

Introduction

OS: A program that acts as an intermediary (supervisor) between a user of a computer and the computer resources.

Duties of OS

1. Provide resource abstraction
2. Manage and coordinate resources
3. Provide security and protection
4. Provide fairness among users (or programs)

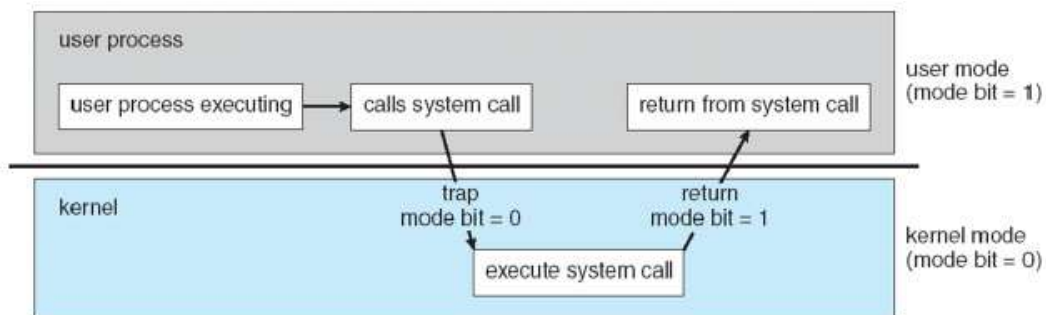
1) Provide resource abstraction

2) Manage and coordinate resources

- OS is a **resource allocator**
 - Manages all resources for processes
 - Decides between conflicting requests for efficient and fair resource use

3) Provide security and protection

- OS is a **control program**
 - Controls execution of programs to prevent errors and improper/malicious use of the computer
 - Dual mode and multimode OS – User mode and kernel mode
- **System Call**
 - How a program requests a service from an OS
 - Results in a transition from user to kernel mode
 - Return from call resets it to user mode
- Software error or request creates **exception or trap**



Interrupts

- An operating system is **interrupt driven**, It sits and waits for an event to occur
- Device or hardware interrupts
 - I/O device is done
 - Hardware throws an exception
- Software interrupts
 - A **trap** or **exception** is a software-generated interrupt caused either by an error or a user request (system call)
- OS has an **interrupt vector**, which contains the addresses of all service routines for interrupt handling.

4) Provide fairness among users (or programs)

- Via multiprogramming

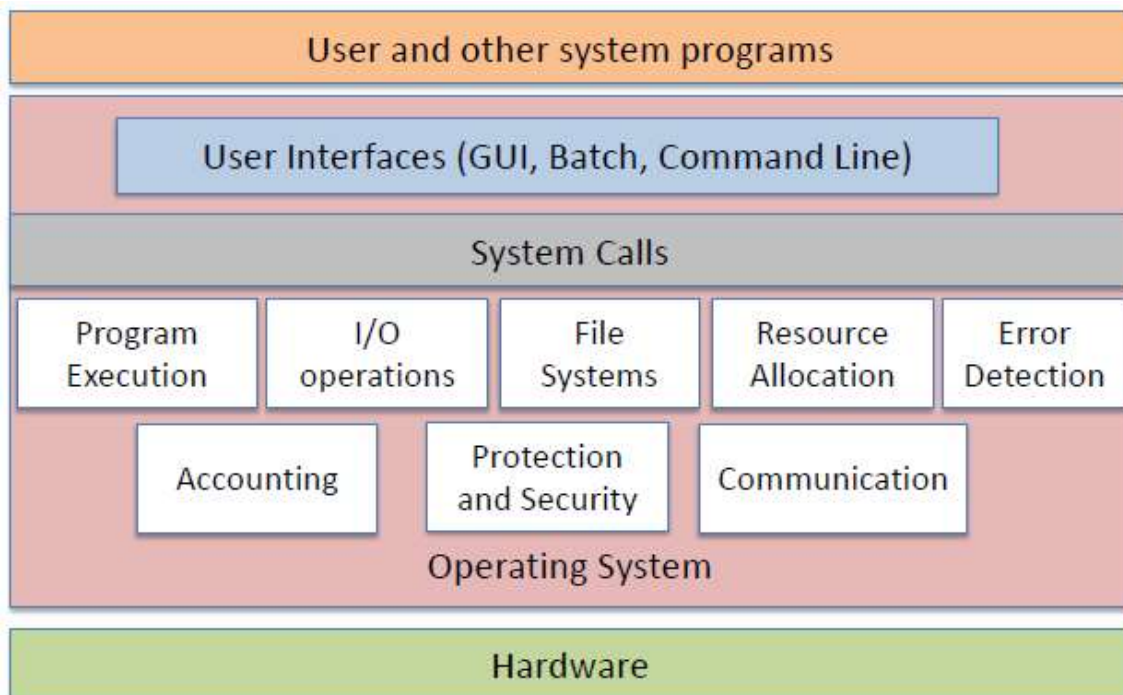
How Multiprogramming Works

- Multiprogramming needed for efficiency
 - Single user or program cannot keep CPU and I/O devices busy at all times
 - Organize process so that CPU always has one process to execute
 - A subset of total jobs is kept in main memory
- One job selected and run via CPU scheduling
 - When it has to wait, OS switches to another job.

Computer Startup

- **Bootstrap program** is loaded at power-up or reboot
 - Typically stored in ROM, generally known as **firmware**
 - Initializes all aspects of a system
 - Loads operating system **kernel** into main memory and starts execution
 - The first system process is 'init' in Linux
 - When the system is fully booted, it waits for some event to occur
- **Kernel**
 - The "one" program running at all the time (the core of OS)
 - Everything else is an application program
- **Process**
 - An executing program (active program)

Operating System Services



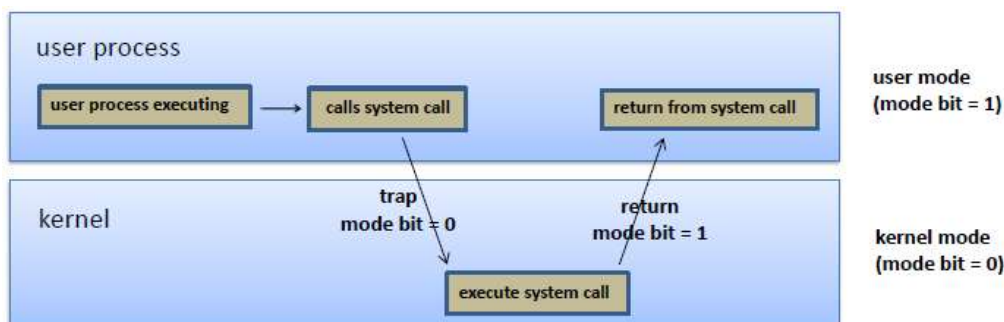
- **User Interface:** Almost all operating systems have a user interface (UI). Varies between Command-Line (CLI), Graphics User Interface (GUI), or Batch
- **Program Execution:** The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
- **I/O Operations:** A running program may require I/O, which may involve a file or an I/O device.
- **File-system Manipulation:** Programs need to read and write files and directories, create and delete them, search them, list file information, manage permissions.
- **Communications:** Process may exchange information, on the same computer or between computers over a network. Communications may be via shared memory or through message passing (packets moved by the OS)
- **Error detection:** OS needs to be constantly aware of possible errors. May occur in the CPU and memory hardware, in I/O devices, in user program. For each type of error, OS should take the appropriate action to ensure correct and consistent computing.
- **Resource allocation:** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them. Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code.
- **Accounting:** To keep track of which users use how much and what kind of computer resources, improve response time to users
- **Protection and security:** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other.

OS Protection: Dual-Mode Operation

- **Dual-mode** operation allows OS to protect itself and other system components.
 - **User mode** and **kernel mode**
 - **Mode bit** provided by hardware, provides ability to distinguish when system is running user code or kernel code
 - Some instructions designated as **privileged**, only executable in kernel mode. For example, I/O related instructions are privileged.
- Ensures that an incorrect program cannot cause other programs to execute incorrectly.

Transition from User to Kernel Mode

- **System call**
 - Results in a transition from user to kernel mode
 - Return from call resets it to user mode
- Software error or a user request creates an **exception or trap**



Privileged Instructions

- The dual mode of operation provides us with the means for protecting the operating system from errant users and errant users from one another.
- We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions**.
 - The hardware allows privileged instructions to be executed only in kernel mode.
 - If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.
- Set value of timer, clear memory, turn off interrupts, modify entries in device-status table, access I/O device should be privileged.

I/O Protection

- All I/O instructions are **privileged instructions**.
- Must ensure that a user program never gain control of the computer in **kernel** mode

Memory Protection

- Must provide memory protection at least for the interrupt vector and the interrupt service routines.
- In order to have memory protection, add two registers that determine the range of legal addresses a process may access.
 - **Base register:** Holds the smallest legal physical memory address.
 - **Limit register:** Contains the size of the range.
- Memory outside the defined range is protected.

Hardware Protection

- When executing in kernel mode, the operating system has unrestricted access to both kernel and user's memory.
- The load instructions for the **base** and **limit** registers are **privileged instructions**.

CPU Protection

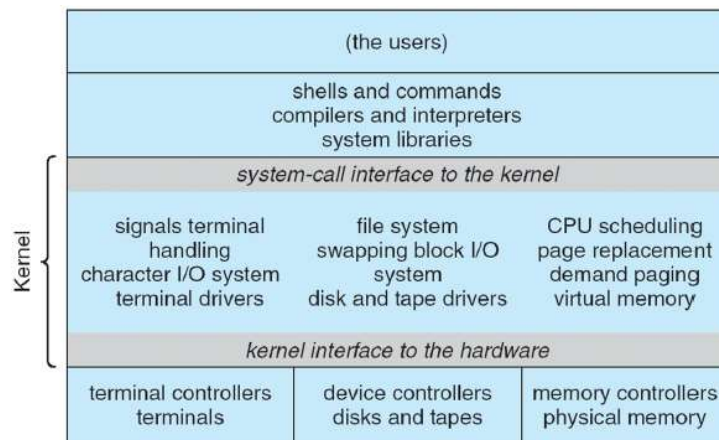
- **Timer** interrupts computer after specified period to ensure operating system maintains control.
 - Timer is decremented every clock tick.
 - When timer reaches the value 0, an interrupt occurs.
- Timer commonly used to implement time sharing systems.
- Clearly, instructions that modify the content of the timer are **privileged**.
- A timer can be used to prevent a user program from never returning control to the os.

OS Structure

- General-purpose OS is very large program.
- Various ways to structure it
 - Monolithic Kernel
 - Microkernel
 - Modular Approach
- Most current OS combines all three approaches nowadays. Ex: win, mac OS, linux

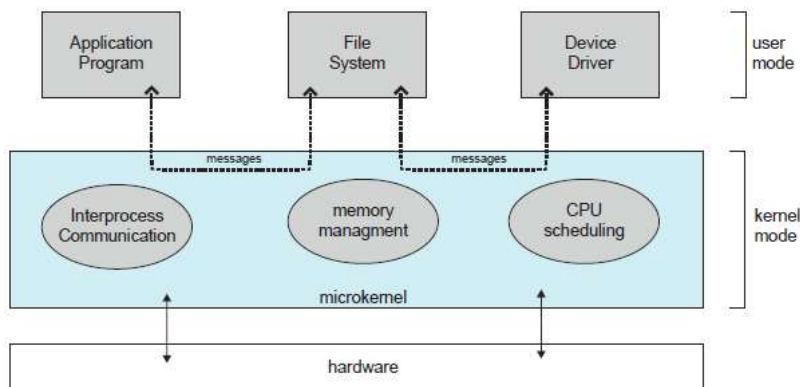
Monolithic Kernel

- All the OS services are implemented in the Kernel.
- Fast OS but hard to extend.
- Ex: MS_DOS, Unix



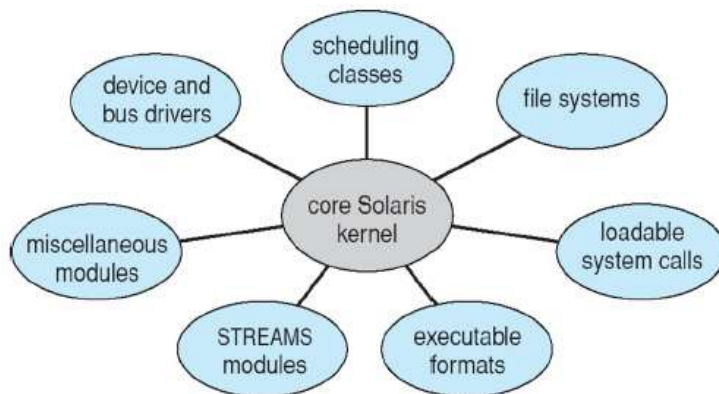
Microkernel

- Moves all the nonessential components from kernel to user level. Smaller kernel
- Uses messages with system and user-level programs.
- Ex: mach



Modular Kernel

- Loadable kernel modules, load additional services if needed at boot or run time:
- Ex: Solaris



Process Management

Process Concepts

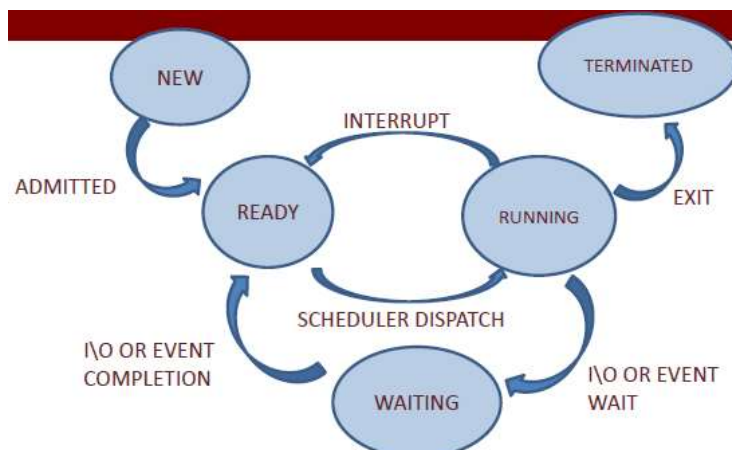
- **Process** is a program in execution.
- A program is **passive** entity stored on disk (executable file); process is **active**.
- A program becomes a process when executable file loaded into memory.
- The terms **job**, **task** and **process** are almost interchangeably used.
- Execution of a program start via GUI mouse clicks, command line entry of its name...
- One program can have several processes.
 - Consider multiple users executing the same program.
 - Ex: Multiple browsers running at the same time
- The program code, also called **text section** or **binary code**.
- Current activity includes **program counter** and processor **registers**.
- **Stack** containing temporary data.
 - Function parameters, return addresses, local variables
- **Data section** containing global variables.
- **Heap** containing memory dynamically allocated during run time.

Concurrent Execution

- OS implements an abstract machine per process
- Multiprogramming enables N programs to be *space-mixed* in executable memory, and *time-mixed* across the physical machine processor.
- Result: Have an environment in which there can be multiple programs in execution *concurrently*, each as a process
 - Concurrently means processes appear to execute simultaneously, they all make some progress over time.

Process State

- As a process executes, it changes its **state**.
 - **new**: the process is being created.
 - **ready**: the process is waiting to be assigned to a CPU.
 - **running**: Instructions are being executed.
 - **waiting**: The process is waiting for some event (e.g., IO) to occur.
 - **terminated**: The process has finished execution.



Process Context

- Also called **Process Control Block (PCB)**
- When an interrupt occurs, what information OS needs to keep around so that we can reconstruct process's context as if it was never interrupted its execution?
- Each process has its own PCB.

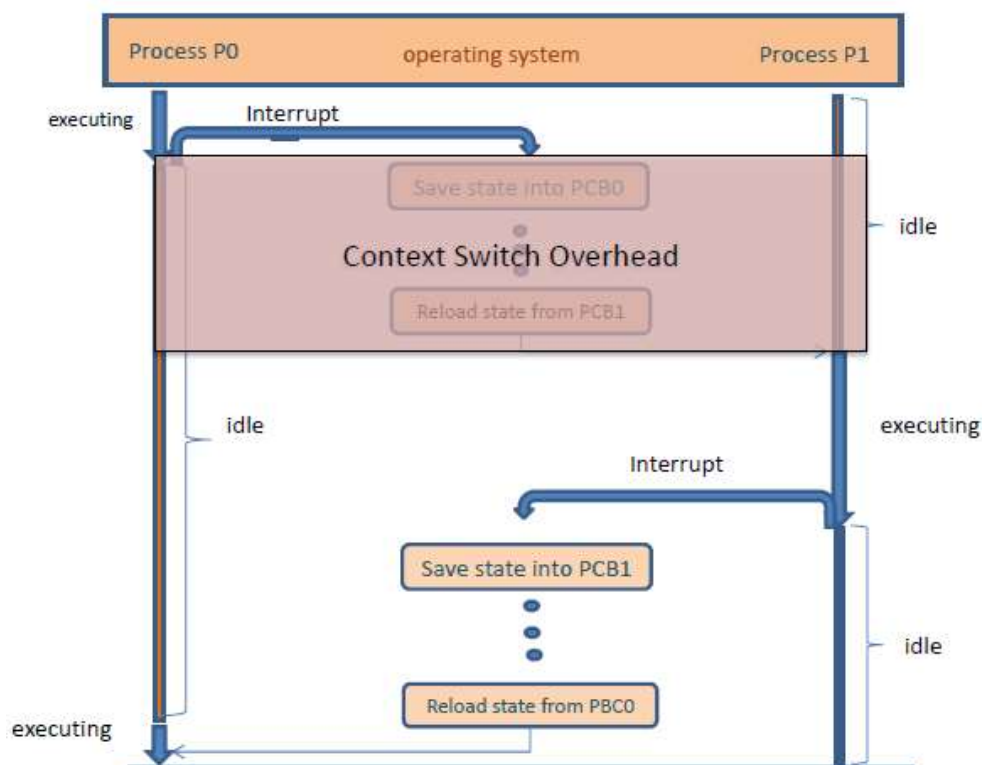
PCB keeps the process context:

- **Process State:**
- **Program Counter:** Location of instruction to next execute
- **CPU Registers:** contents of all process registers
- **CPU Scheduling Information:** Priorities, scheduling queue pointers
- **Memory-management Information:** Memory allocated to the process
- **Accounting Information:** CPU used, clock time elapsed since start, time limits
- **I/O Status Information:** I/O devices allocated to process, list of open files

Context Switch

- OS needs to store and restore the context of a process.
 - So that execution of the process can be resumed from the same point at a later time.
 - This is called **context switch**.
- Context switch is **pure overhead**.
 - Should be very small (couple ms)
 - Hardware support for context switching improves the performance.
- When does OS switch context?
 - In case of an interrupt
 - When process's time is up even though process has still some work to do
 - When a process terminates

→ Each thread has a PCB.

