# Computer Systems: Review



# General Unix and C++

## Unix

- Everything in UNIX like systems is a fucking file.
- Abstraction: as the primary concept underlying all of computation.
- Implementing Abstraction: API, classes, libraries,

- Fucntion pointers are used to create API designs.

```
struct greet_api
{
    int (*say_hello)(char *name);
};
/* Our implementation of the hello function */
int say_hello_fn(char *name)
{
    printf("Hello %s\n", name);
    return 0;
}
int main(int argc, char *argv[])
{
    greet_api.say_hello(argv[1]);
    exit(0);
}
```

# OOPs

1. **Encapsulation**: combine data and functions into a single entity called the class. Allows data hiding.
2. **Inheritence**: Class is based on another class and uses data and implementation of the other class.
    - Single inheritence ($D \rightarrow B$)
    - Multiple inheritence ($D \rightarrow B_1,\ D \rightarrow B_2$)
    - Multi level inheritence ($D_2 \rightarrow D_1 \rightarrow B$)
    - Hierarchical inheritence $\left(D_1 \rightarrow B,\ D_2 \rightarrow B\right)$
    - Hybrid inheritence $\left(D_3 \rightarrow D_1 \rightarrow B,\ D_3 \rightarrow D_2 \rightarrow B\right)$

| Accessibility in public inheritence | Private variables | Protected Variables | Public variables |
|---|---|---|---|
| Accessible from own class | yes | yes | yes |
| Accessible from derived class | no | yes | yes |
| Accessible from second derived class | no | yes | yes |

| Accessibility in protected inheritence | Private variables | Protected Variables | Public variables |
|---|---|---|---|

| Accessibility in protected inheritence | Private variables | Protected Variables | Public variables |
|---|---|---|---|
| Accessible from own class | yes | yes | yes |
| Accessible from derived class | no | yes | yes |
| Accessible from second derived class | no | yes | yes |

Public stuff of base class now act as protected members of the derived class (in protected inheritence).

Similarly, in case of private inheritance, public and protected members of the base class becomes private members of the derived class.

| Accessibility in private inheritence | Private variables | Protected Variables | Public variables |
|---|---|---|---|
| Accessible from own class | yes | yes | yes |
| Accessible from derived class | no | yes | yes |
| Accessible from second derived class | no | no | no |

3. **Polymorphism**: ability to present the same interface for differing underlying forms (data types)

# C++ and similer langauges

Some quirky notes about C++ (and similar languages)

- **Pointers**: yeah that shit.

```
// In Pointers,
int a = 10;
int *p;
int **q; // It is valid.
p = &a;
q = &p;

// Whereas in references,
int &p = a;
int &&q = p; // It is reference to reference, so it is an error
```

- **Dangling pointer**: A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer.
- Void pointers **cannot be dereferenced**. It can however be done using typecasting the void pointer.
- Wild pointer is one which has not been initialized to anything.
- **Garbage Collection**: The basic principles of garbage collection are to find data objects in a program that cannot be accessed in the future, and to reclaim the resources used by those objects. This reduces chances of dangling pointers and memory leaks. But this also has disadvantages since garbage collection consumes a lot of computing resources in deciding which memory to free, even though the programmer may have already known this information.

  C++ doesn't have intrinsic garbage collection, though.

  Algorithm: Mark-sweep is a "stop-the-world" collector, which means that at some point when the program requests memory and none is available, the program is stopped and a full garbage collection is performed to free up space. In mark-sweep, each object has a "mark-bit" which is used during the collection process to track whether the object has been visited. Linear time in the size of heap.

```
mark_sweep_collect() =
   mark(root)
   sweep()

mark(o) =
  If mark-bit(o)=0
    mark-bit(o)=1
    For p in references(o)
      mark(p)
    EndFor
  EndIf
```

```
sweep()
  o = 0
  While o < N
    If mark-bit(o)=1
      mark-bit(o)=0
    Else
      free(o)
    EndIf
    o = o + size(o)
  EndWhile
```

- **Structs vs Classes**

| Classes | Structs |
|---|---|
| Members of a class are private by default. | Members of a structure are public by default. |
| Memory allocation happens on the heap. | Memory allocation happens on a stack. |
| It is a reference type data type. | It is a value type data type. |
| It is declared using the class keyword. | It is declared using the struct keyword. |

- **Namespaces**: Namespaces are basically to allow overloading on steroids. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

```
// first name space
namespace first_space {
   void func() {
      cout << "Inside first_space" << endl;
   }
}

// second name space
namespace second_space {
   void func() {
      cout << "Inside second_space" << endl;
   }
}
```

