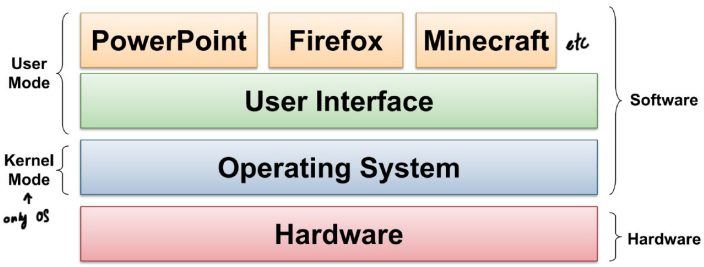
Hardware Virtualisation : All user programs execute as if they have all the resources to themselves

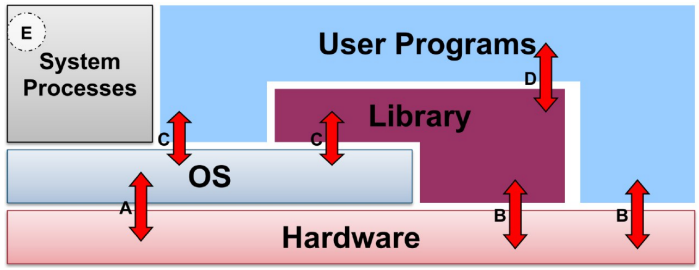
Motivation for OS :

1. Abstraction
   - large variations in hardware configuration
   - high low-level details

   } • efficiency • programmability • portability

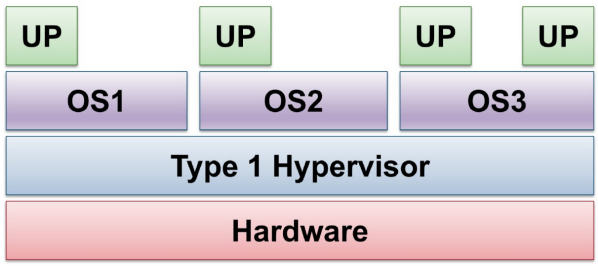
2. Act as a Resource Allocator
   - manage resources ( CPU, Memory, I/O devices, etc. ) ← helps w running multiple programs, BUT does not help run a single program faster
3. Act as a Control Program
   - Prevent errors and improper use
   - Provide security, isolation and protection



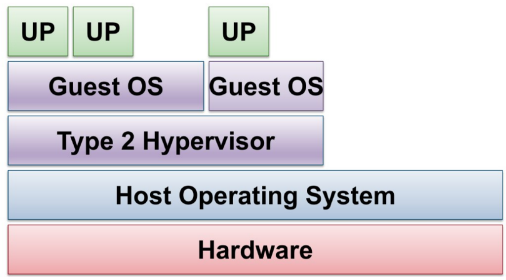




Monolithic OS : Process Management, etc., within the OS
Microkernel OS : Process Management, etc., outside the kernel ( Use IPC to communicate )

## Type 1 Hypervisor




## Type 2 Hypervisor

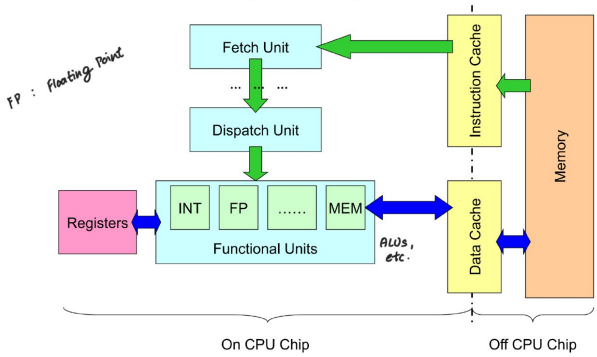


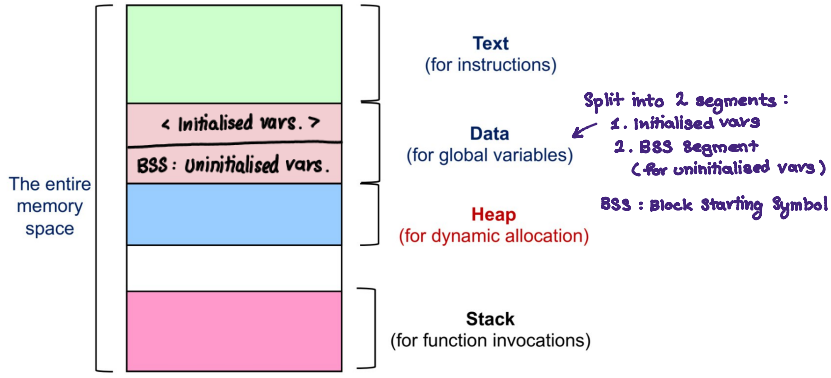

To interact with the OS, system calls are used. These system calls are Synchronous

