# OS Condensed Summary Notes For Quick In-Exam Strategic Fact Deployment

Maksymilian Mozolewski

April 29, 2021

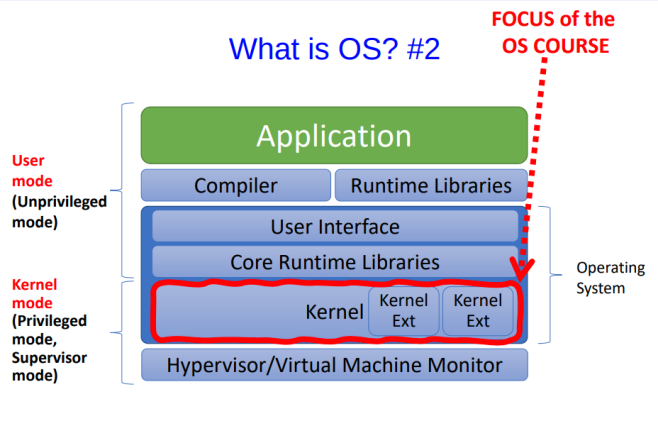
## Contents

# OS

## Introduction & Structure

### OS

An intermediary between the user of a computer and computer hardware. A program itself most intimately connected to the hardware. Everything you don't need to write in order to run your application. Library, all operations on I/O, syscalls. The OS can be an invisible intermediary




Main benefits of OS abstraction:

- Application benefits
  - programming simplicity - see high level abstraction (files) instead of low level hardware details (device registers)
  - abstractions are **reusable** across many platforms
  - **portability** (across machine configurations or architectures) - device independence: 3com or intel card?
- User benefits
  - safety: program sees its own virtual machine, can believe it owns the computer
  - OS **protects** programs from each other
  - OS **fairly multiplexes** resources across programs
  - efficiency - **share** one computer across many users. **Concurrent** execution of multiple programs

## Basic OS Concepts

**Multiprogramming** : ability to keep multiple programs running in parallel (almost). This is done by keeping all possible jobs in the **job pool** on the disk, when a job is ready to take its turn to execute, it's brought to main memory and run for a bit untill the OS decides to switch it for another (because of an interrupt) or it completes.

**Multitasking/timesharing** : while multiprogramming makes teh OS efficient, it does not faciliate for user interaction! For the user to be able to run multiple tasks simultaneously and have a smooth experience, the CPU needs to actually switch jobs much more frequently so as it appears that all of them are being processed simultaneously. A system which faciliates **timesharing** is **interactive** - it frequently awaits user input and has short response time. A time-shared system enables the system to be used by **multiple** users simultaneously.

**Process** : a program which is loaded in main memory. This may be a full on program, or a printer job. Processes may spawn sub-processes if they wish to using syscalls. This is the **unit of work in a system**.

**Job scheduling** : prioritising which jobs should be in main memory at any time.

**CPU scheduling** : prioritising which jobs in main memory should be executed first.

**Virtual memory** : a technique that allows each program to see the entire memory as theirs and not mess with other processes as well as running programs which require more memory than is **physically available**

**Logical memory** : memory as seen by the programmer, abstracted away from the nitty-gritty mechanical details of the OS.

**Interrupt driven** : if there's no demand for action from the OS, it will IDLE. Events are almost always signalled by the occurence of an interrupt or a trap

**Trap** : (exception) is a software generated interrupt caused by either an error or a syscall.

**Dual-mode/multimode operation** : in a multiprogramming system processes are run in parallel, hence protections must be put into place so that processes cannot impede other processes. Most commonly this is done via introduction of **user mode** and **kernel mode**. In user mode certain possibly harmful **priviliged instructions** are forbidden by the **hardware itself** which sends a trap signal (switch to kernel mode, timer management, I/O control)

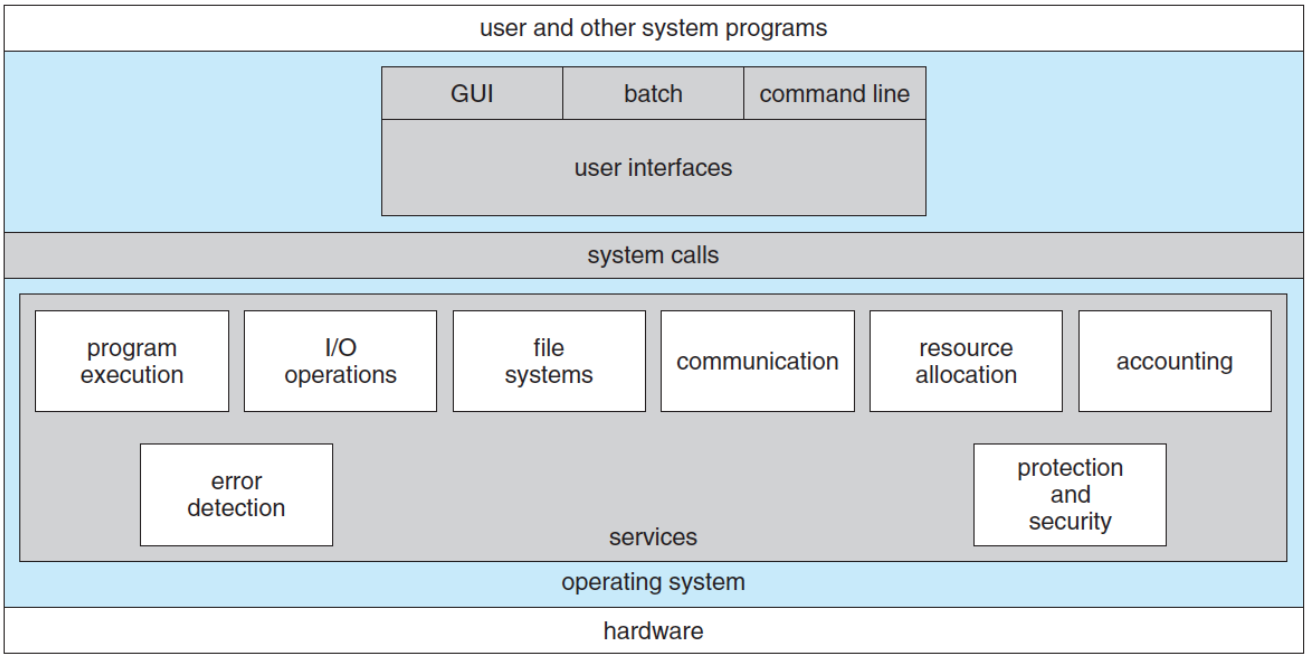
**Virtual machine manager** : virtual machine management software which can be set to run on a third mode (which requires more mode bits), giving it less power than the kernel but more than the user. Virtualisation does not necessarily require its own privilege level

**Mode bits** : reserved bits in the hardware which signify which mode we're in. Typically 0 = user mode and 1 = kernel mode. Always set

**before** passing control to the user program.

**File** : abstract memory concept which is mapped to physical storage space. Each file may contain absolutely any data. The OS is responsible for creating/removing files, creating removing directories to organise files, suporting primitives for manipulating files and directories, mapping files onto secondary storage, backing up files on stable (nonvolatile) storage media.

## OS Services




**User Interface** : can appear in many different forms, CLI, GUI or batch - where commands and directives run directly from files.

**Program execution** : the OS loads programs into memory and runs their instructions, then halts them.

**IO Operations** : interactions with external devices such as the keyboard or mouse.

**File-system manipulation** : search through files and directories, creation and deletion of files, permission management.

**Communications** : facilities for exchanging information between different processes on the same computer or via network. I.e. **Shared memory** or perhaps **message passing**.

**Error detection** : the OS needs to be aware of errors which occur and correct them as they appear. These can happen anywhere in the system, including hardware and software.

**Resource allocation** : distribution of resources available to different jobs and users at the same time efficiently.

**Accounting** : keeping track of who used what resources for either economic purposes or analytics.

**Protection and security** : all data needs to be securely stored and only available to the users who have the correct perissions. Several processes cannot interfere with each other or harm the system.

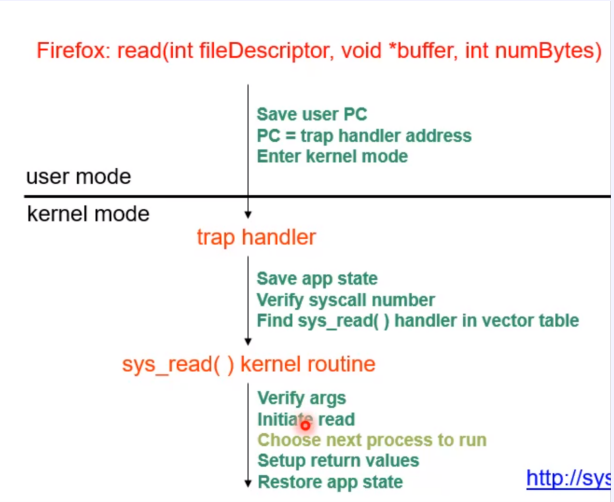
**Command interpreter** : allows the user to directly interface with the system via commands either in the form of a GUI or CLI or in other forms. When multiple are available these are shown as **shells**. The commands themselves may be stored by the shell, or the shell might simply direct the appropriate loading of file-stored directives which run the appropriate commands (i.e. PATH resolution)

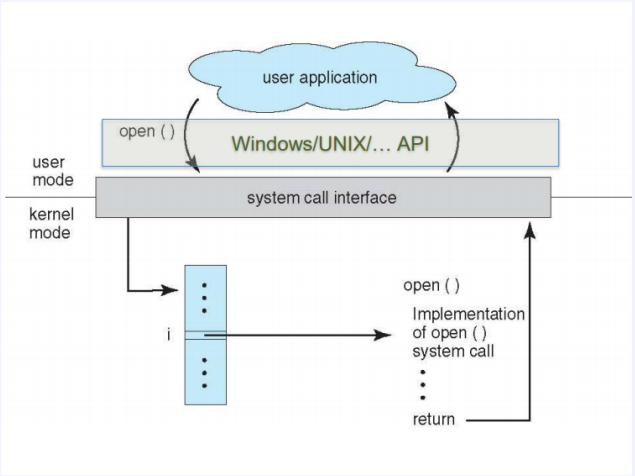
**System calls** : calls to the OS to develop certain specific functions such as opening files or starting sub-processes. Many OS use API's on top of system calls to make portability more achievable.

**System-call interface** : the interface provided by programming languages to interface with the system calls in different OS' which the compiler knows the specifics of.

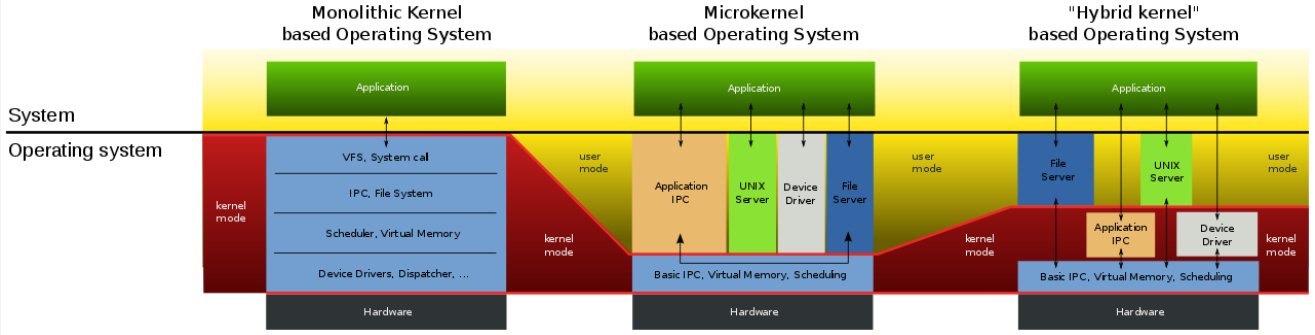
## Syscalls

The user cannot perform IO operations by himself, he must ask the OS to do it for them.
Syscalls define procedures which the OS performs for the user in privileged mode.
Usually implemented as vectors, where each syscall is simply an offeset to a base address.
Mechanically just a procedure call (but is not one! in a normal procedure call the caller knows the location of the procedure, in this case a syscall is just an ID), the caller puts arguments in a place the callee expects, then retrieves output from known place. Usually each system call will have wrapper functions provided by each programming language, i.e. the **system call interface**, which the compiler understands.



## OS Structures




### Monolithic Kernel

All major subsystems implemented in kernel.

+ low system interaction cost (procedure call)
- hard to understand and modify
- unrelible (no isolation between system modules)
- hard to maintain

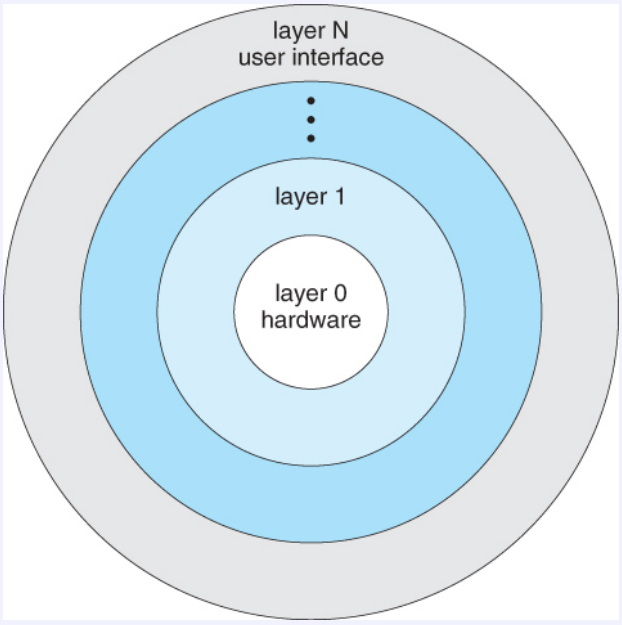
### Microkernel

Minimize what goes into kernel, implement rest of OS as user-level processes

+ better reliability (isolation between components)
+ ease of extension and customization (easy to replace parts)
- poor performance (a lot of user/kernel switches)

### Layered Kernel

Implement OS as a set of layers, each layer interacts only with layer below.




+ more reliable (separation of components)
- strict layering isn't flexible enough - in real life modules might need to communicate with not only nearby layers.
- poor performance, each layer crossing has overhead associated with it (due to API generalization)
- Disjunction between model and reality - system modelled as layers, but not really built that way

### Dynamically loadable kernel modules

Core services in the kernel, others dynamically loaded.

Common in modern implementations:

- **Monolithic**: load the code in kernel space (Solaris, Linux, etc.)
- **Microkernel**: load the code in user space (any)

+ Convenient: no need for rebooting for newly added modules
+ Efficient: no need for message passing unlike microkernel
+ Flexible: any module can call any other module unlike layered model
- Memory fragmentation: fragments OS memory which is normally unfragmented when loaded initially

### Hybrid OS Design

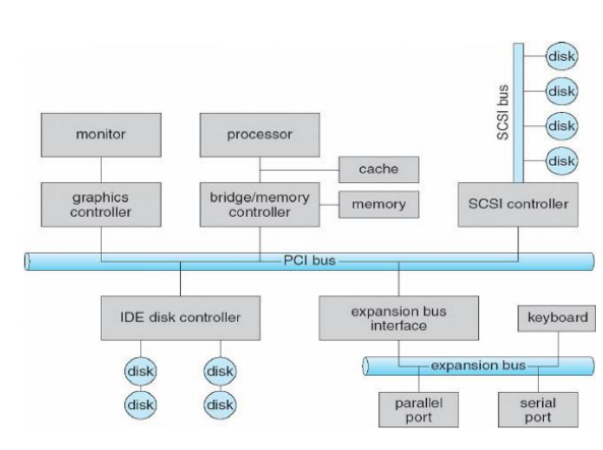
Many different approaches. Key idea: exploit the benefits of monolithic and microkernel designs. Extensibility via kernel modules.

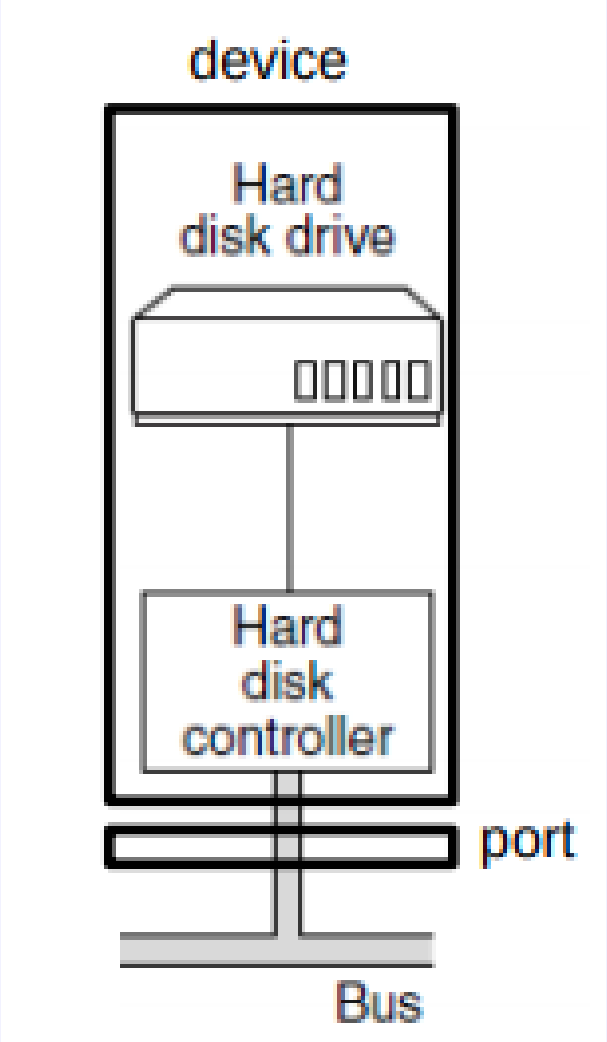
# IO

## I/O

Variety of I/O Devices

**Port** : Connection point for a device (e.g., USB, parallel, serial, ethernet)
**Bus** : Peripheral buses (e.g. PCI/PCIe), Expansion bus connects relatively slow devices
**Device**
**Controller(host adapter)** : electronics that operate port, bus, device (sometimes integrated). Contains processor, microcode, private memory, bus controller etc.



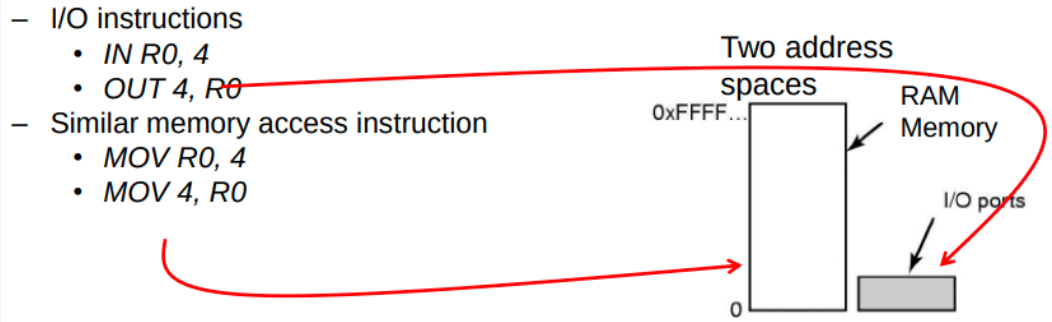



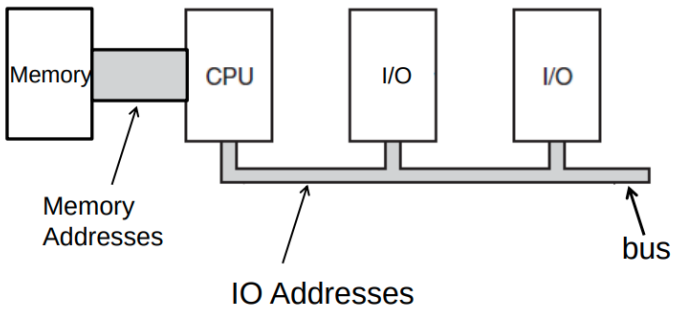

Controllers have **registers** for data and control as well as **buffers**, mostly for data. Communication Methods:

- **IO Ports**
- **Memory-mapped IO**
- Hybrid

## I/O Ports

- Each control register has an I/O port number
- Special instructions exist to access the I/O port space
- CPU reads in from device I/O PORT to CPU register (IN REG, PORT)
- CPU writes to device I/O PORT from CPU register (OUT PORT, REG)
- Instructions are privileged (OS kernel only)
- Seperate I/O Port space and memory space:

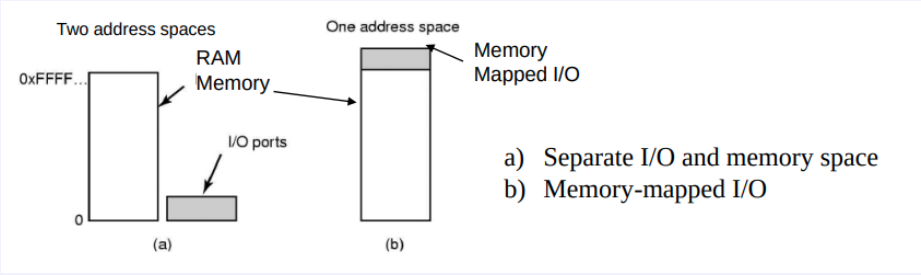


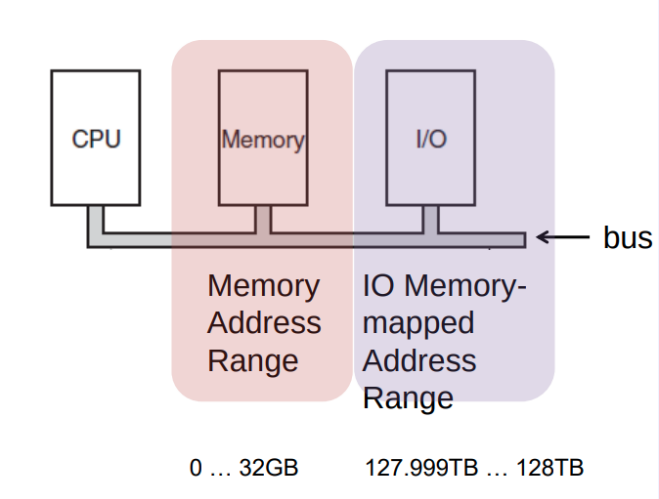




| I/O address range (hexadecimal) | device |
|---|---|
| 000–00F | DMA controller |
| 020–021 | interrupt controller |
| 040–043 | timer |
| 200–20F | game controller |
| 2F8–2FF | serial port (secondary) |
| 320–32F | hard-disk controller |

## Memory Mapped I/O

- All control registers and buffers mapped into the memory space
- Each control register is assigned a unique memory address
- There is no actual RAM memory for that address
- Such addresses may be at the top of the physical address space




## Hybrid I/O

Simply do both, use memory mapped I/O for the **data buffers**, and keep separate I/O ports for **control registers**.