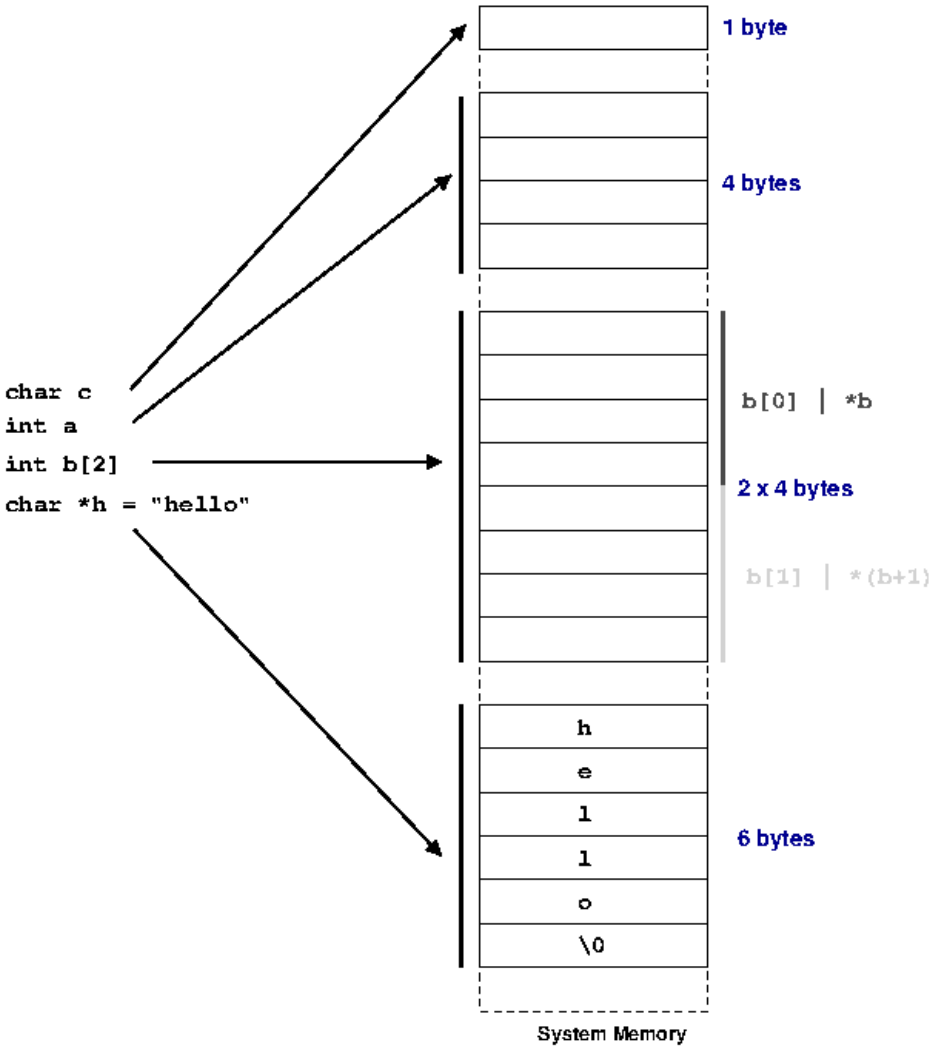
- **Templates**: generalization of functionality based on generalizing type-setting.
  - Stuff done in compile time!
  - Each instance of a template contains its own static variable.
  - Templates are expanded at compiler time. This is like macros. The difference is, the compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.

```
template<typename T, int N>
class Meow {
    // stuff to do
};

template<class T>
class Bhow {
    // stuff to do
};
```

- **Polymorphism in C++**
  - Compile Time: method overloading, operator overloading, templates (in Python, this is done in runtime)
  - Runtime: Function overriding (only a virtual fucntion can be overridden). Virtual function is a member function which is declared within a base class and is re-defined (overridden) by a derived class.

- As long as there is one virtual we need a `vtable`. It has an entry for each function which is virtual which points to the correct code for the function. Each class has its `vtable`, not each instance.
- Each object of a class has pointer to the `vtable` for the class. This pointer is called the `vpointer`.
- This `vpointer` is of generally 4 bytes (in 32 bit systems).

```
#include<iostream>
using namespace std;
class base {
public:
    virtual void print() {
        cout << "print base class\n";
    }
};
class derived : public base {
public:
    void print() {
        cout << "print derived class\n";
    }
};
int main()
{
    base *bptr;
    derived d; bptr = &d;
    // Virtual function, binded at runtime
    bptr->print();
    return 0;
}
```




- Everything inside class is private by default, struct has everything public by default
- Size of object of a class = sum of all attributes + overriding also has vtable so more memory for that
- **Constructors**: always public, types — default, parameterized and copy. C++ Standard allows the compiler to optimize the copy away in certain cases, one example is the **return value optimization**.

```
// Copy constructor
Point(const Point& p1) {
    x = p1.x;
    y = p1.y;
}
```

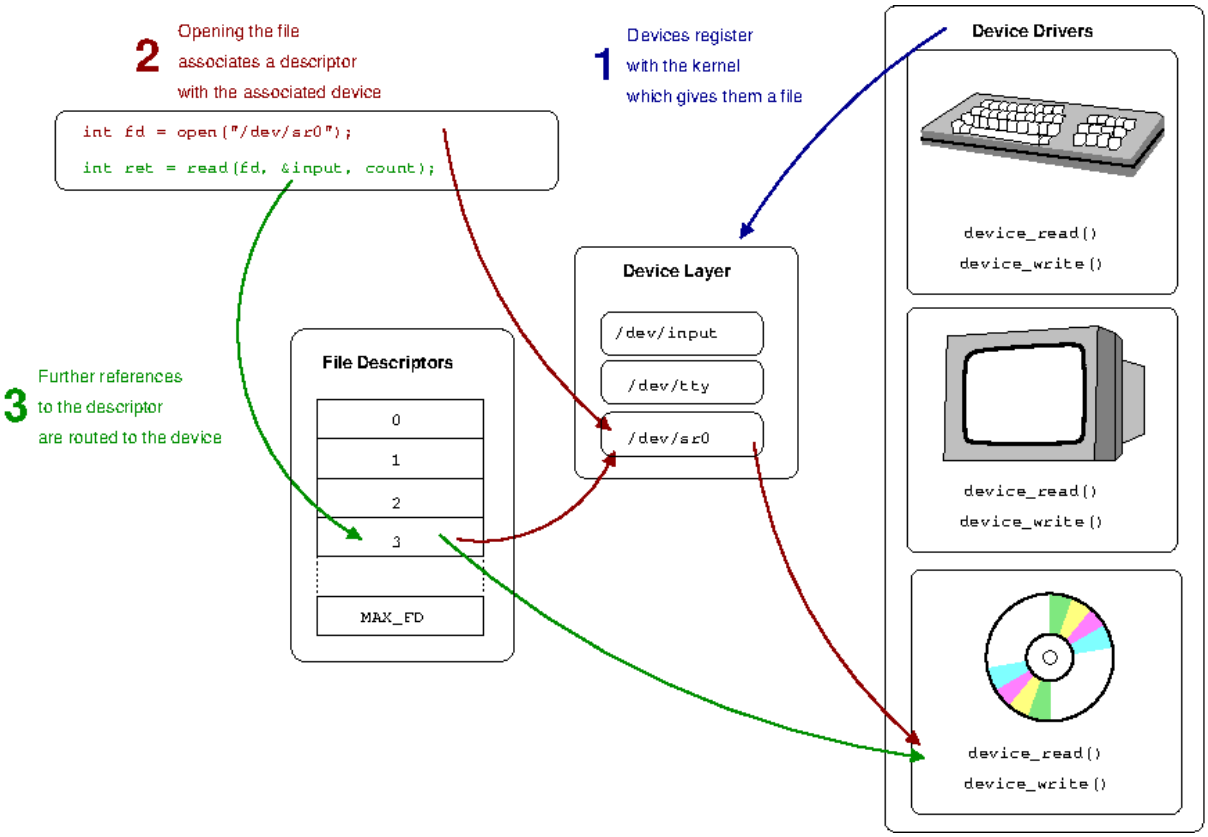
- **Abstract classes**: An abstract class in C++ is a class with at least pure virtual function (or abstract function). It may have other attributes and functions. It is