## Kernel Structures

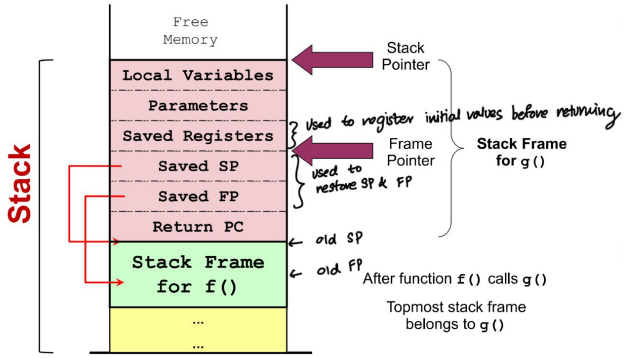
1. Microkernel : Provides basic facilities ( IPC, thread management, etc. )
   - ✓ More robust & extensible
   - ✓ Better isolation & protection
   - ✗ lower performance ( due to increased IPC costs )
2. Monolithic : 1 Big Program, also provides Process & Memory Management, etc. ← traditional approach by most Unix variants & Windows
   - ✓ Well-understood
   - ✓ Provides good performance
   - ✗ highly coupled component
   - ✗ complicated internal structure

## Recap: Generic Computer Organization




* Frame Pointer must be a constant offset from the start of the frame




- On executing function call:
  - **Caller**: Pass arguments with registers and/or stack
  - **Caller**: Save Return PC on stack
  - **Transfer control from caller to callee**
  - **Callee**: Save registers used by callee. Save old FP, SP
  - **Callee**: Allocate space for local variables of callee on stack
  - **Callee**: Adjust SP to point to new stack top; adjust FP
- On returning from function call:
  - **Callee**: Restore saved registers, FP, SP
  - Transfer control from callee to caller using saved PC
  - **Caller**: Continues execution in caller

* With Heap Memory, we can easily encounter scenarios where memory blocks are allocated & deallocated such that there are "holes" in memory.

MEMORY CONTEXT : Memory Space ( Code, Data, Heap, Stack )
HARDWARE CONTEXT : Registers, Pointers, etc.
OS CONTEXT : Process Properties, Resources Used, etc.

FETCH UNIT
- Load instruction from memory
- Location indicated by PC

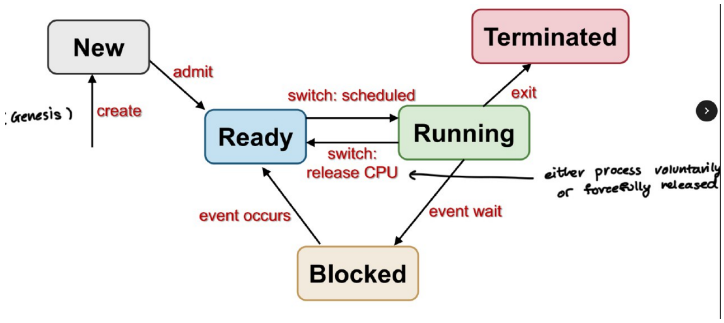
FUNCTIONAL UNIT
- Carry out instruction execution
- Dedicated to different instruction types

Process Table stores PCB
Process Control Block ( PCB )
- stores the entire execution context for a process
- Hardware context stored within is only updated when the process is swapped out

EXCEPTION : Execution of machine level instruction
Synchronous

INTERRUPT : External event trigger
Asynchronous

Traditionally, init processes are the root process ( PID = 1 )




## General System Call Mechanism

1. User program invokes the library call
   - Using the normal function call mechanism as discussed
2. Library call (usually in assembly code) places the **system call number** in a designated location
   - E.g., a register
3. Library call executes a special instruction to switch from user mode to kernel mode
   - That instruction is commonly known as **TRAP**
   - Saves CPU state
6. System call handler ended:
   - Restore CPU state, and return to the library call
   - Switch from kernel mode to user mode
7. Library call return to the user program:
   - via normal function return mechanism

Zombie Process : Child Process has exited but resources are still alive (since Parent might want to acess)
Resources only cleaned up when wait() is called

Orphan Process : Parent exits before child
init process becomes the new parent and kills it by calling wait()

Note that before the orphan process is cleaned up by wait(), it is technically a zombie process

## Copy-on-Write

Processes have their own memory space.
However, it gets expensive copying ē entire memory over each time a process is created
⇒ Copy-on-Write

Memory is copied only when written to

## UNIX Commands

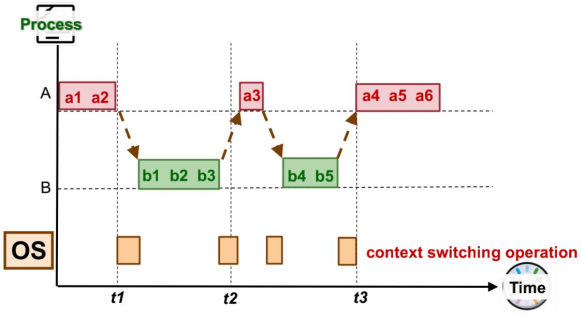
fork() : Returns 0 ← child Process
x ← Parent Process (PID of the child process)

* Wait(NULL) / wait_pid() Required to prevent Zombie Process

exec_ : Execute a different program * NULL at end of arguments

# PROCESS SCHEDULING

Interleaving :




CPU Activity : Computation
I/O Activity : Requesting & receiving sevice from I/O devices

Criteria for Scheduling :
1. Fairness ( No starvation )
2. Utilization

Scheduling Policies : (based on ITI, NOT TQ)
1. Preemptive ( Process can be suspended midway )
2. Non-preemptive ( Process runs until it blocks / gives up CPU )

BATCH-PROCESSING ( No User Interaction , Responsiveness not required )

Criteria :
- Turn-around Time
  finish time - arrival time
- Throughput
  number of tasks finished per unit time
- CPU utilization
  % of time CPU is working on a task

1. First-come-First-served : Tasks stored in a FIFO queue
   Guaranteed no starvation
2. Shortest-Job-First : Always select task w smallest CPU time
   Starvation Possible ⇐ ↑ avg. turn around time
   $Predicted_{n+1} = \alpha Actual_n + (1-\alpha) Predicted_n$
3. Shortest Remaining Time : Preemptive version of SJF.