## Offloaded Communication

The CPU can request data from an I/O controller one byte at at time (Programmed IO). This wastes CPU time for large data transfers, small data transfers are ok.
CPU can instead offload data transfers using DMA:

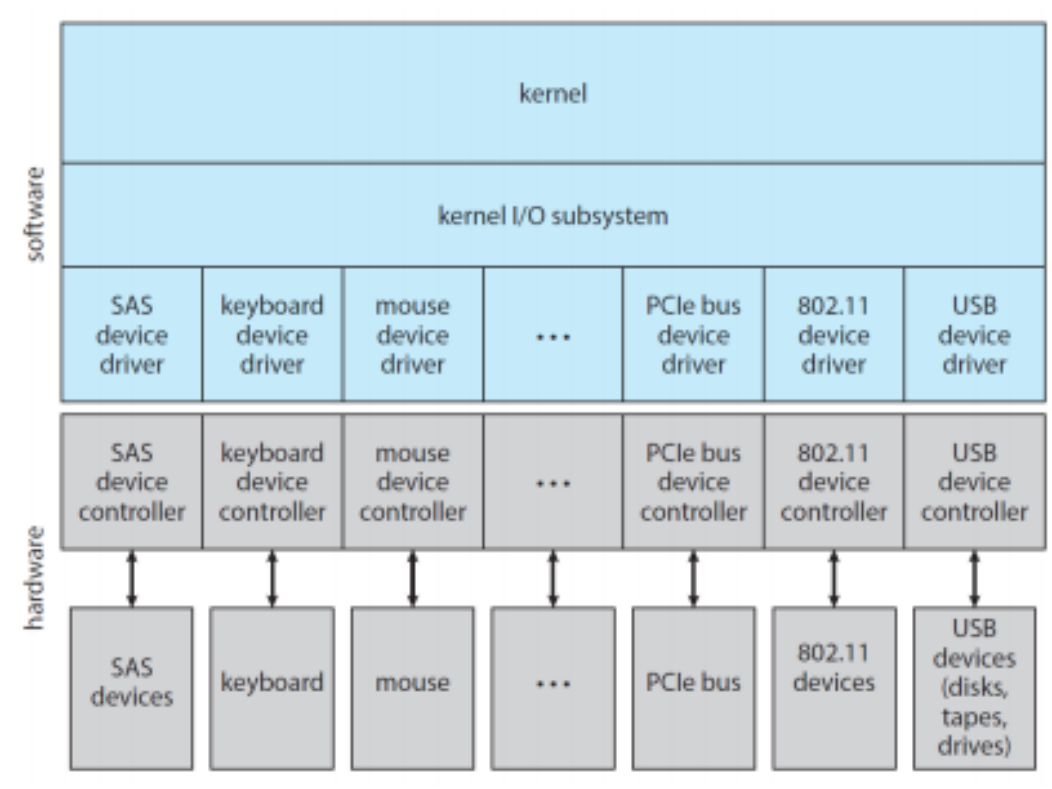
**DMA (Direct Memory Access) controller** transfers data fro the CPU, either from/to IO or between IO devices.

This requires a DMA controller either on the device's host controller, and the motherboard. This controller contains registers to be read/written by the software:

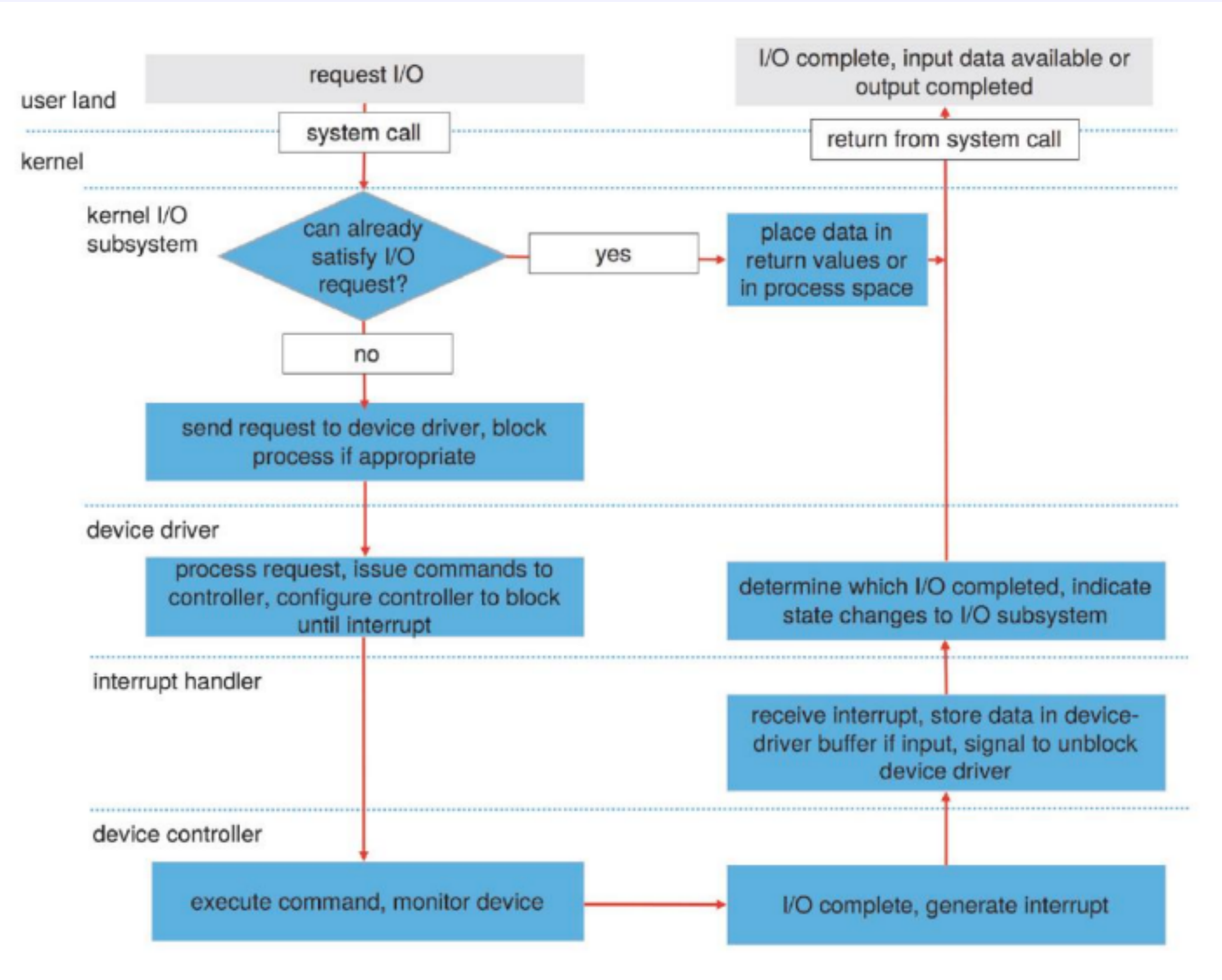
- Memory address register
- byte count register
- Control registers: direction, unit, byte burst size etc..

## OS Device drivers

Great variety of devices, each one has very different specs.
OS Deals with IO devices in a standard and uniform way. Each type of driver is an interface which the vendor can implement as a class. Each OS has its own standard.



**Life cycle of an IO request**




# Processes

## Process

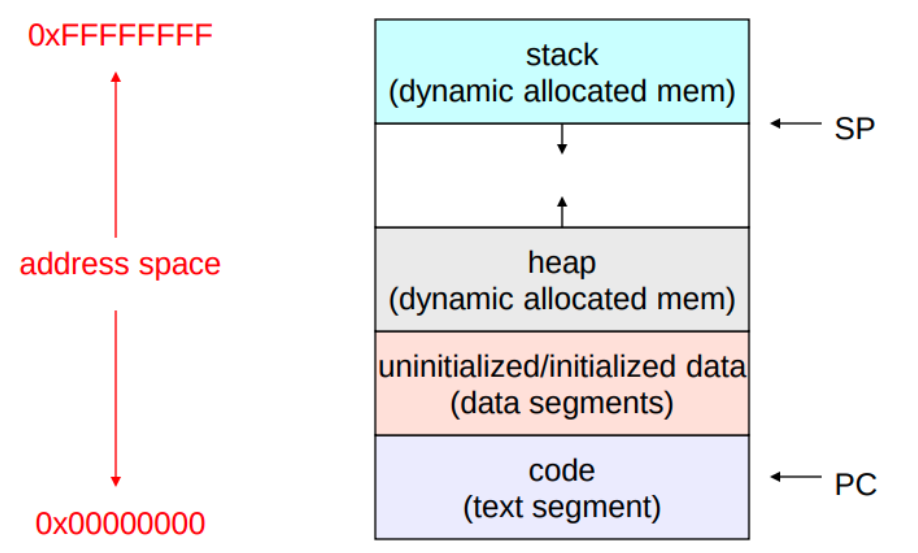
Process is the OS's abstraction for execution.

- Program is the list of instructions, initialized data, etc
- A process is a **program in execution**
- A single flow/sequence of instruction in execution
- An address space (an abstraction of the CPU)

Only one process can be running on a processor core at any instant
Contents:

- An address space, containing:
  - Code (instructions) for running program
  - Data for the running program (static data, heap data, stack)
- A CPU state, consisting of
  - Program counter, indicating the next instruction
  - Stack pointer, current stack position
  - Other general-purpose register values
- A set of OS resources
  - Open files
  - network connections
  - sounc channels

I.e. everything needed to run the program




Each process is identified by a process ID (**PID**). PID's are unique and global. With certain exceptions (cgroups)

Operations that create processes return a PID, and those which operate on processes accept PID's as arguments

### Process representation

Each process is represented internally by the OS with a **Process control block (PCB)** or process/task descriptor, identified by the PID.
OS keeps all of a process's execution state in (or linked from) the PCB when the process isn't running:

- PC, SP, registers etc.
- when a process execution is stopped, its state is transferred out of the hardware into the PCB

When the process is running its state is spread between the PCB and the hardware (CPU regs)
PCB's contain:

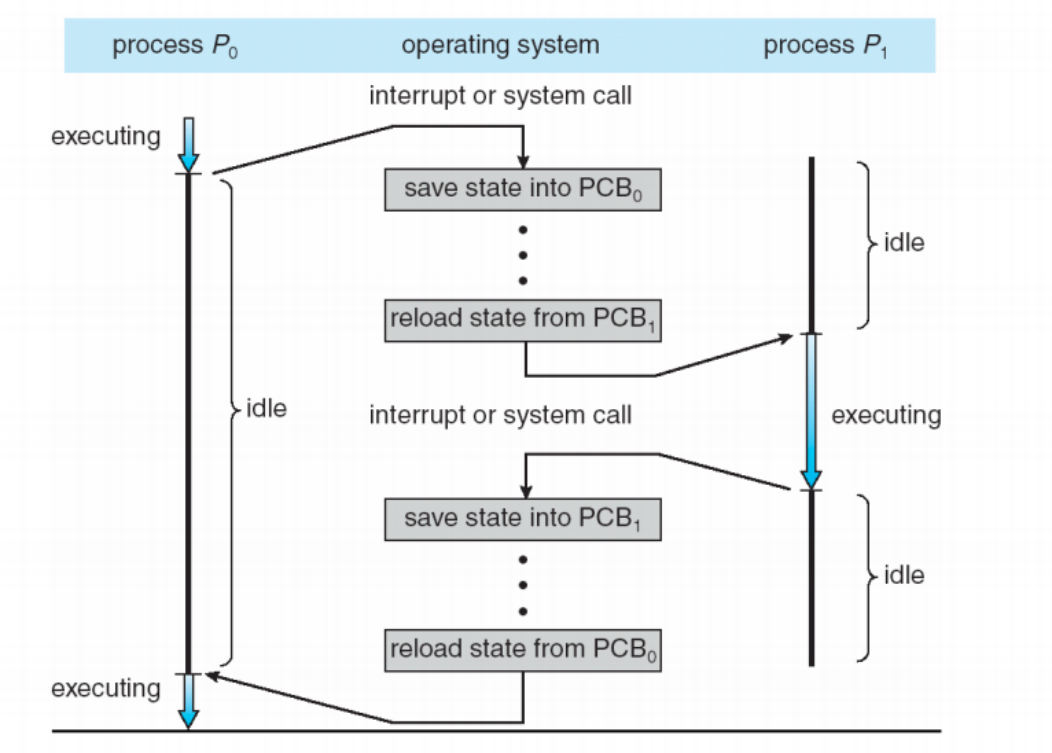
- Process ID (PID)
- Parent process ID
- Execution state
- PC, SP, registers
- Address space info
- UNIX user ID, group ID
- Scheduling priority
- Accounting info
- Pointers for state queues

and likely many more.
Whenever the OS gets control becase of :

- Syscalls
- Exceptions
- Interrupts

The OS then saves the CPU state into the PCB. Whenever the process is resumed into execution again, its PCB is loaded onto the machine registers SP, PC etc.
This is called a **Context switch**

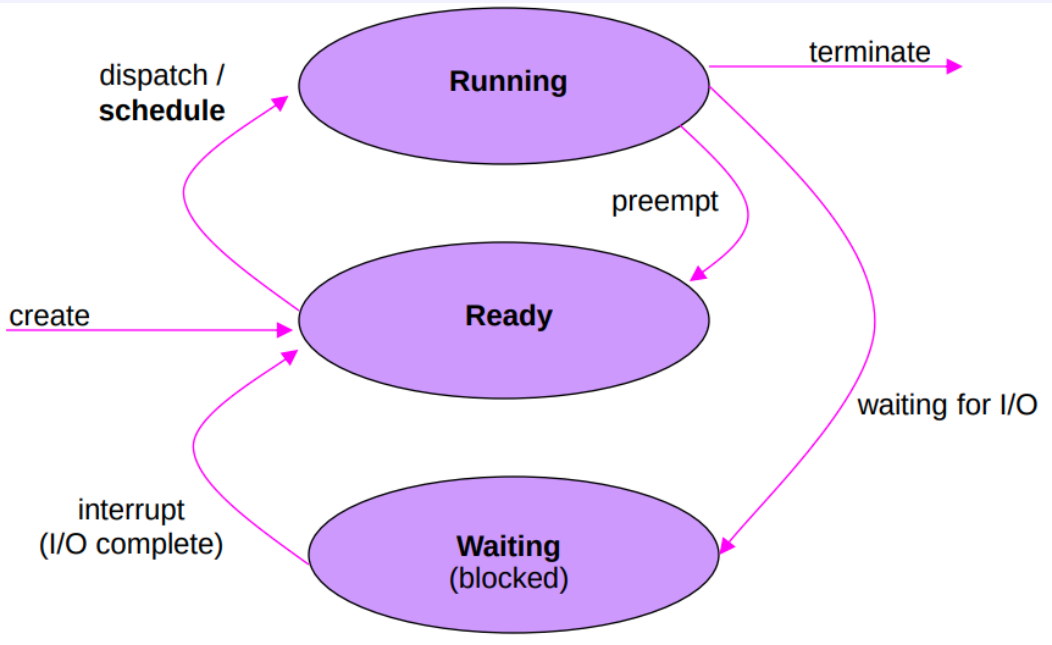
## Context Switch




## Execution states

Each process has an **Execution state**, which indicates what it's currently doing.

- **Ready**: waiting to be assigned to a CPU
- **Running**: executing on a CPU
- **Waiting**: Waiting for an event, e.g. IO completion, or a message from another process.



## State queues

The OS maintains a collection of queues, that represent the state of all processes in the system. Typically one queue for each state (executing, waiting etc..) Each PCB is queued onto a state queue according to the current state of the process it represents. As a process changes state, its PCB is unlinked from one queue, and then linked onto another.

There may be many wait queues, one for each type of wait (specific device, timer, message) etc.

## Process creation

New processes are created by existing processes (parent-child)
The first process is started by the OS, everything else stems from it (init in linux)

Depending on OS, child processes inherit certain attributes of parent, (i.e. open file table: implies stdin/stdout/stderr) Some systems divide resources of parent between children.

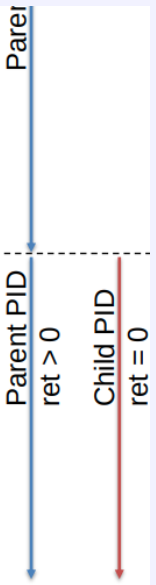
## UNIX - fork()

On UNIX systems, process creation is done through the fork() system call:

- Creates and initializes a new PCB
- Initializes kernel resources of new process with resources of parent (e.g. open files)
- Initializes PC,SP to be same as parent
- Creates a new address space, which is an identical copy of this of the parent's (by value)

The fork call returns "twice" once in the parent, and once in the child. In the child the PID returned is 0 and in the parent, the child's PID is returned.

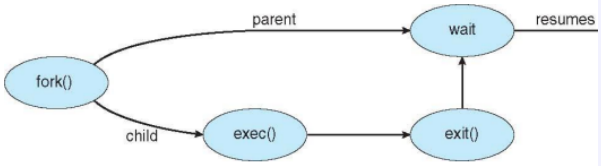
```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  char *name = argv[0];
  int ret = fork();
  if (ret < 0) { /* error */
    printf("Error\n");
    return 1;
  } else if (ret > 0) {/* parent */
    printf("Child of %s is %d\n", name, ret);
    return 0;
  } else {/* child */
    printf("My child is %d\n", ret);
    return 0;
  }
}
```



## UNIX - exec()

In order to start a new program instead of just copying the old one, we must use exec(). Which is the call which stops the current process, loads a new program into the address space, initializes the hardware context and args for the new program and finally places the PCB onto the ready queue

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

int main() {
  pid t pid;
  pid = fork(); /* fork a child process */
  if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
  }
  else if (pid == 0) { /* child process */
    execlp("/bin/ls","ls",NULL);
  }
  else { /* parent process */
    wait(NULL); /* parent waits for the child to complete */
    printf("Child Complete");
  }
  return 0;
}
```

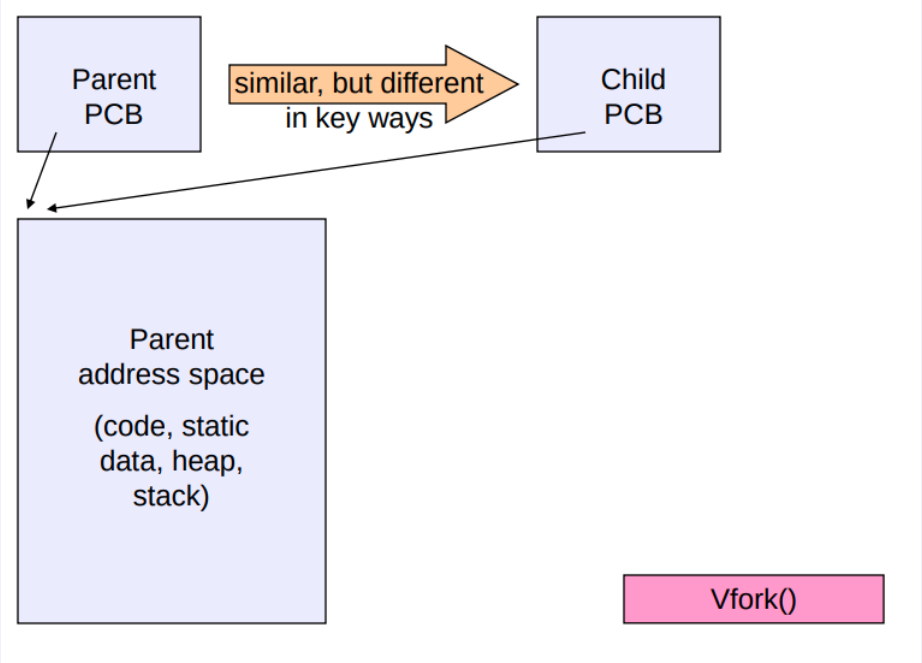



28

Alternatively use vfork() which is faster but less safe

## UNIX - vfork()

Same as fork, but hte child's address space **IS** by address the same space as the parents.
Usage relies on the child not modifying the address space before doing an execve() call, otherwise bad things can happen.



### Copy-on-write (COW) fork()

Retains original semantics, but copies "only what is necessary" rather than the entire address space.
On fork():

- create a new address space
- initialize page tables with same mappings as parents (identical)
- Set both parent and child page tables to make all pages read-only
- if either parent or child writes to memory, an exception occurs
- On exception, OS copies the page, adjusts page tables, etc..

# Interprocess communication

### Shared memory

Allow processes to communicate and synchronize:

- Sharing part of address space
- OS doesn't mediate communication (no overhead)
- Usually OS prevents processes from accessing each other's memory
- Processes should agree to void this restriction

Data:

- Format decided by application
- Direct access (not mediated by the OS) - very fast
- Application programmer fully manages the data transfer - not trivial

Possible use cases:

- Passing of large (single) objects (image)
- Notification variable

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
int main() {
  const int SIZE = 4096; /* the size (bytes) of shared memory object */
  const char *name = "OS"; /* name of the shared memory object */
  const char *message 0 = "Hello"; /* written to shared memory */
  const char *message 1 = "World!"; /* written to shared memory */
  int fd; /* shared memory file descriptor */
  char *ptr; /* pointer to shared memory obect */

  /* create the shared memory object */
  fd = shm_open(name,O CREAT | O RDWR,0666);
  /* configure the size of the shared memory object */
  ftruncate(fd, SIZE);
  /* memory map the shared memory object */
  ptr = (char *)
    mmap(0, SIZE, PROT READ | PROT WRITE, MAP SHARED, fd, 0);
  /* write to the shared memory object */
  sprintf(ptr,"%s",message 0);
  ptr += strlen(message 0);
  sprintf(ptr,"%s",message 1);
  ptr += strlen(message 1);

  return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main() {
  const int SIZE = 4096; /* the size (bytes) of shared memory object */
  const char *name = "OS"; /* name of the shared memory object */


  int fd; /* shared memory file descriptor */
  char *ptr; /* pointer to shared memory obect */

  /* open the shared memory object */
  fd = shm_open(name, O RDONLY, 0666);
  /* memory map the shared memory object */
  ptr = (char *)
    mmap(0, SIZE, PROT READ | PROT WRITE, MAP SHARED, fd, 0);
  /* read from the shared memory object */
  printf("%s",(char *)ptr);
  /* remove the shared memory object */
  shm_unlink(name);

  return 0;
}
```