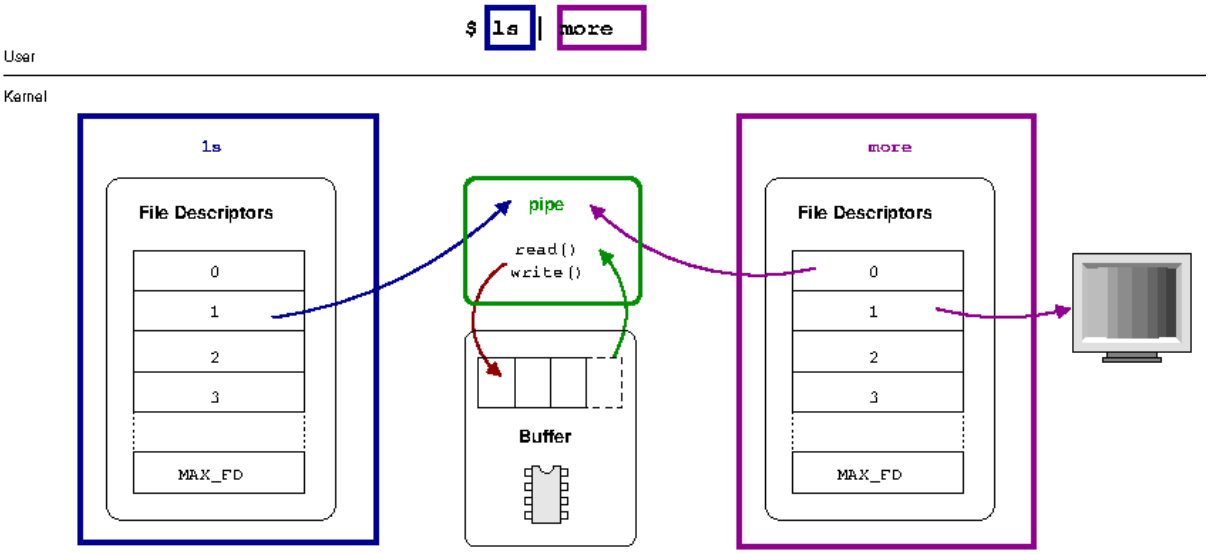
meant to be used as an interface or base class to derive child classes.

- **Interface**: An interface is an abstract class that contains only pure virtual functions.
- We cannot create objects of abstract classes. This will lead to a compiler error. However, we can have a pointer of abstract class type.
- `std::array` is stored on stack, vector on heap

# File Descriptors

| Name | File Number | Description |
|---|---|---|
| stdin | 0 | Input from keyboard |
| stdout | 1 | Output to console |
| stderr | 2 | Error output to console |






# Basic Tree Structures

- Binary Tree
- Binary Search Tree:

  The BST has an important property: every node's value is strictly greater than the value of its left child and strictly lower than the value of its right child. It doesn't allow duplicates.

- Heap (types: binary heap, fibonacci heap):

  BST is an ordered data structure, however, the Heap is not. In computer memory, the heap is usually represented as an array of numbers. The main rule of the Max-Heap is that the subtree under each node contains values less or equal than its root node. Also, the Heap allows duplicates.

# Hash Table with Chaining

Hash Table, also known as hash map or dictionary, is a data structure that implements a set abstract data type, a structure that can map keys to values. A hash table uses a hash function to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found. During lookup, the key is hashed and the resulting hash indicates where the corresponding value is stored.

▼ Code

```
class HashTable:
    def __init__(self):
        self.MAX = 100
```

```
        self.arr = [None for i in range(self.MAX)]

    def get_hash(self, key):
        hash = 0
        for char in key:
            hash += ord(char)
        return hash % self.MAX

    def __getitem__(self, index):
        h = self.get_hash(index)
        return self.arr[h]

    def __setitem__(self, key, val):
        h = self.get_hash(key)
        self.arr[h] = val

    def __delitem__(self, key):
        h = self.get_hash(key)
        self.arr[h] = None

t = HashTable()
t["march 6"] = 310
t["march 7"] = 420
print(t["march 6"])
del t["march 6"]
```