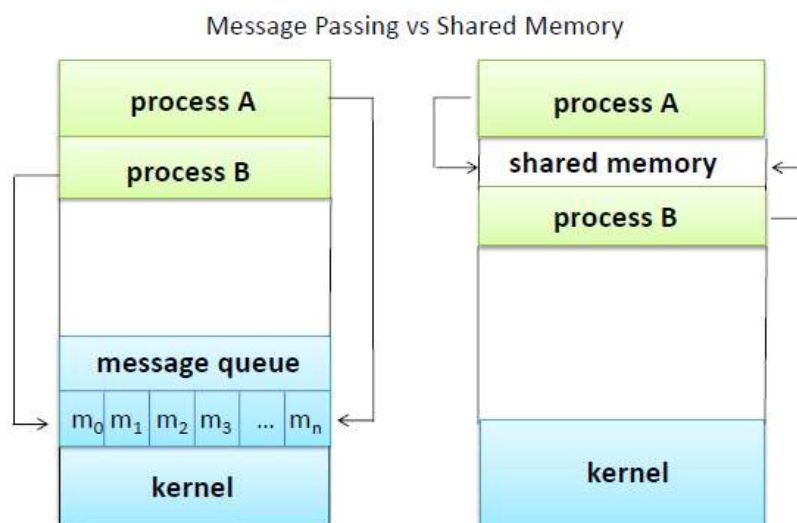


Process Creation and Termination

- UNIX **fork()** creates a process.
 - Creates a new address space.
 - Copies text, data and stack into new address space.
 - Provides child with access to open files of its parent.
- UNIX **wait()** allows a parent to wait for a child to change its state.
 - This is a blocking call, parent waits until it receives a signal.
- UNIX **exec()** system call variants allow a child to run a new program.
- Address space
 - Child duplicates the address space of the parent.
 - Child has a program loaded into it
- Process executes last statement and ask the operating system to delete it (**exit()**)
 - Output data from child to parent via wait
 - Terminated process' resources are deallocated by operating system
- Parent may terminate execution of children process via **kill()** system call
- A terminated process is a **zombie**, until its parents calls wait()
 - Still has an entry in the process table
- Some operating systems do now allow child to continue without its parent
 - All children are terminated – **cascading termination**
- If parent terminates, still executing children processes are called **orphans**
 - Those are adopted by init process
- **Init** periodically calls wait to terminate orphans and zombies

Inter-Process Communication (IPC)

- An **independent process** cannot affect or be affected by the execution of another process.
- **Cooperating processes** can affect or be affected by the execution of other processes.
- Cooperating processes need methods for **inter-process communication (IPC)**
 - **Shared Memory**
 - **Message Passing:** Requires the message A to be copied to a buffer and copied to process B's memory – thus it is slower but safer.



Why Support IPC?

- Sharing information
 - For example, web servers use IPC to share web documents and media with users through a web browser.
- Distributing work across systems
 - For example, Wikipedia uses multiple servers that communicate with one another using IPC to process user requests.
- Separating privilege
 - For example, network systems are separated into layers based on privileges to minimize the risk of attacks. These layers communicate with one another using encrypted IPC.
- Processes within the same computer or across computers use similar techniques for communication

Message Passing

- Processes communicate with each other without resorting to shared variables.
- IPC facility provides two operations:
 - **send(message)**: message size fixed or variable
 - **receive(message)**
- If P and Q wish to communicate, they need to
 - Establish a **communication link** between them.
 - Exchange messages via send/receive
- Implementation of communication link
 - Direct or indirect
 - Synchronous or asynchronous
 - Automatic or explicit buffering

Direct Communication

- Processes must name each other explicitly
 - Send(P, message): send a message to process P
 - Receive(Q, message): receive a message from process Q
- Properties of communication link
 - Links are established automatically.
 - A link is associated with exactly one pair of communicating processes
 - Between each pair there exist exactly one link
 - The link may be unidirectional but is usually bidirectional.

Indirect Communication

- Messages are directed and received from mailboxes (ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox
- Properties of communication link
 - Link established only if processes share a common mailbox
 - A link may be associated with many processes
 - Each pair of processes may share several communication links
 - Link may be unidirectional or bidirectional
- A process may own a mailbox or OS provide operations to create a new mailbox
 - Send and receive messages through mailbox
 - Destroy a mailbox

- Primitive are defined as
 - Send(A, message): send a message to mailbox A
 - Receive(A, message): receive a message from mailbox A

Blocking vs Non-blocking

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**.
 - **Blocking send** has the sender block until the message is received.
 - **Blocking receive** has the receiver block until a message is available.
- **Non-Blocking** is considered **asynchronous**.
 - **Non-blocking send** has the sender send the message and continue
 - **Non-blocking receive** has the receiver receive a valid message or null

Pipes

- Acts as a conduit allowing two processes to communicate
 - Ordinary Pipes
 - Named Pipes

Ordinary Pipes

- Ordinary pipes allow communication in standard producer-consumer style.
- Produced writes to one end (the write-end of the pipe)
- Consumer read from the other end (the read-end of the pipe)
- Ordinary pipes are therefore **unidirectional**.
- Require parent-child relationship between communicating processes.
- Powerful command for I/O redirection
- Connects multiple commands together.
- With pipes, the standard output of one command is fed into the standard input of another.

Examples of IPC Systems – POSIX

- POSIX Shared Memory
 - Process first creates shared memory segment.
 - `shm_fd = shm_open (name, O_CREAT | O_RDWR, 0666);`
 - Also used to open an existing segment to share it.
 - Set the size of the object.
 - `ftruncate (shm_fd, SIZE)`
 - Memory-mapped the file
 - `ptr = mmap (start, length, PROT_WRITE, MAP_SHARED, shm_fd, offset)`
 - Now the process could write to the shared memory.
 - `sprint (ptr, "text")`

CPU Scheduling

Long-term scheduler

- Aka job scheduler
- Selects which process should be brought into the ready queue.
- Invoked very infrequently (seconds, minutes)
- Can be slow.
- Controls the **degree of multiprogramming**.

Short-term scheduler

- Aka CPU scheduler
- Selects which process should be executed next and allocates CPU to that process.
- Invoked very frequently (ms)
- Must be fast.

CPU - I/O Burst Cycle

- Process execution consists of a **cycle** of CPU execution and I/O wait.
- Processes can be described as either
 - I/O – bound process
 - Spends more time doing I/O than computations.
 - Many, short CPU bursts
 - CPU – bound process
 - Spends more time doing computations.
 - Few, very long CPU bursts.

CPU Scheduler

- Selects among the process in memory that are ready to execute and allocates the CPU to one of them.
- CPU scheduling decisions may take place when a process:
 1. Switches from running to waiting state.
 2. Switches from running to ready state.
 3. Switched from waiting to ready state.
 4. Terminates

Non-preemptive: Process voluntarily releases CPU 1 and 4

Preemptive: OS kicks the process out from the CPU 2 and 3

Question: The strategy of making process that are logically runnable to be temporarily suspended is called Preemptive Scheduling.

Scheduling Criteria

CPU Utilization: Keep the CPU as busy as possible.

Throughput: Number of process that completes their execution per time unit

Turnaround Time: Amount of time to execute a particular process (time between entry and exit)

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, not output (for time-sharing environment)

Optimization Criteria

- Max CPU Utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Dispatcher

- **Dispatcher** module is part of the OS that gives control of the CPU to the process selected by the short-term scheduler; this involves:
 - Switching context
 - Switching to user mode
 - Jumping to the proper location in the user program to restart that program
- Dispatch latency: Time it takes for the dispatcher to stop one process and start another running.

Scheduling Algorithms

1. First Come, First Served (FCFS)

- Easy to implement
- Not a great performer
- Non-preemptive
- **Convoy Effect**: Short process behind a long process.

2. 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.
- **Non-preemptive**: Once CPU given to process, it cannot be preempted until completes its CPU burst.
- **Preemptive**: If a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This is known as the **Shortest-Remaining-Time-First (SRTF)**. May starve large jobs.
- SJF is **optimal**. Gives **minimum average waiting time** for a given set of processes.
- Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process.
- **Difficulty**: Determining the length of the next CPU burst
 - Can only **estimate the length**.
 - Can be done by using the length of previous CPU bursts, using **exponential averaging**.
 - $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

3. Priority Scheduling

- A **priority** number is associated with each process.
- CPU is allocated to the process with highest priority (smallest int = highest priority)
- Can be preemptive and non-preemptive.
- FCFS and SJF are special cases of priority scheduling.
 - FCFS priority is based on the arrival time.
 - SJF priority is the predicted next CPU burst time.
- **Problem = Starvation**: Low priority process may never execute.
- **Solution = Aging**: As time progress, increase the priority of the process.

- What if a high-priority process needs to access the data that is currently being held by a low priority process?
 - The high-priority process is blocked by the low-priority process. This is **priority inversion**.
 - This can be solved with priority-inheritance protocol.
 - The low priority process accessing the data inherits the high-priority until it is done with the resource.
 - When the low-priority finishes, its priority reverts back to original.

4. Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** – q), usually 10-100 ms.
- After this time has elapsed, the process is **preempted** and added to the end of the ready 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.
- Typically, **higher average turnaround** than SJF, but **better response**.
- Context Switch Overhead must be considered
- Time Quantum and Context Switch Time
 - q large → FIFO
 - q small → overhead is too high.

5. Multilevel Queue

- Ready queue is partitioned into separate queues:
 - Foreground: interactive
 - Background: batch
- Each queue has its own scheduling algorithm.
 - Foreground: RR
 - Background: FCFS
- Scheduling must be done between queues.
 - Fixed priority scheduling: Serve all from foreground then from background. Possibility of **starvation**.
 - Time Slice: Each queue gets a certain amount of CPU time which it can be schedule amongst its process.

6. Multilevel Feedback Queue

- Multilevel queue with feedback scheduling is similar to multilevel queue, however it allows processes to move between queues.
- **Aging** can be implemented this way.
- Multilevel-feedback-queue-scheduler defined by the following parameter:
 - 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.

Multiple-processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available.
 - **Asymmetric Multiprocessing**: Only one processor accesses the system data structures, alleviating the need for data sharing.
 - **Symmetric Multiprocessing (SMP)**: Each processor is self-scheduling, all process in common ready queue, or each has its own private queue of ready process.
- **Processor affinity**: Process has affinity for processor on which it is currently running
 - **Soft affinity**: Can be changed at a later time.
 - **Hard affinity**: process doesn't move to another processor.
 - Variations including **processor sets**.

Process Migration

- If SMP need to keep all CPUs loaded for efficiency
- **Load Balancing**: attempts to keep workload evenly distributed.
- **Push Migration**: Periodically checks load on each processor, and if found any of them overloaded, pushes task from overloaded CPU to other CPUs.
- **Pull Migration**: Idle processors pull waiting tasks from busy processor.

Real-Time CPU Scheduling

- **Real-time programs** must guarantee response within strict time constraints, often referred to as deadlines.
- **Soft real-time systems**: no guarantee as to when critical real-time process will be scheduled, degrades the system's quality of service.
 - Ex: the flight plan updates for an airline, live broadcasting.
- **Hard real-time systems**: missing a deadline is a total system failure.
 - **Mission critical**: A real-time deadline must be met, regardless of system load.
 - Ex: Anti-lock brakes on a car, heart pacemakers and many medical devices.

➔ Not all the OS are real-time.

- **Event Latency**: Time that elapses from when an event occurs to when it is serviced.
 - Two types of latencies affect performance.
 - **Interrupt Latency**: Time from arrival of interrupt to start of routine that services the interrupt.
 - **Dispatch latency**: Time for schedule to take current process off from CPU and switch to another.

Real-Time OSs

- **Event driven systems** switch between tasks based on their priorities while time sharing systems switch the task based on clock interrupts.
- Design goal is not high throughput, but rather a guarantee of service for high priority job.
- Real-time OS is more frequently dedicated to a narrow set of application.
 - Targeted usage is typically embedded systems, robots etc.
 - Supporting industrial, automotive, smart city and smart home.

Linux Scheduler

Basic Philosophies

- Priority is the primary scheduling mechanism.
- Priority is **dynamically adjusted** at run time.
- Try to distinguish **interactive** process from **non-interactive** ones.
- Use large time quanta for important processes.
 - Modify quanta based on CPU usage for the next run.
- Associate processes to CPUs in multicore systems: Process affinity

Priority

- Each task has a **static priority** that is set based upon the nice value specified by the task. (default 120)
 - For normal tasks, the static priority is 100+ nice.
- Each task has a **dynamic priority** that is set based upon a number of factors.

Niceness

- A process is nicer to others if it has a higher nice value.
- Default is inherited from its parent (usually 0)
- Ranges from -20 to +19
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to global priority 139
- Value can be set via **nice()** system call or **nice** command

Kernel 2.4

- Each task got a number of CPU ticks (jiffies) made available to each scheduling interval, or **epoch**.
- The number of new ticks given was determined from the nice value for the task.
 - It was roughly: $((20 - \text{nice}) * \text{HZ}/800) + 1$
- Each task had a **counter**, which was the number of CPU ticks still left for the task to use in the current epoch.
- Unused ticks in a particular epoch decayed by 50% for use in the next interval.

Linux O(1) Scheduler – Kernel 2.5

- Preemptive, priority based.
- Two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **normal** (time-sharing) range from 100 to 140.
- Higher priority gets larger time quantum.
- Scales well with the number of processes.
- Task runnable as long as time left in time slice (**active**)
- If no time left (**expired**), not runnable until all other tasks use their slices.
- All runnable tasks tracked in per-CPU **runqueue** data structure.
 - Two priority arrays: active and expired.
 - When no more active, arrays are swapped.

Real-Time Scheduling

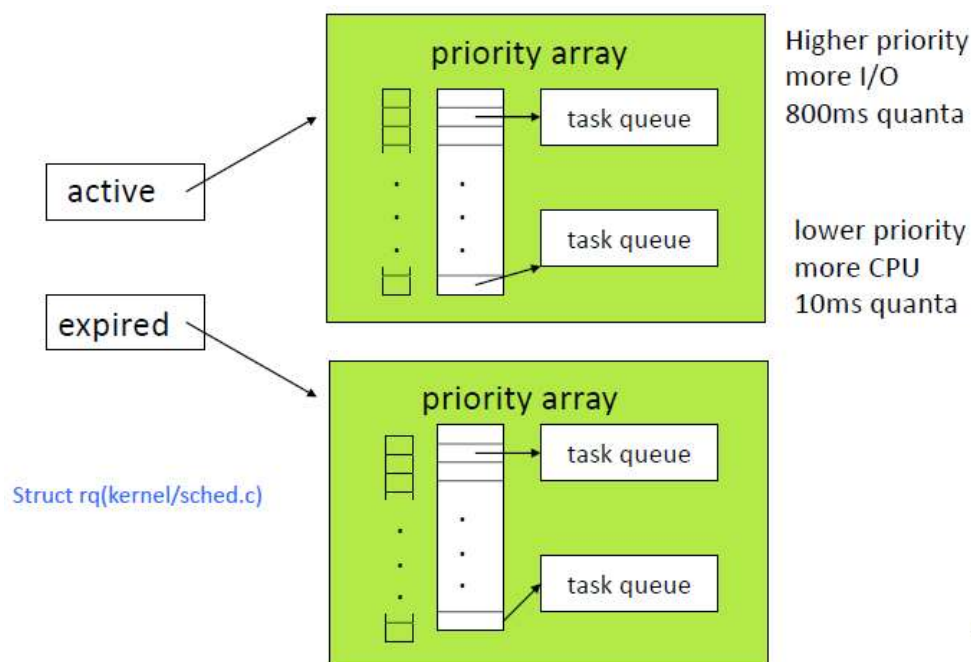
- Linux has a soft real-time scheduler.
 - No hard real-time guarantees.
 - All real-time processes are higher priority than any normal processes.
- Processes with priorities [0, 99] are real-time.
 - Saved in **rt_priority** in the **task_struct**
 - Scheduling priority of a real time task is $99 - \text{rt_priority}$
- A process can be converted to real-time via **sched_setscheduler** system call.

Scheduling Policies

- Real-time processes
 - First in, first out: **SCHED_FIFO**
 - Static priority
 - Process is only preempted for a higher priority process.
 - No time quanta, it runs until it blocks or yields voluntarily.
 - Round Robin: **SCHED_RR**
 - RR within the same priority level
 - There is a time quantum (800 ms)
- Normal processes have
 - **SCHED_OTHER**: standard processes
 - **SCHED_BATCH**: batch style processes
 - **SCHED_IDLE**: low priority tasks

Runqueues

- 140 separate queues, one for each priority level in two sets: **active** and **expired**.
- Total 140 priorities [0, 140)
- Smaller integer = higher priority



Scheduling Algorithm for Normal Processes

- Find the highest-priority non-empty queue in **rq** → **active**; if none, simulate aging by swapping active with expired.
- **Next** = Find the first process on that queue
- Calculate **next's** quantum size and its **next's** priority.
- Context switch to **next**.
- Let it run.
- When its time is up, put it on the **expired list**.
- Repeat

Simulate Aging

- After running all of the active queues, the active and expired queues are swapped.
- There are pointers to the current arrays; at the end of a cycle, the pointers are switched.
- Swapping active and expired gives low priority processes a chance to run.
- **O(1) Advantage:** Processes are touched only when they start or stop running

Find Highest Priority Non-Empty Queue

- Time complexity: O(1)
 - Depends on the number of priority levels, not the number of processes.
- Implementation: A bitmap for fast look up
 - 140 queues. A few comparisons to find the first non-zero bit

Calculating Time Slices

- **time_slice** in the **task_struct**
- if (SP < 120) : Quantum = (140 – SP) x 20
- if (SP >= 120) : Quantum = (140 – SP) x 5
- Higher priority process gets longer quanta.
- Important processes should run longer.