REAL-TIME PROCESSING
- Deadline to be met
- Usually a periodic process

INTERACTIVE ENVIRONMENT ( Active User Interaction, Responsiveness required )

Criteria : Response Time
Predictability

Timer Interrupt → Interrupts OS Scheduler every x seconds
Checks if Time Quantum is up
Time Quantum → Execution Duration given to a process
Must be multiples of Timer Interrupt

* ↑ Time Quantum : Better utilization, Longer waiting time
↓ Time Quantum : Bigger overhead, Smaller utilization, Smaller waiting time

1. Round Robin : Preemptive version of FCFS
   Process moved to end of queue after time quantum
2. Priority Scheduling : Assign a priority value to all tasks
   Preemptive & Non-preemptive version available
   Possible starvation of low priority tasks
   Possible Solution : Decrease priority after each time quantum
   Give the process a time quantum ( & do not regard in next round of scheduling )
3. Multi-Level Feedback Queue : Priority Scheduling w the following rules ( Increased priority for I/O processes )
   * Assume not preemptive for CS2106
   → If same priority , follow RR rules
   → New job == highest priority
   → Time Quantum fully utilised ⇒ Reduce Priority
   → Job gives up / blocks BEFORE Time Quantum ⇒ Maintain Priority
4. Lottery Scheduling : Give "lottery tickets" to each process
   Process can have x tickets , which can be distributed to child processes

Completion Time : end time - arrival time
Waiting Time : time spent in queue

Priority Inversion: Lower priority task running before higher priority task

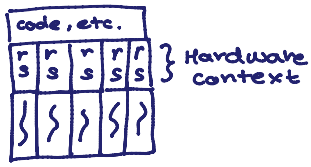
Fork Bomb: More & more processes are created uncontrollably

# PROCESS ALTERNATIVE - THREADS

Processes are expensive → all context information needs to be duplicated (except process-specific info)
→ context switching overhead
→ IPC overhead

Threads → concurrent (interleaving) ∉ ! Parallel
→ Share Memory & OS Contexts
→ Unique information : Thread ID, Registers, etc. (Hardware Context)

∴ Reduced costs




Benefits : Economy
Resource Sharing
Responsiveness
Scalability

* However, requires proper management
- Introduction of race conditions
- Impact of a thread on other threads and process behaviour

## USER THREADS

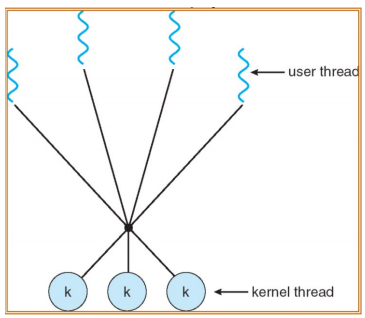
- Implemented as user library
- kernel unaware of threads, only process

✓ Portable
✓ Kernel Mode not required, no IPC
X scheduling is performed at process level

## KERNEL THREAD

- Implemented in OS
✓ Scheduling can be performed at thread level
X Thread Operations are system calls, IPC required
X less configurable

## ∴ HYBRID THREAD MODEL




User Threads are binded to kernel threads
OS schedule on kernel threads

* threadId is not saved / restored during context switching

## UNIX COMMANDS (pthread library)

```
# include <pthread.h>

int pthread_create ( pthread_t * tidCreated ,
                     const pthread_attr_t * threadAttributes,
                     void * ( * startRoutine ) (void *) ,
                     void * argForStartRoutine ) ;

int pthread_exit ( void * exitValue) ;

int pthread_join ( pthread_t threadId ,
                   void ** status ) ;
```

```
gcc x.c -lpthread
```

Datatypes :
pthread_t - Thread ID
pthread_attr - Thread Attributes

# INTER-PROCESS COMMUNICATION

Mechanisms :

- Shared Memory - All changes seen by default
- Message Passing - Changes to be seen should be explicitly defined

## Shared Memory

Process has to create the memory, $m$, and attach to it
Other Processes also have to explicitly attach to $m$
OS involved only in Creation and Attaching

* All processes have to detach before $m$ can be destroyed
( Cleaning up also detaches )

✓ Smaller OS overhead
X Synchronisation needs to be ensured

## Message Passing

Messages stored in kernel memory ( Copy from user buffer to kernel, ← sending
and vice versa ) ← receiving

2 Types :

1. Direct
   - Messages sent by opening a 'socket'
   - Receiver Name & Address needs to be specified
2. Indirect
   - Messages sent to & received from a 'mailbox' (Message Queue)
   - 1-to-Many

# SYNCHRONISATION

Race Conditions : When multiple concurrent threads / processes are 'racing' to use a single shared resource

Synchronisation Problems start to appear with interleaved processes modifying a shared resource

Critical Section : Section of code where multiple processes can modify a resource

∴ To prevent synchronisation problems, only 1 process can be in the CS at a time.