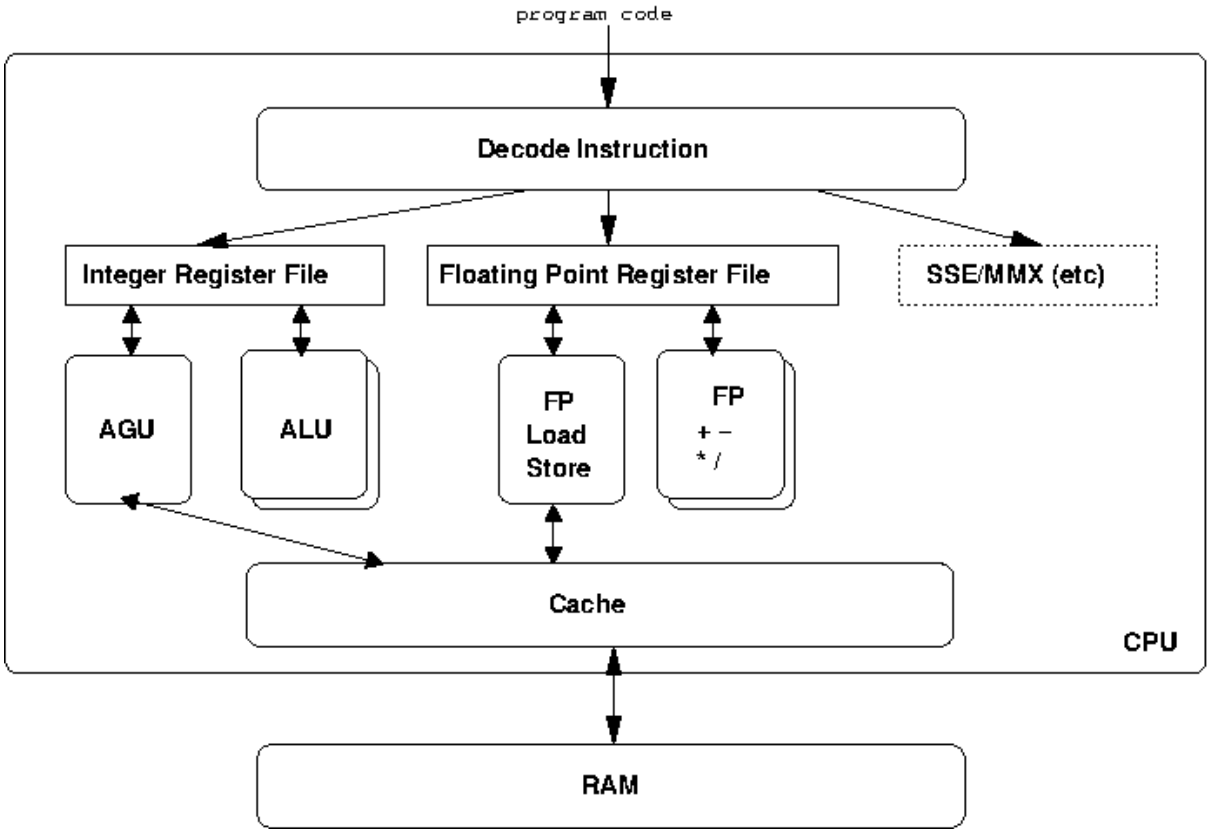
Chaining is implemented to deal with collisions.

▼ Code

```
class HashTable:
    def __init__(self):
        self.MAX = 10
        self.arr = [[] for i in range(self.MAX)]

    def get_hash(self, key):
        hash = 0
        for char in key:
            hash += ord(char)
        return hash % self.MAX

    def __getitem__(self, key):
        arr_index = self.get_hash(key)
        for kv in self.arr[arr_index]:
            if kv[0] == key:
                return kv[1]

    def __setitem__(self, key, val):
        h = self.get_hash(key)
        found = False
        for idx, element in enumerate(self.arr[h]):
            if len(element)==2 and element[0] == key:
                self.arr[h][idx] = (key,val)
                found = True
        if not found:
```

```
            self.arr[h].append((key,val))

    def __delitem__(self, key):
        arr_index = self.get_hash(key)
        for index, kv in enumerate(self.arr[arr_index]):
            if kv[0] == key:
                print("del",index)
                del self.arr[arr_index][index]
```

# Computer Architecture




Stuff happenning inside the CPU:

1. Fetch: get the instruction from memory into the processor
2. Decode: internally decode what it has to do (in this case add)
3. Execute: perform the instruction
    - load values from memory into registers and store values from registers to memory, or
    - operate on values stored in registers

4. Store: store the result back into another register (retiring the instruction)

**Register file** is the collective name for the registers inside the CPU. **AGU** is the address generation unit.

# Pipelining

- Modern architectures are **superscalar**. Fucking club fetch, execute, etc. shit together to parallelize business of several parts of the CPU.
- **Branch prediction**: pipeline flush, predict taken, predict not taken, branch delay slots (this is done because branch instruction )
- Reordering instructions is often done for streamlining execution.
- However, when writing very low level code some instructions may require some security about how operations are ordered. We call this requirement **memory semantics** (security regarding operation order violation).
  - **Acquire semantics**: ensures completion of results of prev instructions.
  - **Release semantics**: ensures that next instructions must see this current result.
  - **Memory barrier**: operations have been committed to memory before continuing
- Pipeline hazards:
  1. Data dependency hazards: data overwritten
  2. Control dependency hazards: instructions overridden or flow is fucked up
- Prevent hazards:
  1. Stalling and memory semantics
  2. Forwarding

# CISC vs RISC

- RISC is Reduced Instruction Set Computer
- CISC is Complex Instruction Set Computer
- RISC is stramlined and simple.
- CISC approach might have only one instruction which takes values from memory adds them and writes the result back. This takes multiple cycles.

- RISC allows pipelining.
- Because the instructions in a RISC processor are much more simple, there is more space inside the chip for registers. And, more registers imply better performance.

### EPIC (Explicitly Parallel Instruction Computing)

- Traditionally organising the incoming instruction stream has been the job of the hardware. Thus the processor needs to look ahead and decide stuff.
- In case of EPIC, the axiom is that there is more information at higher levels which can make these decisions better.
- Analysing a stream of assembly language instructions, as current processors do, loses a lot of information.
- Thus the logic of ordering instructions can be **moved from the processor to the compiler**. This means compiler designers must be smarter.
- This way the processor is also significantly simplified, since a lot of its work has been moved to the compiler.

## Memory

| Speed | Memory | Description |
|---|---|---|
| Fastest | Cache | Embedded in CPU, takes 2-4 cycles to retrieve, limited, leveled (L1, L2, L3) with each level having different speed |
| | RAM | All instructions and storage addresses for the processor must come from RAM. Has latency but is larger. |
| Slowest | Disk | Laegest but slowest. |

**Tradeoff**: speed vs size — the faster the memory the smaller it is.

Caches are very effective because of:

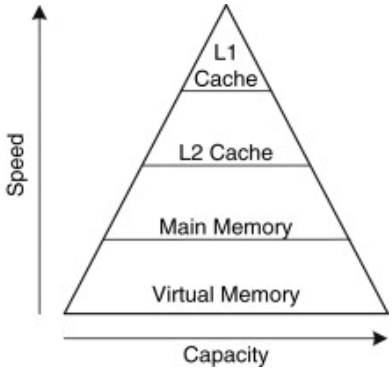
1. Spatial locality: data within blocks will prolly be accessed together.
2. Temporal locality: data used recently will prolly be accessed again.