Priority:	Static Pri	Niceness	Quantum
Highest	100	-20	800 ms
High	110	-10	600 ms
Normal	120	0	100 ms
Low	130	10	50 ms
Lowest	139	20	5 ms

Issues with O(1) RR Scheduler

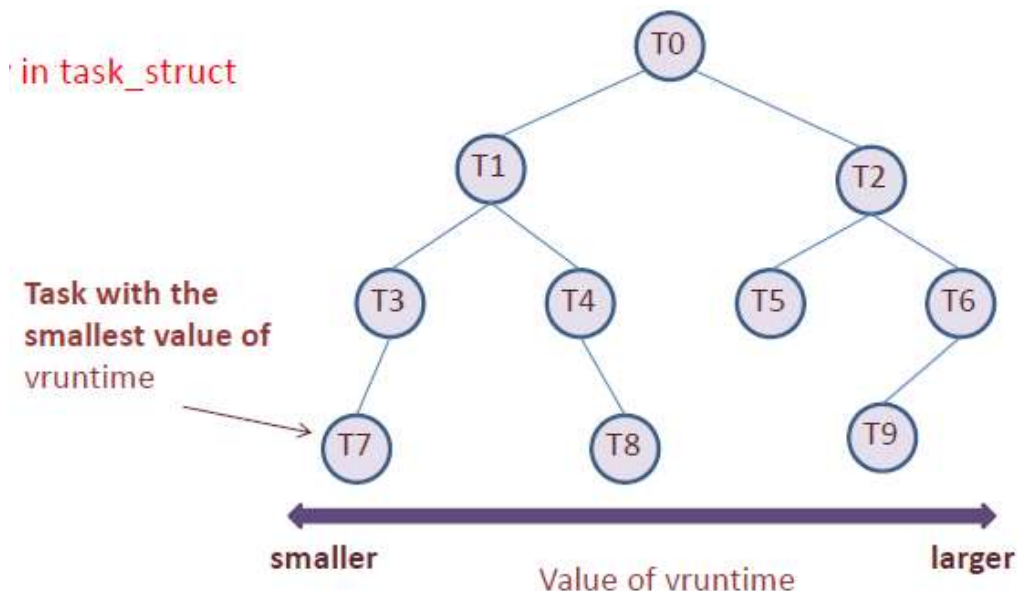
- Not easy to distinguish between CPU and I/O bound
 - I/O bound typically needs better interactivity.
- Finding right time slice isn't easy.
 - Too small: Good for I/O but high overhead.
 - Too large: Good for CPU bound but poor interactivity.
- Priority is relative but time slice absolute.
 - Nice 0, 1: Time slice 100 and 95 ms. 5% difference
 - Nice 19, 20: Time slice 10 and 5 ms. 100% difference

Completely Fair Scheduler (CFS)

- Not based on runqueues as in O(1) scheduler.
- Not based on time slices.
- Note that CFS is used only for normal processes, for real-time processes, Linux still use priority based FCFS and RR schedulers.
- Core ideas: Dynamic time slice and order
- Don't use fixed time slice per task.
 - Instead, fixed time slice across all tasks.
 - Scheduling Latency.
- Don't use RR to pick next task.
 - Pick task which has received the least CPU time so far
 - Equivalent to dynamic priority
- CFS calculates how long a process should run as a function of the total number of runnable processes.
 - If there are N runnable processes, then each should be afforded 1/N of the processor's time.
 - CFS adjusts the allocation by weighting each process's allocation by its nice value.
 - Smaller nice value → higher weight
 - Larger nice value → lower weight
 - Then process's time slice is proportional to its weight divided by the total weight of all runnable processes.
- $\text{Timeslice}(\text{task}) = \text{Timeslice}(t) * \text{prio}(t) / \text{sum_all_t'}(\text{prio}(t'))$
- $\text{Timeslice}(t) = \text{latency} / \text{nr_tasks}$

CFS Tree

- Each runnable task is placed in a red-black tree (a balanced binary search tree whose key is based on the value of vruntime)
- When a task becomes runnable, it is added to the tree.
- Not runnable tasks (e.g. waiting for I/O) are removed from the tree.
- Smallest vruntime has the highest priority
- Insertion/deletion on the requires $O(\log n)$ time
- Linux scheduler uses a cache to retrieve small vruntime tasks



- Two tasks have the same nice values. One task is I/O bound, other is CPU-bound.
 - I/O bound normally runs for a short period before it is interrupted for an I/O operation.
 - CPU-bound normally exhausts all its quantum.
- Vruntime eventually be lower for the I/O bound task than for the CPU-bound task.
 - Vruntime is weighted by process priority.

Picking the next process

- Pick task with minimum runtime so for
- Every time process runs for t ns. $Vruntime += t$
- How does this impact I/O vs CPU bound tasks?
 - Task A needs CPU for 1 ms every 100 ms (I/O bound)
 - Task B, C need CPU for 80 ms every 100 ms (CPU bound)
 - After 10 times that A, B and C have been scheduled.
 - $Vruntime(A) = 10$
 - $Vruntime(B, C) = 800$
 - Overtime task a gets priority, but it quickly releases CPU:

CFS Algorithm

- The leftmost node of the scheduling tree is chosen (as it will have the lowest spent execution time) and sent for execution.
- If the process simply completes execution, it is removed from the system and scheduling tree.
- If the process reaches its maximum execution time or is otherwise stopped (voluntarily or via interrupt) it is reinserted into the scheduling tree based on its new spend execution time.
- The new leftmost node will then be selected from the tree, repeating the iteration.

Multiprocessor Scheduling

- Each processor maintains a red-black tree.
- Each processor only selects processes from its own tree to run.
- It is possible for one processor to be idle while others have hobs waiting in their run queues.
- Periodically rebalance
 - `Void load_balance()`: attempts to move tasks from one CPU to another

Processor Affinity

- Each process has a bitmask saying what CPUs it can run on
- Normally, of course, all CPUs are listed.
- Processes can change the mask.
- The mask is inherited by child processes (and thread), thus tending to keep them on the same CPU.
 - Not allowed to run on the current CPU

Process Synchronization

Shared Memory

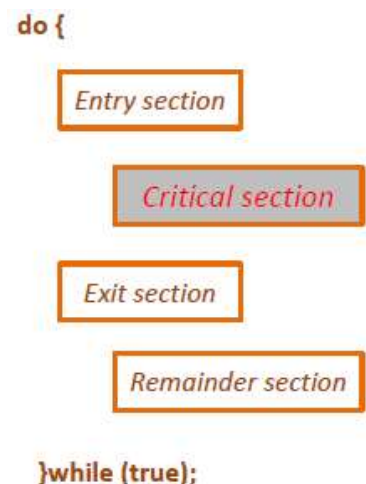
- Communication through shared memory takes place with shared variables.
- Threads or processes share common variables.
- Access these variables should be coordinated (synchronized) so that the data is not corrupted.
- Processes can execute concurrently. They might be interrupted at any time, partially completing execution.
- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperation processes.

Race Condition

- The situation where several processes access and manipulate shared data concurrently. The final value of the shared data is **non-deterministic** and depends upon which process finishes last.
- To prevent race conditions, concurrent processes must be **synchronized**.

Critical-Section

- Each process has a code segment, called **critical section**, in which the shared data is accessed.
- Problem: Ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- A Critical Section Environment contains:
 - Entry Section: Code requesting entry into critical section.
 - Critical Section: Code in which only one process can execute at any one time.
 - Exit Section: The end of the critical section, releasing or allowing others in.
 - Remainder Section: Rest of the code **after** the critical section.



Solution to Critical Section Problem

1. Mutual Exclusion: If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. Progress: If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. Bounded Waiting Time: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Preemptive vs Non-preemptive Kernels

- A **non-preemptive** kernel is free from race conditions on kernel data structures.
- A **preemptive** kernel allows a process to be preempted while it is running in kernel mode.

- Need to ensure that shared kernel data are free from race conditions.
- Ex: list of open files, interrupt handlers, process list/queues, memory alloc.
- Preemptive kernel is more suitable for real-time programming.

Peterson's Solution

- Is a software-based solution.
- Is **not correct** on today's modern computers because update to turn is not specified as atomic.
 - **Atomic** = indivisible instructions
 - **Atomic operation** means an operation that completes in its entirety **without interruption**.
- Also, we have **caches and multiple copies** of the same data (turn var) in the hardware.
- May resulting in data race.
- We need hardware support for avoiding race conditions.

Hardware Support for Synchronization

Atomic Test-and-Set Instruction

- The terms **locks** are used to indicate getting a key to enter a critical section.
- The **test-and-set** instruction is an instruction used to write to a memory location and return its old value as a single atomic (i.e., non-interruptible) operation.
- If multiple processes may access the same memory location, and if a process is currently performing a test-and-set, other process may begin another test-and-set until first process is done.

```
boolean TestAndSet(boolean *lock) {
    boolean initial = *lock;
    *lock = true;
    return initial;
}
```

- Test-and-set does two things atomically:
 - Test a lock (whose value is returned)
 - Set the lock
- Lock obtained when the return value is FALSE
- If TRUE, someone already had the lock (and still has it)

Mutual Exclusion with Test-and-Set

- Shared variable: boolean lock = false;
- Process P_i

```
do {
    while (TestAndSet(&lock)) { };
    critical section
    lock = false;
    remainder section
} while (true);

boolean TestAndSet(boolean *lock) {
    boolean initial = *lock;
    *lock = true;
    return initial;
}
```

The calling process obtains the lock if the old value was *false*. It spins until it acquires the lock. When it acquires, the value turns to *true* preventing other processes to acquire the lock.

Must be careful if these approaches are to satisfy a bounded wait condition. Must use round robin.

Compare-and-Swap (CAS) Instruction

- It compares the contents of a memory location to a given value and, **only if they are the same**, modifies the contents of that memory location to a given new value.
- Done as a **single atomic operation**.
- It is executed atomically.
- If the value had been updated by another process in the meantime, the write would fail.

```
int compare_and_swap(int *value,
                    int expected, int new_value) {
    int oldValue = *value;
    if (*value == expected)
        *value = new_value;
    return oldValue;
}
```

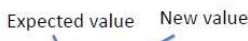
- Shared boolean variable *lock* initialized to FALSE (0)

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = FALSE;
    /* remainder section */
} while (true);
```

Expected value New value



Mutex Locks

- Software interface for locks.
- Previous hardware solutions are complicated and generally inaccessible to application programmers.
- OS designers build software tools to solve critical section problem using the support in the hardware.
- Enter critical regions by first **acquire()** a lock then **release()** it
 - Boolean variable indicates if lock is available or not.
- The solutions **still** use hardware solutions/support underneath.
- Calls to **acquire()** and **release()** must be atomic.
 - Can be implemented using the *CAS*.
- Main disadvantage is that it requires **busy waiting**.
 - Busy waiting wastes CPU cycles.
- This type of lock is called a **spinlock**.
 - Because the process “spins” while waiting for the lock to become available.
 - Spinlocks have an advantage; no context switch is required when a process must wait on a lock.

```
do {
    acquire lock
    critical section
    release lock
    remainder section
} while (true);
```

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}

release() {
    available = true;
}
```


Semaphores

- We want to be able to write more complex construct.
 - Need a language to do so. We define semaphores which we assume are atomic operations.
- Semaphores are more general synchronization tools.
 - Operating System Primitive
 - 2 standard atomic operations modify semaphore variable S: **wait()** and **signal()**

WAIT (S):

```
while ( S <= 0 );  
S = S - 1;
```

SIGNAL (S):

```
S = S + 1;
```

- As given here, these are not atomic as written in “macro code”. We define these operations, however, to be **atomic** (Protected by a hardware lock).

Critical Section for n Processes

- Shared semaphore:
 - semaphore mutex = 1; // initial value
- Process P_i:

```
do {  
    wait(mutex)  
    critical section  
    signal(mutex)  
    remainder section  
} while (true);
```

Example: Shared Balance

```
semaphore mutex = 1;
```

```
proc_0() {  
    . . .  
    /* Enter the CS */  
    wait(mutex);  
    balance += amount;  
    signal(mutex);  
    . . .  
}
```

```
proc_1() {  
    . . .  
    /* Enter the CS */  
    wait(mutex);  
    balance -= amount;  
    signal(mutex);  
    . . .  
}
```

Semaphore Usage

- Provide **mutual exclusion**.
- **Counting Semaphore**: integer value can range over an unrestricted domain.
- **Binary Semaphore**: integer value can range only between 0 and 1; can be simpler to implement. Same as **mutex locks**
- Semaphores can be used to force synchronization (precedence) if the **preceeder** does a signal at the end, and the **follower** does wait at the beginning.

Example:

- We want P₁ to execute before P₂
- Execute B in P_j **only after** A is executed in P_i
- Use semaphore flag initialized to 0.



Classical Problems of Synchronization

Bounded-Buffer Problem

- **Spinlock** (mutexes) are useful in a system since no context switch is required.
- A disadvantage of mutex solutions so far that they all require **busy waiting**.
- To overcome busy waiting → **blocking a process**

Blocking Semaphores: No busy waiting

- With each semaphore there is an associated waiting queue
 - Keeps list of processes waiting on the semaphore.
 - **wait()** operation adds one process to the list.
 - **signal()** operation removes one process from the list.
- Two operations
 - **Block**: Place the process invoking the operation on the appropriate waiting queue if semaphore == false (is not available)
 - **Wakeup**: Wakes up one of the blocked processes upon getting a signal and places the process to ready queue.

Semaphore Implementation

```
typedef struct {  
    int    value;  
    struct process *list; /* list of processes waiting on S */  
} SEMAPHORE;
```

```
SEMAPHORE s;  
wait(s) {  
    s.value = s.value - 1;  
    if ( s.value < 0 ) {  
        add this process to s.list;  
        block ();  
    }  
}
```

Block – place the process invoking the operation on the appropriate waiting queue if semaphore is not available

```
SEMAPHORE s;  
signal(s) {  
    s.value = s.value + 1;  
    if ( s.value <= 0 ) {  
        remove a process P from s.list;  
        wakeup(P);  
    }  
}
```

Wakeup – Wakes up one of the blocked processes upon getting a signal and places the process to ready queue

Deadlock and Starvation

- **Deadlock**: Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- **Starvation**: Indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
- May occur if we add and remove processes from the list in LIFO order or based on priority.

Readers and Writes Problem

Semaphores vs Locks

- **Semaphores**: Processes that are blocked at the level of program logic are placed on queues, rather than busy waiting.
- **Locks**: Busy waiting may be used for the mutual exclusion. But these should be very short critical sections.
- Unlike locks, counting semaphores can take an integer value representing total number of resources.

Problems with Semaphores and Locks

- Semaphores are shared global variables. Can be accessed from anywhere.
- Used for both critical sections (mutual exclusion) and for coordination (scheduling or ordering operations)
- Incorrect use of semaphore operations
 - Call signal first and later on call wait
 - signal(mutex) ... wait(mutex)
 - Call wait after another wait
 - wait(mutex) ... wait(mutex)
 - Omitting of wait or signal.
- Thus, they are prone to bugs.
- To deal with issues, introduce a high-level synchronization construct: **monitors**.

Monitors

- A monitor is a **programming language construct** that supports controlled access to shared data.
 - It resembles an object-oriented approach for synchronization.
- A monitor encapsulates;
 - **Shared data** structures
 - **Procedures** that operate on the shared data (protects shared data)
 - **Synchronization** between concurrent processes that invoke those procedures.
- Monitor construct ensures that only one process at a time can be active within the monitor.

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) {.....}

    procedure Pn (...) {.....}

    initialization(...) { ... }
}
```

Example: Shared Balance

- Balance – **shared variable**
- Deposit and withdraw – **procedures**
- Processes do not directly read/write into shared variables but access them via these procedures.

```
monitor sharedBalance {

    double balance;