## Correct Implementation

1. Mutual Exclusion : only 1 Process can be in a Critical Section at a time
2. Progress : If there is a waiting process and no process in the Critical Section, the waiting process should be allowed to run
3. Bounded Wait : There exists an upperbound on the number of processes that can enter before an arbitrary process
4. Independence : Processes not in Critical Section should not block waiting processes

## Incorrect Implementation

1. Deadlock : All processes blocked
2. Livelock : Processes not blocked but change of state occurs indefinitely
3. Starvation : Some processes blocked forever

**Test and Set** : An Atomic instruction which ① loads the current context in a MEM location into a REGISTER
② stores a '1' into MEM location

```
void enterCS (int * lock) {
    while (TestAndSet (lock) == 1);
}
```

```
void exitCS (int* lock) {
    * lock = 0;
}
```

**Peterson's Algorithm** : Uses both an int / bool var, 'turn', and an array, 'want' to keep track of who is running

- Just using 'turn' : Only works when we have exactly 2 processes
- Just using array : Possibility of deadlocks

∴ Use 'turn' to break deadlocks

P0

```
want[0] = 1;
turn = 1;
while (want[1] && turn == 1);
<CS>
want[0] = 0;
```

P1

```
want[1] = 1;
turn = 0;
while (want[0] && turn == 0);
<CS>
want[1] = 0;
```

Assumption : writing to 'turn' is an atomic operation

Limitation : Does not work with > 2 processes
Busy waiting occurs
No high-level abstraction, logic must be written by dev.
Not general

∴ Introducing SEMAPHORES

**Semaphore** : Generalised Synchronisation Mechanism ( ē usage of it is also called Mutex )

Can be initialised to any non-negative value initially ( similar to a quota )

Uses 2 Atomic Operations : wait (s)
signal (s)

Allow only 1 process in CS at a time
⇒ Always initialise to '1'

wait (s) : Blocks if $s \leq 0$ ( sends process to sleep ) signal (s) : Increments s

Decrements s after block passes

* This specifies the behaviour of semaphores , not the implementation

Given $s_{initial} \geq 0$ , $s_{current} = s_{initial} + \#signal(s) - \#wait(s)$

$\#signal(s)$: no. of signal(s) executed

$\#wait(s)$: no. of wait(s) completed

* With improper usage , deadlock is still possible:

| | |
|---|---|
| wait (P) ① | wait (Q) ② |
| wait (Q) ③ | wait (P) ④ |
| ⋮ | ⋮ |
| signal (Q) | signal (P) |
| signal (P) | signal (Q) |

# Memory Management

## Properties of Memory
1. Speed of access
2. Size of memory
3. Cost of memory

## Types of Address
1. Logical : A process / program's view of the memory space
2. Physical : The physical hardware addresses

* Mapping between logical and physical addresses required

## Types of data in a process
1. Transient Data : Data which is valid only for a short duration
   e.g. local variables, parameters
2. Persistent Data : Data which is valid for the duration of a program
   e.g. global variable, constant variables

## Tasks handled by the OS
1. Allocating memory for new processes
2. Managing & Protecting memory space
3. Providing memory syscalls

## Memory Abstraction
* W/O memory abstraction, processes always point to the same physical address for a particular variable
  ∴ It is not possible to ensure independence of variables across processes

### Methods
1. Address Relocation : Recalculating memory references by adding an offset to all memory references in other processes
   - X Slow loading time
   - X Not easy to distinguish memory reference from normal integer constants
2. Base + Limit Registers : (Base) Maintain the base value from which all memory references are calculated as an offset of
   (Limit) Additionally, maintain the range of memory space a process can access

   * A segmentation fault occurs when a process tries to access a memory reference beyond the limit

Load - Store Execution Model : Only Load / Store instructions can access memory
Other instructions can only access registers
Most CPUs, except Intel, AMD, follow this model

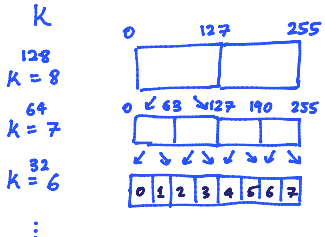
Multitasking : Multiple processes allowed in the physical memory at a time
(to allow context switching)

## Memory Partitioning
### 2 Types
1. Fixed - Size : Each process has the same partition size
   Causes **Internal Fragmentation** when processes do not require so much memory space
   - ✓ Easy to manage
   - ✓ Fast allocation
2. Variable - Size : Each partition size is different, depending on the process requirements
   Causes **External Fragmentation** when there are many (small) free spaces in memory
   - ✓ Flexible
   - ✓ No Internal Fragmentation
   - X Needs to maintain more information in OS
   - X Slower - need to locate an appropriate region

### Allocation Algorithm
1. First - Fit : Use the first free memory space large enough
2. Best - Fit : Use the smallest free memory space large enough ← Could result in many small 'holes'
3. Worst - Fit : Use the largest free memory space ← Reduces possibility of many small 'holes'

### Buddy Allocation
When allocating space, split the free block into 2 equal halves recursively until it is the smallest $2^x$ block able to store the new process memory