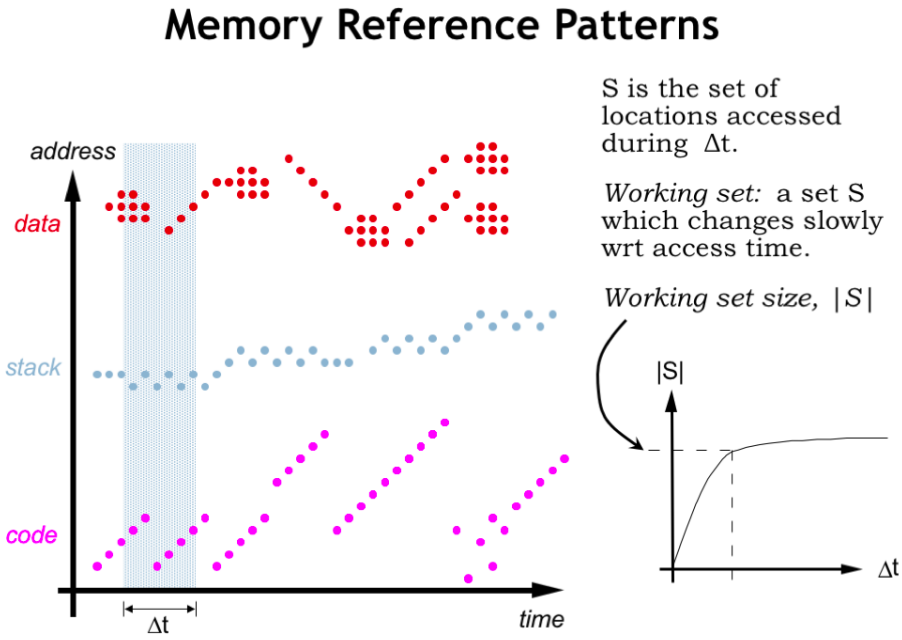
## Cache

- Made up of small chunks of

mirrored main memory called line size (32 or 64 bytes).

- The cache can only load and store memory in sizes a multiple of a cache line.
- If the Cache size will be bigger then **the CPU seek time will be increase** to find out the desirable address.




$$CPU \xrightarrow{address} Cache$$
$$Cache \xrightarrow{data} CPU$$

$$\text{AMAT} = \text{Hit Time} + \text{Miss Ratio} \times \text{Miss Penalty}$$

$$\text{Miss Ratio} = 1 - \text{Hit Ratio} = 1 - \frac{hits}{hits + misses}$$

Here, AMAT stands for Average Memory Acess Time. We wanna minimize this. Also, AMAT is recursively applied in multi-level cache systems.

## Direct Cache

- $2^w$ cache lines require $w$ bits to store index.

| Valid bit | Tag (27 bits) | Data (32 bits) |
|---|---|---|
| ... | ... | ... |
| ... | ... | ... |

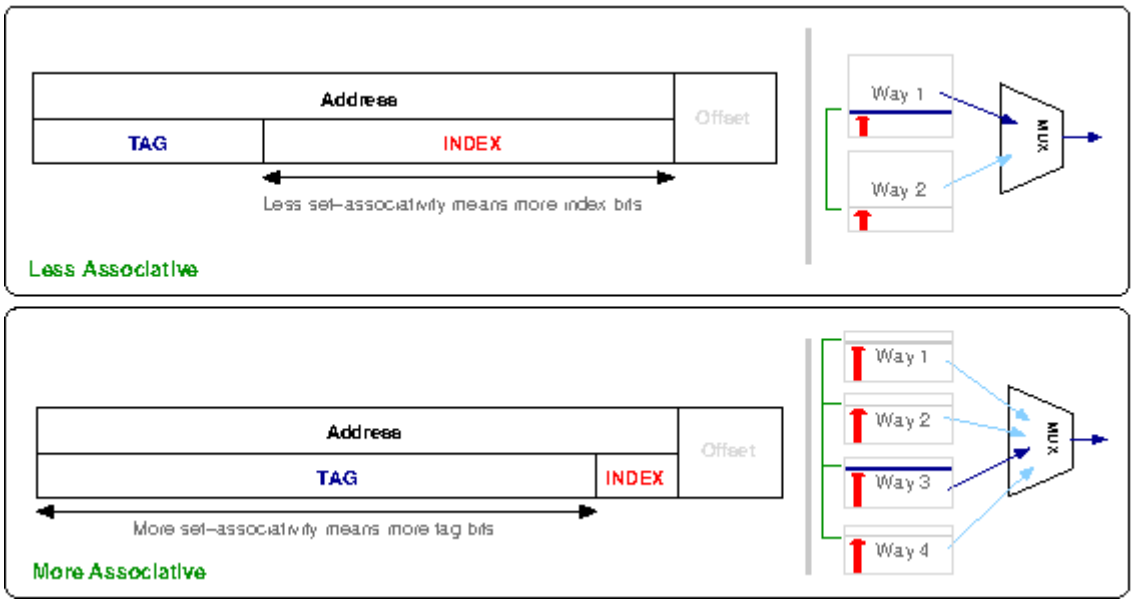
- 32-bit address is given as follows.

| Tag bits | Index bits (w) | Offset bits (2 bits) |
|---|---|---|
| 00010011101110111010110110110 | 010 | 00 |

- We can also store multiple words per data block in a cache line.
- As block size increases, miss penalty increases linearly.
- As block size increases, miss rate decreases first due to exploitation of spatial locality and then increases due to fuck up in temporal locality.
- Overall, AMAT decreases and then increases with increase in block size.
- 32 or 64 bytes serve as optimal block size.
- Direct mapped caches result in a lot of conflict misses.
- Allows cache line to exist only in a singe entry in the cache