    void deposit(double amount) {
        balance += amount;
    }
    void withdraw(double amount) {
        balance -= amount;
    }
    . . .
}
```

Example: Produce-Consumer Problem

Conditional Variables

- Conceptually a condition variable is a queue of processes, associated with a monitor on which a process may wait for some condition to become true.
- Sometime called a rendezvous point
- To allow a process to wait within the monitor, a condition variable must be declared as
 - **condition c;**
- Three operations on condition variables 'c'
 - **wait(c):** The calling process is suspended until another invokes it

- **signal(c):** wake up at most waiting process. If no waiting processes, signal effect. This is different than semaphores: no history!
- **broadcast(c):** wake up all waiting processes.
- A **monitor** is a synchronization construct that allows processes to have both mutual exclusion and the ability to wait (block) for a certation condition to become true.
- Monitors also have a mechanism for signaling other processes that their condition has been met.
- A monitor consists of a mutex object and **condition variables**, procedures to access them.

Summary of Synchronization

- A race condition occurs when processes have concurrent access to shared data and the final result depends on the particular order in which concurrent accesses occur. Race conditions can result in corrupted values of shared data.
- Mutual exclusion is required to ensure no two concurrent processes are in their critical section at the same time to prevent race conditions (corruption of shared data)
- A critical section is a section of code where shared data may be manipulated and a possible race condition may occur. The critical-section problem is to design a protocol whereby processes can synchronize their activity to cooperatively share data.
- A solution to the critical-section problem must satisfy the following three requirements: (1) mutual exclusion, (2) progress, and (3) bounded waiting. Mutual exclusion ensures that only one process at a time is active in its critical section. Progress ensures that programs will cooperatively determine what process will next enter its critical section. Bounded waiting limits how much time a program will wait before it can enter its critical section.
- Hardware solutions
 - Test-and-Test
 - Compare-and-Swap
- Software Solutions
 - Locks (busy wait)
 - Semaphores (blocking, counting semaphores)
 - Monitors (high level language constructs)
- Software solutions to the critical-section problem, such as Peterson's solution, do not work well on modern computer architectures.
- Hardware support for the critical-section problem includes memory barriers; hardware instructions, such as the compare-and-swap instruction; and atomic variables.
- A mutex lock provides mutual exclusion by requiring that a process acquire a lock before entering a critical section and release the lock on exiting the critical section.
- Semaphores, like mutex locks, can be used to provide mutual exclusion. However, whereas a mutex lock has a binary value that indicates if the lock is available or not, a semaphore has an integer value and can therefore be used to solve a variety of synchronization problems.
- A monitor is an abstract data type that provides a high-level form of process synchronization. A monitor uses condition variables that allow processes to wait for certain conditions to become true and to signal one another when conditions have been set to true.
- Solutions to the critical-section problem may suffer from liveness problems, including deadlock.
- The various tools that can be used to solve the critical-section problem as well as to synchronize the activity of processes can be evaluated under varying levels of contention. Some tools work better under certain contention loads than others.

Threads

Threads ← Processes

- A process is an executing program with at least one thread of control.
- A process can contain multiple threads of control.
- A thread is a basic unit of CPU utilization.
- Threads run within application in parallel (concurrently)
- Process creation is relatively **heavy-weight** while thread creation is **light-weight**.
- Modern kernels are generally multithreaded.

Why use threads?

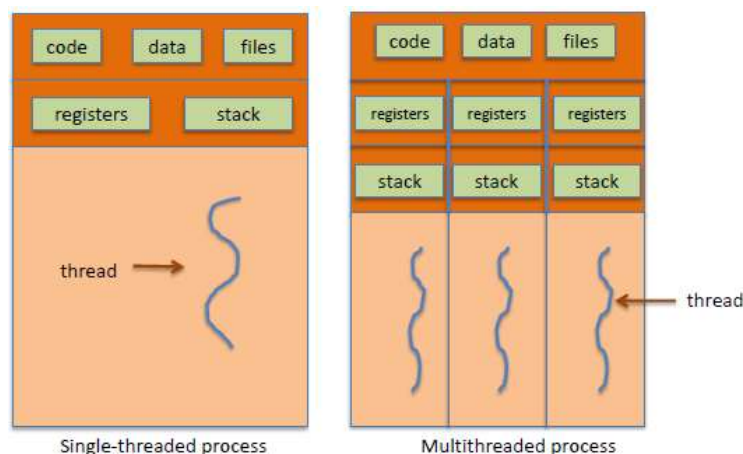
- Responsiveness: May allow continued execution if part of process is blocked, especially important for user interactivity.
- Resource Sharing: Threads share the memory and resources of the process to which they belong by default. It is easier than shared memory or message passing communication between processes.
- Economy: Cheaper than process creation. Context switching is typically faster between threads than between processes.
- Scalability: The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

Process and Thread

- Process is an infrastructure in which execution takes place: address space + resources.
- Thread is a program in execution within a process context: each thread has its own stack.

Single vs Multithreaded Process

- A thread has an ID, a program counter, a register set, and a stack.
- Shares the code section, data section and OS resources with other threads within the same process.



Thread States

- Like a process, a thread can be in one of three states: ready, blocked (waiting), running.
- A thread may wait:
 - For some external event to occur (such as I/O completion)
 - For some other thread to unblock it.
- When a program runs, it starts as a single-threaded process then it can create other threads
 - Typically, the first thread is referred as the **master** thread and the others are **worker** threads.

Thread Libraries

- A thread library provides a programmer with API for creating and managing threads.
- Two primary ways of implementing
 - Library entirely in user space
 - Kernel-level library supported by the OS

User Threads and Kernel Threads

- User Threads: Management done by user-level thread library.
 - The primary thread libraries:
 - POSIX pthreads
 - Win32 threads
 - Java threads
- Kernel Threads: Supported by the Kernel

Java Threads

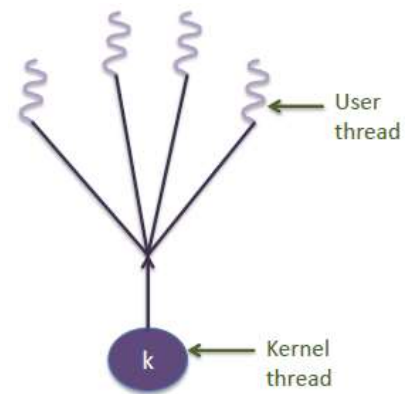
- Java threads are managed by the JVM.
- Typically, implemented using the thread model provided by underlying OS.
- Java thread may be created by
 - Extending Thread class
 - Implementing the Runnable interface.

Multithreading Models

- Many systems support both **user threads** and **kernel threads** that results different multithreading models.
- User threads are supported above the kernel and are managed without kernel support.
- Kernel threads are supported and managed directly by the operating system.

Many-to-One Model

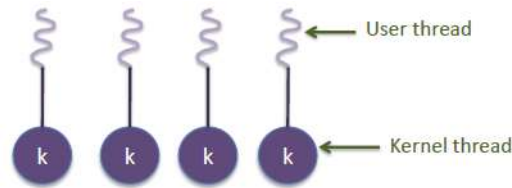
- Maps many user level threads to one kernel thread.
- Thread management is done by the thread library in user space, so it is efficient.
- Entire process will block if a thread makes a blocking system call.
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time.
- Used on systems that do not support kernel threads.
 - Solaris Green Threads
 - GNU Portable Threads
- Not common anymore due to its inability to take advantage of multiple processing cores.



One-to-One Model

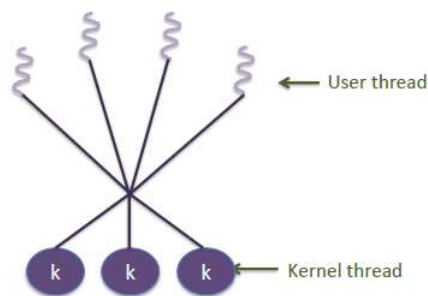
- Maps each user-level thread to a kernel thread.
 - Creating a user level thread creates a kernel thread (-)
- Provides more concurrency than many-to-one by allowing another thread to run when a thread makes a blocking system call.
- Also, allows multiple threads to run in parallel on multiprocessors.
- Number of threads per process sometimes restricted due to overhead.

- Examples: Windows NT/XP/2000, Linux, Solaris 9



Many-to-Many Model

- Allows many user-level threads to be mapped to many kernel threads (usually fewer kernel threads)
- Developers can create as many user threads as necessary.
- Corresponding kernel threads can run in parallel on a multiprocessor.
- When a thread performs a blocking system call, the kernel can schedule another thread for execution.
- Allows the OS to create a sufficient number of kernel threads.



Exercise: Which of the following components of program state are shared across threads?

Register values

Heap memory

Stack

Global variables

Program counter

Scheduling properties (e.g. policy, priority)

Threading Issues

The fork() and exec() System Calls

- Semantics of fork() and exec() system calls change in a multithreaded program.
- Threads and fork: **think twice before mixing them!**
- Does **fork()** duplicate only the calling thread or all threads?
 - v1: The child has only one thread.
 - v2: The child has all the threads.
- **Exec()** usually works as normal – replace the entire process including all threads.
 - Call exec right after fork

Signal Handling

- A **signal** is used in UNIX systems to notify a process that a particular event has occurred.
- A signal may be received either synchronously or asynchronously.
 - Synchronous: Delivered to the same process that performed the operation caused the signal. Ex: Illegal memory access, division by zero
 - Asynchronous: When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Ex: Keystrokes to cancel a process.

- The signal handling pattern:
 - A signal is generated by the occurrence of a particular event.
 - The signal is delivered to a process.
 - Once delivered, the signal must be handled.
- In a single-threaded program, a signal is delivered to the process.
- In a multi-threaded program, where should a signal be delivered?
 - Deliver the signal to the thread to which the signal applies.
 - Deliver the signal to every thread in the process.
 - Deliver the signal to certain threads in the process.
 - Assign a specific thread to receive all signals for the process.
 - Depends on the type of signal generated.

Thread Cancellation

- Terminating a thread before it has finished.
- Thread to be canceled is called **target thread**.
- Two general approaches
 - Asynchronous Cancellation: Terminates the target thread immediately.
 - Deferred Cancellation: Allows the target thread to periodically check if it should be cancelled.
- The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while during updating data it is sharing with other threads.
- pthread code to create and cancel a thread.

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid, NULL);
```

Thread-Local Storage (TSL)

- Threads belonging to a process share the data of the process.
- TSL allows each thread to have its own copy of data.
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
 - Local variables visible only during single function invocation
 - TLS visible across function invocations.
 - TLS is similar to static data but TLS data are unique to each thread.

POSIX Threads API (pthreads)

- Functions of pthreads API provide:
 - Thread management
 - Creation/termination of threads
 - Set/Query thread attributes
 - Mutexes, semaphores
 - Condition variables
- All identifiers in the thread library begin with **pthread_**

Creating Threads

- Initially, a main() program comprises a single, default thread. All other threads must be explicitly created by the programmer.
- pthread_create: creates a new thread and makes it executable.
- The maximum number of threads that may be created by a process is implementation dependent.
 - Programs that attempts to exceed the limit can fail or produce wrong results.
- Threads can create other threads but there is no hierarchy.

Signature:

```
int pthread_create(pthread_t *,
                  const pthread_attr_t *,
                  void * (* start_routine) (void *),
                  void *);
```

Example call:

```
errcode = pthread_create(&thread_id, &thread_attribute,
                        &thread_func, &func_arg);
```

- **thread_id** is the thread id or handle (used to halt, etc.)
- **thread_attribute** various attributes, standard default values obtained by passing a NULL pointer
- **thread_func** the function to be run (takes and returns void*)
- **func_arg** an argument can be passed to thread_fun when it starts
- **errorcode** will be set to nonzero if the create operation fails

Thread Creation

- Each thread executes a specific function, thread_func
 - For the program to perform different work in different threads, the arguments passed at thread creation distinguish the thread's "id" and any other unique features of the thread.
- After thread is created, various attributes of it can be set
 - Priority of the thread
 - Stack size
 - Its scheduling policy

Because threads share resources

- Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
- Two pointers having the same value point to the same data, because of the **shared address space**
- Reading and writing to the same memory location is possible, and therefore requires **explicit synchronization** by the programmer.

Shared Data

- Variables declared outside of "main" are **global variables**. Those variables are shared by all threads.
- Object allocated on the **heap** may be shared if pointer is passed as an argument to the thread function.
- Variables on the stack are private (locally defined variables.)
 - Passing pointer to these around to other threads can cause problems.

Thread Synchronization

- Need to protect the shared data synchronize threads.
- Pthread provides several ways to synchronize threads:
 - Mutexes (locks)
 - Semaphores
 - Condition Variables
 - Barriers: Synchronizing the threads to make sure that they all are at the same point in a program is called a barrier.

Condition Variables in pthreads

- While **mutexes** implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
- Without condition variables, the programmer would need to have threads continually polling to check (possibly in a critical section) if the condition is met.
- A condition variable is always used in conjunction with a mutex lock.
- Set/Query condition variables attributes...

Barriers

- Synchronizing the threads to make sure that they all are at the same point in a program is called a barrier.
- No thread can cross the barrier until all the threads have reached it.

Thread Scheduling

- Threads can be scheduled by the operating system and run as independent entities.
- Many-to-one and may-to-many models, thread library schedules user-level threads
 - Known as **process-contention scope (PCS)** since scheduling competition is within the process.
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** competition among all threads in system.

Summary of Threads

- A thread represents a basic unit of CPU utilization, and threads belonging to the same process share many of the process resources, including code and data.
- There are four primary benefits to multithreaded applications: (1) responsiveness, (2) resource sharing, (3) economy, and (4) scalability.
- Concurrency exists when multiple threads are making progress, whereas parallelism exists when multiple threads are making progress simultaneously. On a system with a single CPU, only concurrency is possible; parallelism requires a multicore system that provides multiple CPUs.
- There are several challenges in designing multithreaded applications. They include dividing and balancing the work, dividing the data between the different threads, and identifying any data dependencies. Finally, multithreaded programs are especially challenging to test and debug.
- Data parallelism distributes subsets of the same data across different computing cores and performs the same operation on each core. Task parallelism distributes not data but tasks across multiple cores. Each task is running a unique operation.

- User applications create user-level threads, which must ultimately be mapped to kernel threads to execute on a CPU. The many-to-one model maps many user-level threads to one kernel thread. Other approaches include the one-to-one and many-to-many models.
- A thread library provides an API for creating and managing threads. Three common thread libraries include Windows, Pthreads, and Java threading. Windows is for the Windows system only, while Pthreads is available for POSIX-compatible systems such as UNIX, Linux, and macOS. Java threads will run on any system that supports a Java virtual machine.
- Implicit threading involves identifying tasks—not threads—and allowing languages or API frameworks to create and manage threads. There are several approaches to implicit threading, including thread pools, fork-join frameworks, and Grand Central Dispatch. Implicit threading is becoming an increasingly common technique for programmers to use in developing concurrent and parallel applications.
- Threads may be terminated using either asynchronous or deferred cancellation. Asynchronous cancellation stops a thread immediately, even if it is in the middle of performing an update. Deferred cancellation informs a thread that it should terminate but allows the thread to terminate in an orderly fashion. In most circumstances, deferred cancellation is preferred to asynchronous termination.
- Unlike many other operating systems, Linux does not distinguish between processes and threads; instead, it refers to each as a task. The Linux `clone()` system call can be used to create tasks that behave either more like processes or more like threads.

Deadlocks

Deadlock: Every process in a set of processes is waiting for an event that can be caused only by another process in the set.

Deadlock Characterization

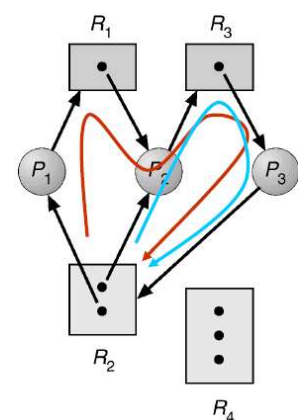
Deadlock can arise if four conditions hold simultaneously.

1. **Mutual Exclusion:** Only one process at a time can use a resource.
2. **Hold and Wait:** A process holding at least one resource is waiting to acquire additional resources held by other processes.
3. **No preemption:** A resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait:** There exist a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_n is waiting for a resource that is held by P_0 .

Resource Allocation Graph

Deadlock can be described in terms of a directed graph.

- If a graph contains **no cycles** → No deadlock
- If graph contains a **cycle** → Possibility of deadlock



Methods for Handling Deadlock

1. Deadlock Prevention

- Ensure that the system will **never enter** a deadlock state.
- Methods for ensuring that at least one of the four conditions cannot hold.
 - **Mutual Exclusion:** Not required for sharable resources; must hold for non-sharable resources.
 - **Hold and Wait:** Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when process has none → Low resource utilization, starvation possible
 - **No Preemption:** If a process that is holding some resource request, another resources that cannot be immediately allocated to it, then all resources currently being held are released (preempted). Process will be restarted only when it can regain its old resources, as well as the new one that is requesting.
 - **Circular Wait:** Impose a total ordering of all resource types and require that each process request resources in an increasing order of enumeration.
- Prevention leads to
 - Low utilization of devices
 - Low throughput
 - Frequent starvation

2. Deadlock Avoidance

- The **system knows** the complete sequence of request and releases for each process.
 - **Priori information** is available.
- The **system decides** for each request whether or not the process should wait in order to avoid a deadlock.

- Each **process declares** the maximum number of resources of each type that it may need.
- The system should always be at **safe state**.
- Dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist.

Safe, Unsafe, Deadlock State

- We can define avoidance algorithms that ensure the system will never deadlock.
- A resource is granted only if the allocation leaves the system in a **safe state**.
- If a system is in **safe state** → No deadlocks.
- If a system is in **unsafe state** → Possibility of deadlock
- **Avoidance:** Ensure that a system will never enter an unsafe state.

Safe State

- System is in safe state if there exist a **safe sequence** of all processes.
- Sequence $\langle P_1, P_2, \dots, P_n \rangle$ is **safe** if for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j , with $j < i$

Deadlock Avoidance Algorithms

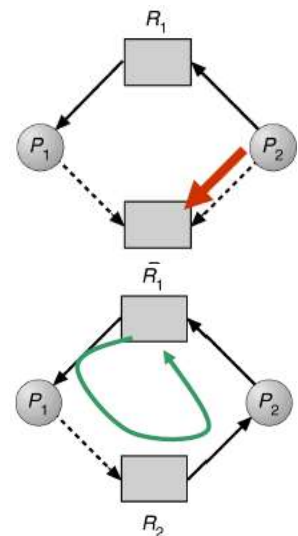
- Single instance of a resource type → Resource-Allocation Graph
- Multiple instances of a resource type → Banker's Algorithm

Resource-Allocation Graph

- Works only if each resource type has on instance.
- A cycle indicates an **unsafe state**.
- Algorithm
 - Add a claim edge $P_i \rightarrow R_j$ indicating that process P_i may request resource R_j
 - A request $P_i \rightarrow R_j$ can be granted only if adding assignment edge $R_j \rightarrow P_i$ does not result in a cycle in the graph.

→ A cycle indicates an **unsafe state**!

- Suppose that process P_i request a resource R_j
- The request can be granted only if converting the **request edge** to an **assignment edge** does not result in the formation of a **cycle** in the resource-allocation graph.

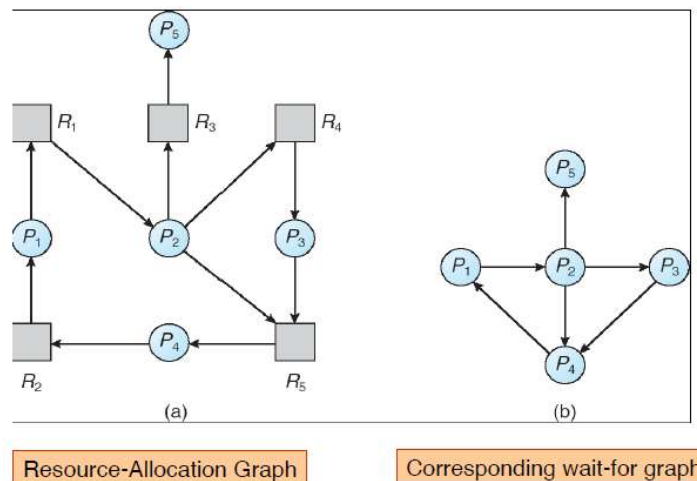


Banker's Algorithm

- Multiple instances of resource types.
- Each process must a priori claim **maximum use**.
- When a process requests a resource, it may have to wait.
- When a process requests a set of resources:
 - Will the system be at a safe state after the allocation?
 - Yes: Grant the resources to the process.
 - No: Block the process until the resources are released by some other process.

3. Deadlock Detection and Recovery

- Allow system to enter deadlock state.
- Single instance of each resource type
 - Maintain **wait-for** graph.
 - Periodically invoke an algorithm that searches for a cycle in the graph.



- Multiple instances of a resource type

Let n = number of processes, and m = number of resources types.

```

n: integer    # of processes
m: integer    # of resource-types

Available[1:m]
  #Available[i] is # of avail resources of type i
Request[1:n,1:m]
  #Current demand of each  $P_i$  for each  $R_j$ 
Allocation[1:n,1:m]
  #current allocation of resource  $R_j$  to  $P_i$ 
finish[1:n]
  #true if  $P_i$ 's request can be satisfied
  
```

Detection Algorithm

1. **Initialize:**
 - (a) $Work = Available$
 - (b) For $i=1:n$,
 - if $Allocation[i] \neq 0$, then $Finish[i] = false$
 - otherwise, $Finish[i] = true$.
2. **Find an index i such that both:**
 - (a) $Finish[i] == false$
 - (b) $Request[i] \leq Work$
 If no such i exists, go to step 4.
3. **$Work = Work + Allocation[i]$**
 $Finish[i] = true$
 go to step 2.
4. **If $Finish[i] == false$, for some i , then**
 the system is in deadlock state with P_i is deadlocked.

Requires an order of $O(mn^2)$ operations to detect whether the system is in deadlocked state.