* When merging partitions together, we have to ensure we merge the correct partitions




i.e., merge partitions 0 & 1, rather than 0 & 7

This gets harder when we can randomly allocate partitions out-of-order as long as we find the best-fit

## How to find the Buddy?

Notice : The original block is of size $2^k$, and each split block is of size $2^j$ where $0 \leq j \leq k$

∴ The start address of each block is also of value $2^j$

$Start_A = 0\ (000000_2)$ of size 32

↙ $Start_B = 0\ (000000_2)$ of size 16

↘ $Start_C = 16\ (010000_2)$ of size 16

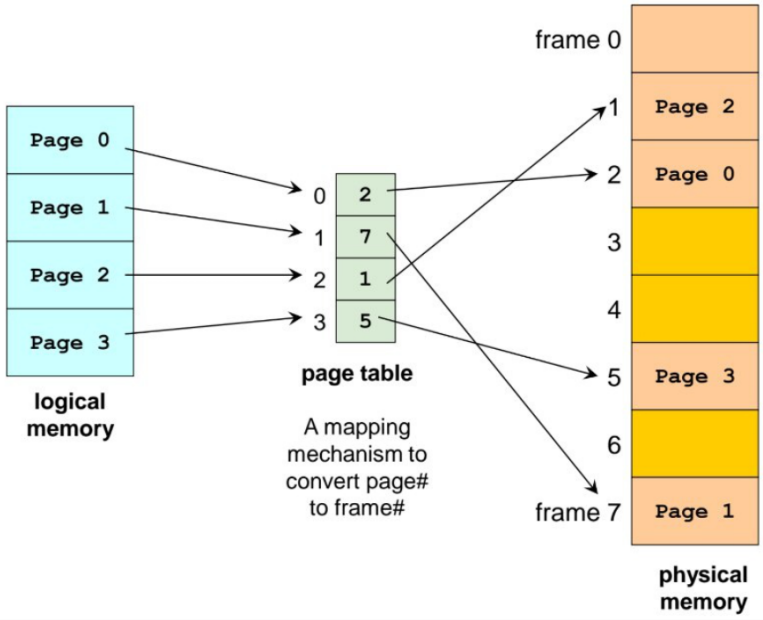
∴ Two blocks, B and C, are buddies if all bits leading up to the $j^{th}$ position are the same, whereas the bit at the $j^{th}$ position are complements of each other ( assuming B and C are of the same size )

# Disjoint Memory Schemes

- Works with non-contiguous memory using a **Paging Scheme** ← The rest of this topic will mainly deal with this scheme
- Both the Physical and Logical Memory is split into regions of the same size ← Note that regions are all of the same size
  The regions are known as Physical Frame and Logical Page respectively
- The pages of a process are loaded into any available physical frame

* While the Physical Addresses are disjoint, the Logical Page is continuous

## Paging: Illustration




We use a **Page Table** to map the Logical Page to the Physical Frame

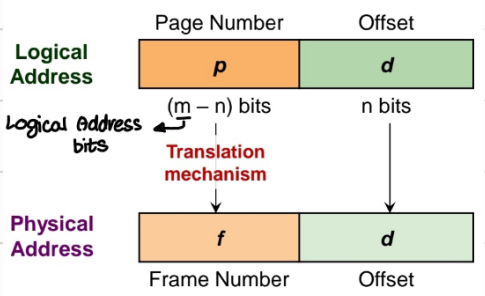
The formula to locate the Physical Address is:

$PA = (F \times \text{Physical Frame Size}) + \text{Offset}$

where,

F = Physical Frame Number

Offset = Displacement from the beginning of the Physical Frame




✓ No External Fragmentation

✓ Clear seperation of logical and physical address space

✗ Internal Fragmentation still possible

✗ 2 Memory Access required for each lookup (page table & physical address)

**Translation Look-Aside Buffer (TLB)** ← Caching yay!s

- Acts as a cache for a few page table entries (typically 32 - 1024) ← Double-Check this
- Fully-associative, that is, no conflict misses, only capacity / cold misses
- Part of a process' hardware context
  → During Context Switching, the TLB will be flushed ← ↑ TLB misses

TLB-Hit : Physical Frame number is retrieved and Physical Address generated

TLB-Miss : Additional Memory Access to access the full page table, and update TLB

## TLB: Impact on Memory Access Time

- Suppose:
  - TLB access takes 1ns
  - Main memory access takes 50ns
  - What is the average memory access time if TLB contains 40% of the whole page table?

w/o TLB : 50ns + 50ns = 100ns

- Memory access time
  = TLB hit + TLB miss
  = 40% x (1ns + 50ns) + 60%(1ns + 50ns + 50ns)
  = 81ns

  (40% x (1ns + 50ns): Hit; 60%(1ns + 50ns + 50ns): Miss)

- Note:
  - Overhead of filling in TLB entry and impact of cache ignored.