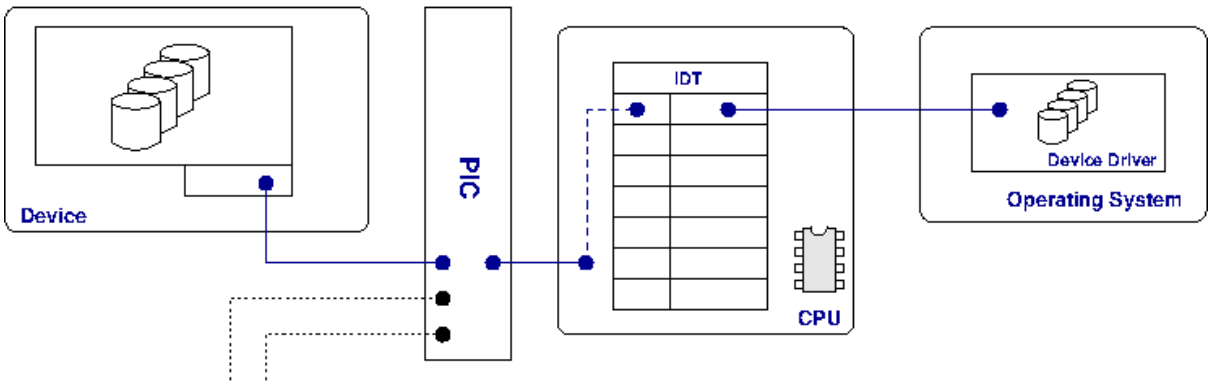
## N-way associative Cache

- Bunch of direct mapped caches working in parallel.
- Expense is higher but conflict misses are lower.
- **Fully associate cache**: no bits are used as index bits, no conflict misses, gotta check the entire TAG and hence is expensive.
    - Allow a cache line to exist in any entry of the caches.




## Peripherals and Buses

- Peripheral buses connect processor to peipherals.
- **Interrupt**: allows the device to literally interrupt the processor and flag some information.
- **Programmable Interrupt Controller** (PIC): Each device has a physical interrupt line between it an one of the PIC's provided by the system. When the device wants to interrupt, it will modify the voltage on this line.




# Operating System

## Basics

There are a lot of processes. They all want time on CPU and access to memory. So how the fuck do we do management of processes? We use an operating system

which implements limited direct execution and serves as a middleman between processes and CPU + Memory.

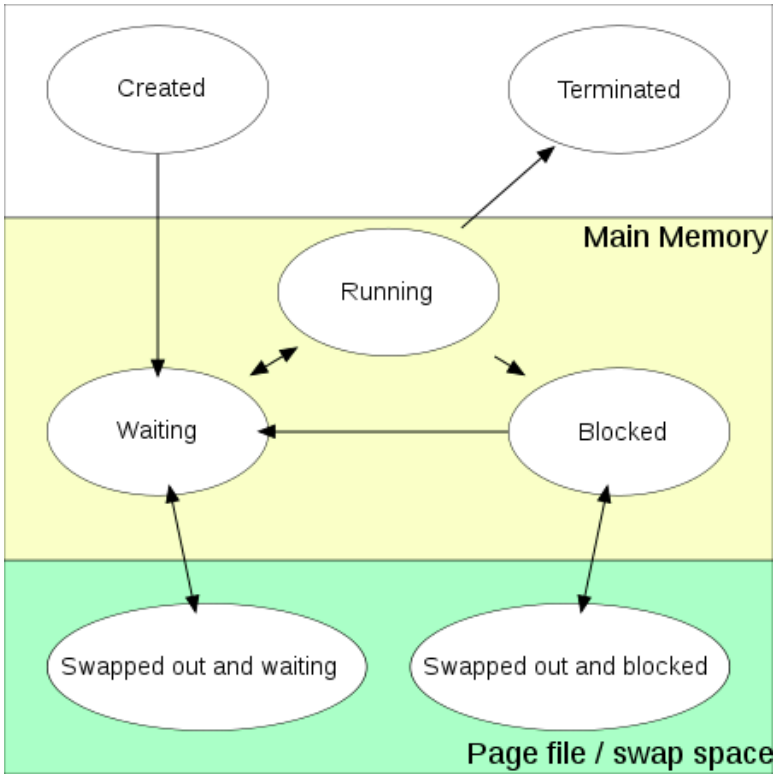
Purpose of an OS is to provide reliable and secure time sharing and memory sharing.

1. Virtualization
2. Security and Reliability
3. Persistancy
4. Concurrency

## Processes

A process is an encapsulation of code, the memory it uses or references and the CPU (time it asks for).

The types of process are shown below. Also, if a process has exited but has not cleaned up by the OS then it is said to be in the zombie state.




- Program is transformed into a process by loading it into memory. Modern OS do it lazily.
- Each process has its own private virtual address space.

- Process consists of:

| Process | |
|---|---|
| Code | Memory |
| | CPU |

1. Memory space
2. Contents of Register
3. IO info (files open)
4. Code

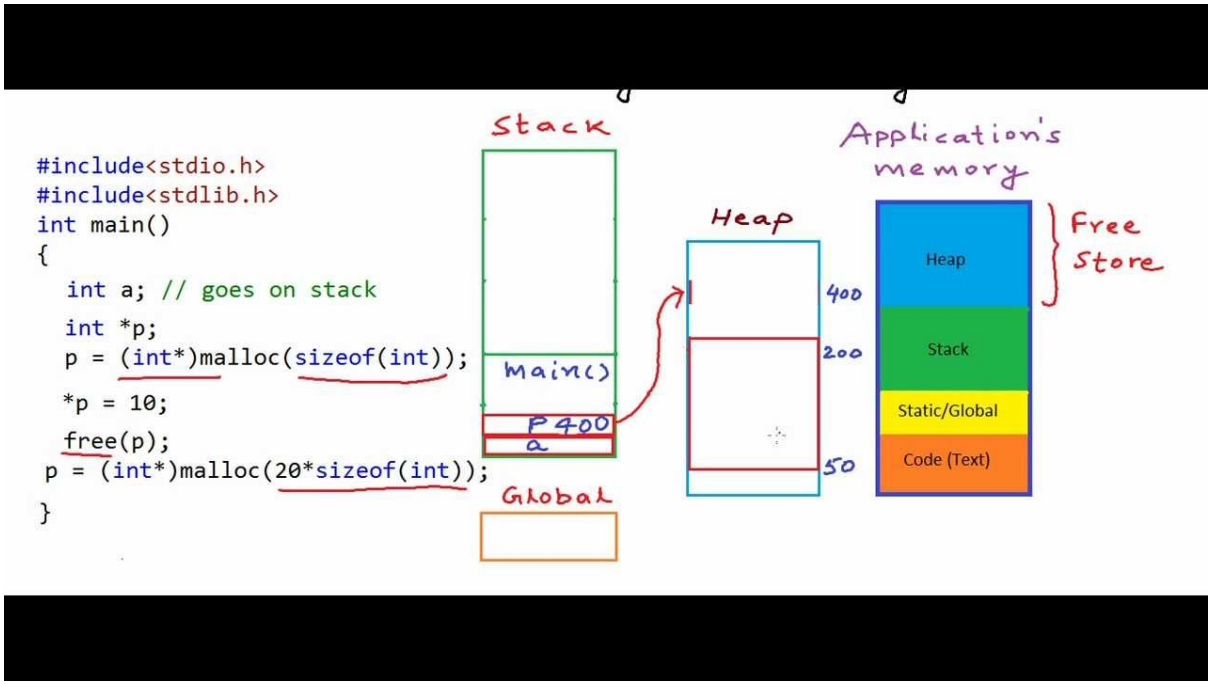
- `open(), close(), write()` are system calls routed through kernel space of file system.
- **Fork system call**:
  - return code 0 in child
  - return code child pid in parent
  - return code < 0 means failed
  - creates exact copy of current program (except return code of fork)
  - clones address space, registers and open files
- **Exec system call**:
  - transforms current process to the given exec
  - address space etc are reinititalized
  - no code after exec call is ever run

    `>>> wc p3.c > a.txt`
  - first close std out and open the file a.txt
  - then call fork
  - then call exec
  - open fd are preserved across the exec call
- **Wait sys call**:
- kil sys call used to send signals like sigint or sigstp
- signal() to catch the signals sent by kill and handle them properly
- System calls
  - A program must execute trap instruction
  - Raises privelage and kernel does the work needed

    - Return from trap is called
    - Each system call has a number
    - Number is mapped to address of which code to be executed
    - mapping is called trap table is set up in boot time
    - kernel mode and user mode implemented at hardware level
- How does OS regain control when a process is running? **Timer interrupt**.

## Scheduling

- FIFO or FCFS: one big job fucks up shit
- SJF: Shortest job first (non-preemptive)
- Preemptive scheduling is where the process is interrupted to stop it to allow another process to run. Co-operative scheduling is where the currently running process voluntarily gives up executing to allow another process to run.
- Turnaround time $=$ Time of completion $-$ Time of arrival
- Response Time $=$ Time of first schedule $-$ Time of arrival
- Round Robin: Runs job for a time slice, RR good for response time but bad for turnaround time
- MLFQ: if same priority do RR and then use history of the job to change its priority. MLFQ assumes a new job to be short and gives it highest priority.
- CFS: Linux completely fair scheduler. Equal virtual runtime to each process.




## Virtualization

- All address we in programs are from virtual address space.
- **Malloc errors**:
  - Use after free (dangling pointers)
  - Double free
  - Free not malloced pointers
- Paging allows for faster access and mapping of physical memory to virtual memory.

# Paging

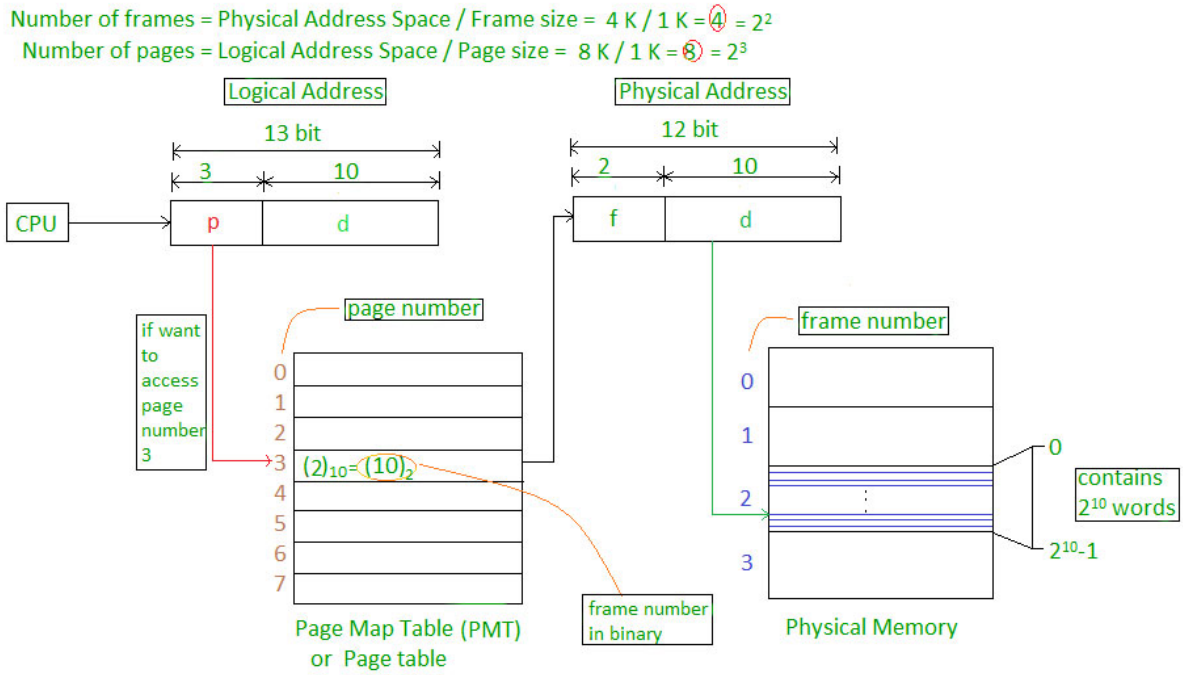
Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non – contiguous.