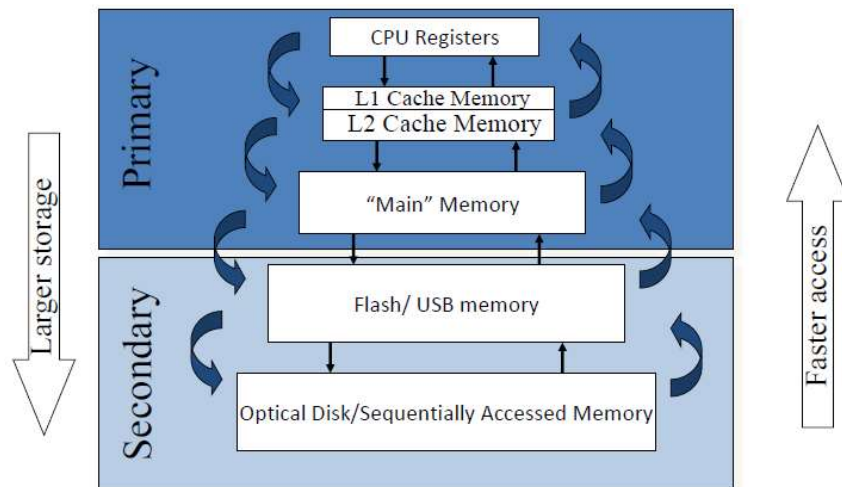
- How often should we call deadlock detection algorithm?
 - When there is a low CPU utilization.
 - Periodically but not too often.

Recovery from Deadlock

- **Killing** one/all deadlock processes.
 - Keep killing processes, until deadlock broken.
 - Repeat the entire computation.
- **Preempt** resource/processes until deadlock broken.
 - Selecting a victim (based on # resources held, how long executed)
 - Rollback: Return to same safe state, restart process for that state.
 - Starvation: Same process may always be picked as victim, include number of rollbacks in cost factor.

Memory Management

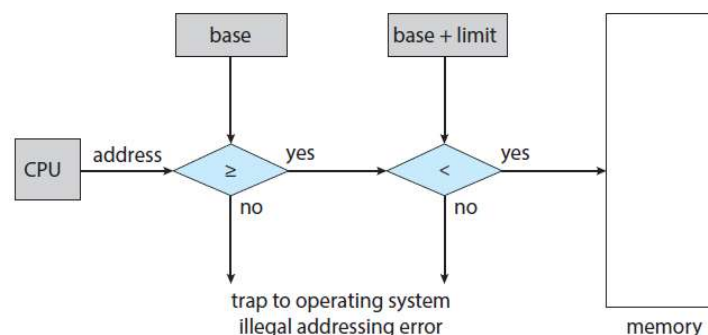
- CPU reads instructions and read/writes data from/to memory.



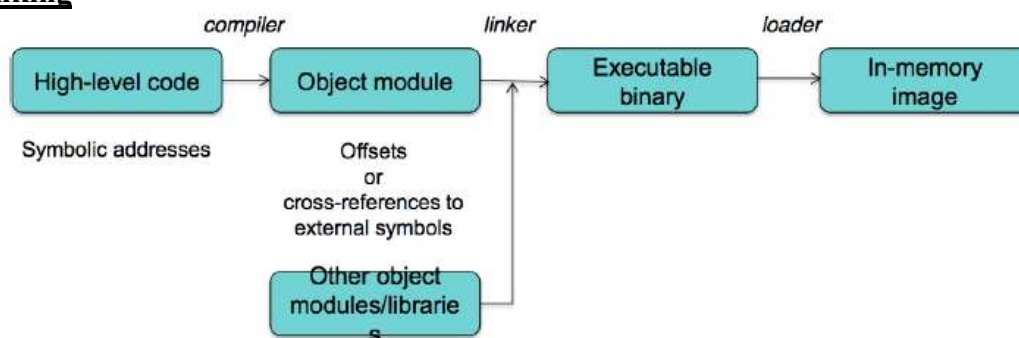
- Program must be brought (from disk) into memory and placed within a process for run.
- **Main memory** and **registers** are only storage CPU can access directly.
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a stall.
- **Cache** sits between main memory and CPU registers.
- Protection of memory required to ensure correct operation.

Hardware Address Protection

- We need to sure that each process has a separate memory space.
- This protects the processes from each other and is fundamental to having multiple processes.
- **Base and Limit Register** are used to provide this protection.

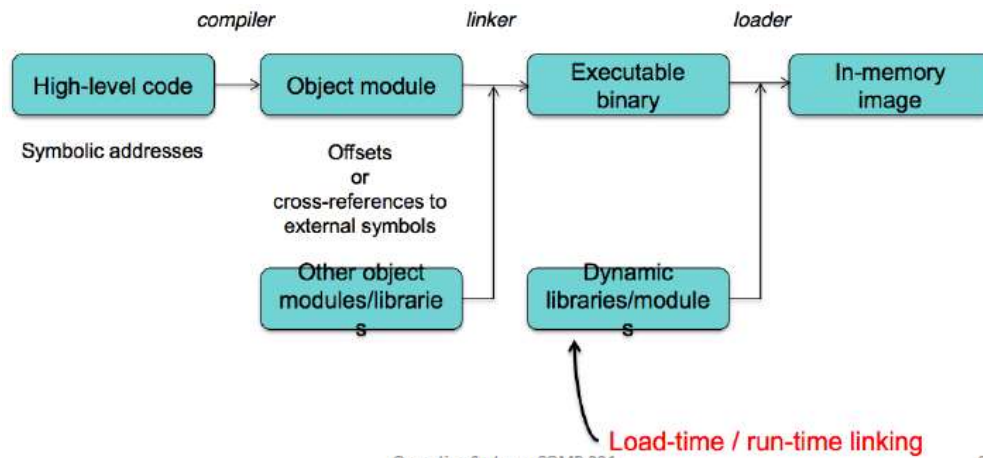


Static Linking



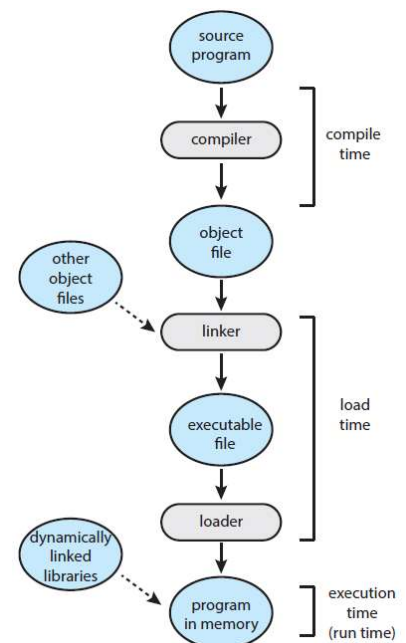
Dynamic Linking

- A process loads library at load time.
- OS loader finds the dynamic libraries and brings them into the process' memory address space.



Address Binding

- Addresses in the source program are generally symbolic. A compiler typically **binds** these symbolic addresses to relocatable addresses.
- Address binding of instruction and data to memory addresses can happen at 3 different stages.
- **Compile time:**
 - If you know at compile time where the process will reside in memory, then **absolute code** can be generated.
 - If starting location changes, code must recompile.
- **Load time:**
 - If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code**.
 - Final binding is delayed until load time.
 - If the starting location changes, only reload the code
- **Execution time:**
 - Binding delayed until run time if the process can be moved during its execution from one memory segment to another.
 - Need hardware support for address maps (e.g., base and limit registers).



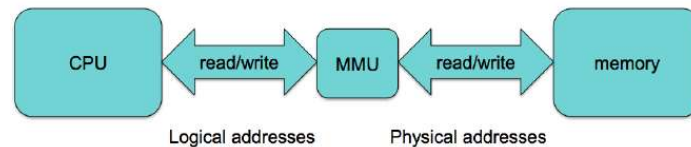
Logical vs Physical Address Space

- **Logical Address:** Generated by the CPU; also referred to as virtual address.
- **Physical address:** Address seen by the memory unit.
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes.
- Logical (virtual) and physical addresses differ in execution-time address-binding scheme because addresses are set to different spaces.
- User program deals with **logical** address. It never sees **real physical address** → **protection!**

Logical Addressing

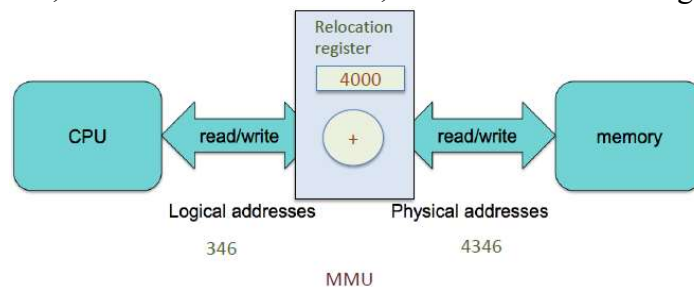
Memory Management Unit (MMU):

- Real time, on-demand translation between logical (virtual) and physical addresses.
- Maps the logical address dynamically by adding the value in the relocation register.



Relocation Register

- User program only generates logical addresses and thinks that the process runs in location 0 to max. But in fact, it runs $R + 0$ to $R + \text{max}$, where R is the base register.



Relocatable Addressing

- $\text{Physical Address} = \text{Logical Address} + \text{Base Register}$
- But first check that: $\text{logical address} < \text{limit}$

Base and Limit Registers

- A pair of base and limit registers define the logical address space.
- CPU must check every memory access generated in user mode to be sure it is between in base and limit for that user.

Hardware Protection

- When executing in kernel mode, the OS has unrestricted access to both kernel and user's memory.
- The load instructions for the base and limit registers are privileged instructions.

Memory Allocation

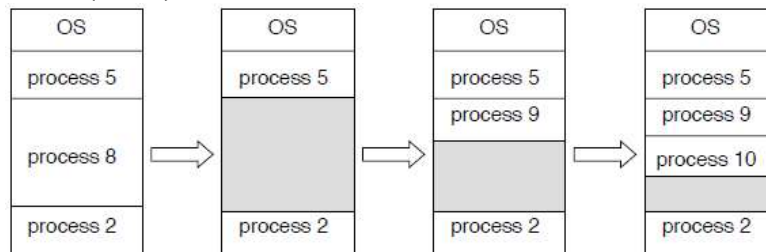
- Main memory must support both OS and user processes.
- Limited resource must allocate efficiently.

1. Contiguous Allocation

- Main memory is usually divided into two partitions:
 - **Resident operating system**, usually held in low memory with interrupt vector.
 - **User processes** then held in high memory.
- **Relocation registers** used to protect user processes from each other, and from changing operating-system code and data
 - **Base (relocation) register** contains value of smallest physical address
 - **Limit register** contains range of logical addresses – each logical address must be less than the limit register
 - MMU (memory management unit) maps logical address dynamically

- **Multiple-partition allocation**

- **Hole** – block of available memory; holes of various sizes 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 **allocated partitions** and free **partitions (holes)**



Dynamic Storage-Allocation Problem

- First-fit: Allocate the **first** hole that is big enough.
- Best-fit: Allocate the **smallest** hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- Worst-fit: Allocate the **largest** hole; must search entire list. Produces largest leftover hole.

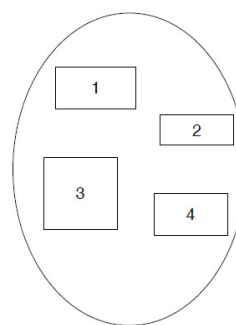
➔ First-fit and best-fit perform better than worst-fit in terms of decreasing time (speed) and storage utilization, however they cause fragmentation.

Fragmentation

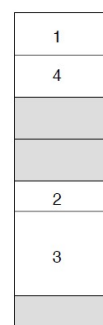
- External Fragmentation
 - Total memory space exists to satisfy a request, but it not contiguous.
 - Also, a common problem in disk as well.
- Internal Fragmentation
 - Allocated memory may be slightly larger than requested memory. This size difference is memory internal to a partition, but not being used.
- What if a process needs more memory?
 - Always allocate some extra memory just in case
 - Find a hole big enough to relocate the process.
- Reduce **external fragmentation** by **compaction**.
 - Shuffle memory contents to place all free memory together in one large block.
 - Compaction is possible only if relocation is dynamic and is done at execution time.

2. Segmentation

- Memory allocation mechanism that supports user view of memory.
- Users prefer to view memory as a collection of variable-sized segments
- A program is collection of segments. A segment is a logical unit such as
 - Main program, function object, local/global variables...



user space

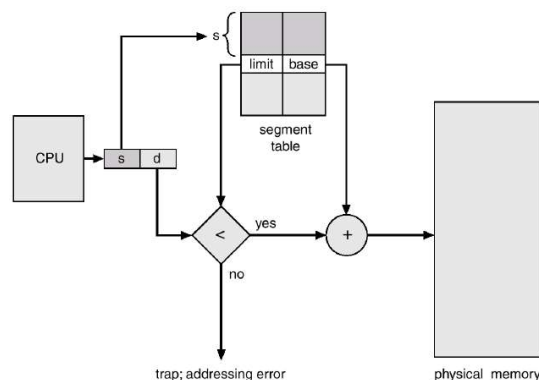


physical memory space

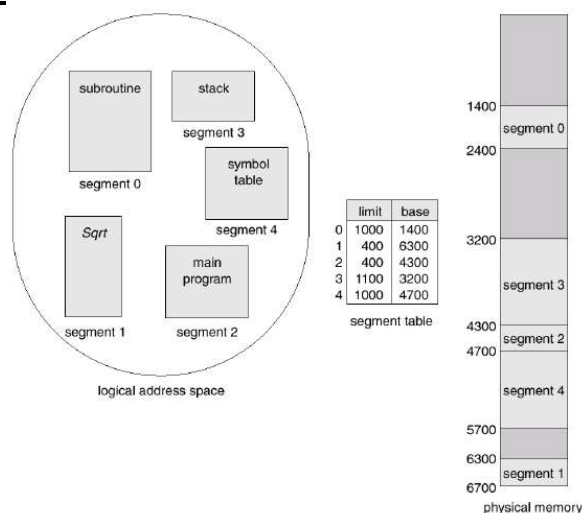
Segmentation Architecture

- Logical address consists of a two tuple: Segment-number, offset.
- **Segment Table:** maps two dimensional physical addresses. Each table entry has
 - **Base:** Contains the starting physical address where the segments reside in memory.
 - **Limit:** specifies the length of the segment.
- **Segment-table base register (STBR)** points to the segment table's location in memory.
- **Segment-table length register (STLR)** indicates number of segments used by a program.
 - Segment number s is legal if $s < \text{STLR}$
- Protection
 - With each entry in segment table associate
 - Validation bit = 0 → illegal segment
 - Read/write/execute privileges.
- Protection bits associated with segments. Code sharing occurs at segment level.
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

Segmentation Hardware



Example of Segmentation



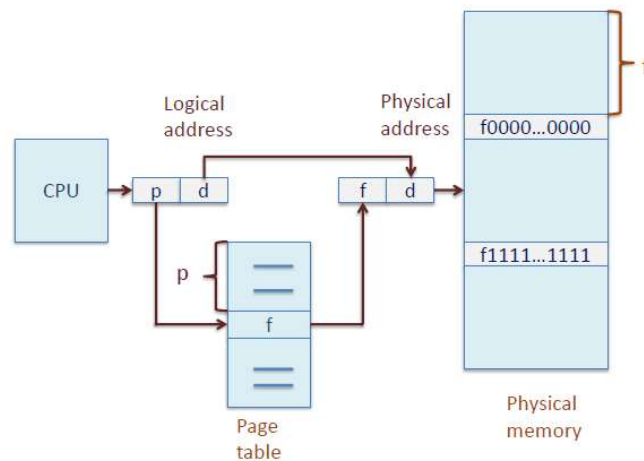
3. Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.
- Divide **physical memory** into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 16MB).
- Divide **logical memory** into blocks of the same sized blocks called **pages**.
- Keep track of all free frames. Set up a page table to translate logical to physical addresses.

Address Translation Scheme

- Address generated by CPU (logical address) is divided into two parts
 - **Page number (p):** used as an index into a page table which contains base address of each page in physical memory.
 - **Page offset (d):** combined with base address to define the physical memory address that is sent to the memory unit.
- Steps of MMU to translate a logical address (generated by the CPU) to a physical address
 - Extract page number p and use it as an index into the page table.
 - Extract corresponding frame number f from the page table
 - Replace page number p in the logical address with the frame number f

Paging Hardware



- Paging is a form of dynamic relocation.
- No external fragmentation.
- May have internal fragmentation (in the last frame of a process.)

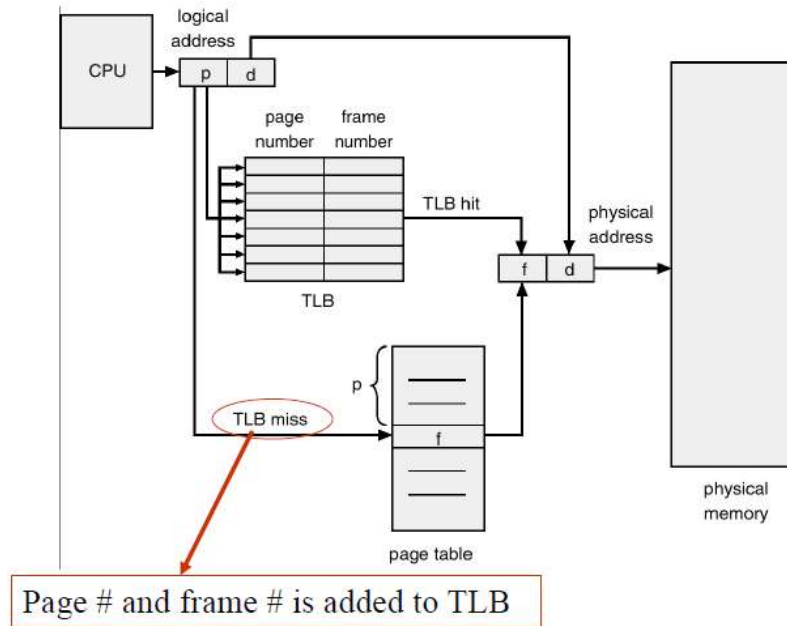
Implementation of Page Table

- Page table is kept in main memory.
- **Page-table base register (PTBR)** points to the page table.
- **Page-table length register (PTLR)** indicates size of the page table.
- **Problem:** Every data/instruction access requires two memory accesses:
 - one for the page table and
 - one for the data/instruction.
- **Solution:** The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**

Associative Memory

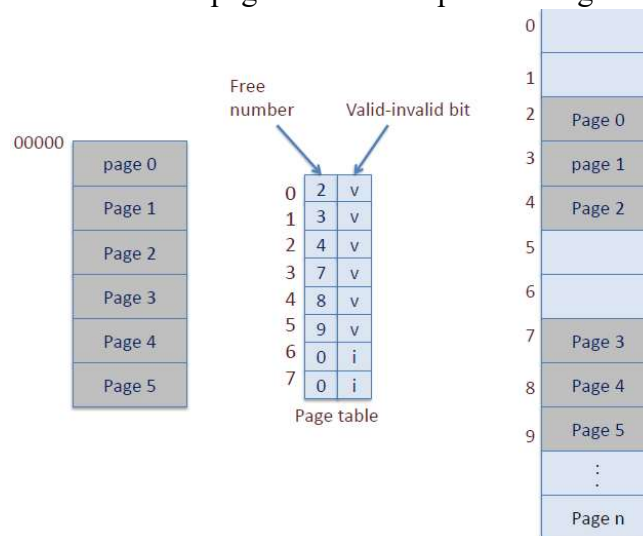
- Associative memory – parallel search
- Address translation (p, d)
 - If p is in associative register, get frame number.
 - Otherwise get frame number from page table in memory.
- Search is fast.
- TLB contains some of the page table entries but not all

Paging Hardware with TLB



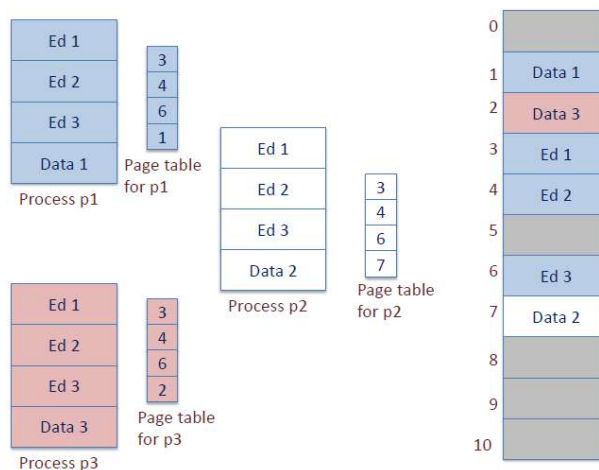
Memory Protection

- Memory protection implemented by associating **protection bit** with each page/frame.
- **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.