5

## Message Passing

Allow processes to communicate and synchronize:

- Without sharing part of address space
- OS mediates communication (overhead likely)

Works with processes on the same machine and also those on different inter-networked machines! This is not possible with shared memory.
Message passing facility provides at least two operations:

- Send(message)
- Receive(message)

Communication link:

- Several implementation tradeoffs, .e.g. messages size
- Fixed
- Variable

Communicating processes must refer to each other:

- Direct communication:
  - Symmetric: explicit name of sender and receiver
    - send(P, message) - send message to P
    - receive(P, message) - receive message from P
  - Assymetric: Explicit at least on one end:
    - send(P, message) - send to p
    - receive(id, message) - receive from any process, sender saved in id
- Indirect communication:
  - No need to know/explicitly in advance sender and/or receiver
  - Mailboxes (e.g., POSIX mailbox)
    - send(A, message) - send message to mailbox A
    - receive(A, message) - receive message from mailbox A
  - A mailbox can be accesed by more than two processes
  - Multiple mailboxes might exist between processes

send() and receive() calls might be implemented as **blocking** or **synchronous** as well as **nonblocking** or **asynchronous**. different combinations of these might be offered:

- Blocking send
- Nonblocking send
- Blocking receive
- Nonblocking receive

**Rendezvous** - When both send and receive are blocking

## Buffering

Messages exchanged reside in temporary buffers/queues with either:

- Zero capacity (no buffering) - no message waiting, sender must block until recipient receives message
- Bounded capacity - n messages may reside. If the queue is not full, then nonblocking, otherwise blocking
- Unbounded - never blocks

**Example implementation: Pipes**

A pipe acts as a conduit allowing two processes to communicate **one-way** only and either **Annonymously** or **Named**.

Example of P

```c
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define READ_END 0
#define WRITE_END 1
int main(void) {
   char write msg[256] = "Greetings";
   char read msg[256];
   int fd[2];
   pid t pid;

   if (pipe(fd) == -1) {/* create the pipe */
      fprintf(stderr,"Pipe failed");
      return 1;
   }
   pid = fork(); /* fork a child process */
   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   if (pid > 0) { /* parent process */
      close(fd[READ_END]); /* close the unused end of the pipe */
      write(fd[WRITE_END], write msg, strlen(write msg)+1); /* write to the pipe */
      close(fd[WRITE_END]); /* close the write end of the pipe */
   }
   else { /* child process */
      close(fd[WRITE_END]); /* close the unused end of the pipe */
      read(fd[READ_END], read msg, 256); /* read from the pipe */
      printf("read %s",read msg);
      close(fd[READ_END]); /* close the read end of the pipe */
   }
   return 0;
}
```

Mechanically act like file descriptors (streams)

## Client-Server communication

- **Sockets** abstraction
  - endpoint for communication
  - identified by an IP address concatenated with a port number
  - servers implementing specific services (SSH, FTP, HTTP) listen to well-known ports
  - an SSH server listens to port 22, an FTP server listens to port 21, a web or http server listens to port 80
- **Remove procedure call** (RPC)
  - Abstract the procedure-call mechanism
  - for use between systems with network connections
  - similar in many respects ot the IPC
  - uses message-based communication to provide remote service

## Signals

- OS mechanism to notify a process (one way)
- From the OS POV can be thought as a software-generated interruption/exception (synchronous or asynchronous)
- From other processes POV, it is only a notification, no data is transferred. A communication method for: management, synchronization etc.

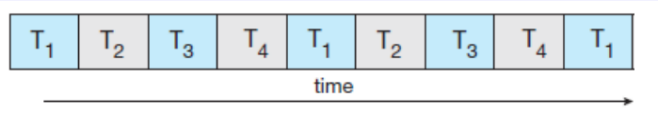
UNIX: signal handlers must be registered, and provide code for handling the signal.

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#define SIG_STOP_CHILD SIGUSR1

main() {
  pid_t pid;
  sigset_t newmask, oldmask;