- Logical Address or Virtual Address (represented in bits): An address generated by the CPU.
- Logical Address Space or Virtual Address Space( represented in words or bytes): The set of all logical addresses generated by a program.
- Physical Address (represented in bits): An address actually available on memory unit.

- Physical Address Space (represented in words or bytes): The set of all physical addresses corresponding to the logical addresses.
- If Logical Address = $31$ bit, then Logical Address Space = $2^{31}$ words = $2$G words ($1$G = $2^{30}$).

The mapping from logical to physical address is done by memory management unit (hardware device):

- The Physical Address Space is conceptually divided into a number of fixed-size blocks, called **frames**.
- The Logical address Space is also splitted into fixed-size blocks, called **pages**.
- Page Size = Frame Size.

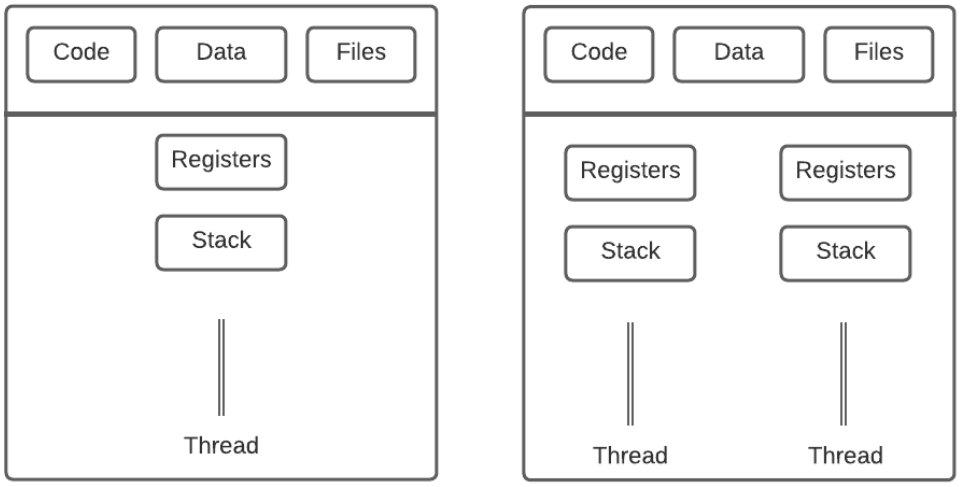



- **Page (or frame) number:** Number of bits required to represent the pages (or frames) in Logical addr Space (or Physical addr space).
- **Page (or frame) offset or word number:** Number of bits required to represent particular word in a page or page (or frame) size of Logical addr space (or Physical addr space).

The hardware implementation of page table can be done by using dedicated registers. But the usage of register for the page table is satisfactory only if page table is small. If page table contain large number of entries then we can use TLB

(translation Look-aside buffer), a special, small, associative, fast look up hardware cache.

## Concurrency




All threads share code, data and files. But, they do have their respective registers and stack.

For threads, we have:

- Inexpensive creation.
- Inexpensive context switching.
- If a thread dies, its stack is reclaimed by the process.

Mutex or Mutual Exclusion Object is used to give access to a resource to only one process at a time. The mutex object allows all the processes to use the same resource but at a time, only one process is allowed to use the resource. Mutex uses the lock-based technique to handle the critical section problem.

Semaphore is an integer variable S, that is initialized with the number of resources present in the system and is used for process synchronization. It uses two functions to change the value of S i.e. wait() and signal(). Both these functions are used to modify the value of semaphore but the functions allow only one process to change the value at a particular time i.e. no two processes can change the value of semaphore simultaneously. There are two categories of semaphores i.e. Counting semaphores and Binary semaphores.

## Processes vs Threads

| Comparison Basis | Process | Thread |
|---|---|---|
| Definition | A process is a program under execution i.e an active program. | A thread is a lightweight process that can be managed independently by a scheduler. |
| Context switching time | Processes require more time for context switching as they are more heavy. | Threads require less time for context switching as they are lighter than processes. |
| Memory Sharing | Processes are totally independent and don’t share memory. | A thread may share some memory with its peer threads. |
| Communication | Communication between processes requires more time than between threads. | Communication between threads requires less time than between processes . |
| Blocked | If a process gets blocked, remaining processes can continue execution. | If a user level thread gets blocked, all of its peer threads also get blocked. |
| Resource Consumption | Processes require more resources than threads. | Threads generally need less resources than processes. |
| Dependency | Individual processes are independent of each other. | Threads are parts of a process and so are dependent. |
| Data and Code sharing | Processes have independent data and code segments. | A thread shares the data segment, code segment, files etc. with its peer threads. |
| Treatment by OS | All the different processes are treated separately by the operating system. | All user level peer threads are treated as a single task by the operating system. |
| Time for creation | Processes require more time for creation. | Threads require less time for creation. |
| Time for termination | Processes require more time for termination. | Threads require less time for termination. |

## Multi-threading

## Semaphores and Mutex

- Mutex uses a locking mechanism i.e. if a process wants to use a resource then it locks the resource, uses it and then release it. But on the other hand, semaphore uses a signalling mechanism where wait() and signal() methods are used to show if a process is releasing a resource or taking a resource.
- A mutex is an object but semaphore is an integer variable.
- In semaphore, we have wait() and signal() functions. But in mutex, there is no such function.
- A mutex object allows multiple process threads to access a single shared resource but only one at a time. On the other hand, semaphore allows multiple process threads to access the finite instance of the resource until available.
- In mutex, the lock can be acquired and released by the same process at a time. But the value of the semaphore variable can be modified by any process that needs some resource but only one process can change the value at a time.

## Deadlock

A deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource.

Deadlock can arise if the following four conditions hold simultaneously (necessary):

- **Mutual Exclusion**: Two or more resources are non-shareable (Only one process can use at a time)
- **Hold and Wait**: A process is holding at least one resource and waiting for resources.
- **No Preemption**: A resource cannot be taken from a process unless the process releases the resource.
- **Circular Wait**: A set of processes are waiting for each other in circular form.

How to be safe? Follow a strategy of avoidance.