## Access Right Bits & Valid Bit used to protect memory between processes

- Can be maintained in the Page Table
- Access Right Bits : Writable, Executable, Readable } Follows usual 1 & 0 to True / False convention
- Valid Bit : Whether the page is valid for access
  OS sets this when a process is running
- Memory access checked against both bits

## (Back to) Copy-on-Write

- To reduce overhead as a result of copying memory for each process
- When a new process is created, copy the Parent's page table to the child's page table;
  Both processes use the same page table and physical addressing space as long as there is no write operations;
  If there is a write operation, only that specific Physical Frame & corresponding page table entry is altered;
  the child points at a new Physical Frame and its page table is updated

## Segmentation Scheme

Motivation : Each memory region has different uses, and thus, different requirements

i.e., Some regions (Stack, Heap, etc.) grow and shrink ⇒ difficult to manage in contiguous memory space

Basic Idea : Segmenting ē memory space of a process based on ē region (i.e., Data, Stack, etc.)

⇒ Logical Memory space is now a collection of segments

Each segment has :

1. Name : For ease of reference (id)
2. Limit : Indicates the range of the segment

} All Memory References are now : Segment Name + Offset

✓ Each segment is an independent contiguous memory space
⇒ Grow / Shrink and Protected / Shared separately

✗ Requires variable-size contiguous memory regions
⇒ External Fragmentation

## < combining Segmentation & Paging >

### Segmentation w Paging

- Each segment is now composed of several pages instead of a contiguous memory region
  (Each segment has its own page table)
- Growing / Shrinking is done by allocating & adding / deallocating & removing from the page table respectively

# Virtual Memory Management

Motivation : What if our Physical Memory is not large enough to contain the complete memory space of a process?

Basic Idea : Some parts of the Logical Address Space is stored in the Physical Memory, but others are stored in the Secondary Storage (Disk Space) ← Store data not immediately required in Secondary Storage

Extension of the Paging Scheme ← Instead of translating from logical address, translate from virtual address to physical address

## 2 Types of Pages

1. Memory Resident ← Physical Memory
2. Non-Memory Resident ← Secondary Storage

} Additional bit now required to denote where the page is stored

Steps :

1. Check Page Table
   (If yes, access and exit. Else, continue to other steps)
2. Page Fault : Trap to OS
3. Locate page X in Secondary Storage
4. Load page X in a Physical Memory
5. Update page table
6. Go to Step 1 to retry

} Hardware-Level

} OS-Level (Additional info. required to denote where the page is located in Secondary Storage)

Recall that Access to Secondary Storage is much, much slower than to Physical Memory (RAM)

∴ If too many Page Faults occur, "Thrashing" occurs (See : Frame Allocation)

However, Thrashing is unlikely to occur due to the concepts of Temporal & Spatial Locality

Now, also notice that it is not efficient to load all pages on startup ⇒ Demand Paging

Demand Paging : only load to Physical Memory on Page Fault

✓ Fast Start-up Time

✓ Small Memory Footprint

X Multiple Page Faults at the start (Process slightly slower behaviour)

X May cause thrashing for other processes ← "Cascading Page Fault"

## Page Table Structure

1. Direct Paging
2. 2-Level Paging
3. Inverted Page Table

## Page Replacement Algorithm ← Implicit Assumption of Local Replacement

* If a page is 'dirty' (has been modified), the Secondary Storage also needs to be updated
  ↑ Denoted by a dirty bit

1. Optimal Page Replacement ← Used mainly as a 'best-case' comparison for other Page Replacement algorithms
2. FIFO ← Doesn't exploit Temporal Locality (See : Belady's Anomaly)
3. LRU : needs to be done in Hardware
   high overhead due to having to keep track & search for appropriate page
   exploits temporal locality ⇒ generally good approximation of OPT
   two implementation approaches: Counter (incremented for every memory reference)
   'Stack' (more accurately, a Queue)
4. Second-Chance Page Replacement (aka. Clock) :
   - FIFO Algorithm, with the addition of a Reference Bit to account for temporal locality
   - If accessed, Reference Bit = 1
     Else, Reference Bit = 0

   * If all pages have Reference Bit = 1, this Algorithm loops back around and essentially degenerates into a normal FIFO Algorithm

## Frame Allocation

Deals with the following Question : How do we allocate N Physical Frames to M Processes, all competing for frames ?

2 Approaches :

1. Equal Allocation : Each process gets $\frac{N}{M}$ frames
2. Proportional Allocation : Each process gets $\left(\frac{size_p}{size_{total}}\right)(N)$ frames, where $size_p$ = Size of process, $size_{total}$ = Total size of all processes

Thrashing occurs as a result of Insufficient Physical Frames
⇒ Heavy I/O is required to bring non-resident pages into RAM

Local Replacement : Pages to be replaced are selected among the pages of the process that causes the page fault
Frames allocated to a process remains constant

✓ Performance is stable across multiple runs
✓ Thrashing is limited to one process ← That one process can hog I/O, degrading the performance of other processes
X May hinder the progress of a process if no. of frames allocated to the process is not enough

Global Replacement : Pages to be replaced are selected among all physical frames, that is, a process P can evict the frames of another process Q

✓ Allows for self-adjustment between processes
X Badly behaved processes can affect others by hogging pages
X The frames allocated to a process can differ from run to run
X Can cause Cascading Thrashing ← Thrashing process 'steals' pages from another process and cause it to thrash

Notice that some program code can be sectioned into different parts whereby the set of pages referenced by a process is relatively constant in a period of time ( locality )

I.e.,

```
for i in range ( 100 000 000 ):
    ⋮
```
} Locality 1 : Pages are constant

```
for i in range ( 100 000 000 ):
    ⋮
```
} Locality 2 : Pages are constant