  if ((pid = fork()) == 0) { /* Child */
     struct sigaction action;
     void catchit();
     sigemptyset(&newmask);
     sigaddset(&newmask, SIG_STOP_CHILD);
     sigprocmask(SIG_BLOCK, &newmask, &oldmask);
     action.sa_flags = 0;
     action.sa_handler = catchit;
     if (sigaction(SIG_STOP_CHILD, &action, NULL) == -1) {
        perror("sigusr: sigaction");
        _exit(1);
     }
     sigsuspend(&oldmask);
  }
  else {  /* Parent */
     int stat;
     sleep(10);
     kill(pid, SIG_STOP_CHILD);
     pid = wait(&stat);
     printf("Child exit status = %d\n", WEXITSTATUS(stat));
     _exit(0);
  }
}
void catchit(int signo) { /* Signal Handler */
     printf("Signal %d received from parent\n", signo);
     _exit(0);
}
```

# Threads

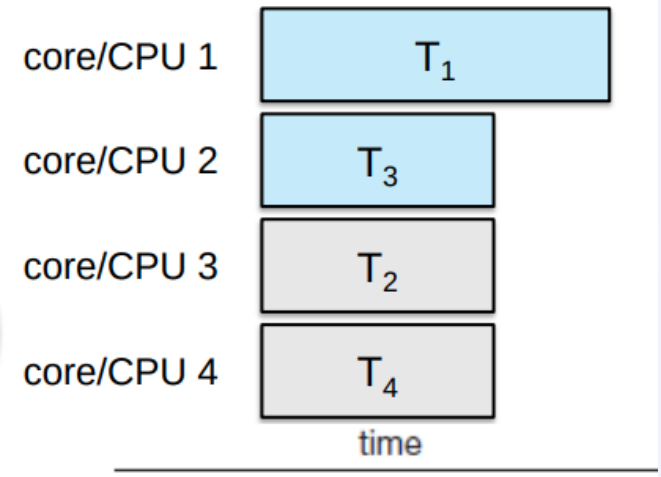
## Concurrency

Concurrency is carrying out multiple tasks in parallel but only one at the same time instance:




## Parallelism

Parallelism is carrying out multiple tasks in parallel at the same time instance:




## Communication problems

Multiple processes are required for both concurrency and parallelism, this requires communication. But the methods discussed thus far have limited usability.

Message passing:

- slow, OS mediates

Shared Memory:

- limited shareability, not all pointers work (both processes have different virtual memory layouts)
- OS resources not shared by default - cumbersome

Possible solution:

1. Fork several processes
2. cause each of them to map to the same shared memory (shmget())
3. make them open the same OS resources

Couple problems:

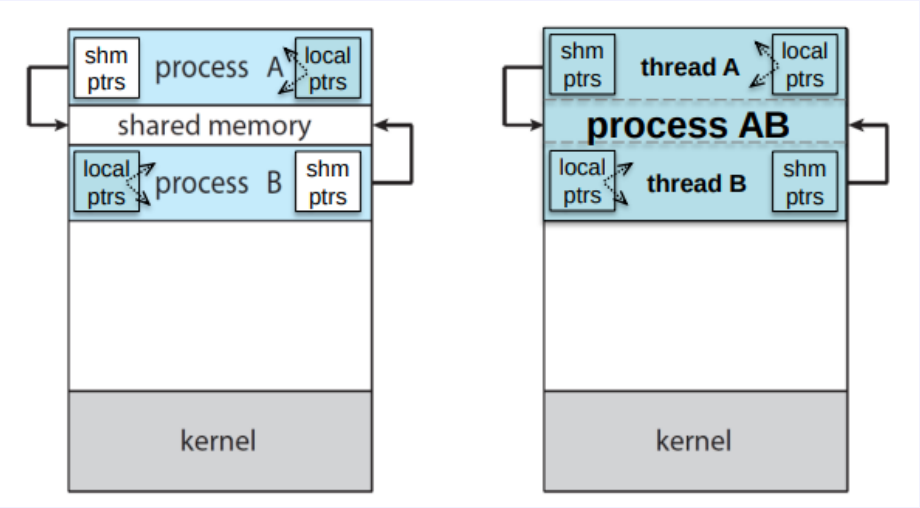
- cumbersome
- has limited shareability again
- inefficient - takes a long time to create all this, requires a PCB per process etc..

## Threads

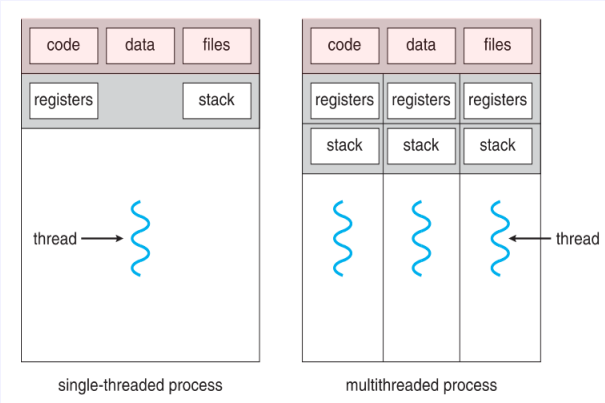
Threads are a great solution to the communication problem: Instead of spawning a new process per task, use **threads**.

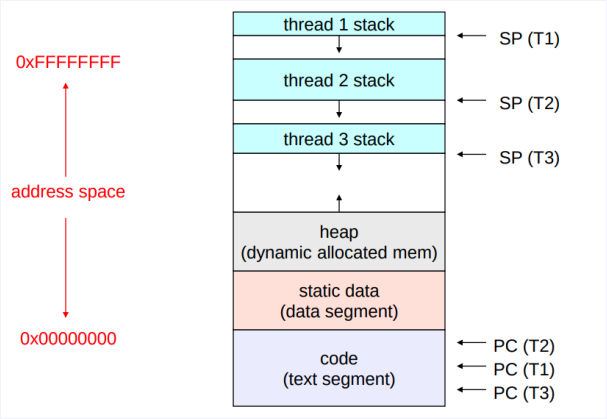
Each thread is part of the same spawning process, shares **address space & OS resources**.

Threads only differ in their **execution state (instruction flow)** i.e. private stack, and CPU state




A thread abstracts the execution state away from a process, and now a process represents the static parts of a task (address space, OS resources etc)




Threads become the **unit of scheduling** (depending on implementation of course).
Think processes are boxes for threads in which they execute.

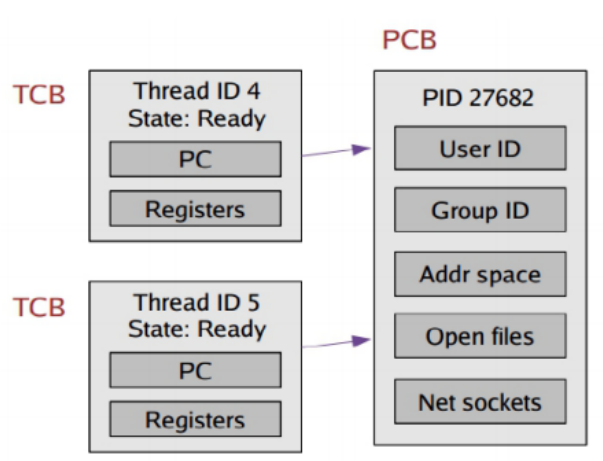
## Thread Control Block - TCB

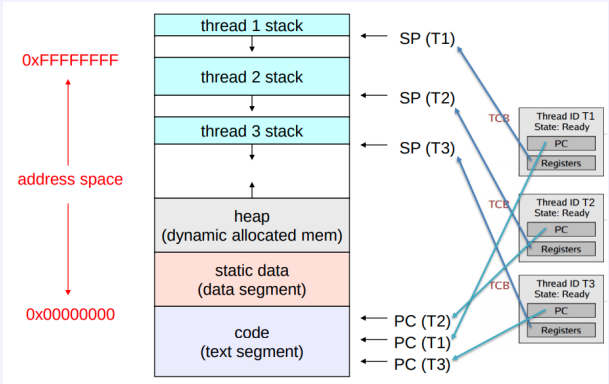
On the OS side, the PCB need to be adjusted to accomodate threads, easiest way is to create sub blocks for each thread representing execution state:
TCB contains:

- Program counter
- CPU registers
- Scheduling information
- Pending I/O information

PCB stores:

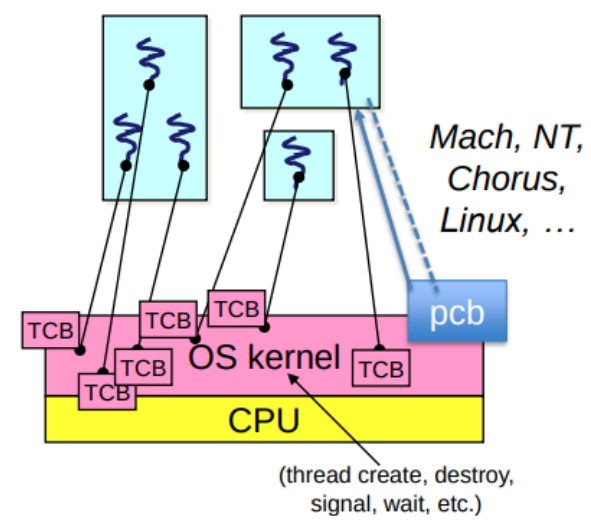
- Memory management information
- Accounting information

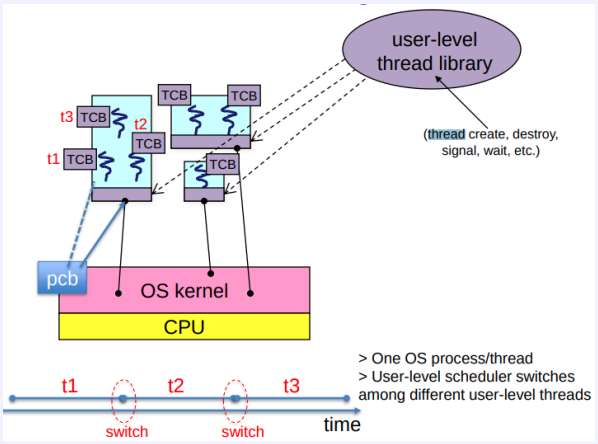


## User vs Kernel level threading

Threading can be either implemented as part of the OS, or as a library in the user space




### Kernel level threading 1:1

- OS allocates and manages threads
- TID's are used to identify threads

\+ if one thread blocks, the OS can run other threads within the same process

\+ possiblity to efficiently overlap IO and CPU time within a process

\+ Threads are cheaper than processes - less state to manage

\- pretty expensive for fine-grained use

  - Orders of magnitute more expensive than procedure calls
  - thread operations are syscalls (context switches + argument checks)
  - Must maintain kernel state for each thread

### User Level threading 1:N

- Threads managed at the user level, within the process
- A library in the program manages the threads
- the thread manager doesnt need to manipulate address spaces (Only OS can)
- Threads differ only in hardware contexts (PC,SP,registers), which can be manipulated by user-level code
- The thread package multiplexes user-level threads in a process
- TID's are now unique per process not globally
- No context switching between thread operations, these are done via procedure calls now (10-100x times faster than kernel threads)

\- if one thread tries to do IO, the whole process is blocked!

### N:M Threading

Best of both worlds, can start OS level threads for threads which will use IO.

# Scheduling

### Dispatcher

Mechanism used to switch between tasks (save and restore state)

### Scheduler

Decides on policy (implemented by scheduling algorithms) for ordering excution of tasks (threads/processes)

### CPU Bursts

Bursts of CPU processing done by a task. **Application dependent**

### IO Bursts

Similar to CPU Burst but for IO operations

### Performance goals

- CPU Utilization
- Throughput (processes completed per unit time)
- Turnaround time (time from submission of task to completion)
- Waiting time (all periods spent waiting in the ready queue from submission)
- Response time (time from submission of request to when response produced)
- Energy (joules per instruction) subject to some constraint (fps)

In most cases we optimize the **average metric**. Goals may be conflicting

### Fairness

No single compelling definition of fair for process resource allocation.

Sometimes goal is to be unfair and prioritize some classes of requests higher.

We want to avoid starvation - everyone needs at least some service

### Classes of schedulers

- Batch - throughput / utilization oriented
- Interactive - response time oriented
- Real time - deadline driven

### Preemptive scheduling

**Non-preemptive** scheduling:

- Processes/threads execute until completion or until they want
- The scheduler gets involved only at exit or on request

**Preemptive** scheduling:

- While a process/thread executes, its execution may be paused, and another process/thread resumes its eecution
- Involountary process switch

# Scheduling algorithms

## First-come First-served (FCFS)

Processes/tasks served in the order they arrive:

| Process | CPU time | Turnaround time |
|---|---|---|
| P1 | 24 | 24 |
| P2 | 3 | 24 + 3 = 27 |
| P3 | 3 | 24 + 3 + 3 = 30 |

Execution order: P1,P2,P3

Avg. Turnaround time: $\frac{24+27+30}{3} = 27$

- Non pre-emptive
- Poor average response time
- poor utilisation of **other resources** - a CPU-intensive job prevents I/O- intensive job from tiny bit of computaion on the CPU before returning to IO and keeping disk busy

## Shortest Job First (SJF)

Associate with each process the length of its CPU time Sort jobs, shortest CPU time goes first Can be preemptive (simply re-sort including the current process - Shortest Remaining Time Next, SRTN)

### Non-Preemptive

| Process | Arrival time | CPU time | Turnaround time |
|---|---|---|---|
| P1 | 0 | 7 | 7 |
| P2 | 2 | 4 | 12 - 2 = 10 |
| P3 | 4 | 1 | 8 - 4 = 4 |
| P4 | 5 | 4 | 16 - 5 = 11 |

Execution order: P1,P3,P2,P4

Avg. Turnaround time: $\frac{7+10+4+11}{4} = 8$

### Preemptive

| Process | Arrival time | CPU time | Turnaround time |
|---|---|---|---|
| P1 | 0 | 7 | 16 |
| P2 | 2 | 4 | 7 - 2 = 5 |
| P3 | 4 | 1 | 5 - 4 = 1 |
| P4 | 5 | 4 | 11 - 5 = 6 |

Execution order: P1 (2s),P2 (2s),P3(1s),P2(2s),P4(4s),P1(5s)

Avg. Turnaround time: $\frac{16+5+1+6}{4} = 7$

\+ Preemptive is optimal

- Too complex, to be implemented in practice
- not always possible to determine the CPU/IO burst lenghts

## Round-robin (RR)

Processes run in discrete time slots, after each time slot a new process/task is chosen to be run

Time quantum = 20

| Process | CPU time | Turnaround time |
|---|---|---|
| P1 | 53 | 125 - 0 = 125 |
| P2 | 8 | 28 - 0 = 28 |
| P3 | 68 | 153 - 0 = 153 |
| P4 | 24 | 112 - 0 = 112 |

Execution order: P1(20s),P2(8s),P3(20s),P4(20s),P1(20s),P3(20s),P4(4s),P1(13s),P3(20s),P3(8s)

Avg. Turnaround time: $\frac{125+28+153+112}{4} = 104.5$

Long time quanta cause poor response times with a lot of processes, a too low time quanta, causes a lot of context switching loss.

+ Solves fairness and starvation
+ Fair allocation of CPU across jobs
+ Low average waiting time when job lengths vary
+ Good for responsiveness (interactivity) if small number of jobs
- Context switching time may add up for long jobs

## Priority (PRIO)

Always execute highest-priority runnable jobs to completion

| Process | CPU time | Priority | Turnaround time |
|---|---|---|---|
| P1 | 10 | 3 | 16 |
| P2 | 1 | 1 | 1 |
| P3 | 2 | 4 | 18 |
| P4 | 1 | 5 | 19 |
| P5 | 5 | 2 | 6 |

Execution order: P2,P5,P1,P3,P4

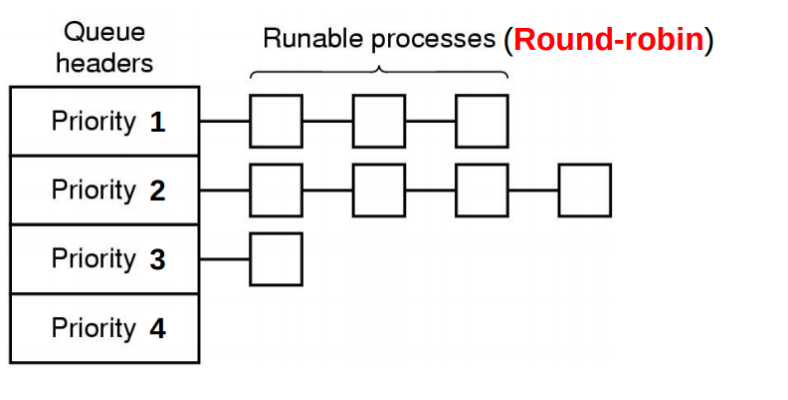
Avg. Turnaround time: $\frac{16+1+18+19+6}{5} = 12$

How to assign priorities ? Based on process type, User, price paid etc. or dynamically, based on how long the process ran etc..

- Starvation - lower priority jobs dont get to run because higher priority always running
- deadlock - priority inversion - happens when a low priority task has lock needed by high priority task (busy waiting)

## Multiple Queues(MQ)

Multiple round-robin scheduled queues, with queues of higher priority always scheduled first




Time quantum = 2

| Process | CPU time | Priority | Turnaround time |
|---|---|---|---|
| P1 | 10 | 3 | 19 |
| P2 | 1 | 1 | 1 |
| P3 | 2 | 3 | 10 |
| P4 | 1 | 3 | 11 |
| P5 | 5 | 2 | 6 |

Execution order: P2(1s),P5(5s),P1(2s),P3(2s),P4(1s),P1(8s)

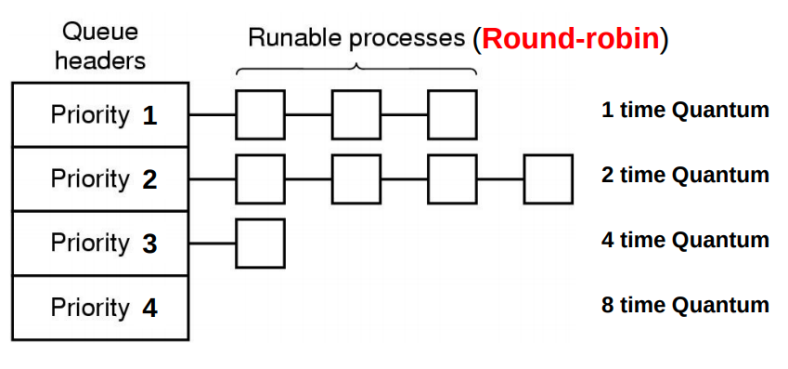
Avg. Turnaround time: $\frac{19+1+10+11+6}{5} = 9.4$

How to assign priorities ? Based on process type, User, price paid etc. or dynamically, based on how long the process ran etc..

- Starvation - lower priority jobs dont get to run because higher priority always running
- deadlock - priority inversion - happens when a low priority task has lock needed by high priority task (busy waiting)

### Multilevel Feedback Queue(MLFQ)

Same as MQ but each queue has a different time quanta. Time quanta are increasing inversely with priority of queue (higher priority lower time quanta). Each process starts in queue 1 - but when it exceeds its quanta it's pushed lower in the queues. When a process becomes inactive it is moved to a higher priority. This can be gamed by making a process interactive.




Time quanta = (1,2,4,8,16,32,64)

| Process | CPU time | Turnaround time |
|---|---|---|
| P1 | 100 | 102 |
| P2 | 2 | 3 |

Execution order: P1(1s),P2(1s),P1(2s),P2(1s),P1(4s),P1(16s),P1(37s)

Avg. Turnaround time: $\frac{102+3}{2} = 52.5$

8 context switches vs 101 with fixed quanta

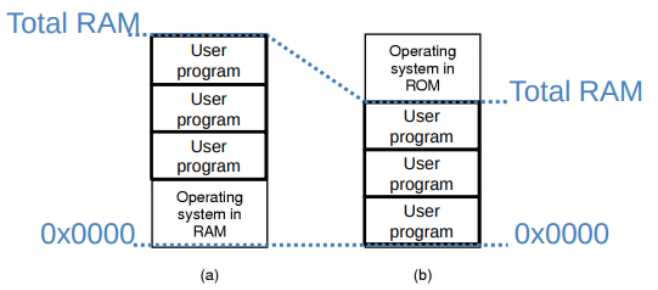
- Starvation - lower priority jobs dont get to run because higher priority always running
- deadlock - priority inversion - happens when a low priority task has lock needed by high priority task (busy waiting)

# Memory

### Why use abstraction

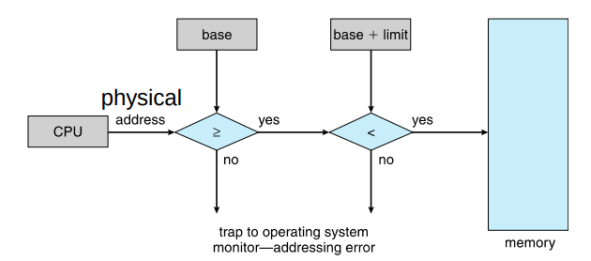
In the case that a program sees the physical memory directly, the program can mess with the OS and BIOS code, whether intentionally or unintentionally. There is a need for a protection system.

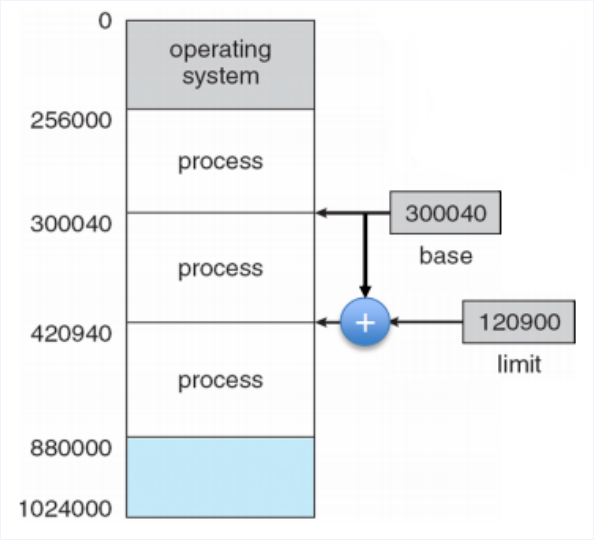
The situation gets worse when multiple programs are running with access to direct physical memory. In which case each program can alter all the other running programs. The total number of programs that can be run is then limited by the memory size!



## Protection - Base and Limit registers

A simple solution to prevent bad memory accesses, is to use base and limit registers.
This requires hardware support, but is enough to allow multiple **relocatable** programs to be run on the same physical address space.

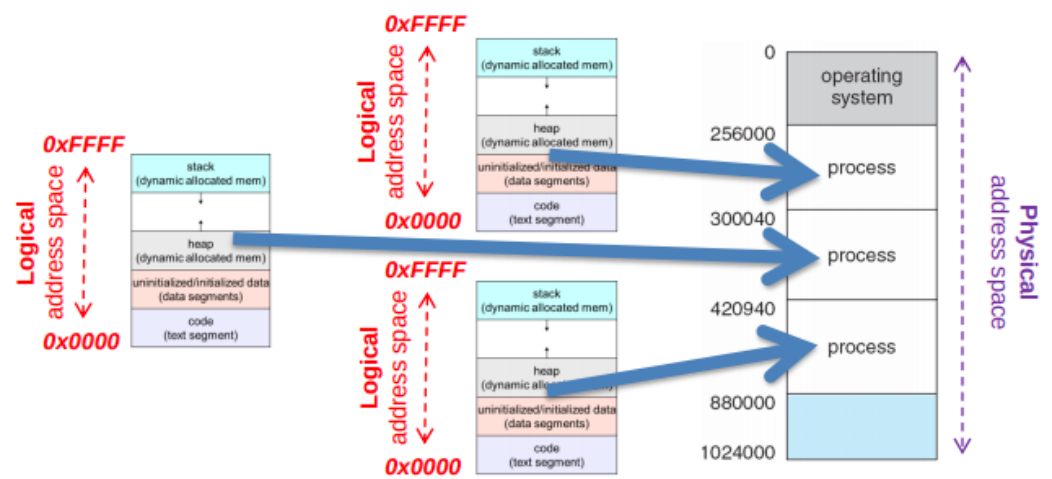



## Address Space

Abstraction from physical memory space. Defines the set of memory addresses that a process can use independently from other processes.

To make it easier to manage memory of multiple processes, we can make processes use **logical addresses** instead of physical ones!
These logical addresses are independent of physical addresses, yet the data lives in physical memory, so the OS manages the data location in the physical memory. Logical addresses are **translated by hardware** with the OS' aid.

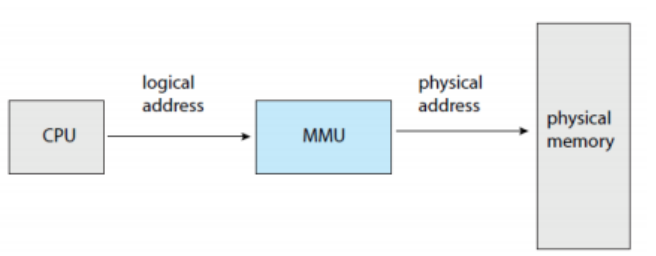


## Memory-Management Unit (MMU)

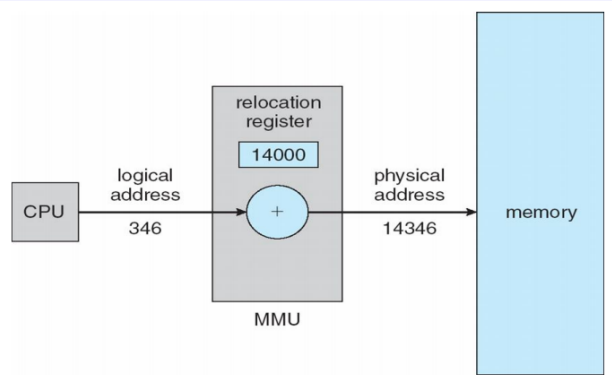
A hardware component which translates CPU generated addresses to physical addresses.

Programs deal with logical addresses and never see the physical ones.

Can be implemented in many ways (paging, relocation+limit registers, segmentation etc.)

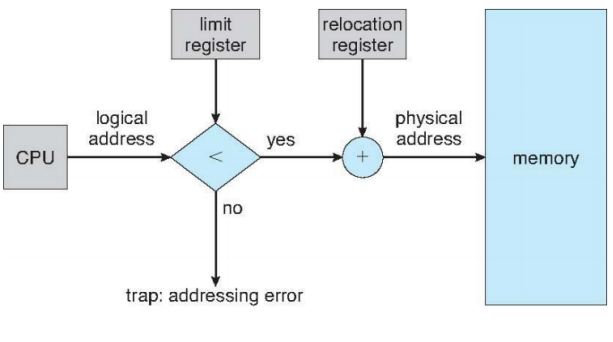



### Relocation register




- no protection, nothing checks if address is outside of range

### Relocation + limit registers




+ program doesn't have to be relocatable (it operates on "fake" physical memory) = simpler loader and faster load time
+ protection - each program has its own private address space
+ each program can have a different partition size (amount of physical memory allocated)

## Memory allocation

The process of allocating physical memory to programs requiring it.
We want efficient and fast allocation with smallest memory waste. Allocation is done with hardware support (MMU)

## Contigious Allocation

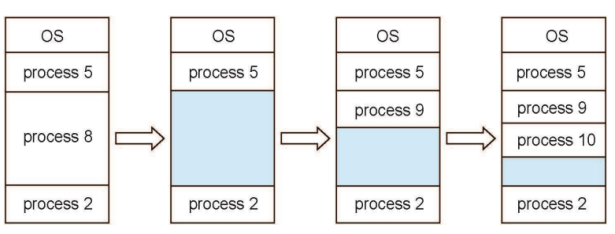
All techniques above allocate memory in contigious chunks.

### Multiple-partition allocation

Allocation of contigious block of physical memory of various sizes (to minimize loss)

Hole: a block of available memory; holes have various sizes. When a process arrives, OS allocates memory from a hole large enough to accomodate it. On process exiting, the partition is returned to OS, and adjacent holes are merged. OS keeps information about allocated partitions and free partitions.

- degree of multiprogramming limited by number of partitions used




Many possible policies:

#### 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

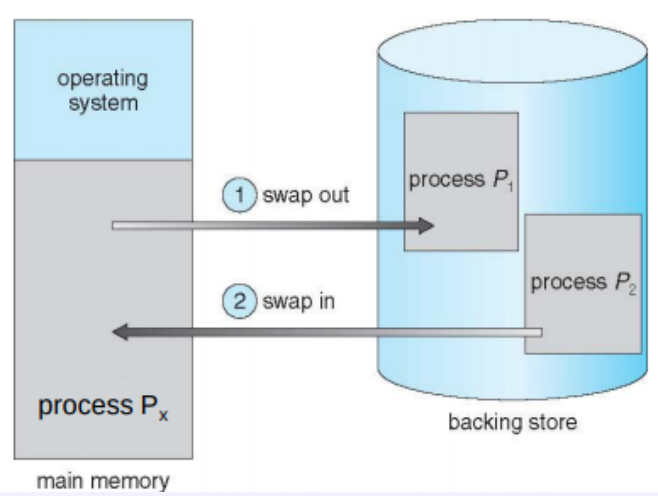
#### Worst-fit

Allocate the **largest** hole; must also search entire list - produces the largest leftover hole

First and best fit better than worst fit in terms of speed and storage utilization.

## Swapping

What to do when we run out of memory? We have to keep some programs in the backing storage. To do this we need to be able to swap in and out programs from memory.