Shared Pages

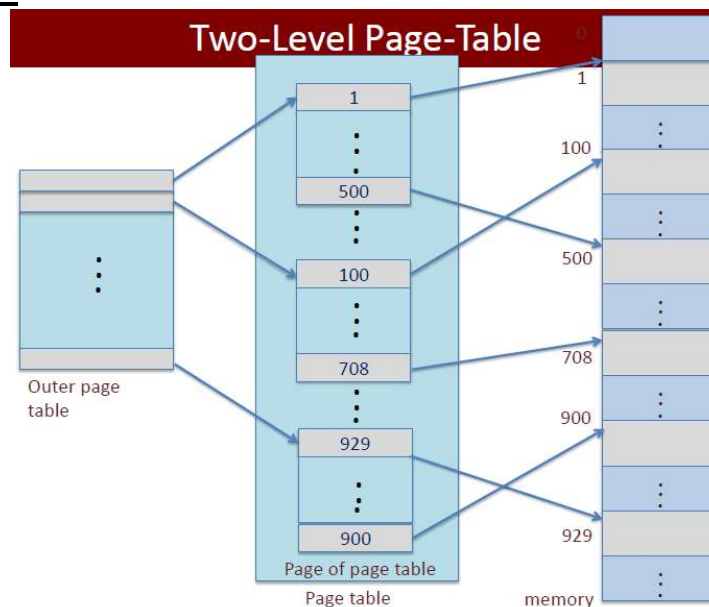
- **An advantage of paging is the possibility of sharing common code.**
- **Shared code**
 - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
 - Shared code must appear in same location in the logical address space of all processes.
- **Private code and data**
 - Each process keeps a separate copy of the code and data.
 - The pages for the private code and data can appear anywhere in the logical address space.



Hierarchical Page Tables

- Break up the logical address space into multiple page tables.
- A simple technique is a two-level page table.
- Page table itself is also paged.

Two-Level page table

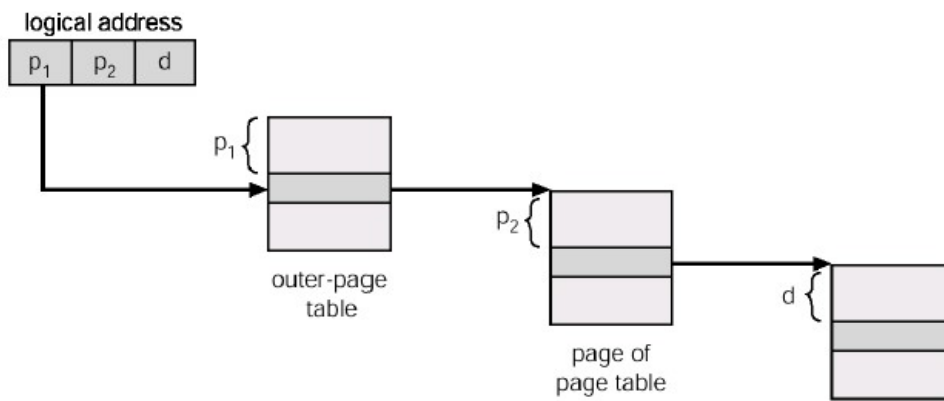


- A logical address (on 32-bit machine with 4K page size) is divided into:
 - a **page number** consisting of 20 bits.
 - a **page offset** consisting of 12 bits.
- Since the page table is paged, the **page number** is further divided into:
 - a 10-bit page number.
 - a 10-bit page offset.
- Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

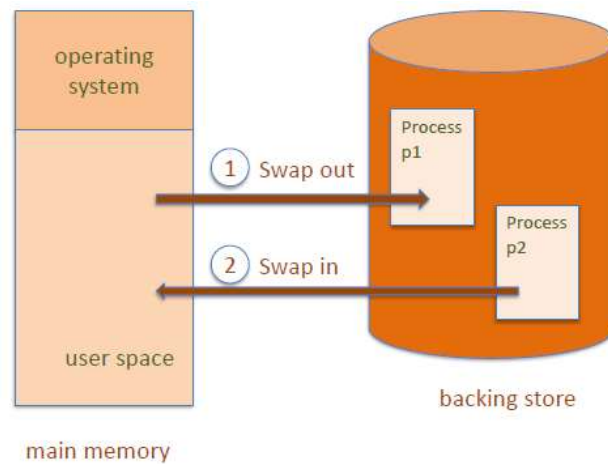
where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table.

Address Translation scheme

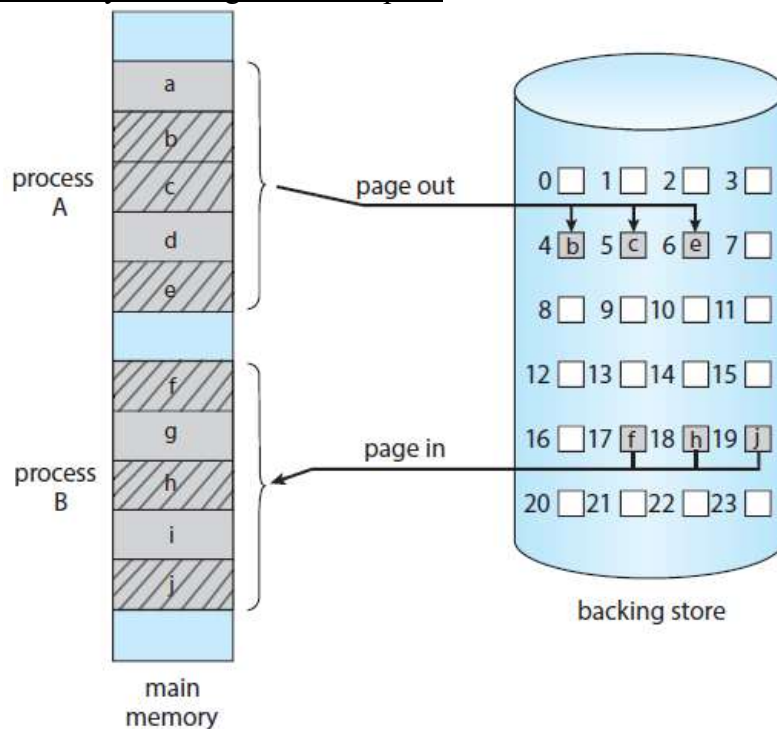


Swapping

- A process can be **swapped** to temporarily out of memory to backing store, and then brought back into memory for continued execution.



Transfer of a Paged memory to contiguous disk space



Summary

- Memory is central to the operation of a modern computer system and consists of a large array of bytes, each with its own address.
- One way to allocate an address space to each process is through the use of base and limit registers. The base register holds the smallest legal physical memory address, and the limit specifies the size of the range.
- Binding symbolic address references to actual physical addresses may occur during (1) compile, (2) load, or (3) execution time.
- An address generated by the CPU is known as a logical address, which the memory management unit (MMU) translates to a physical address in memory.
- One approach to allocating memory is to allocate partitions of contiguous memory of varying sizes. These partitions may be allocated based on three possible strategies: (1) first fit, (2) best fit, and (3) worst fit.
- Modern operating systems use paging to manage memory. In this process, physical memory is divided into fixed-sized blocks called frames and logical memory into blocks of the same size called pages.
- When paging is used, a logical address is divided into two parts: a page number and a page offset. The page number serves as an index into a per process page table that contains the frame in physical memory that holds the page. The offset is the specific location in the frame being referenced.
- A translation look-aside buffer (TLB) is a hardware cache of the page table. Each TLB entry contains a page number and its corresponding frame.
- Using a TLB in address translation for paging systems involves obtaining the page number from the logical address and checking if the frame for the page is in the TLB. If it is, the frame is obtained from the TLB. If the frame is not present in the TLB, it must be retrieved from the page table.
- Hierarchical paging involves dividing a logical address into multiple parts, each referring to different levels of page tables. As addresses expand beyond 32 bits, the number of hierarchical levels may become large. Two strategies that address this problem are hashed page tables and inverted page tables.
- Swapping allows the system to move pages belonging to a process to disk to increase the degree of multiprogramming.
- The Intel 32-bit architecture has two levels of page tables and supports either 4-KB or 4-MB page sizes. This architecture also supports page address extension, which allows 32-bit processors to access a physical address space larger than 4 GB. The x86-64 and ARMv9 architectures are 64-bit architectures that use hierarchical paging.

Virtual Memory Management

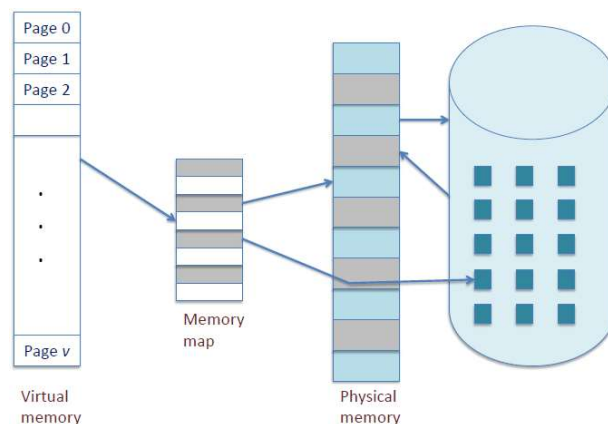
What happens if a process needs more memory than there is available physical memory?

Main Memory

- Both data and code reside in main memory for a program
- In main memory, there are multiple programs (processes) exist at the same time.
- OS as well needs some space in the main memory.

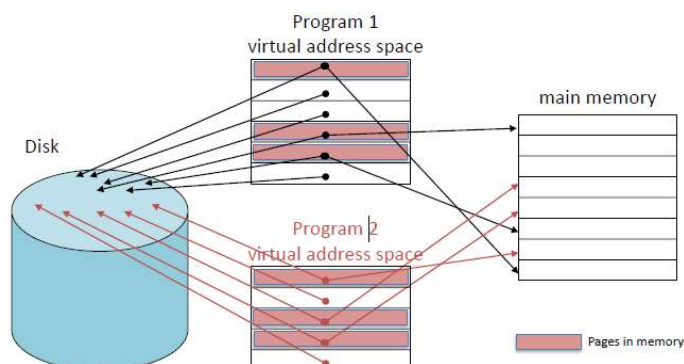
Virtual Memory

- Separation of user logical memory from physical memory.
 - Only part of the program needs to be in memory for execution.
 - Logical address space can therefore be much larger than physical address space.
 - Allows address spaces to be shared by several processes.
 - Allows for more efficient process creation.
- Use main memory as a **cache** for secondary memory (disk)
- Allows efficient and **safe** sharing of memory among multiple programs
- Provides the ability to easily run programs larger than the size of physical memory.
- Automatically handles bringing in data from disk.
- Can be implemented via demand paging.



Two Programs Sharing Physical Memory

- Each program is compiled into its own address space – a “**virtual**” address space.
- Address space is divided into **pages**.
- All these pages are in disk but only some of them are in main memory (physical memory) during program execution.

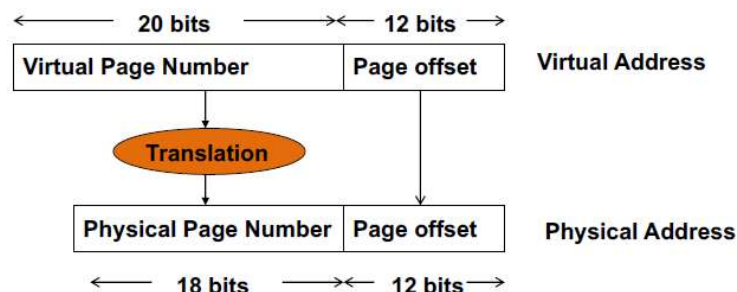


Overview of Paging

1. Based on the notion of a **virtual address space**
 - A large, contiguous address space that is only an illusion
 - Virtual address space → Physical address space
 - Each program gets its own separate virtual address space
 - Each **process**, not each thread
2. Divide the address spaces into fixed-sized pages
 - **Virtual Page:** a chunk of the virtual address space
 - **Physical page:** A chunk of the physical address space (also called a **frame**)
 - Size of virtual page == size of physical page.
3. **Map** virtual pages to physical pages
 - By itself, a virtual page is merely an illusion
 - Cannot actually store anything
 - Needs to be backed-up by a physical page
 - Before a virtual page can be accessed
 - It must be paired with a physical page
 - It must be **mapped** to a physical page
 - This mapping stored somewhere
 - On every subsequent access to the virtual page
 - Its mapping is looked up. Then, the access is directed to the physical page

Virtual and Physical Addresses

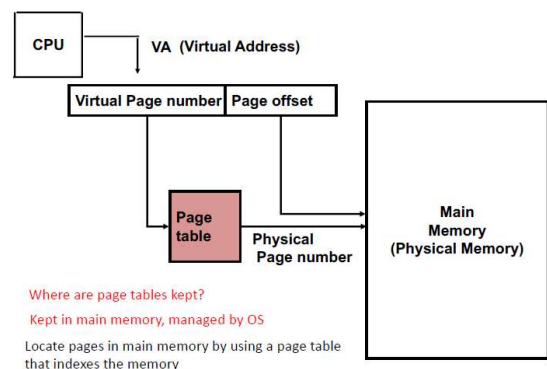
- Each memory request first requires an **address translation** from virtual space to physical space.
- Translation is done by a combination of **hardware and OS support**



- Page Size: $2^{12} = 4 \text{ KB}$
- How many pages are allowed in physical memory: $2^{18} * \text{Page Size} = 1 \text{ GB}$
- How many pages are allowed in virtual memory: $2^{20} * \text{Page Size} = 4 \text{ GB}$

Address Translation (VPN → PPN)

- Page Table: a “lookup table” for the mappings
 - Can be thought of as an array
 - Each element in the array is called a page table entry (PTE)



Demand Paging

- Similar to a paging system with swapping
- **Bring a page into memory only when it is needed.**
 - Less I/O needed
 - Less memory needed
 - Faster response
 - More users
- Page is needed → reference to it
 - invalid reference → abort
 - **not-in-memory → bring it to memory**
- **Lazy swapper** – never swaps a page into memory unless page will be needed

Valid-Invalid Bit

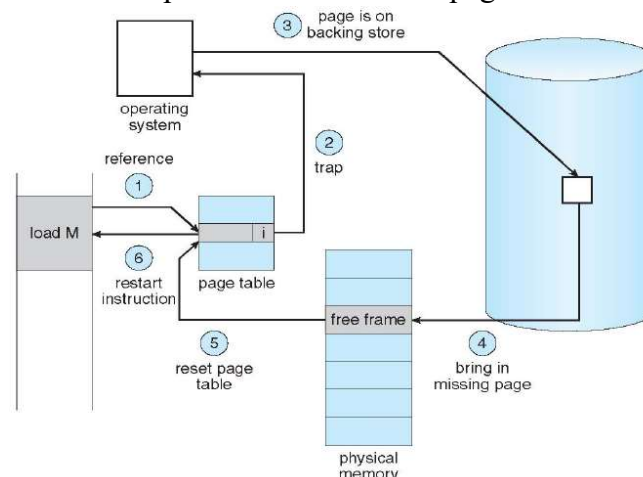
- While a process is executing, some pages will be in memory, and some will be in secondary storage. **Valid-invalid** bit distinguish between these two.
- Each page table entry has a valid-invalid bit
 - **Valid** → page is legal and in memory
 - **Invalid** → page is not valid or valid but not in memory (in secondary storage)
- Initially set to i for all entries
- During address translation, trying to access to a page marked **invalid**, causes a **page fault**.

Page Fault

- If the valid bit of the page table is zero, this means that the page is **not in main memory**.
- In this case, reference to an invalid page, then a **page fault** occurs, and the missing page is read in from disk.
- If there is a reference to a page, first reference to that page will trap to OS: page fault

Steps in Handling a Page Fault

1. OS looks at another table (kept in Process control Block) to determine whether the reference was a valid or an invalid.
2. If it was invalid, then terminate the process. If it was valid but we have not yet brought in that page, we now page it in
3. Get an empty frame.
4. Swap page into frame
5. Reset table. Set validation bit to v
6. Restart the instruction of the process that cause the page fault



Page Fault in detail

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
 1. Wait in a queue for this device until the read request is serviced
 2. Wait for the device seek and/or latency time
 3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Aspects of Demand Paging

- Extreme case – start process with no pages in memory
 - OS sets instruction pointer to first instruction of a process, nonmemory-resident → page fault
 - And for every other process page on first access → **Pure demand paging**
- Actually, a given instruction could access multiple pages → multiple page faults

→ Hardware support needed for demand paging

- **Page Table**
 - This table has the ability to mark an entry invalid through valid-invalid bit
- **Secondary memory**
 - This memory holds those pages that are not present in main memory.
 - Known as the **swap device**.
 - The section of storage used for this purpose is known as **swap space**.
- **Instruction restart**
 - Crucial requirement for demand paging is the ability to restart any instruction after a page fault.
 - When the page fault occurs, we must be able to restart the process in exactly the same place and state

→ On a page fault the thread state is set to blocked as an I/O operation is required to bring the new page into memory.