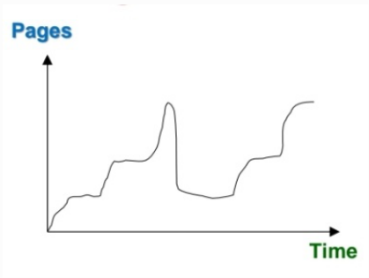
Using the above observation on locality, we can infer the following :

1. In a new locality, a process will cause page fault for the new set of pages
2. With the set of pages in a frame, there are no / few page faults within the same locality

Thus, we can define a Working Set Model, which defines a Working Set Window ( an interval of time ), such that :

$$w(t, \Delta) = \text{active pages in the interval at time } t$$

Therefore, we simply have to allocate enough frames for the pages in $w(t, \Delta)$ to reduce the possibility of a page fault




Transient Region : Working Set changes in size

Stable Region : Working Set about the same

* The accuracy of the working set model is directly affected by the choice of $\Delta$
  If $\Delta$ too small, may miss pages in the current locality
  If $\Delta$ too large, may contain pages from a different locality

# File Systems (Introduction)

Motivation: Physical Memory (RAM) is volatile and non-persistent
⇒ External Storage needed to store persistent data
⇒ Direct access to storage media is not portable (& is hardware specification and organisation dependent)
∴ File Systems abstract the interaction with the physical media,
and provides high-level resource management

Generally, File Systems have to be:
1. Self-contained: "plug-and-play"
all data has to be able to be stored in the File System
2. Persistent
3. Efficient

Files are actually logical units of information created by process(es) ← Directories are special files that contains data about other files
It contains:
1. Data: Information structured in some format
2. Metadata: Additional Information, i.e. File Attributes

Operations Required: Read, Write, Open, Close, etc.

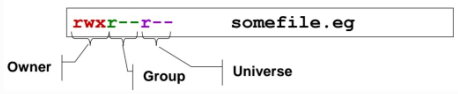
* Information kept for an Open File:
- File Pointer: Current location in file
- Disk Location
- Open Count: Number of processes opening this file
⇒ used to know when to commit & clear buffer, and remove from Open File Table

## File Protection Mechanisms

Different types of file access: Read, Write, Execute, etc.

General Method: Access Control List, restrict access to files based on user identity
✓ Very Customizable
X Too much information associated with a file
⇒ Results in large directories & directory entries
⇒ Directories are usually stored in memory
⇒ Slow searching of files

∴ To reduce associated information, classify users into 3 classes: Owner, Group, Universe
Give access based on these classes, instead of to individual users

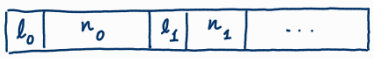



2 Types of ACL:
1. Minimal: same as permission bits
2. Extended: added named users / groups

## File Data Structure

Each file record is an Array of Bytes, s.t. each byte has a unique offset (distance) from the start of the file

1. Fixed-Length Records:
An array of records, which can grow/shrink
We can jump to any record easily
⇒ Offset of the $N^{th}$ record = Record Size * (N-1)
(Database files structured this way)
2. Variable-Length Records:




✓ Less redundant space
✓ Flexible
X Harder to search for records ← Sequential Search required
X Slower

## File Data Access Method ← OS provides these as system calls

1. Sequential Access (traditional method)
Cannot skip ahead ⇒ Slower
2. Random Access
Data can be read in any order
Can be done in 2 ways:
1. Seek(Offset): uses a file pointer (more common method currently)
2. Read(Offset): Always starts from beginning of file
3. Direct Access
For files that contain Fixed-Length Records
Allows for Random Access to any record directly
Particularly useful for when there is a large volume of data

* Organising Open Files is usually done in 1 of 2 ways: ← When a file is created, it is added to both the System OFT and the Process OFT

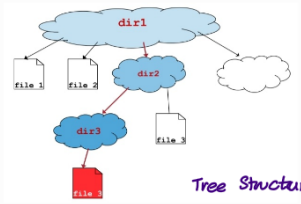
1. System-wide Open-file Table : 1 entry / unique file across all users and processes
2. Per-process Open-file Table : 1 entry / file in a process

Notice that, if multiple processes point to the same file entry, the operation of 1 process can affect other processes

Directories (folders) provide logical grouping of files (user view), and helps keep track of files (actual system usage) ← This is especially true for if multiple files have the same name, but are in different directories

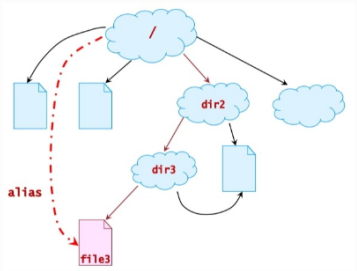
Structuring directories can be done in different ways:

1. Single-Level :
   Basically only Root exists
   Not good for large file systems, usually only used in CDs
2. Tree Structure :
   Directories recursively embedded in other directories
   2 ways to refer to a file : Absolute and Relative Pathnames



Tree Structure

3. Directed Acyclic Graph :
   Happens when files can be shared across directories, i.e., via Hard Links or Symbolic Links