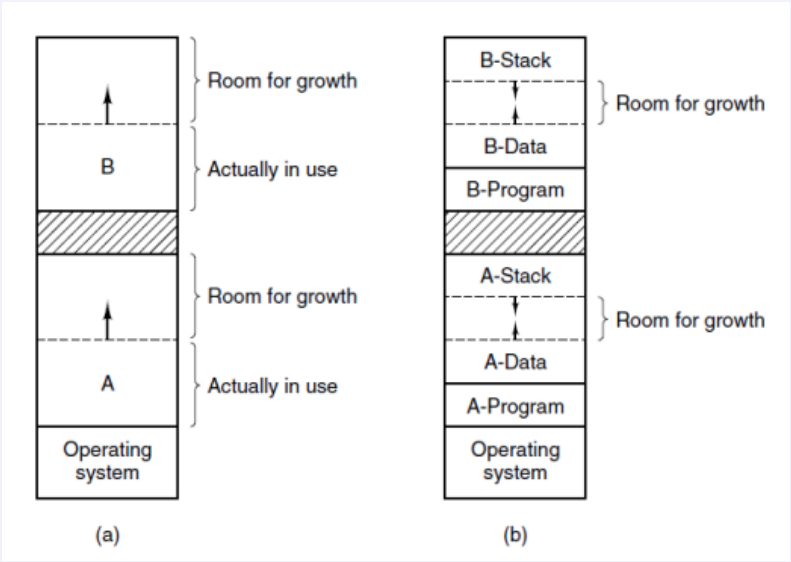


## Growing programs problem

Programs are not actually static! They can allocate memory dynamically, and hence grow in memory.
Two solutions possible:

a) Allocate more space than needed, if more needed, relocate

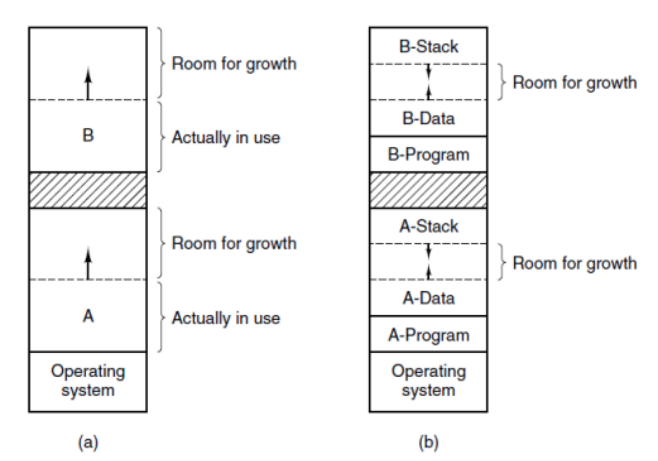
b) Allocate space for growing program in its address space




Both of these aren't ideal

## Memory fragmentation

Processes create memory holes, which might not be convieniently sized for other programs to occupy:




**External fragmentation** - you allocate the exact amount of memory requested, total memory space exists to satisfy a request, but it is not contiguous.
**Internal fragmentation** - you allocate more than what required, allocated memory may be slightly larger than requested memory
*First fit analysis - N blocks allocated = 0.5 N blocks lost to fragmentation, 1/3 may be unusable*

# Paging

### Non-Contigious Allocation

Issues with Contigious allocation:

- loads the entire program at once into memory when needed
- external fragmentation
- long swap times
- long **compaction** times (tight reallocation)

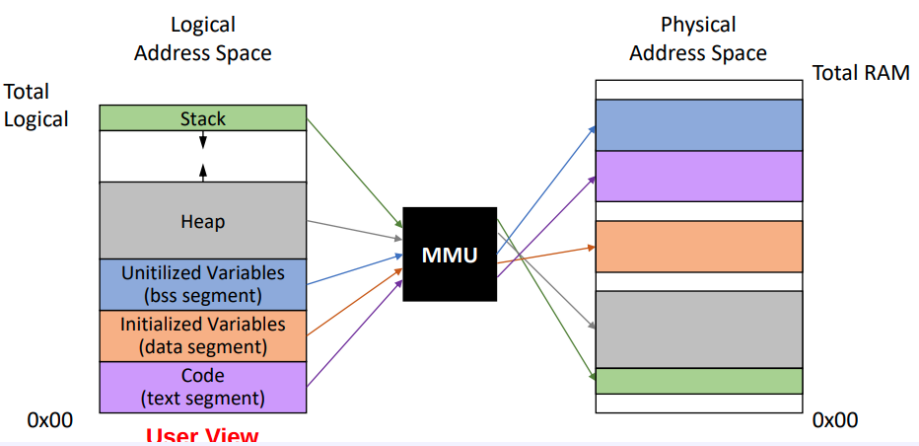
Presenting Non-contigious allocation:

- request physical space for each program's chunk when required
- split logical address space in chunks
- contigious in logical memory, non-contigious on physical memory
- segmentation + paging

## Segmentation

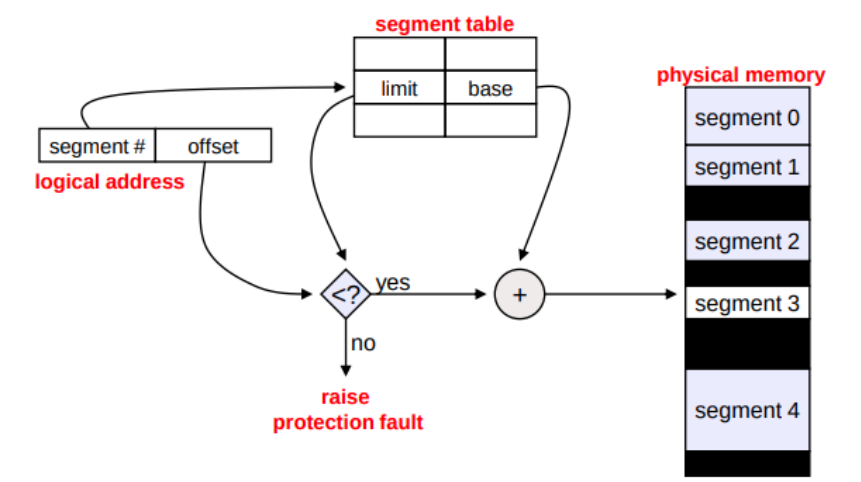
Partition an address space into **variable size** chunks/units. **Logical units** = stack,heap,data,code,subroutines ..

A logical address is split into (segment#, offset)
The logical address space is divided in **variable size** chunks




## Hardware support

Segment table:



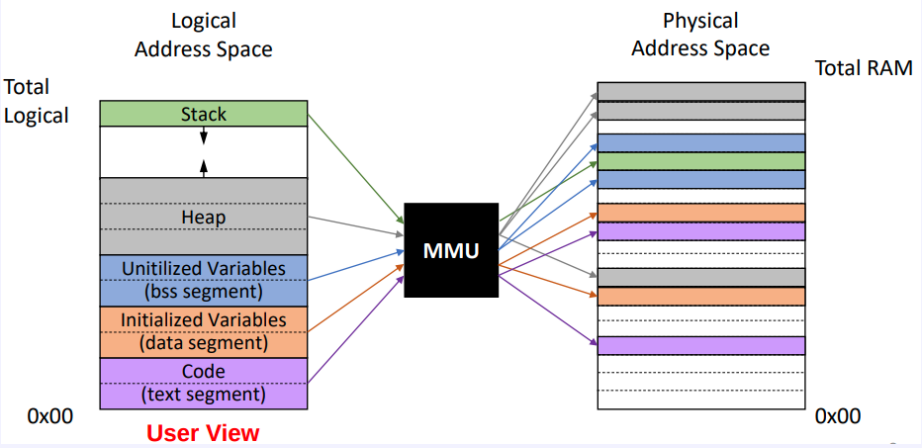

Multiple base/limit pairs, one for each part of the program (segment) the segment number is used as the index into the segment table
The segments can roughly be associated with addresses based on conventions.

+ allows non-contiguous physical addresses
+ reduces fragmentation by exploting varying sized holes
+ allocated chunks are smaller than the entire program address space
+ enables sharing (same segment can be shared across processes, think threads)
- process view and physical memory are extremely different
+ protection, process can only access its own memory
- external fragmentation still occurs

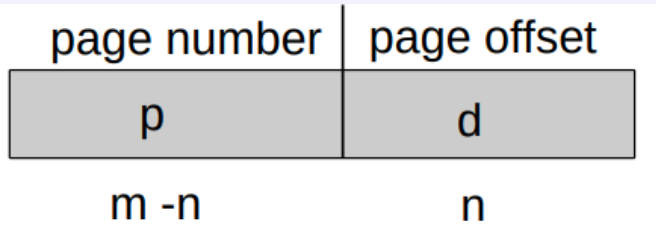
## Paging

Similar to segmentation, but logical address space divided in **fixed size** chunks Here we **combine** base address with offset instead of adding to entire address by single offset, this establishes a mapping. Each process might have its own page table, or a single one can be used.

Pages mapped to **frames**, page size = frame size

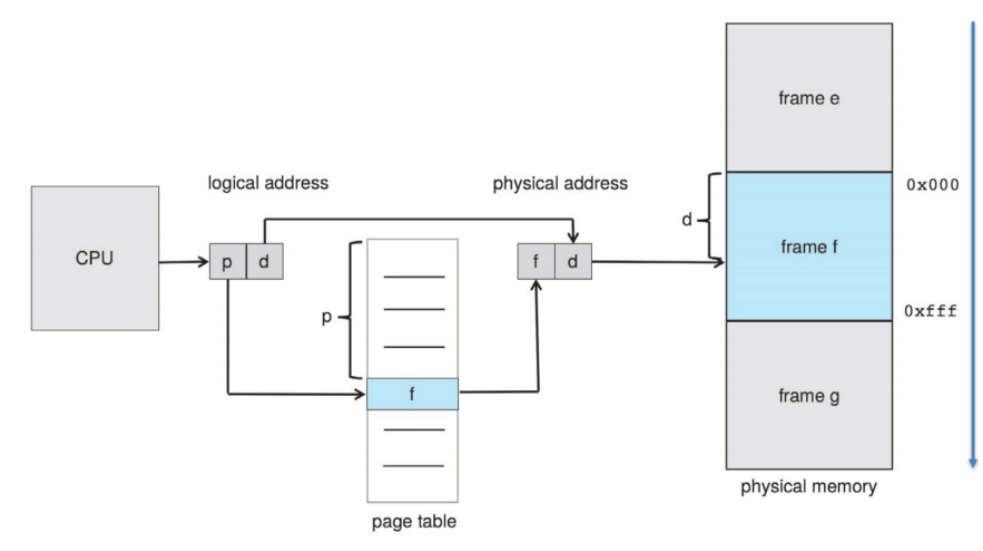



Logical address is (page #, page offset). page # is used as index into **page table** which again contains base addresses, page offset is **combined** with base address to create physical memory address.




*logical address space size* $= 2^m$ *bytes. Page and frame size* $= 2^n$ *bytes*

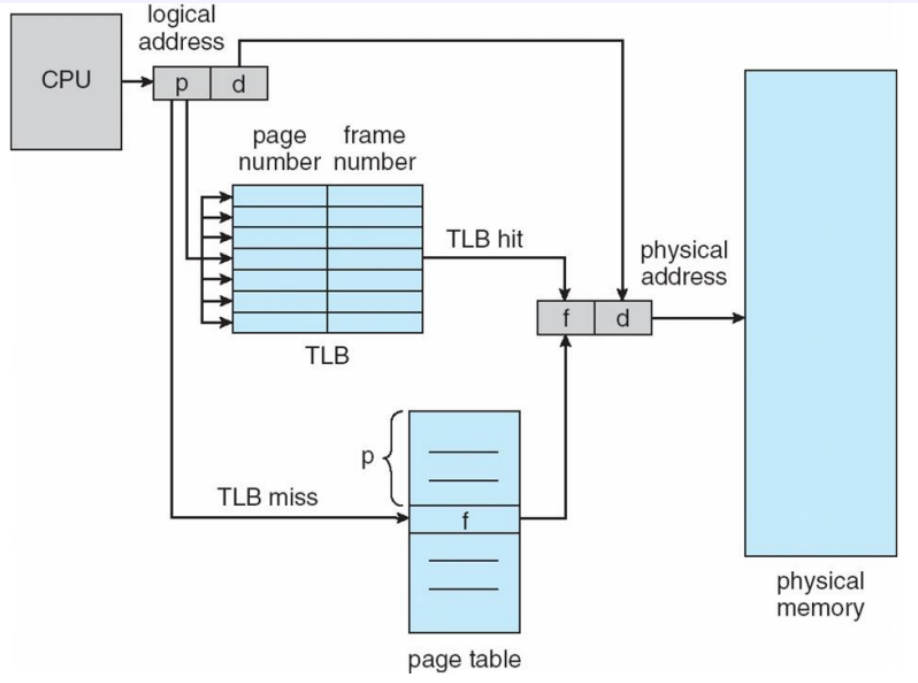
### Hardware Support




+ no external fragmentation!
- internal fragmentation depends on page size, if program "overflows" when dividing by page size.
- translations may require memory accesses (costly)
- different page sizes available on every system
- *average fragmentation = .5 frame size, worst case = frame size - 1 byte*
- page tables get huge, require more complex arrangements to cut size down

## Translation look-aside buffers (TLB)

In order to avoid wasteful memory lookups, a TLB cache is used when paging.
If translation exists in TLB, use it (free), otherwise fetch translation from memory (expensive)



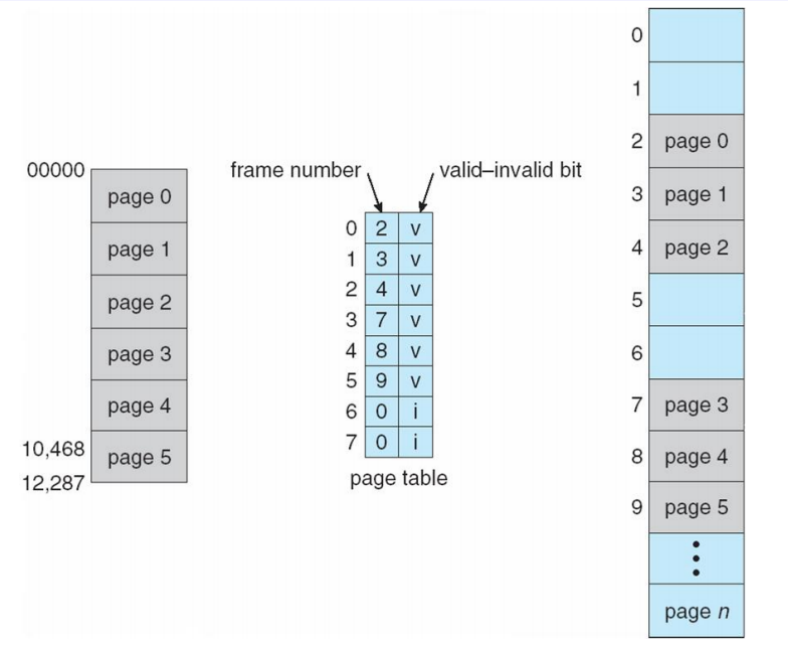

**Memory access time** - time the CPU waits to access main memory directly
**Hit ratio** ($\alpha$) - percentage of times taht a page number is found in TLB
**Miss ratio** - 1 - Hit ratio
**Effective access time (EAT)** - the average experienced access time:

with $\alpha = 80\%$, 100ns access times $= 0.8 * 100 + 0.2 * 200 = 120ns$
with $\alpha = 99\%$, 100ns access times $= 0.00 * 100 + 0.01 * 200 = 101ns$

## Memory protection

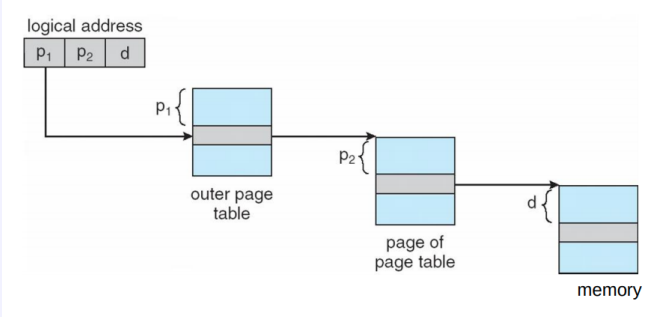
Page table entries (PTEs) can contain more information:

- protection bits - read-only, read-write,execute-only
- valid bits - valid = translation exists and is in the logical address space of process otherwise not in process' space (when more frames than logical pages)
- violations result in traps



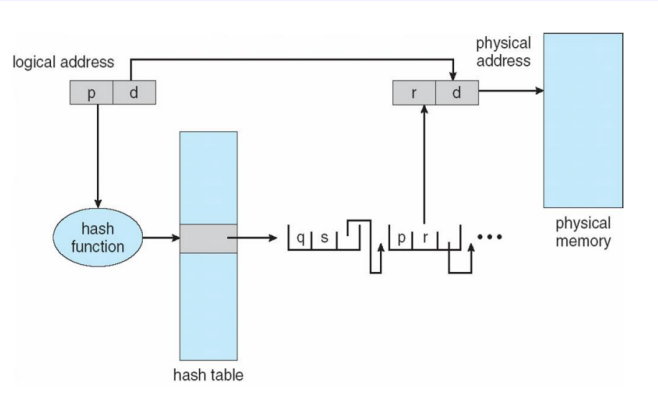

## Multi level TLBs

Consider 32-bit logical address space, with page size of 4kB ($2^{12}$). The entire address space is divided into 1 million frames ($2^{32}/2^{12}$) Each entry is 32 bits and so we need 4MB in total for the page table
We can create a table for the page table itself, to reduce the amount of "page translations" held in memory, we would only need to store the outer table entries:
So the first half of the address gets split according to the hierarchy, i.e. 22 bit page number can be divided into 12 bits for the outer and 10 for the inner tables. this means we only have to store the $2^{12}$ entries of the page table



## Hashed page tables

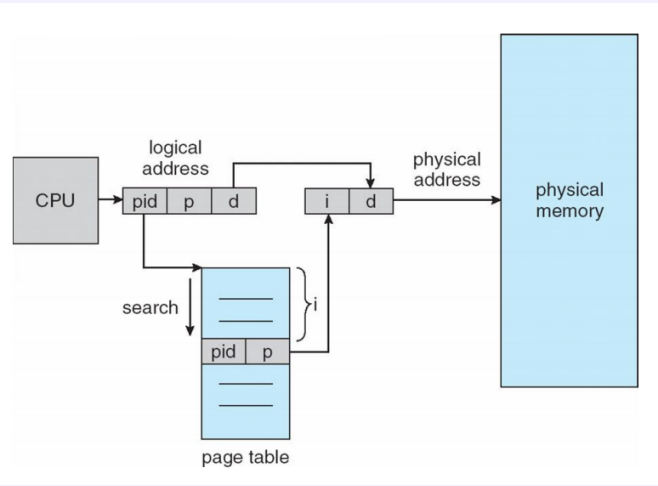
The logical page number can be hashed to generate the page table index, then the elements at that index need to be searched against (each entry would have the logical page number, the page frame, and the pointer to the next element)




Advantages: speed, fast access times

## Inverted Page table

Instead of each process having a page table and keeping track of all possible logical pages, track all physical pages each physical page = one entry
each entry has the virtual address owning that memory location along with the pid of the process whose virtual memory that is
this requires less memory for translations, but much more time is needed to search through the table, we can use hash tables to limit the search to one/few page-table entries
Shared memory requires OS intervention




## Advantages of paging

- inter-process memory protection
- prevents code being rewritten (protection bits)
- detects null pointer dereferencing at runtime
- reduce memory usage (shared libraries) - one copy of a library only necessary, not one per process
- generalizes use of "shared memory"
- can implement **copy-on-write(CoW)** mechanics easilly with read only pages and page faults
- memory-mapped files - can load a file on demand into logical addresses

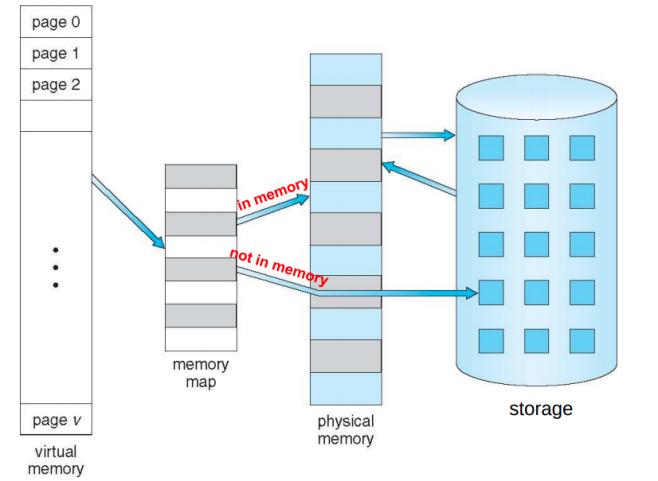
# Virtual Memory

## Overlays

What if the logical address space is greater than the available physical memory ? Can a program correctly execute even if it is not all in memory at all times?
One solution is to simply divide the program into **overlays**, which are smaller than the physical memory, then load and unload them as needed via some root overlay manager (user program) which determines when a new overlay needs to be loaded.
This is something you'd need to use when really constrained on physical memory. However there are better solutions

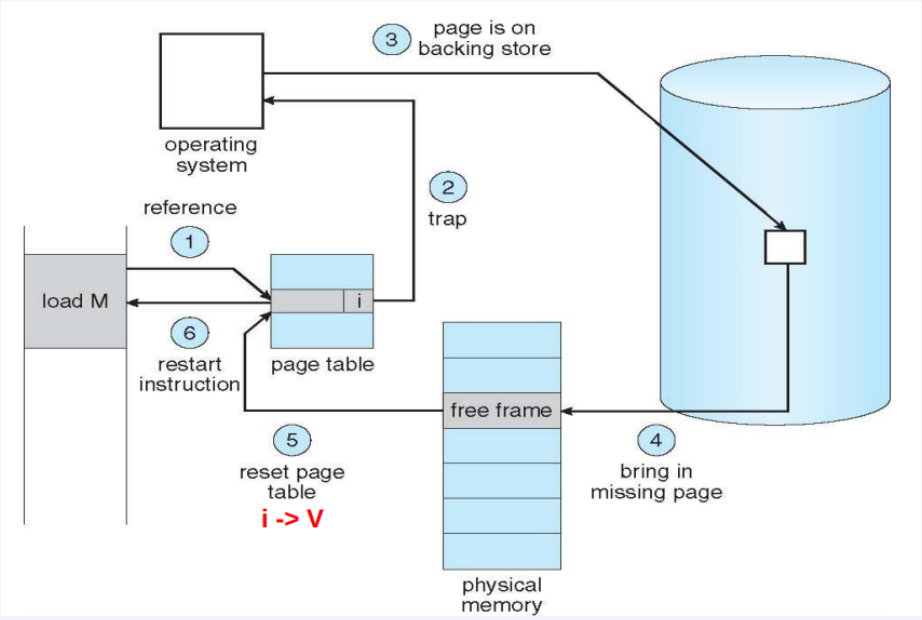
## Virtual memory

fully decouples address space from physical memory, allows larger logical address space than physical memory.
We've already seen an example: **Paged virtual memory**!
We can simply keep some of the pages out of the main memory and on the disk, and swap pages into memory when needed. This requires the **eviction** of older pages.
Main memory becomes a **cache** for pages!



## Page Fault

1. Software accesses a page that is not in memory
2. Hardware triggers a **page fault**
3. Operating system checks internal data structures:
   - If invalid reference, abort the original program
   - If not in main memory currently, continue
4. Operating system finds a free frame - swaps page into frame via disk operation
5. Operating system sets internal data structures to indicate page now in memory + set valid bit in page table
6. Operating system restarts the instruction that caused the page fault



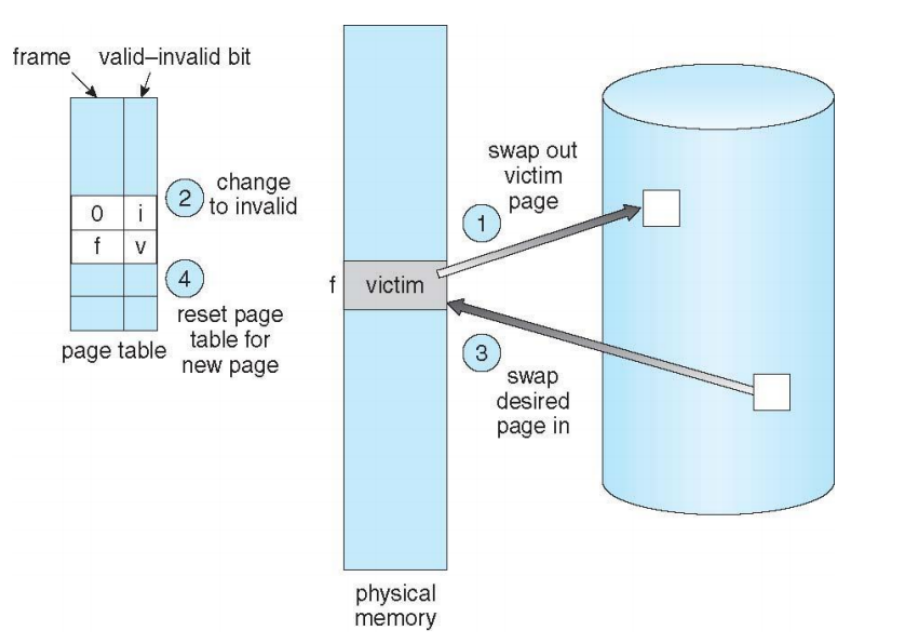

## Demand paging

Pages brought into memory **when accessed first**
Few systems try to anticipate future needs, pages may be clustered - OS can anticipate which pages usually get used together and bring them all in when one is accessed

- expensive heavilly depends on storage latency

## Page allocation and replacement

When there are no more free frames, we must **evict** unfortunate pages in memory.



## Page replacement policies

What page to evict ?
Reduce page-fault rate by selecting **best victim page** - one that we will never touch again or in the near future.

**Belady's Theorem** - evicting the page that won't be used for the longest period of time minimizes page fault rate

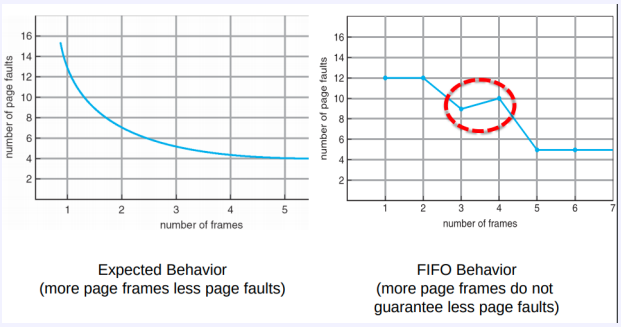
evict **unmodified** pages first - since don't have to write them back to disk

### First-in-First-Out (FIFO)

replace page that has been inserted the longest time ago

+ simple, just a linked list or queue

- Belady's anomaly, more page frames do not guarantee less page faults!




Expected Behavior
(more page frames less page faults)

FIFO Behavior
(more page frames do not guarantee less page faults)

### Least Recently Used (LRU)

Replace page that has not been used in the most amount of time

+ uses past knowledge rather than predictions

+ does not suffer from Belady's anomaly (stack algorithm)

- requires substantial **hardware assistance** - keeping track of last usage of each page

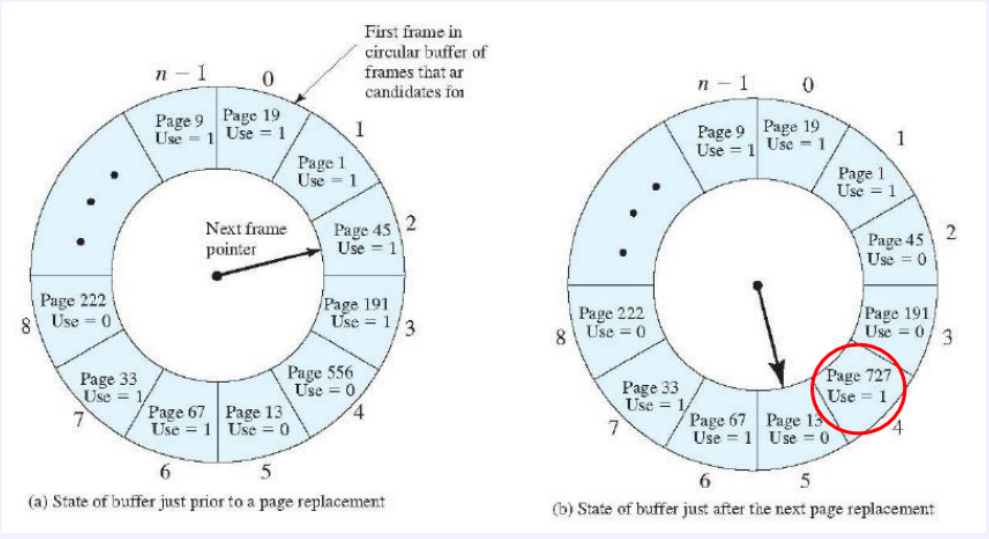
Need the following page table entry bits (**hardware**):

- Page referenced (if accessed or not)
- Page modified (if access was in write)

and need to keep track of a history for each page in the kernel.

### Second chance

Variant of FIFO, makes use of page usage data. use circular queue and **reference bits**, keep "victim" pointer, if the victim was referenced, set reference bit to 0, and move pointer along by one, if not replace that page with new one, and move pointer along.




(a) State of buffer just prior to a page replacement

(b) State of buffer just after the next page replacement

## Frames as resources

We can decide to limit the number of physical frames available for certain processes. Frame Allocation can be:

- Equal: an equal share for all processes
- Proportional: a share based on the program size or other factors

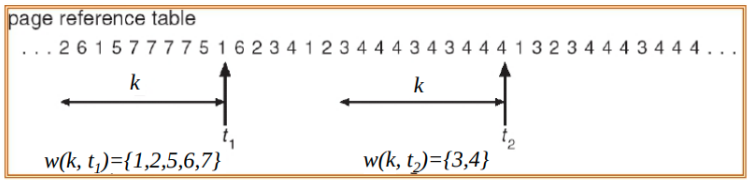
Frame Replacement can be:

- Local: each process is given a limit of pages it can use (pages against itself, i.e. evicts its own pages):
  - \+ doesn't affect other processes
  - \- poor overall utilization of frames, and long access times
- Global: the "victim" is chosen from all available frames regardless of owner
  - \- risk of global **thrashing**
  - \+ processes' page frame allocation can vary dynamically

## Working set

let t be the time, k the working set **window** (measured in page refs) then
WS(k,t) = **Working set** in time interval t,t-k.

Then a page is in the WS, iff it was referenced in the last k references by the process.

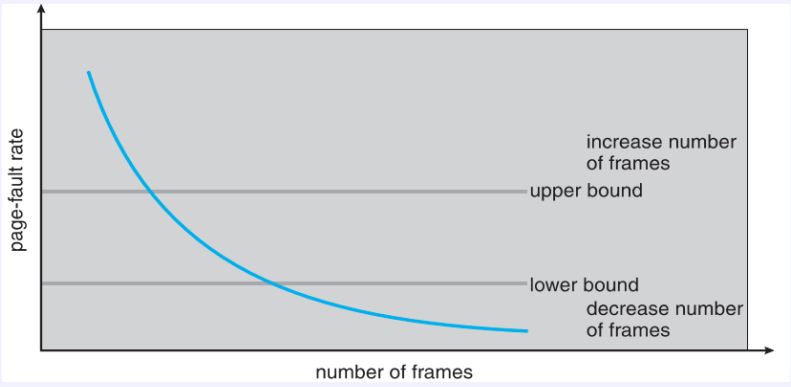



*Examples of working set for k = 10*

The size of the working set changes with program locality - when program has poor locality, more pages are referenced and the working set grows
The working set must be wholly in memory, otherwise heavy faulting - **thrashing**

## Page-Fault Frequency Allocation

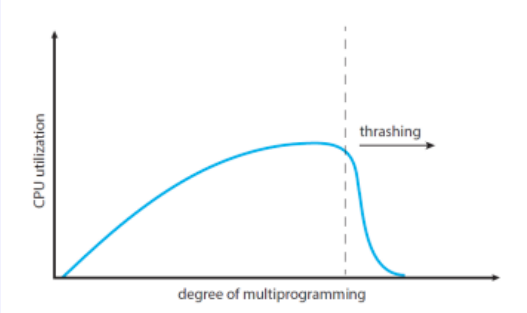
We can make use of the working set to come up with a great allocation algorithm.
Notice: there is a relationship between the size of the working set and the page-fault rate of a process.
Use a local replacement algorithm (i.e. process pages against itself, with limited number of pages)
The higher the page fault frequency (PFF), the more the process is struggling to keep its working set in memory.
set an acceptable PFF range (lower and upper bound), if actual frequency higher than upper bound, increase the number of frames allocated to the process, if frequency lower than lower bound, remove one of the process' frames.



## Thrashing

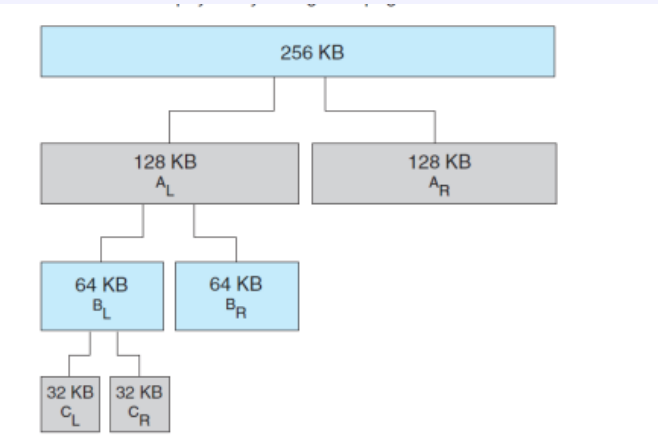
When system spends most of its time servicing page faults, little time spent doing useful work.
Reasons:

- Could be that there is enough memory but replacement algorithm is not working well - it's incompatibile with the program behaviour
- Could be that memory is over-committed - OS sees CPU poorly utilized and adds more processes



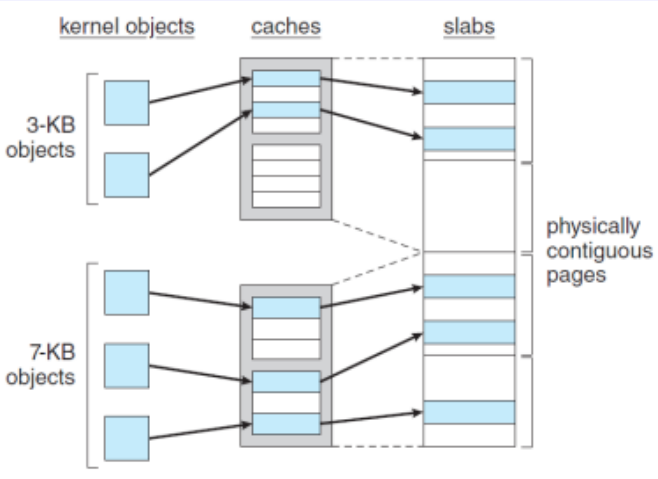

## Buddy system

A contiguous power of 2 memory allocator for kernel structures. Satisfies request in units sized in powers of 2. all requests are rounded up to nearest power of 2.




## Slab allocation

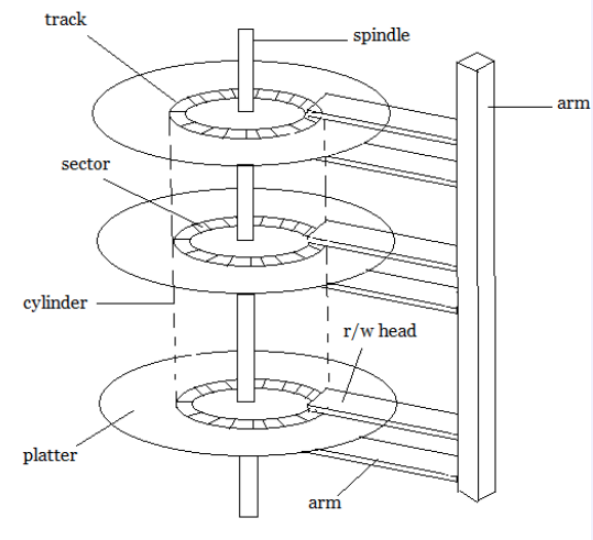
Contiguous memory allocator for kernel structures. Slabs are made up of one or more physically contiguous pages. A cache consists of one or more slabs. There is a cache for each **unique kernel data structure**. Each cache populated with objects If there are any **free slabs**, the allocation is immediate (no search for memory space)

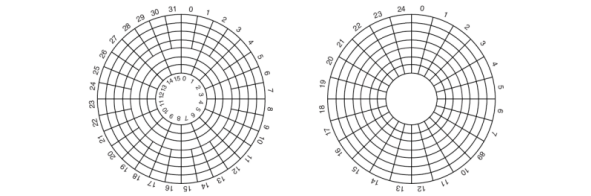


# Secondary Storage

## Magnetic Disk

Traditional cheap HDD storage. Addressed either geometrically (Cylinder, head, sector) or more recently using **Logical block addresses (LBA)**



(**left**) Physical geometry of a disk with two zones. (**right**) A possible virtual geometry (addressing) for this disk

Performance depends on:

- **Seek time** - moving the disk arm to the correct cylinder. Depends on arm speed, does not diminish quickly due to physics
- **Rotation (latency)** - waiting for the sector to rotate under head. Depends on RPM of disk, rates are slowly increasing
- **Transfer time** - transferring data from surface to disk controller. Depends on density of bytes on disk, increasing relatively quickly

OS tries to minimise disk usage, mostly the seeking and rotation time.