→ On a TLB-miss, the thread continues running if the address is resolved in the page table

Address Space of Processes

- Each process has its own virtual address space.
 - X writing to its virtual address does not affect the data stored in Y's virtual address.
 - This was the entire purpose of virtual memory.
 - Each process has its own page directory and page tables.

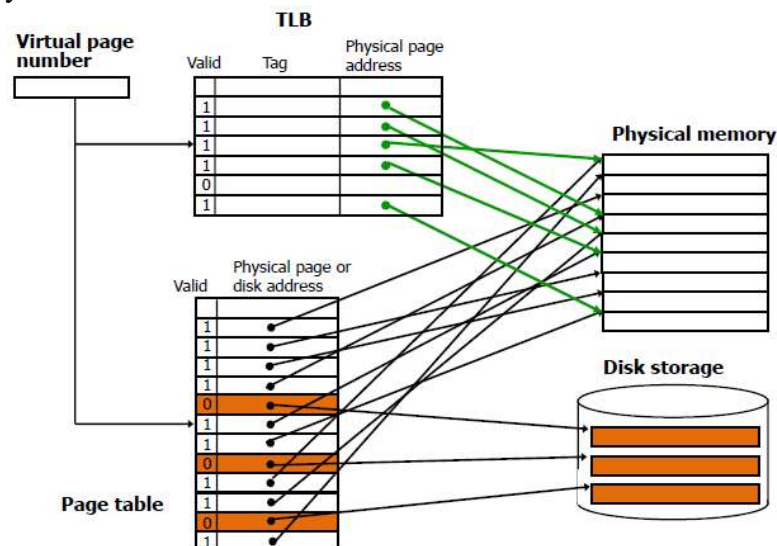
- When process starts running (process X stops running), the page directory base register's value must be updated.

Accessing Page Tables

- Problem
 - Need to translate Virtual addresses into Physical address for every load/store
 - Translation is done through page tables
 - Page tables are in memory!
- Accessing page tables is slow.
 - Must access memory for load/stores **even cache hits!**
 - Worse, if translation is not completely in memory, may need to go to disk before hitting in cache.
 - **Page tables are kept in pages as well**

Translation-Lookaside Buffer (TLB)

- A TLB acts as a cache for the page table, by storing physical addresses of pages that have been recently accessed.



Copy-on-Write

- **Copy-on-Write (COW)** allows both parent and child processes to initially share the same pages in memory.
 - If either process modifies a shared page, only then is the page copied.
- COW allows more efficient process creation as only modified pages are copied.
- In general, free pages are allocated from a **pool of zero-fill-on-demand** pages.

What happens if there is no free frame?

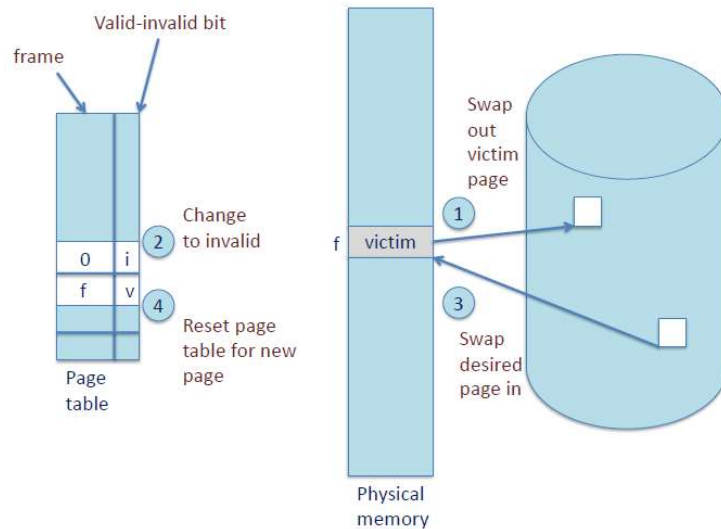
- **Page replacement:** find a frame in memory, but not really in use, swap it out.
- Same pages may be brought into memory several times.

Page Replacement

- **Page Replacement is basic to demand paging.**
- Page replacement completes separation between logical memory and physical memory.
- **Large virtual memory can be provided on a smaller physical memory.**
- Use modify bit (dirty bit) to reduce overhead of page transfers – only modified pages are written to disk.

Basic Page Replacement

1. Find the location of the desired page on disk.
2. Find free frame
 - If there is a free frame, use it
 - Otherwise, use a **page replacement algorithm** to select a **victim** frame
 - Write the victim frame to secondary storage, change the page and frame tables accordingly.
3. Save the victim frame to the disk if the frame is modified.
4. Read the desired page into the (newly) free frame. Update the page and frame tables.
5. Restart the process from where it is left.



Page and Frame Replacement Algorithms

- **Page replacement algorithm** wants lowest page-fault rate on both first access and re-access
- **Frame-allocation algorithm** determines how many frames to give each process and which frames to replace.
- Evaluate an algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.
 - String is just page number, not full addresses.
 - Repeated access to the same page does not cause a page fault.

1. FIFO Page Replacement

- Each page is associated the time when it was brought into memory.
- When a page must be replaced, the oldest page is chosen.
- **Belady's Anomaly:** Page fault rate may increase as the number of allocated frames increases
 - Ideally → more frames = less page faults

2. Optimal Page Replacement

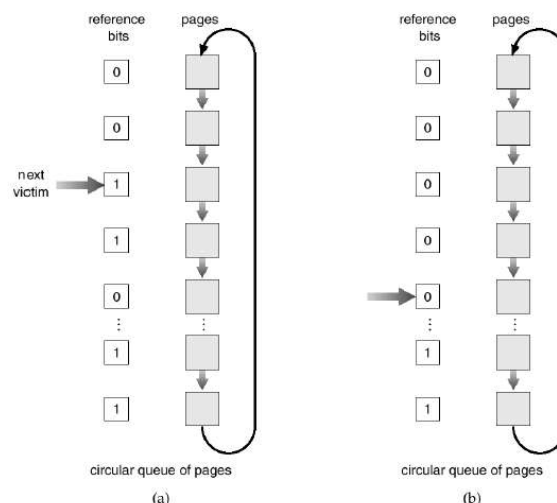
- Replace page that will not be used for the longest period of time.
- Lowest page-fault rate of all algorithms
- Never suffers from Belady's anomaly.
- It requires future knowledge of reference string which is not possible! Therefore, it used for measuring how well your placement algorithm performs because you cannot be better.

3. Least Recently Used (LRU) Algorithm

- Replace page that has not been used for longest period of time.
- Does not suffer from Belady's anomaly.
- Counter Implementation of LRU
 - Every page entry has a counter. Every time a page is referenced, copy the clock into counter of the page.
 - When a page needs to be replaced, look at the counter to determine which one to change (replace the page with the smallest counter) → least recently used.

4. LRU Approximation Algorithms

- LRU needs hardware support.
- **Reference bit**
 - With each page associate a bit, initially = 0
 - When page is referenced, bit set to 1.
 - Periodically reset all the bits to 0
 - Replace ant with reference bit = 0 (if one exists)
- **Additional reference bits**
 - 8-bit shift register
 - At regular intervals OS shifts the reference bit for each page into the high-order bit
 - 1100100 → 0110010 if not used.
 - 1100100 → 1110010 if used.
- **Second chance algorithm**
 - Generally FIFO, plus hardware-provided reference bit
 - Clock replacement
 - If page to be replaced has
 - Reference bit = 0 then replace it
 - Reference bit = 1 then
 - Set reference bit 0, leave page in memory
 - Replace next page, subject to same rules



5. Counting Algorithms

- Keep a counter of the number of references that have been made to each page.
- **LFU (Least Frequently Used):** replaces page with smallest count. Set the counter to zero when a page is moved into memory.
- **MFU (Most Frequently Used):** based on the argument that the page with smallest count was probably just brought in and has yet to be used.

Application and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some application have better knowledge (databases)
- Memory intensive application can cause double buffering.
 - OS keeps copy of a page in memory as I/O buffer
 - Application keeps page in memory for its own work.

Allocation of Frames

- Each process needs **minimum** number of pages. There must be enough frames to hold all the different pages that any single instruction can reference.
- **Two major allocation methods:**
 - Fixed allocation (equal, proportional)
 - Priority allocation

1. Fixed allocation

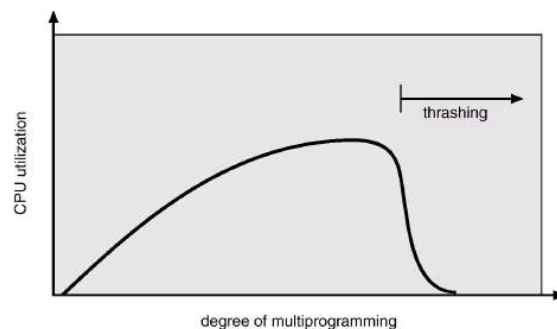
- Equal allocation: Give each process equal pages.
- Proportional allocation: Allocate according to the size of process.

2. Priority Allocation

- Use a proportional allocation method **using priorities rather than size.**
- Of a process generates a page fault
 - Select for replacement one of its frames
 - Select for replacement a frame from a process with lower priority number
- **Global replacement**
 - Process selects a replacement frame from the set of all frames, one process can take a frame from another.
 - Generally results in greater system throughput
- **Local replacement**
 - Each process selects from only its own set of allocated frames

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high, leading to
 - Low CPU utilization
 - OS thinks that it needs to increase the degree of multiprogramming
 - Another process is added to the system, make it worse
- **Thrashing** = a process is busy swapping pages in and out
- A process is thrashing if it is spending more time for paging than executing.



- Thrashing occurs when total size of locality > total memory size

Working-Set Model

- It prevents thrashing while keeping the degree of multiprogramming as high as possible → optimizes CPU utilization
- To prevent thrashing, provide a process as many frames as it needs
- Working set model defines the **locality model** of process execution
 - An approximation of the set of pages that the process will access in the future
- The OS
 - Monitors the working set of each process and allocated to that working set enough frames to provide it with its working-set size
 - If there are enough frames left, it can increase degree of multiprogramming
 - Otherwise, it will suspend some of the processes

Summary

- Virtual memory abstracts physical memory into an extremely large uniform array of storage.
- The benefits of virtual memory include the following: (1) a program can be larger than physical memory, (2) a program does not need to be entirely in memory, (3) processes can share memory, and (4) processes can be created more efficiently.
- Demand paging is a technique whereby pages are loaded only when they are demanded during program execution. Pages that are never demanded are thus never loaded into memory.
- A page fault occurs when a page that is currently not in memory is accessed. The page must be brought from the backing store into an available page frame in memory.
- Copy-on-write allows a child process to share the same address space as its parent. If either the child or the parent process writes (modifies) a page, a copy of the page is made.
- When available memory runs low, a page-replacement algorithm selects an existing page in memory to replace with a new page. Page replacement algorithms include FIFO, optimal, and LRU. Pure LRU algorithms are impractical to implement, and most systems instead use LRU-approximation algorithms.
- Global page-replacement algorithms select a page from any process in the system for replacement, while local page-replacement algorithms select a page from the faulting process.
- Thrashing occurs when a system spends more time paging than executing.
- A locality represents a set of pages that are actively used together. As a process executes, it moves from locality to locality. A working set is based on locality and is defined as the set of pages currently in use by a process.

File System

- A collection of related bytes having meaning only to the creator. The file can be “free formed”, indexed, structured, etc.
- The file is an entry in a directory.
- The file may have structure (OS may or may not know about this). It’s a tradeoff of capabilities versus overhead. For example
 - An OS understands program image format to create a process.
 - The UNIX shell understands how directory file look. (In general the UNIX kernel doesn’t interpret files.)
 - Usually the OS understands and interprets file type.

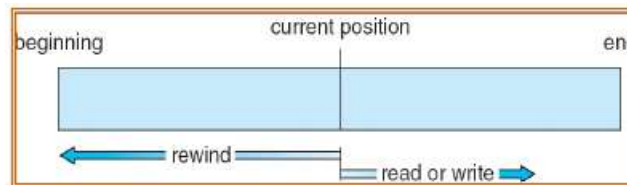
File Attributes

- Name: only information kept in human-readable form
- Identifier: unique tag (number) identifies file within file system
- Type: needed for systems that support different types
- Location: pointer to file location on device
- Size: current file size
- Protection: controls who can do reading, writing, executing
- Time, date, and user identification: data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk
- For large files, the files themselves may contain structure, making access faster.

Access methods

Sequential Access

- Simplest access method.
- Implemented by the file system.
- Data is accessed one record right after the last
- Reads cause a pointer to be moved ahead by one
- Writes allocate space for the record and move the pointer to the new End Of File
- Such a method is reasonable for tape



Direct (Relative) Access

- Method useful for disks.
- The file is viewed as a numbered sequence of blocks or records.
- There are no restrictions on which blocks are read/written in any order.
- User now says "read n" rather than "read next".
- "n" is a number relative to the beginning of file, not relative to an absolute physical disk location.

Other Access Methods

- Built on top of direct access and often implemented by a user utility.
- **Indexed ID + pointer**

- An index block says what's in each remaining block or contains pointers to blocks containing particular items. Suppose a file contains many blocks of data arranged by name alphabetically.

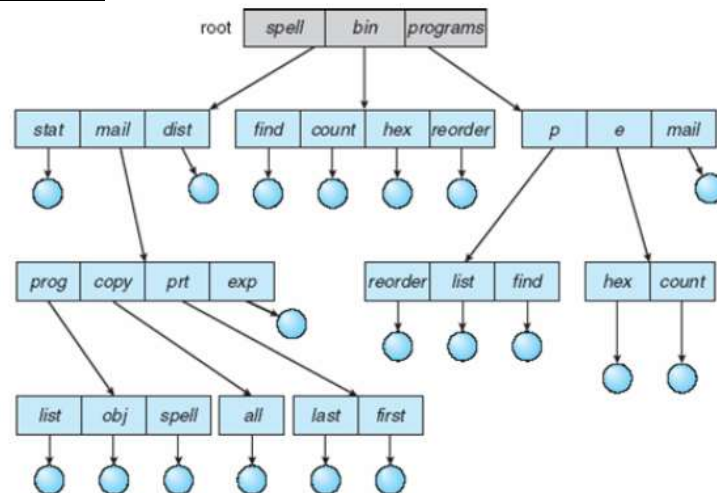
Directory Structure

- **Directories** maintain information about files, a directory itself is a file.
- For a large number of files, may want a directory structure – directories under directories.
- Information maintained in a directory:

Name	The user visible name.	
Type	The file is a directory, a program image, a user file, a link, etc.	
Location	Device and location on the device where the file header is located.	
Size	Number of bytes/words/blocks in the file.	
Position	Current next-read/next-write pointers.	
Protection	Access control on read/write/ execute/delete.	
Usage	Open count	In Memory only!
Usage	time of creation/access, etc.	
Mounting	a filesystem occurs when the root of one filesystem is "grafted" into the existing tree of another filesystem.	

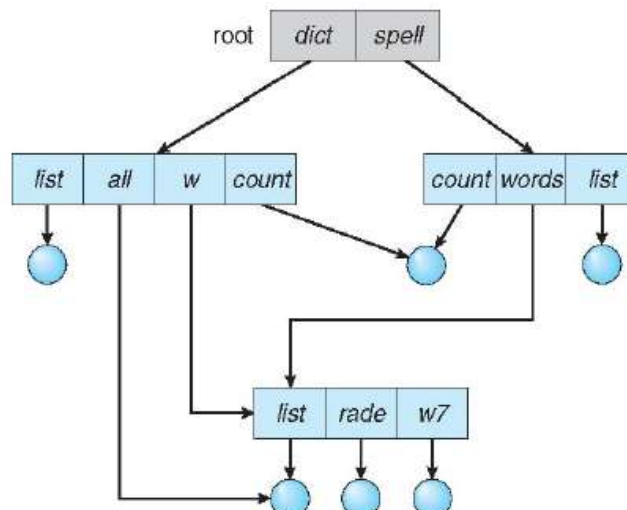
○

Tree Structured Directories



Acyclic-Graph Directories

- Have shared subdirectories and files.
- Implemented via links.



Symbolic vs Hard Links

- Both are used for aliasing files (Multiple names for the same file)
- Symbolic Link
 - contains a reference to another file or directory in the form of an absolute or relative path
 - When the link is deleted, file remains
 - When the file is deleted, the link remains – user has to clean up
 - `ln -s target_path link_path`
- Hard Links
 - Keep a reference (link) count
 - When count is zero, delete the file as well
 - Not recommended to use
 - Only few OSs support with a root access

File System Mounting

- A file system must be **mounted** before it can be accessed
- An unmounted file system is mounted at a **mount point**
- Mount Point
 - Mac OS X searches for a file system on the device at boot time or while the system running. It automatically mounts the file system under the /Volume directory
 - UNIX requires explicit mount usually under /mnt. The ones listed in in configuration file containing list of devices automatically mount

File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a **protection** scheme
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method
- In multi-user system
 - **User IDs** identify users, allowing permissions and protections to be per user
 - **Group IDs** allow users to be in groups, permitting group access rights
 - Owner of a file / directory
 - Group of a file / directory

Access List and Groups in UNIX

- Mode of access: read (4), write (2), execute (1)
- Three classes of users on UNIX RWX
 - Owner Access 7 ➔ 1 1 1
 - Group Access 6 ➔ 1 1 0
 - Public Access 1 ➔ 0 0 1
- Ask manager to create a group (unique name) and add some users to the group
- For a particular file or subdirectory, define an appropriate access.

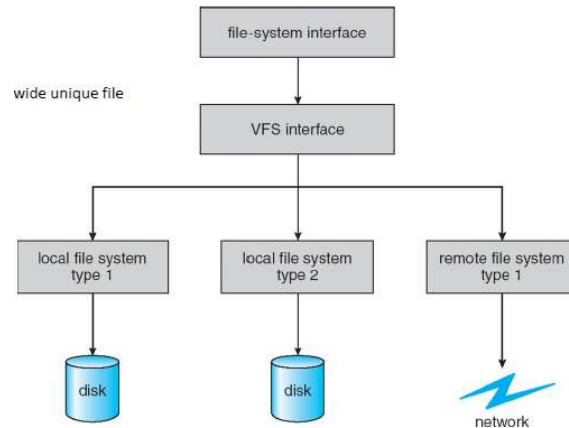
Virtual File Systems

- Virtual File Systems (VFS) on Unix 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

- Separates file-system generic operations from implementation details
- Implementation can be one of many file systems types, or network file system
- Then dispatches operation to appropriate file system implementation routines
- The API is to the VFS interface, rather than any specific type of file system

Schematic View

- **vnode**: network wide unique file
- **inode**: unique within only a single file system



Implementation

- For example, Linux VFS has four object types:
 - **Inode Object**: represents an individual file
 - **File Object**: represents an open file.
 - **Superblock Object**: represents entire file system.
 - **Dentry Object**: individual directory entry
- VFS defines set of operations on the objects that must be implemented. Such as open, close, read, write,

Kernel Space

- **System Open File Table**
 - the kernel keeps a data structure called the system open-file table which has an entry for each connection (process-to-file). Each entry contains
 - the connection status, e.g. read or write,
 - the current offset in the file, and
 - a pointer to a vnode, which is the OS's structure representing the file, irrespective of where in the file you may currently be looking.
- **Vnode Table**
 - has an entry for each open file or device.
 - contains information about the type of file and pointers to functions that operate on the file.
 - Typically for files, the vnode also contains a copy of the inode for the file, which has "physical" information about the file, e.g. where exactly on the disk the file's data resides.

Physical Drive

- The physical device: inodes, etc.
 - a file may be broken up into many data blocks, which may be widely distributed across the physical drive.
 - The inode for a file contains the locations of each of the data blocks comprising the file.

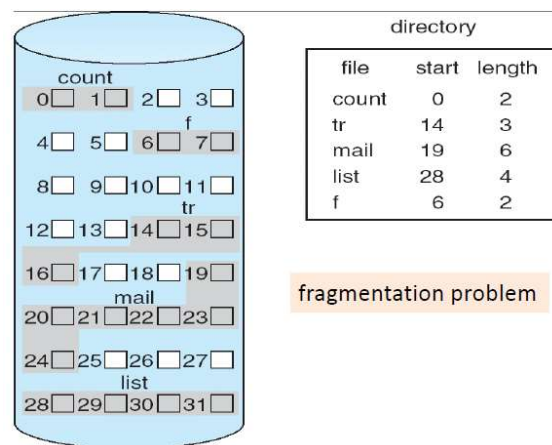
- Directories don't have data blocks, but in a similar fashion have directory blocks, which contain inode/filename pairs; i.e. the names of the files/directories in the directory, along with the inodes for each. Each directory contains entries for "." and ".." --- the current directory and its parent.

Allocation methods

- Memory is divided into pages, similarly disk is divided in block.
- An allocation method refers to how disk block are allocated for files so that disk space is utilized effectively and files can be accessed efficiently
 - Contiguous allocation
 - Linked allocation
 - Indexed allocation

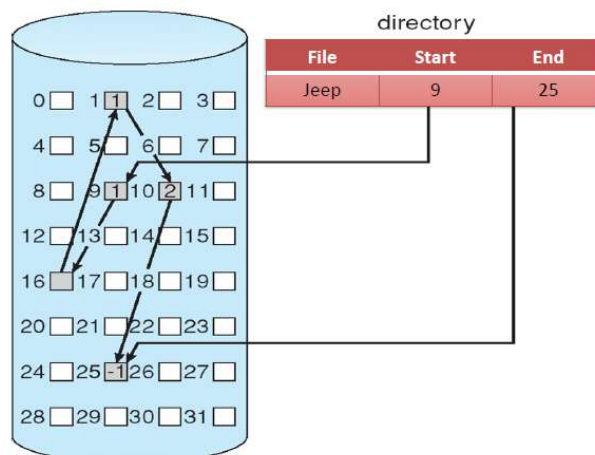
Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk.
- Simple, only starting location (block #) and length (number of blocks) are required.
- Random access
- Wasteful of space: dynamic storage-allocation problem
- Files may not grow.



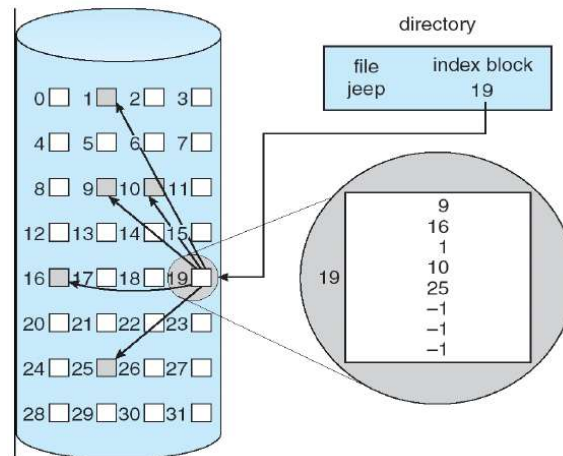
Linked Allocation

- Each file is a linked list of disk blocks. Blocks may be scattered anywhere on the disk
- Simple, need only starting address
- No random access
- Free-space management system: no waste of space



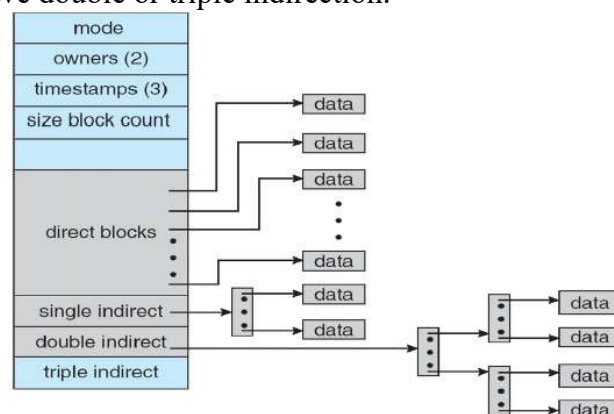
Indexed Allocation

- Brings all pointers together into the **index block**.
- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block

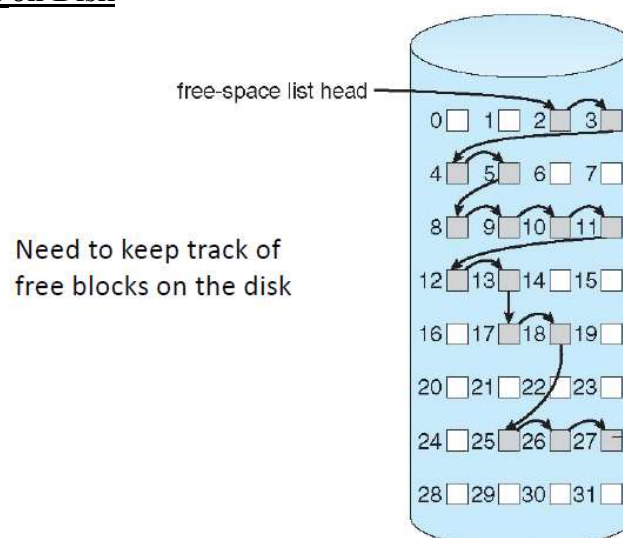


Combined Method: UNIX (4K bytes per block)

- Small files do not have an index block
- Large files can have double or triple indirection.



Linked Free Space List on Disk



Summary

- A file is an abstract data type defined and implemented by the operating system. It is a sequence of logical records. A logical record may be a byte, a line (of fixed or variable length), or a more complex data item. The operating system may specifically support various record types or may leave that support to the application program.
- A major task for the operating system is to map the logical file concept onto physical storage devices such as hard disk or NVM device. Since the physical record size of the device may not be the same as the logical record size, it may be necessary to order logical records into physical records. Again, this task may be supported by the operating system or left for the application program.
- Within a file system, it is useful to create directories to allow files to be organized. A single-level directory in a multiuser system causes naming problems, since each file must have a unique name. A two-level directory solves this problem by creating a separate directory for each user's files. The directory lists the files by name and includes the file's location on the disk, length, type, owner, time of creation, time of last use, and so on.
- The natural generalization of a two-level directory is a tree-structured directory. A tree-structured directory allows a user to create subdirectories to organize files. Acyclic-graph directory structures enable users to share subdirectories and files but complicate searching and deletion. A general graph structure allows complete flexibility in the sharing of files and directories but sometimes requires garbage collection to recover unused disk space.
- Remote file systems present challenges in reliability, performance, and security. Distributed information systems maintain user, host, and access information so that clients and servers can share state information to manage use and access.
- Since files are the main information-storage mechanism in most computer systems, file protection is needed on multiuser systems. Access to files can be controlled separately for each type of access—read, write, execute, append, delete, list directory, and so on. File protection can be provided by access lists, passwords, or other techniques.

Distributed File System (DFS)

Basic Features

- Highly fault-tolerant
- High throughput
- Suitable for application with large data sets
- Streaming access to file system data
- Can be built out of commodity hardware

Data characteristic

- Applications need streaming access to data
- Batch processing rather than interactive user access.
- High aggregate data bandwidth
- Scale to hundreds of nodes in a cluster
 - Tens of millions of files in a single instance
 - Large data sets and files: gigabytes to terabytes size
- Write-once-read-many: a file once created, written and closed need not be changed – this assumption simplifies coherency
- A map-reduce application or web-crawler application fits perfectly to this model.

Questions

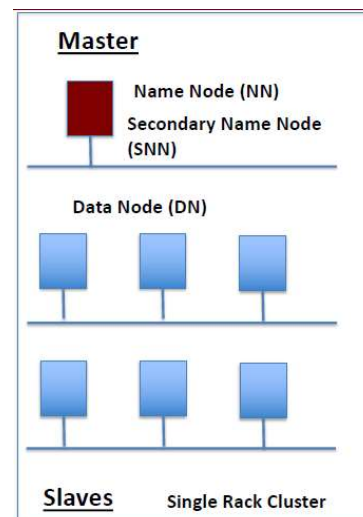
- **Problem 1:** Data is too big to store on one machine → Store the data on multiple machines
- **Problem 2:** Very high-end machines are too expensive → Run on cheap hardware
- **Problem 3:** Cheap hardware will fail → Software is intelligent enough to handle hardware failure
- **Problem 4:** What happens to the data if the machine stores the data fails? → Replicate data
- **Problem 5:** How can distributed machines organize the data in a coordinated way? → Master-Slave Architecture

Fault Tolerance

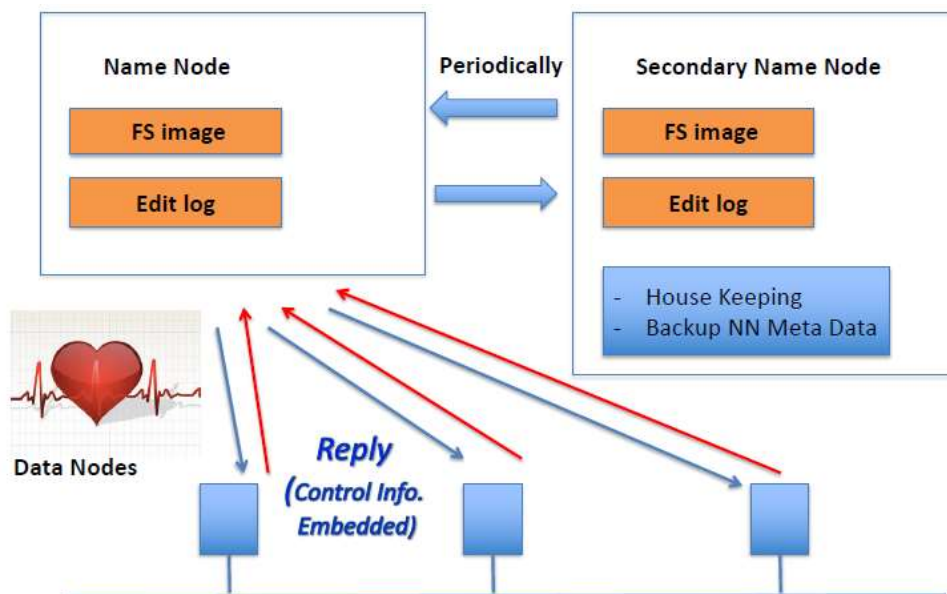
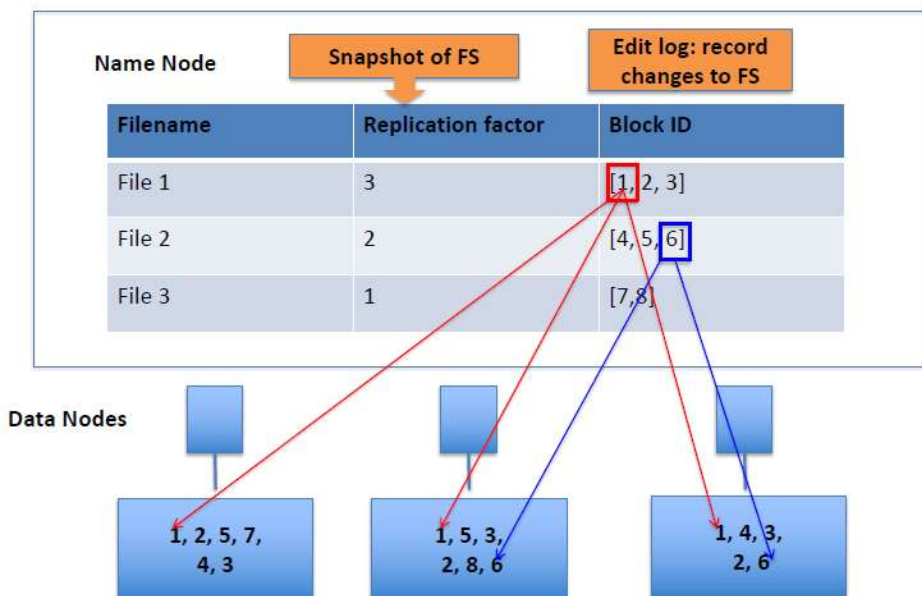
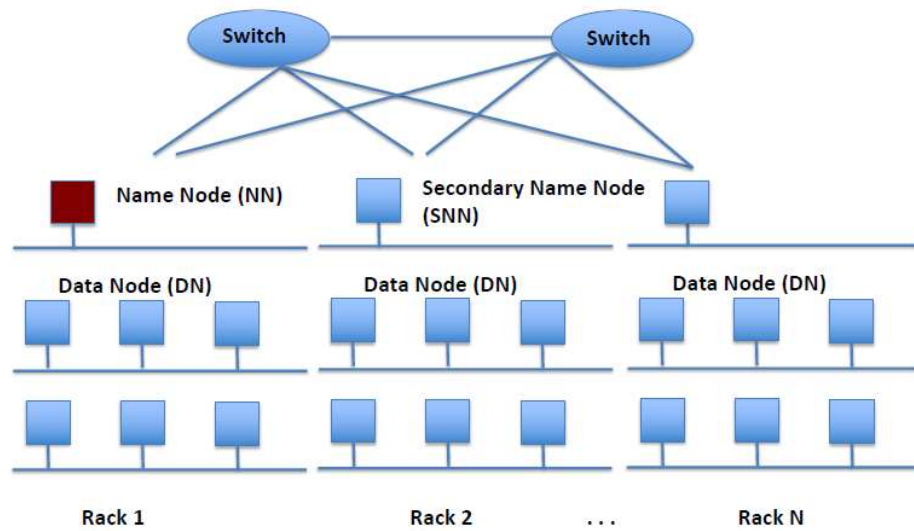
- Failure is the norm rather than exception
- An DFS instance may consist of thousands of server machines, each storing part of the file system's data.
- Since we have huge number of components, and that each component has non-trivial probability of failure means that there is always some component that is non-functional.
- Detection of faults and quick, automatic recovery from them is a core architectural goal of DFS.

Master-Slave Architecture

- Name Node: Controller
 - File system name space management
 - Block mappings
- Data Node: Work Horses
 - Block operations
 - replication
- Secondary Name Node
 - Checkpoint node



Multiple-Rack Cluster



Blocks

→ Why do we need the abstraction “Block” in addition to “Files”

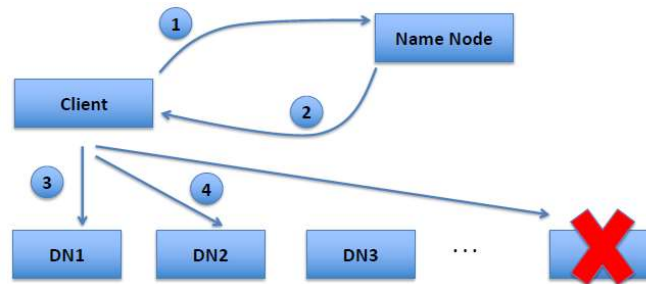
- Files can be larger than a single disk
- Block is of fixed size, easy to manage and manipulate
- Easy to replicate and do more fine-grained load balancing

→ HDFS Block size is by default 64 MB. Why it is much larger than regular file system block?

- Minimize overhead, disk seek time is almost constant

Read

1. Client connects to NN to read data
2. NN tells clients where to find the data blocks
3. Clients reads blocks directly from data node (without going through NN)
4. In case of node failures, client connects to another node that serves the missing block



→ Why does HDFS choose such a design for read? Why not ask client to read block through NN?

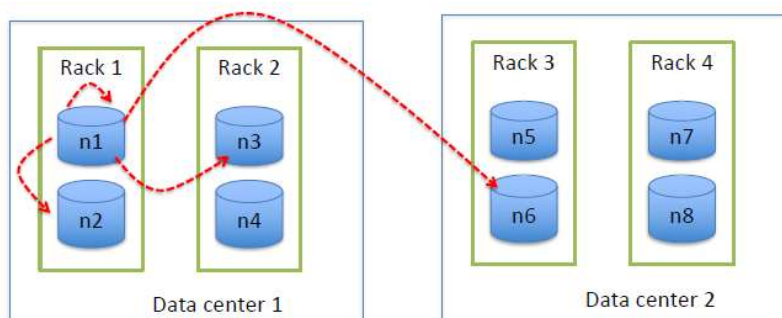
- Prevent NN from being the bottleneck of the cluster
- Allow HDFS to scale to large number of concurrent clients
- Spread the data traffic across the cluster

→ Given multiple replicas of the same block, how does NN decide which replica the client should read?

- Rack awareness based on network topology.

Network Topology

- The critical resource in HDFS is **bandwidth**, distance is defined based on that
- Measuring bandwidths between any pair of nodes is too complex and **does not scale**
- Basic Idea:
 - Processes on the same node
 - Different nodes on the same rack
 - Nodes on different racks in the same data center (cluster)
 - Nodes in different data centers
- HDFS takes a simple approach
 - See the network as a tree
 - **Distance between two nodes is the sum of their distances to their closest common ancestor**









Write

1. Client connects to NN to write data
2. NN tells client write these data nodes
3. Clients writes block directly to data nodes with desired replication factor
4. In case of node failures, NN will figure it out and replicate the missing blocks

➔ Where should HDFS out the three replicas of a block? What tradeoffs we need to consider.

- Reliability
- Write Bandwidth
- Read Bandwidth

	Reliability	Write Bandwidth	Read Bandwidth
Put all replicas on one node			
Put all replicas on different racks			
HDFS: 1-> same node as client 2-> a node on different rack 3-> a different node on the same rack as 2	