Unix Index Node

Hard Links : All directories have seperate pointers to the INode
- ✓ Low overhead
  ⇒ only pointers added to directory
- X Difficult to handle deletion
  ⇒ Solved by using Reference Count in INode, only delete INode & file in disk when Reference Count = 0

Symbolic Link : Special File created
- ✓ Simple Deletion
- X Large overhead (takes up actual disk space)
- X Requires care when setting up (cycles are possible)

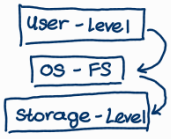
4. General Graph ← Not desirable due to difficult traversal
   Symbolic Link to a directory is possible
   ⇒ Especially useful for heavily-nested directories, allowing for easier access (i.e., Network Drives)
   X Hard to determine when to remove a file / directory

# File Systems (Implementation)

* The idea for HDDs & SSDs is similar. Mainly, they differ in that the location of the logical block does not matter in SSDs.

## General Structure

1D Array of logical blocks, which are mapped to ≥ 1 Disk Sectors




Mapping logical blocks to folders / files

Mapping data to logical blocks

## Unix Index Node (INode)

Directory: | File Name | INode no. |

Contains: Metadata
- Information on user who created the file
- Timestamps (Created, Updated, etc.)
- No. of logical blocks
- Reference Count
- etc.

## Disk Organisation

Master Boot Record (MBR)

Located at Sector 0, followed by ≥ 1 Partitions ← 1 partition = 1 (completely indepent) File System

Stores:
1. Basic Boot Information
2. Information about the different partitions (including Partition Table)

Partition Table: Tells where each partition starts and ends

A File System generally contains:
1. OS Boot Information
2. Partition Details → Total Number of Blocks
3. Directory Structure → Number & Location of Free Disk Blocks
4. File Information
5. Actual File Data

/swap Partition: Section of Disk Drive dedicated to Virtual Memory

## File Implementation

How do we allocate file data on the disk?

Files are simply a collection of logical blocks

⇒ File Size must be a multiple of number of logical blocks, else Internal Fragmentation may occur ← number? or file size?

Criteria (for good File Implementation):
- Keeps track of data location (in terms of logical blocks)
- Efficient access
- Efficient Disk Space Usage ← Persistent data has to be stored on Disk Drives

2 Common approaches to store file information:
1. Store everything in directory entry (Windows - FAT)
2. Store only file name and point to some other data structure for other information (Unix)

Methods:

### 1. Contiguous Block Allocation

Allocate consecutive disk blocks to a file

Typically used by DVDs / CD ROMs ← file size known in advance; generally, only 1 write (burning) required

✓ Simple to keep track of
✓ Fast Access
⇒ Seek to 1st Block, then seek to $n^{th}$ block

X External Fragmentation possible
X File size needs to be specified in advance, and cannot be changed easily

### 2. Linked List

Each block contains a pointer to the next block (not necessarily consecutive blocks)

✓ No external fragmentation

X Slow Random Access O(n)
X Part of each block has to be reserved for the pointer information
X Pointer could be corrupted, losing file data

### 3. Linked List V2

Similar to linked list method above, except have the pointers in a seperate table (i.e., FAT - File Allocation Table)

The table is stored in Memory at all time
⇒ O(1) Random Access
⇒ 1 entry in FAT for each logical block

FAT32 ⇒ $2^{28}$ Blocks long, each block being 32 bits

FAT stores: File name, Metadata, Starting Block no.

✓ Fast Random Access

X Consumes a lot of valuable memory space
⇒ FAT keeps track of all disk blocks in memory
⇒ 4GB drive ↔ $4 \times 2^{20}$ entries in FAT

↓ file size ↔ ↓ access time

4. Indexed Allocation

Only the indexed block of opened file(s) in memory

✓ Less memory overhead
✓ Fast direct access
X Limited max. file size ← Max. no. of blocks == No. of indexed file entries
X Overhead from index block

There are some variations to resolve the issue of limited max. file size

- Linked Scheme : Keep a linked list of index blocks
- Multi-level Index : <Similar to multi-level paging>
- Combined Scheme : <Mix of linked scheme & multi-level index> ← used in unix systems

## Free Space Management ← Performed when allocating / deallocating logical blocks

1. Bitmap

Each bit in the (2-D) bitmap represents the status of the logical block ← 1 bit / block
Following general convention, 0 = occupied and 1 = unoccupied

✓ Easily manipulable using bit operations
X Takes up a lot of memory space

2. Linked List

Each free block has a pointer to the next largest, in terms of block no., free block

X Need to consecutively read free blocks
X Pointer information can be corrupted

## Directory Implementation

A directory contains : File Name ← Just an 'alias'
INode no. ← The actual file object is in the INode, not the directory entry

Given a file path,

Directory → File → Get INode No. → Find in disk → Load into OFT
(↑ if nested)

1. Linked List

Directory consists of a list, where each entry represents a file :

- Store filename (at minimum) and possibly other metadata
- Store file information, or pointer to file information
  (file information: Starting Block No.; pointer to file information: INode No.)

X Inefficient for large directories and/or deep tree traversals ← Common Solution : use cache to remember the latest free search
⇒ Requires linear search to locate a file

2. Hash Table

Hash based on file name, using Seperate Chaining Collision Resolution
1 Hash Table maintained per directory

X Limited size
X Difficult to get a general hash function