OS can inflate file block size , or co-locate "related items" to reduce seeking (blocks of the same file, data and metadata for a file)
OS may keep data or metadata in memory to reduce physical disk access.
OS may fetch blocks into memory before requested (to hide slow disk accesses)

## Block scheduling

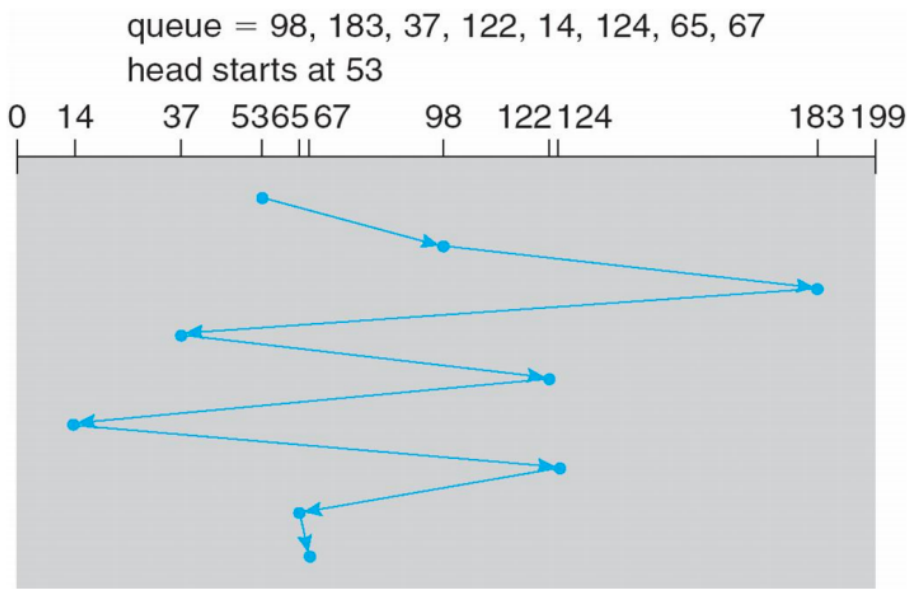
Applications request data accesses to the OS, OS maintains **request queues** and generates **transfer commands** to/from the disk(s) (seeks,waits for rotations, data transfers, all the low level commands)
App waiting time can be reduced by modifying the order of blocks requested: This of course brings up a number of factors that need to be considered as with any scheduling:

- fairness
- efficiency
- speed

Many block scheduling algorithms exist.

## First come first served (FCFS)

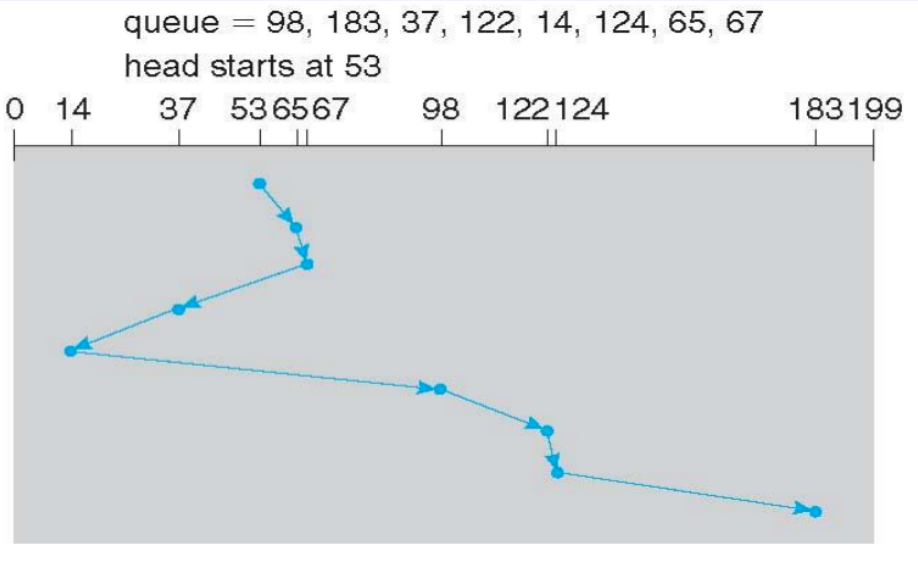
Simply keep a queue.




+ reasonable when load is low
- long waiting time for long request queues

## Shortest seek time first (SSTF)

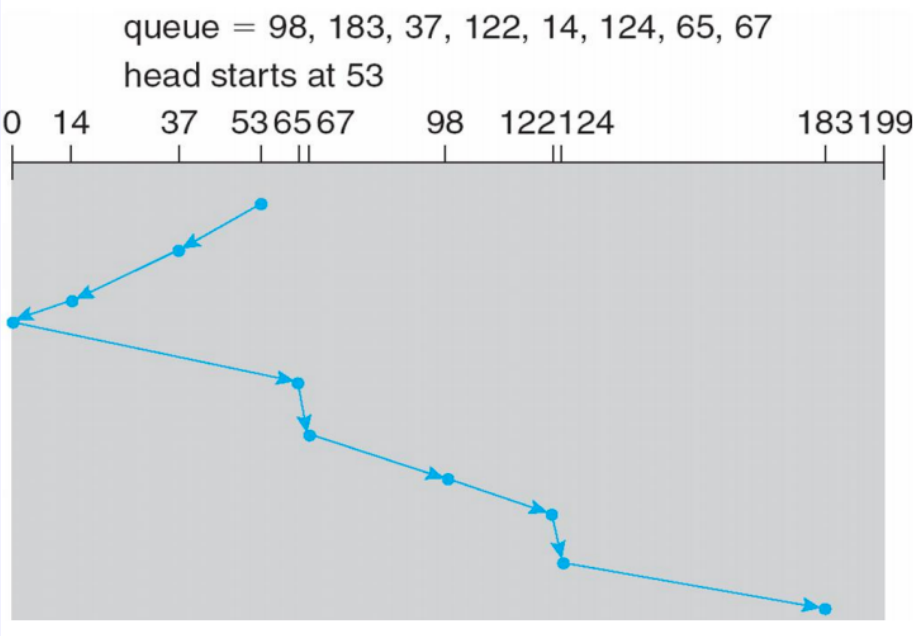
Order requests according to seek time, and serve them in that order




+ minimizes seek time, and therefore maximizes request rate
- unfairly favors middle (clustered) blocks

## SCAN

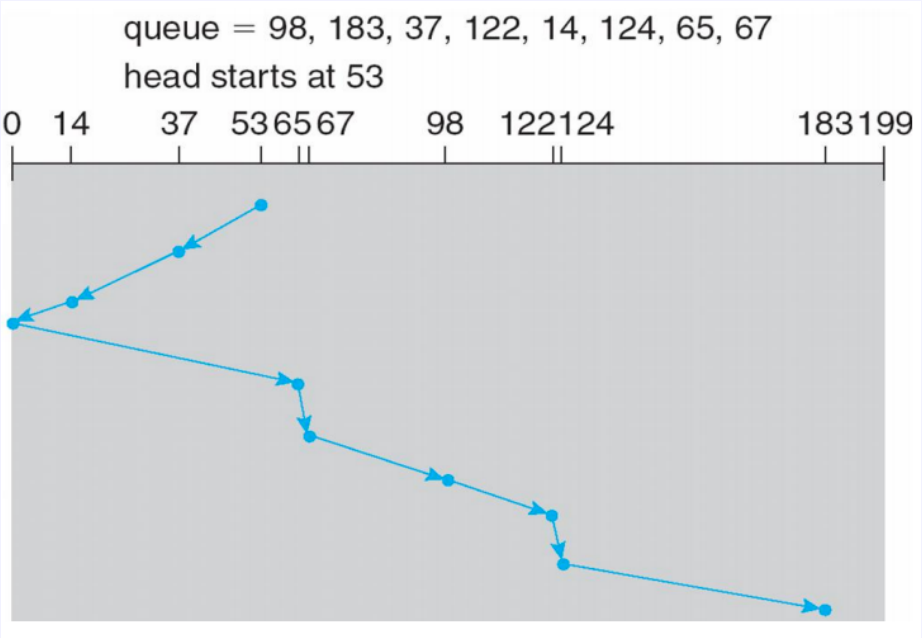
Disk arm starts at one end of the disk, then moves toward the other end, at which the direction is reversed and this repeated




+ simple, easy to understand
+ no starvation
+ better than FCFS
+ good under heavy load
- not fair - long waiting time for cylinders just visited by the head
- bad when all requests are uniformly spaced out ()
- skews wait times non-uniformly

## C-SCAN

Same as SCAN, but once other side is reached, it returns to start position and repeats.



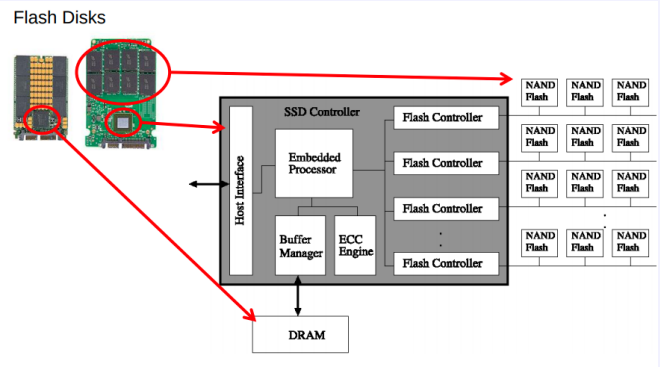

+ more uniform wait times than SCAN
+ good under heavy load

## Selecting an algorithm

All algorithms behave like FCFS with one request.
Performance usually depends on number and type of requests
Requests for disk service can be influenced by file-allocation methods and the metadata layout.

## Solid state drives (SSD)

A version of storage with no moving parts. More expensive, more robust.



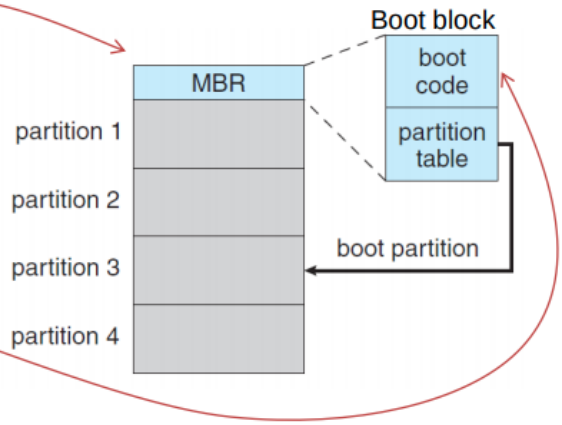

Read access time is mostly independent of the device geometry - block scheduler is not needed.
Unit of read/write is a page (typically 4kB).
Usually lower write vs read speed. Higher write lag than read lag. write usually lower throughput than read
Flash media must be **erased before you can write to them**
Unit of erase is typically 64-256 pages. Limited number of times before unusable (10,000 - 1,000,000 times)

- \- higher cost per GB
+ more energy efficient than HDD
+ much more physicaly robust

### Storage device management

Storing data is not enough - need metadata
Before storing data, device needs to be initialized

- Low-level formatting
- For each partition
  - Volume creation (lvm2)
  - Logical formatting (file system)




- **Booting**
  1. Firmware, or BIOS
  2. Reads code in MBR
  3. MBR also contain partition table
  4. Code in MBR reads boot sector of the selected partition
  5. Pass control to code in selected partition

# File System

### File System

**Abstracts** away secondary storage. **Files** are unit of abstraction, organized into **directories**.
Enables **sharing** of data between:

- Processes
- People
- Machines
- etc.

Provides additional:

- Access control
- Consistency
- Reliability
- etc.

## File

A **named collection of related information** with some properties:

- Content
- Size
- Owner
- Protection
- Last read/write time

File **types** are understood by **file system**:

- Directories, symbolic links, **devices**
- Programs, Data

Types need to be encoded into file name on windows, not on linux (.com,.exe etc..)

## File access methods

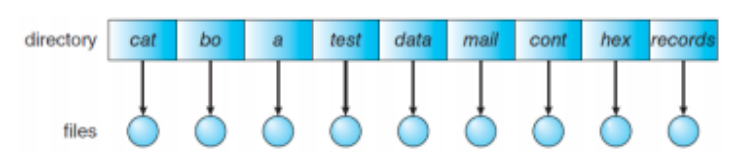
File systems provide different **access methods**

- **Sequential** - read/write bytes one at a time, in order
- **Direct** - random access given by a byte #
- **Record** - file is an array of fixed- or variable-sized records
- **Indexed** - One file contains an index to a record in another file

## Directory structures

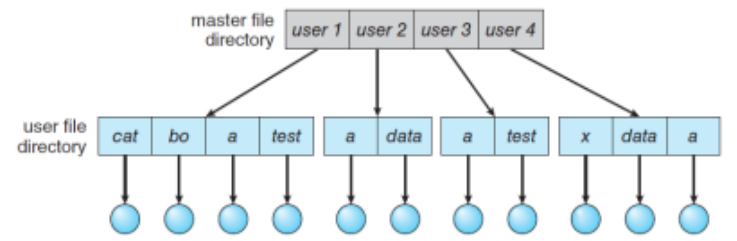
### Single-level

Files must have unique names



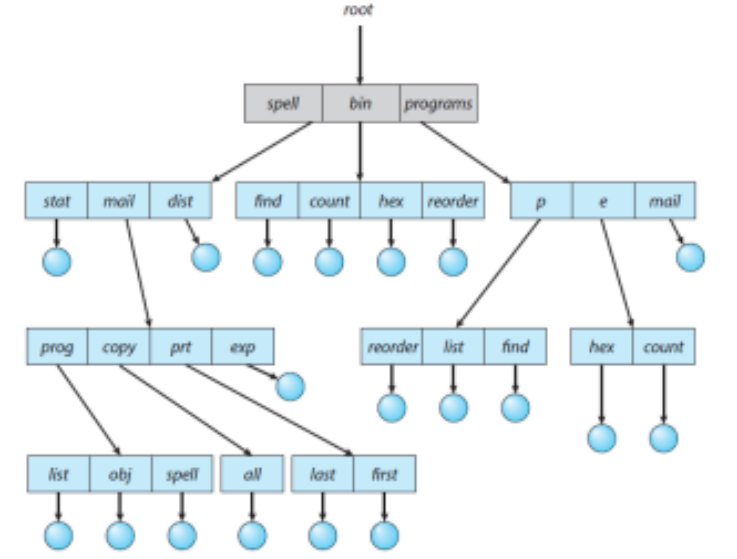

### Two-level

Sharing requires path abstraction



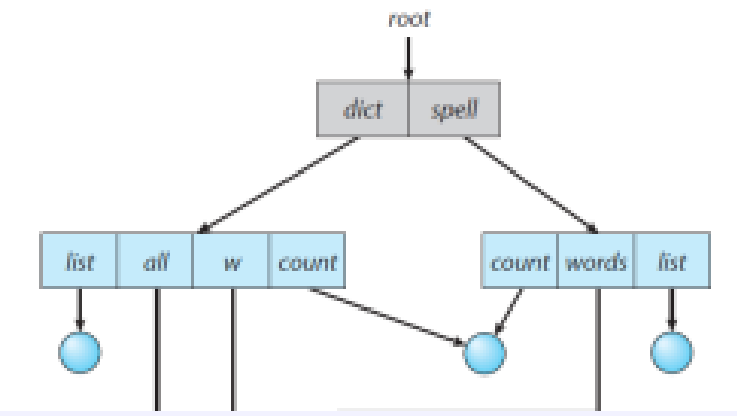

### Tree structured

Eventual replication of files




### Acyclic graph

Links as a solution to avoid duplication



## Directories

Directories provide way for users to **organize their files** under conenient **file name spaces** for user and FS
Most file systems support **multi-level directories** (/,/usr,/usr/local,/usr/local/bin)
Most file systems also support notion of **current directory**

**Absolute names** - fully-qualified starting from root of FS
**Relative names** - specified with respect to current directory

### Internals

Usually just a **file** with special metadata.
Organized as a **symbol table** - list/hash table of name & file references
Contains **attributes** such as:

- Size
- Protection
- Location on disk
- Creation time
- Access time

Usually unordered ('ls' sorts the results)

## File protection

File system implements protection system:

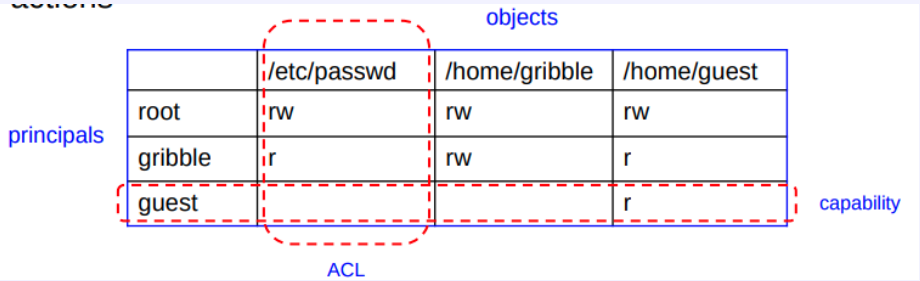
- Control **who**(user/group) can access **what** (file/directory)
- Control **how** the file can be accessed (read/write/exec)

**Objects** - the **what**
**Principals** - the **who**
**Actions** - the **how**
Protection system dictates whether a given action performed by a given principal on a given object should be allowed.

## Protection models

Two different models:

- Access Control Lists (ACLs) - for each **object** keep list of **principal's** allowed actions
- Capabalities - for each **principal**, keep list of **objects** and **principal's** allowed actions




**Principals** usually summarised as groups:

- Owner
- Group (customized lists of users)
- Other

## Data Structures

Some metadata is needed for a file system:

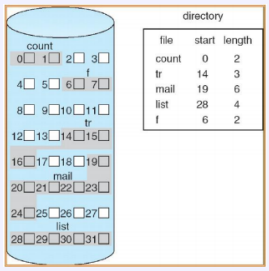
### In backing storage

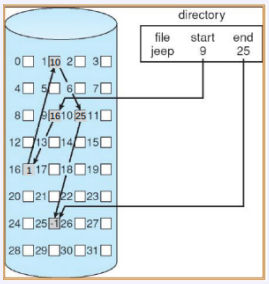
- Boot control block - information needed to boot an OS
- A per file File Control Block (FCB) - details about each file, unique ID
- Volume control block - volume details, number of blocks, size of blocks, free-block count and pointers, free-FCB count and pointers
- Directory structure - used to organise files

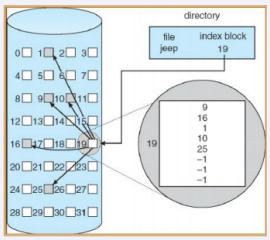
### In memory

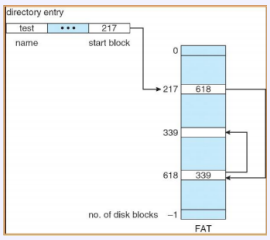
- Mount table - information about each currently mounted volume
- Directory-structure cache - information about most recently accessed directories (for speeding up queries)
- The system-wide open-file table - contains copies of FCB's for each open file with other information
- The per-process open-file table - contains pointers to the appropriate entries in the system-wide open-file table for all files that the process has opened
- Buffers - hold file-system blocks when they are being read/written to a FS

## Disk allocation strategies




### Contiguous Allocation

Each file is required to occupy a contiguous part of the disk. A file entry contains **Address of starting block** and **length of file in blocks**

- \- External fragmentation, holes that cannot be occupied due to size
- \+ reading file is quick
- \+ most resilient to file damage (know when there is memory errors)

### Linked Allocation

Each file entry contains pointer to the next block, each directory entry contains the first and last block of a file.

- \+ no external fragmentation
- \+ files can grow as long as there is space
- \+ only really good for sequential access
- \- no random access, need to traverse all of previous blocks - O(N)

### Indexed Allocation

Each file has a corresponding index block, which contains the indices of all the blocks which make up the file in order. Directory entries tie files to their index blocks.

- \- wastes a lot of space if files are made up of few blocks (internal fragmentation)
- \+ no external fragmentation, files can grow as long as there is space
- \+ fast random access unlike linked allocation

### File allocation table (FAT)

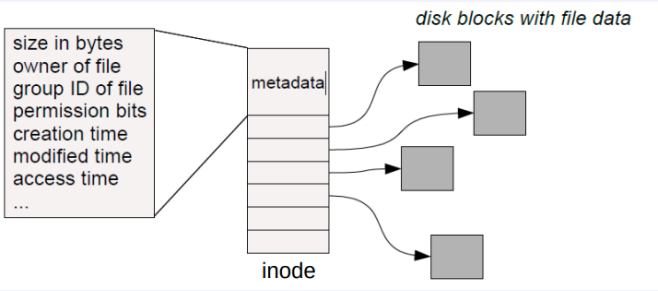
FAT table directly correlates entries to sectors, with each entry pointing to next sector of owning file. directories contain sector numbers of start of each file.

- \+ more resistant to damage than linked allocation, can store multiple FAT tables easilly
- \+ allows random access, but not too fast still need to traverse FAT table (but this is more localized and smaller)
- \+ only FAT needs to be traversed for each file operation (not various parts of disk)
- \- FAT might be spacious and depends on number of entries
- \- internal fragmentation depending on block siz

## INodes - Indexed allocation

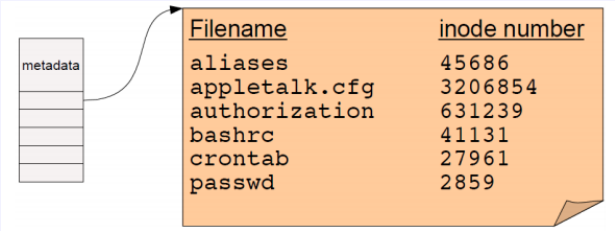
Every file and directory is represented by an inode with a unique id (**index node**)
every inode contains:

- Metadata - file owner, access rights etc.
- Location - of file blocks on disk

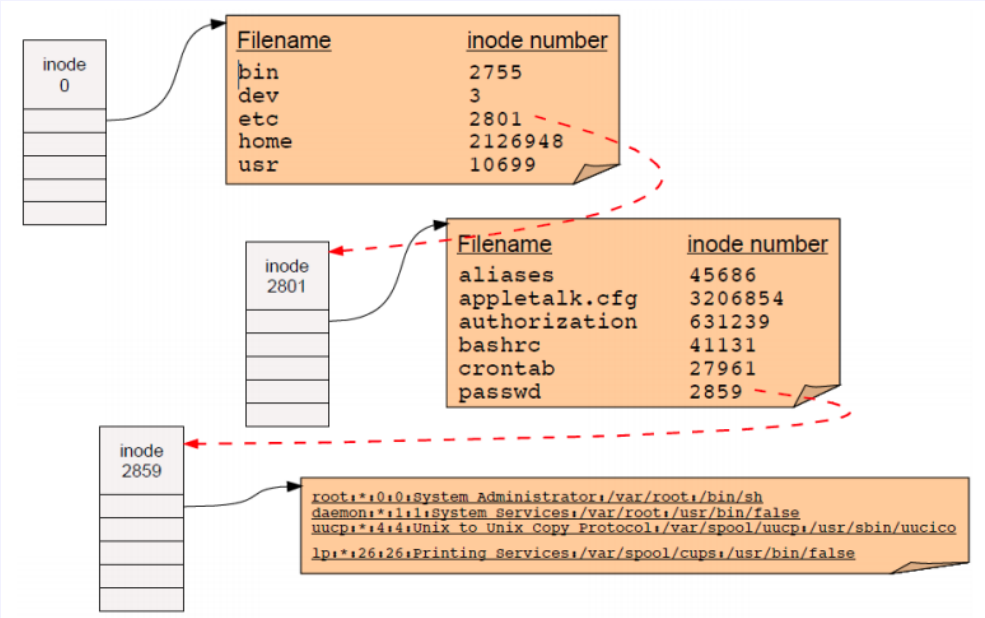



## Directories

A directory is a flat file of fixed-size entries. Each entry links filename to inode ID




To resolve a pathname, e.g. "/etc/passwd", start at root and walk down chain of nodes:

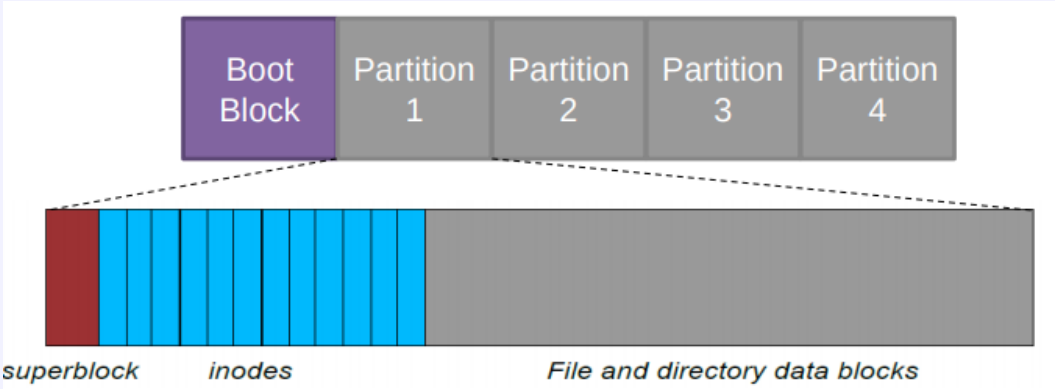



Multiple directories can contain the same file! (hard link in unix)

## Inode file systems - data layout

The data on the disk contains different areas:

- Superblock - specifies boundaries of other areas, contains head of freelists of inodes and file blocks
- Inode area - contains inodes for each file on the disk (each the same size)
- File contents area - fixed-size blocks referenced by inodes

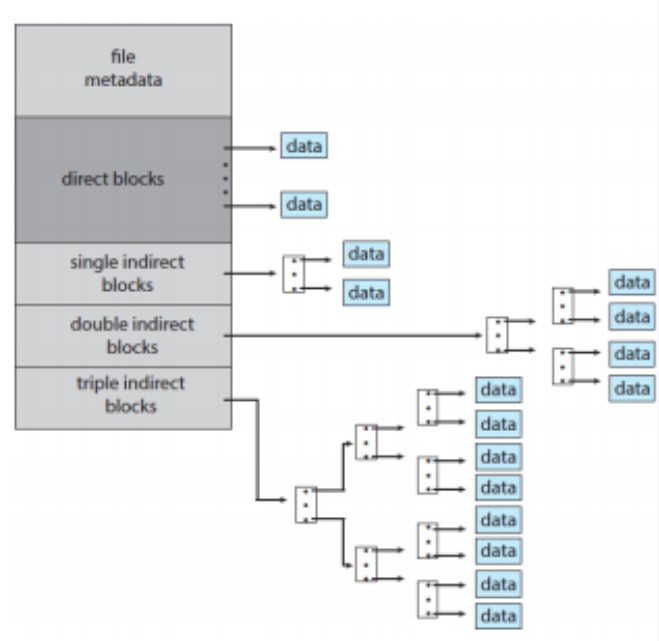



To find any inode, simply use their ID as index to the inode area! Can do this since they are fixed size.

- filesystem has fixed number of potential inodes - set when FS is created

## Inode file system - block list

Each inode contains metadata and the **block list**.
To be able to store very small and very large files, the block pointers contain a number of direct pointer slots (i.e. ones that point directly to data blocks) and a number of single, double and triple indirect pointers which point to blocks containing more pointers!



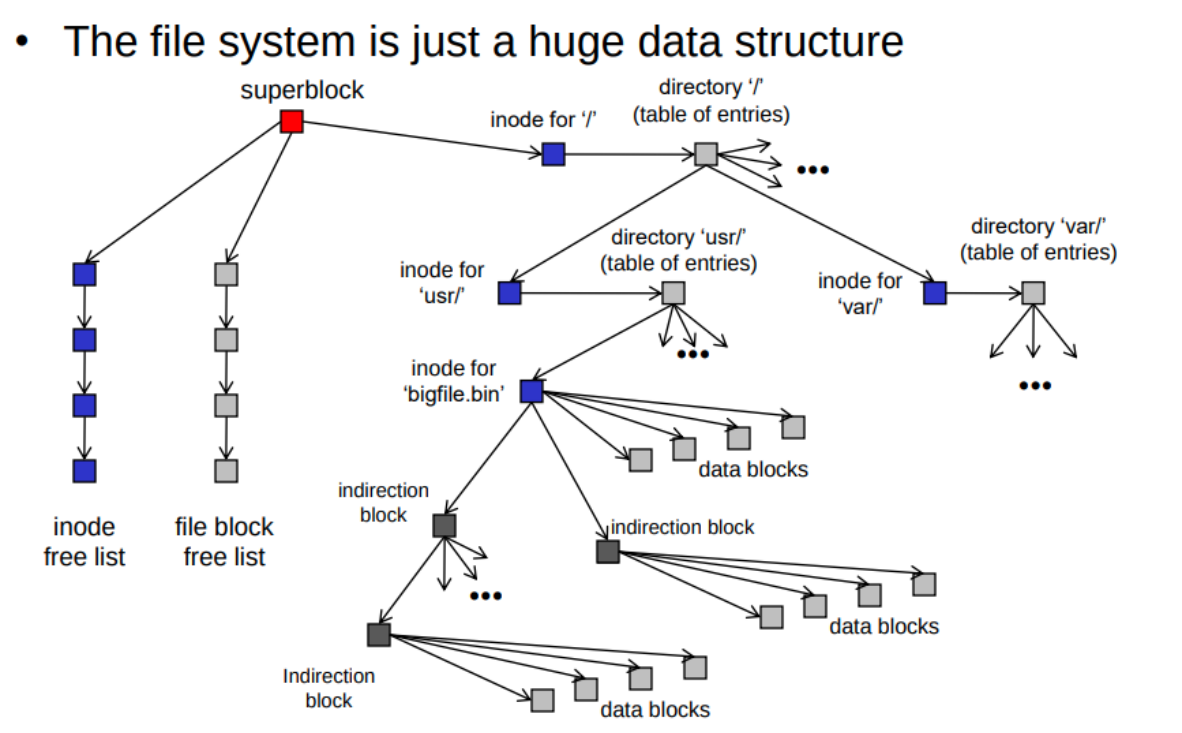

### Example

ters and if each inode contains 10 direct pointers, one single, double and triple indirect pointers (13 in total), then: With 512B data block sizes,32b(4B) poin

- Can access files as large as $10 \cdot 512B = 5120B$ with direct pointers
- The single indirect pointer, can link to a block containing $512B/4B = 128$ direct pointers, each pointing to a block: this lets us access $128 \cdot 512B = 64kB$
- The double pointer points to 128 single pointers which point to 128 blocks of data each, this lets us access $128 * 128 * 512B = 8MB$
- Similarly, we can access $128^3 * 512B = 1GB$ with a triple indirect pointer
- In total: $5120B + 64kB + 8MB + 1GB \approx 1GB$

### Inode file system - overview

The file system is simply just a huge data structure:




An important part of a file system is to lay all this data out on a disk efficiently, also keeping in mind the physical constraints of the disk, i.e. trying to keep locality within a directory, and allowing sequential access to many files for when seek times are expensive.

# Synchronization

### Synchronisation

Cooperating tasks access the same data:

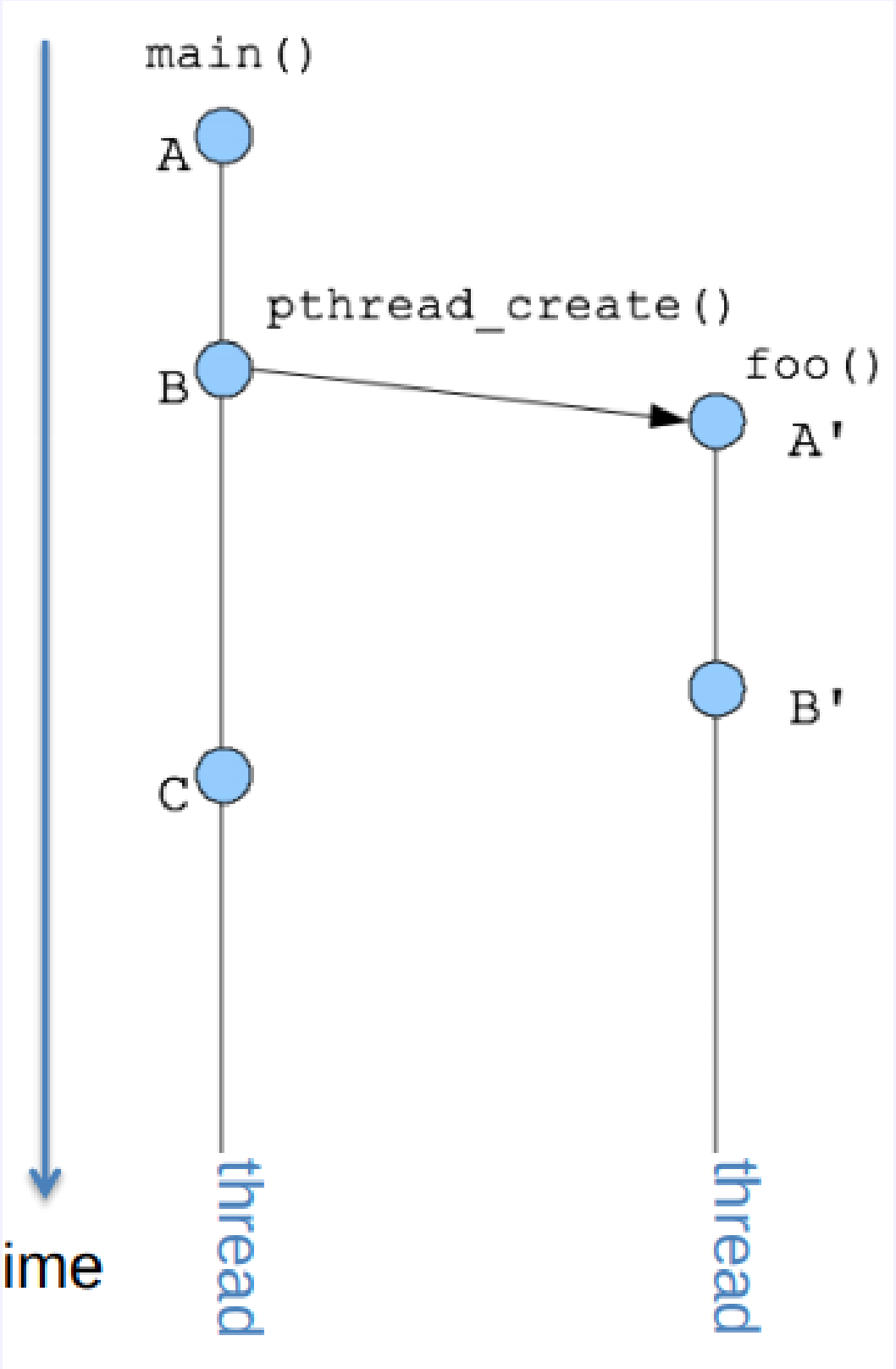
- Two or more threads of the same process share **data** and code
- Two or more processes can share data with **shared memory**

**Concurrent or parallel** access to shared data is either virtually done at the same time or actually at the same time for parallel access. This may result in **data inconsistency** (two processes may overwrite some data in a way which makes it invalid).
How to ensure **ordered** execution of cooperating tasks?

## Ordered execution

Instructions executed by a single thread/process are **totally ordered**: $A < B < C$. i.e. A happens before B and C etc..

Without synchronization, instructions executed by a distinct thread/process must be considered **unordered/simultaneous**: cannot guarantee any of: $X < X'$ or $X' < X$ or C == A' or C == B'
Creation relations always hold: $A < B < A' < B'$

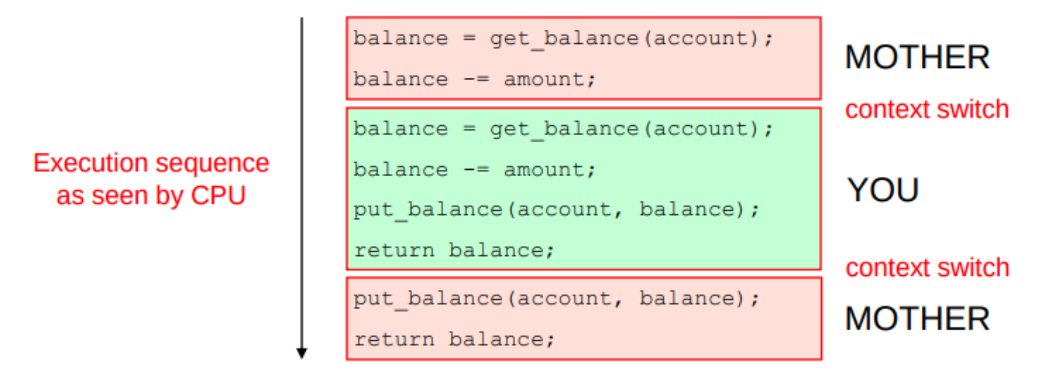


## Race condition

When the result of concurrent or parallel execution is non-deterministic and depends on which thread runs precisely when.

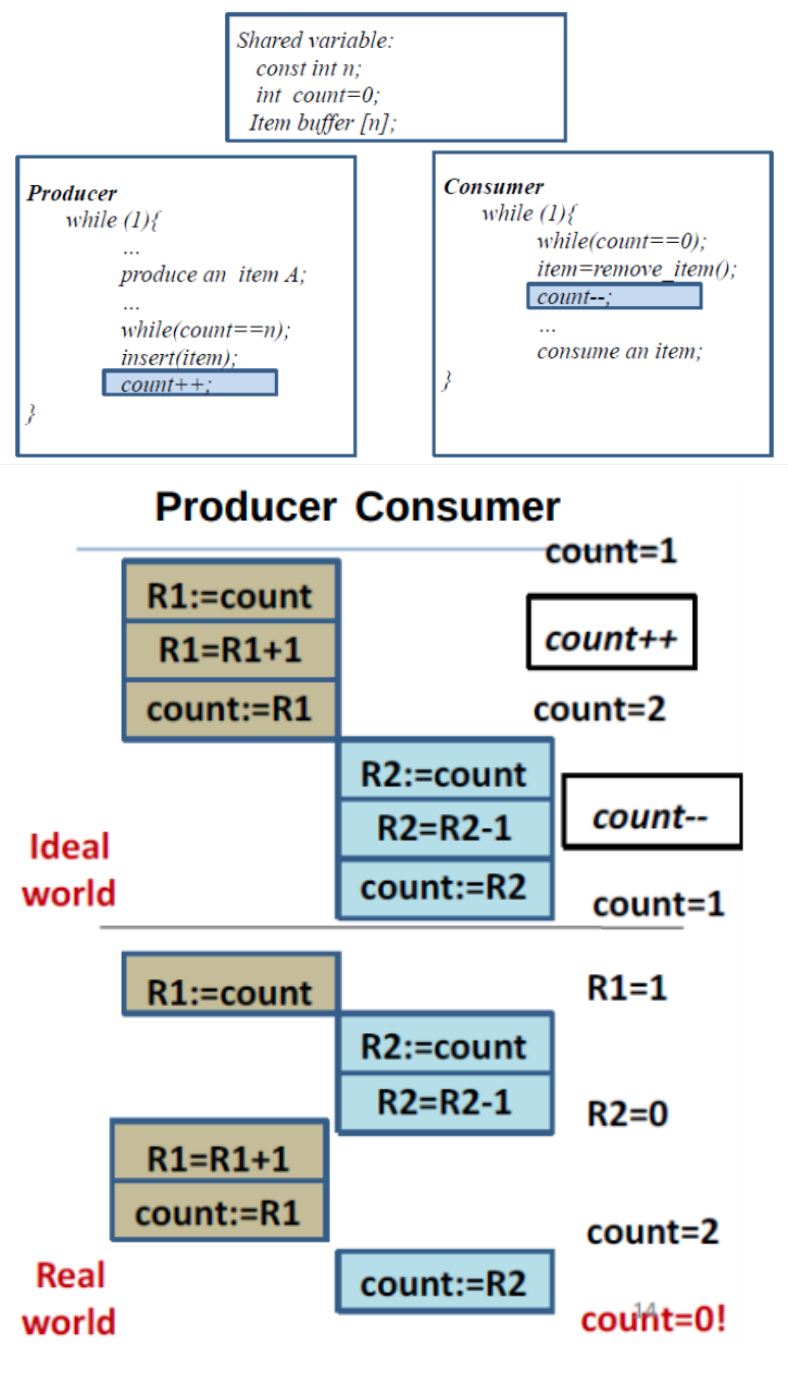
## Example Race condition

Race conditions happen when two **interleaved** sections of the same code, execute in a harmful way.
The below code shows a case where the bank account does not register one of the transactions!



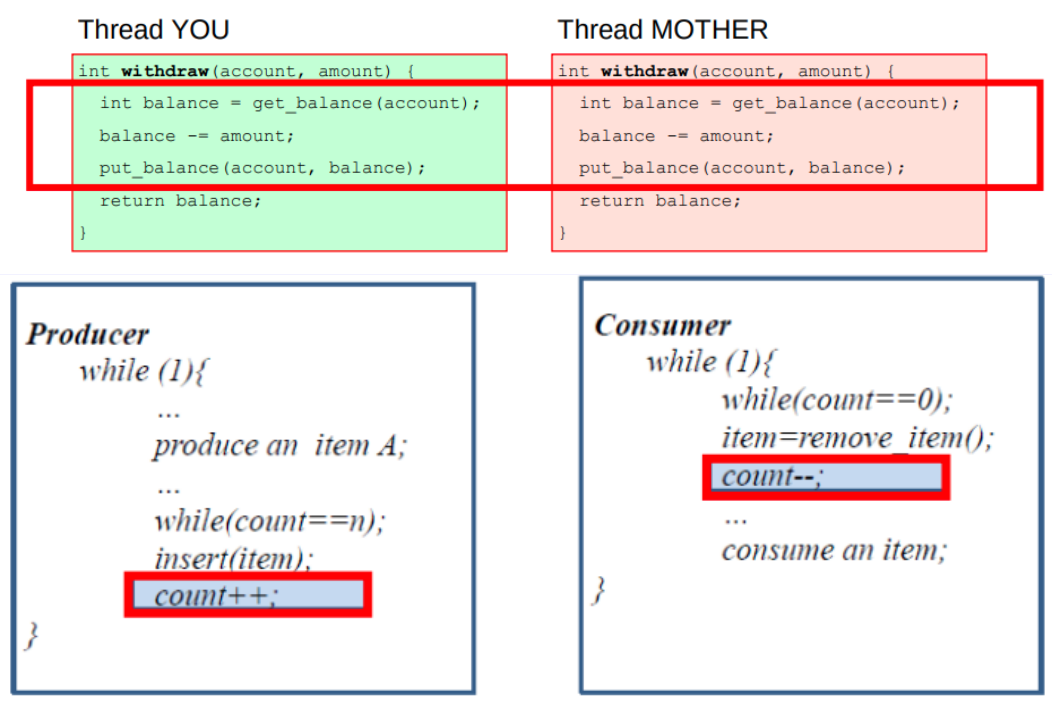
## Example Race condition # 2

Race conditions can also happen between **seemingly** atomic instructions, in reality $count++$ would be executed as multiple assembly instructions:



## Critical region

Critical regions are the areas of code which write/read to shared state, and can cause non-determinism:




## Avoiding race conditions

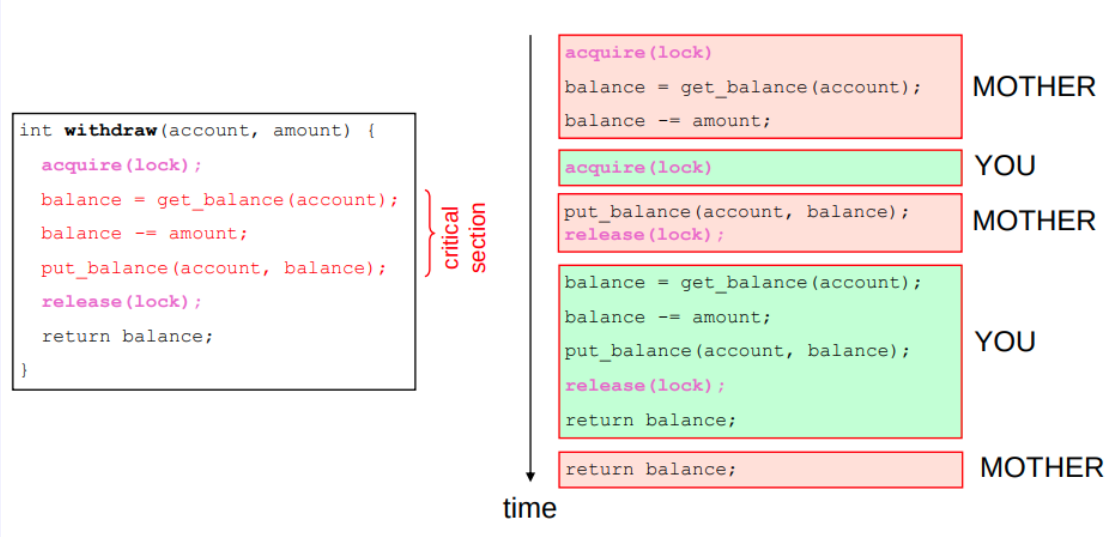
Many factors need to ideally be taken under consideration:

- **Mutual exclusion** - at most one thread/process is in one critical region
- **Progress** - No process running outside its critical region may block other processes
- **Bounded waiting** - No process should have to wait forever to enter its critical region
- **Performance** - Overhead of entering and exiting critical section should be small wrt to work done inside it

## Locks

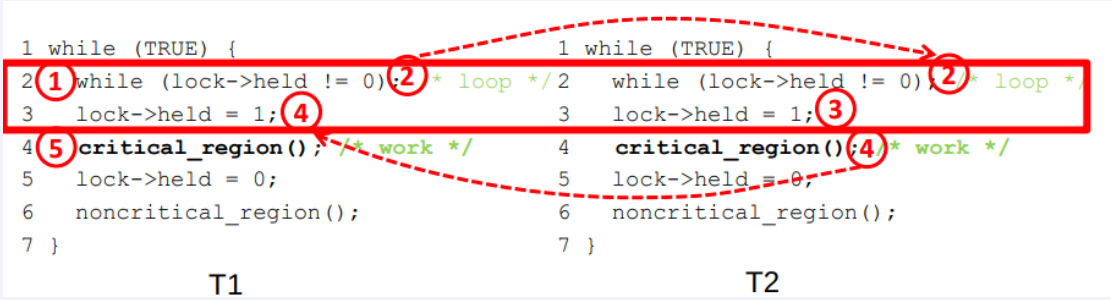
Locks are ways of ensuring mutual exclusion in critical parts of the code.
When one thread holds a lock, other threads wait untill this lock is released.

### Example




### Bad implementation

Still has critical region itself!




### Good implemetation

need **atomic** instruction ideally:

```
struct lock_t {
  int held = 0;
}
void acquire(lock_t* lock) {
   while(test_and_set(&lock->held)) { };
}
void release(lock_t* lock) {
  lock->held = 0;
}
```

\+ simple easy to use and do the job effectively

\- burn a lot of CPU cycles for no reason

# Semaphores and Monitors

**sleep() and wakeup()**

sleep() - caller guves up CPU for some time, untill a thread/process wakes it up

wakeup() - caller wakes up some sleeping thread/process
in practice sleep() might just be calling yield() (giving up CPU scheduling priority), signals might be used for waking up etc..

## Semaphore

A sleep/wakeup version of a lock, which prevents the burning of CPU cycles!

### Implementation

```
typedef struct {
    int value;
    struct thread *list;
} semaphore;
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this thread to S->list;
        sleep();
    }
}
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a thread T from S->list;
        wakeup(T);
    }
}
```

*note: the tread objects might be TCB's as either user or kernel level constructs. Each function must be executed atomically! (spinlocks)*

- threads/processes are blocked at the level of program logic

\+ no busy-waiting (burning cpu)

\+ scheduler is aware about thread/process waiting, other tasks can execute

\- still use spinlocks so some cpu cycles are still burned

\- easy to make mistakes, more complex problems can be solved but require more complex control

\- no guarantee of proper usage, signal() call is not enforced

### Binary semaphore

Value initialzied to 1. Protects single resource

### Counting semaphore

Value initialized to # of protected resources. Positive means resource available, negative or zero = number of waiting for resources or no resources available

### Synchronisation semaphore

Value initialized to 0, 0 means no events pending, **positive** value means there are events pending (waiting in sleep).
*If you have a routine S1 which needs to happen before S2, you call **signal** after S1, and **wait** before S2. This means that any process wanting to execute S2, will have to wait (value = -1) untill S1 is executed by some process which reduces value to 0.*

## Semaphore example

```
Shared variables
const int N=100;
semaphore full=N, empty=0;
semaphore mux=1;
Producer
while (1){
  produce an item A;
  wait(full);
  wait(mux); // acquire(mux)
  insert item;
  signal(mux); // release(mux)
  signal(empty);
}
Consumer
while (1){
  wait(empty);
  wait(mux); // acquire(mux)
  remove item;
  signal(mux); // release(mux)
  signal(full);
  consume an item;
}
```

*note: semaphores here are used to limit burning of CPU when buffer is full or empty, but we still need another semaphore on the actual buffer operations*

## Semaphore example - reading

Sometimes we want to let multiple reading threads in, but only one writer:

```
var mutex: semaphore = 1      ; controls access to readcount
    wrt: semaphore = 1        ; control entry for a writer or first reader
    readcount: integer = 0    ; number of active readers
```

```
writer:
      P(wrt)              ; any writers or readers?
            <perform write operation>
      V(wrt)              ; allow others
```

```
reader:
      P(mutex)                          ; ensure exclusion
        readcount++                     ; one more reader
        if readcount == 1 then P(wrt)   ; if we're the first, synch with writers
      V(mutex)
            <perform read operation>
      P(mutex)                          ; ensure exclusion
        readcount--                     ; one fewer reader
        if readcount == 0 then V(wrt)   ; no more readers, allow a writer
      V(mutex)
```

*note: how the first reader blocks the write semaphore, and the last releases it, while every writer does the same. We still need to lock the manipulation of the shared variables*

## Semaphore example - forever blocked

**Starts with full=N, empty=0, mux=1**

```
Producer                          Consumer
while (1){          (4)           while (1){          (1)
  produce an item A;  (5)           wait(mux);        (2)
  wait(full);       (6)             wait(empty);      (3) Blocked!
  wait(mux);        (7) Blocked!    remove item;
  insert item;                      signal(mux);
  signal(mux);                      signal(full);
  signal(empty);                    consume an item;
}                                 }
```

*note: order of locks is important! bug prone*

## Monitor

Address key usability issues with semaphores, a **programming language construct**. An **abstract data type/class** in which every method automatically acquires a lock on entry and releases it on exit.
includes:

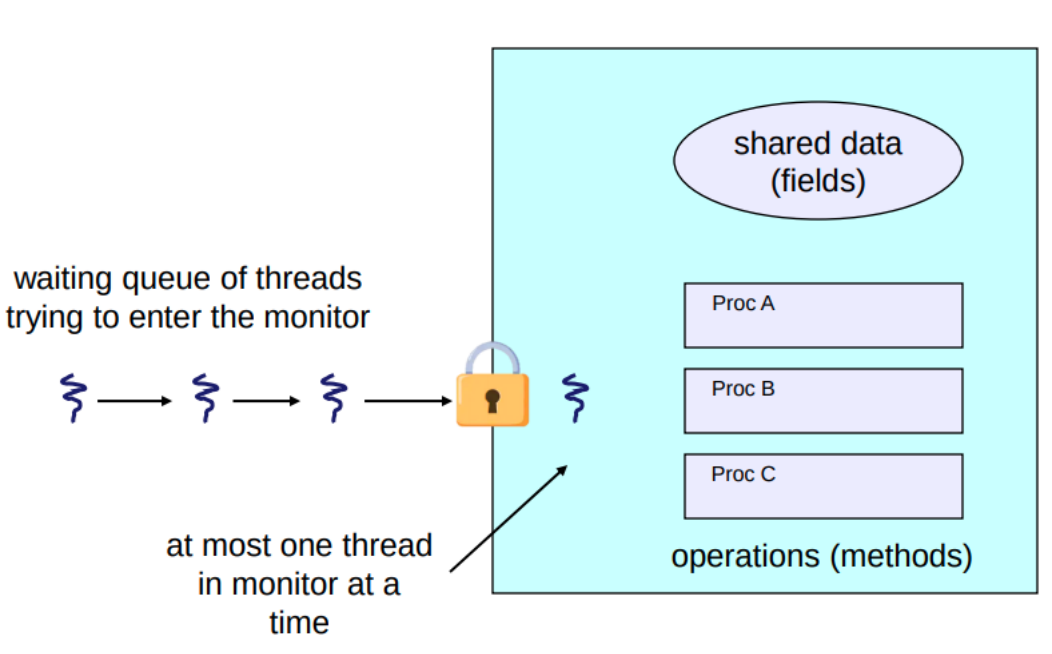
- **shared data** structures
- **procedures** that operate on the shared data
- **synchronisation** between concurrent execution flows that invoke those procedures

data is encapsulated from within:

- protects the data from unstructured access
- prevents ambiguity about synchronisation

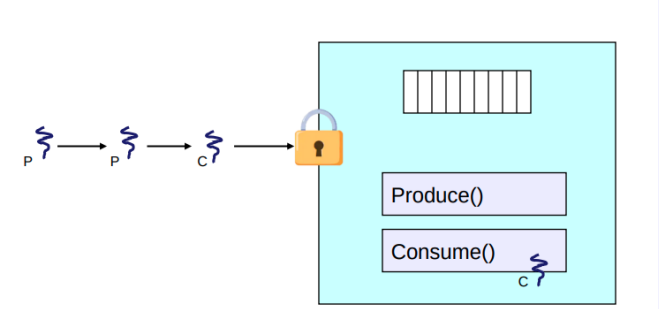
**Automatic** mutual exclusion

- Only **one thread** can be executing inside at any time
- Synchronization is implicitly associated with the monitor (free)
- If a second thread tries to execute a monitor procedure it **blocks** untill the first has left the monitor
- more restrictive than semaphores but easier to use



## Deadlock problems

Not as flexible, what to do in deadlocks ?




- Buffer is **empty**
- Now what?

## Condition variables

- **cond_wait(c)**
  - Release monitor lock, so somebody else can get in
  - Wait for someone else to call **cond_signal(c)**
  - Condition variables have associated wait queues
- **cond_signal(c)**
  - Wake up at most one waiting thread
  - Wake up immediately, signaller steps outside
  - If no waiting threads, signal is lost (difference from semaphores, no history)
- **cond_broadcast(c)**
  - Wake up all waiting threads (for c)

## Monitor example

The compiler can be made to insert enter and exit monitor calls wheerever the programer uses wait and signal calls:

```
Monitor producer_consumer {
  buffer resources[N];
  condition not_full, not_empty;

produce(resource x) {                                   EnterMonitor(m)
   if (array "resources" is full, determined maybe by a count)
        wait(not_full);
   insert "x" in array "resources"
   signal(not_empty);
  }                                                     ExitMonitor(m)

 consume(resource *x) {                                 EnterMonitor(m)
   if (array "resources" is empty, determined maybe by a count)
        wait(not_empty);
   *x = get resource from array "resources"
   signal(not_full);
  }                                                     ExitMonitor(m)
}
```

# Deadlocks

# Virtualization

# Data-center technologies