is not modified, and 20 ms if the replaced page is modified. Memory access time is 100 ns.
Assume that the page to be replaced is modified 70% of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 ns?

EAT=(1-p) x ma x p x "page fault overhead"
8ms - page fault with no page write
20 ms - page fault with page write
100 ns – memory access time
200 ns=(1-p) x 100 ns + p x ((0.3 x 8 000 000 ns + 0.7 x 20 000 000 ns)+100 ns) ⇔ p = 6.098 x $10^{-6}$

Consider a system with 4 pages (1 to 4) and 3 page frames. The following reference string, where R denotes a reading operation and W denotes a writing operation, will be used:

**W(2) ; R(1) ; W(1) ; W(3) ; R(4) ; R(2) ; W(1) ; W(3) ; W(2) ; R(4) ; R(1) ; W(2) ; R(1)**

The system uses a Modify-bit
Which is the number of Page-Faults and Swap-outs (writing of memory pages to disk) when using the following algorithms: