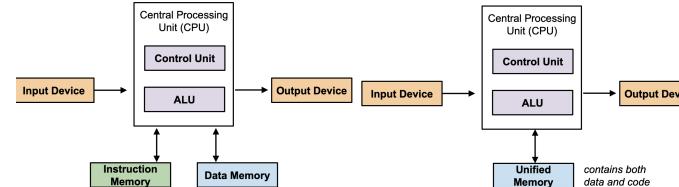


## 01. Introduction

OS - Program that acts as an intermediary between user and hardware

### Different architectures



**Difference** Separate vs common storage pathway for code and data

Why do we need OS?

- Should not have bugs
- Treat process as malicious

### Mainframe

Old analog "computers" using physical cards for programming

### Improvements

- Problem: Batch processing inefficient
- Solution: Multiprogramming
  - Loading multiple jobs that runs while other jobs using I/O
  - Overlapping computation with I/O
- Problem: Only one user
- Solution: Time sharing OS
  - Multiple concurrent users using terminals
  - User job scheduling
  - Memory management
  - **Hardware virtualization** - Each program executes as if it had all resources

### Motivation

1. Abstraction
  - Hide low level details and present common, high-level functionality to users
2. Resource allocation
  - Allow concurrent usage of resource and execute programs simultaneously
  - Arbitrate conflicting request fairly and efficiently
3. Control programs
  - Restrict resource allocation
  - Security, protection and error prevention (Defensive)
  - Ensure proper use of device

### Advantage

- Portable and flexible
- Use computer resources efficiently

### Disadvantage

- Significant overhead

### OS vs User Program

#### Similarities

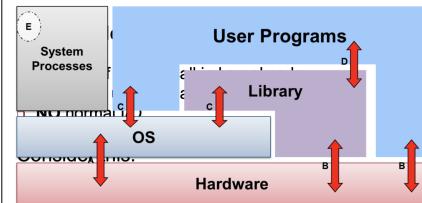
- Both softwares

#### Difference

- OS runs in **kernel mode** - Access to all hardware resources

- User programs run in **User mode** - Limited access

- User programs use syscalls to communicate with OS for hardware processes
- User - Full virtual address space for code but cannot write outside of virtual address space
- Kernel - All code share same address space

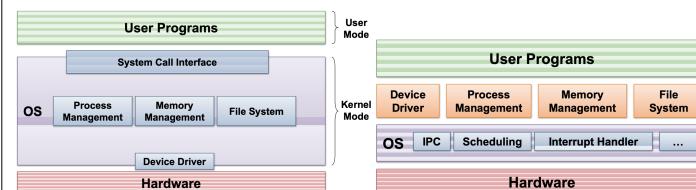


- A OS executes machine instructions
  - B Normal machine instructions executed
  - C Calling OS using **syscall interface**
  - D User programs call library code
  - E System processes providing high level services
- Why OS don't occupy entire hardware layer
- Slow to have all operations pass through intermediary
  - User programs can have direct interaction with hardware (eg. Arithmetic) during low risk operations

### OS structure

#### Monolithic OS

- One big kernel program
- Well understood and has good performance
- Highly **coupled** - internal structure interconnected that unintentionally affect each other

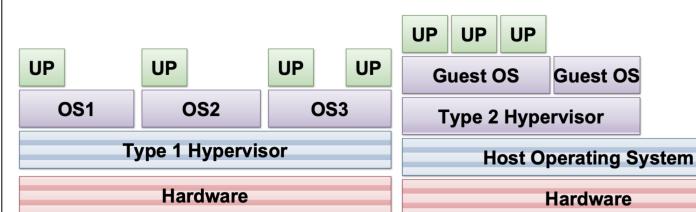


#### Microkernel

- Small clean
- Basic and essential facilities
- IPC communication OR run external programs outside OS
- Robust and more **modular** - Extendible and maintainable
- Better isolation btw kernel and services
- Lower performance
- Unix is monolithic kernel, Windows is hybrid

#### Virtual Machines

- Software emulation of hardware
- Virtualization of underlying hardware to run additional operating systems concurrently



## 02. Process abstraction

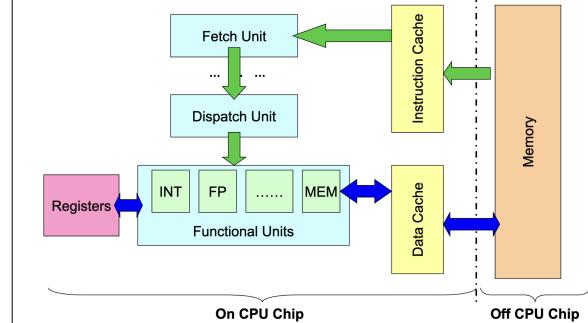
### Program vs process

- Program is set of instructions to complete a certain task stored in secondary memory of computer
- Process is instance of program that is currently executed
- Note that threads are not new processes and share the same file descriptor
  - Share the heap with the process but differ in stack

### Motivation

- Allow concurrent usage of hardware
- Multiple programs sharing the same processors/IO

### Computer organisation



### Memory

- Storage for instruction and data
- Managed by the OS
- Normally accessed via load/store instructions

### Cache

- Fast and invisible to software
- Duplicate part of the memory for faster access
- Usually split into instruction and data cache

### Fetch

- Load instructions from memory
- Location indicated by **Program Counter**

### Functional units

- Carry out instruction execution
- Dedicated to specific instruction type

### Registers

- Internal storage for fastest access speed

### Information needed

- Memory context
  - Code, data
- Hardware context
  - Register, PC value, frame pointer
- OS context
  - Process properties, resource used, files

### Function calls

Suppose a function f() calls g()

- f is caller and g is callee
- Steps of control flow

- 1. Setup parameters
- 2. Trf ctrl to callee
- 3. Setup local var

- Remember the initial variables especially if registers are to be reverted back when jumping back to caller
- Callee does not know the registers callers used, so prevent accidental overwrite of content to registers caller use

- 4. Store any results
- 5. Return ctrl to caller

## Issues

### Control Flow

- Need to jump to functional body when callee called
- Need to resume to next instruction in caller after done

### Data storage

- Need to pass parameters to function
- Need to capture return result

### Additional

- May lead to overriding of data in caller by callee (interference)
- Calling g() multiple times may lead to insufficient space and overriding

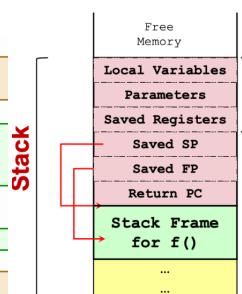
## Stack memory

Memory to store function invocation and control flow

**Stack Pointer** - Indicates the first free location in the stack region

**Frame Pointer** - Points to the frame and is used for traversing around the stack easily

- 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



Information needed for function invocation - Stack frame

- Return address of caller
- Arguments for the function
- Local variables
- Stack and frame pointer of caller
- GPR values (register spilling)

Callee stack frame will be on top of the caller

## Dynamic memory (Heap)

Memory that the program/user specifies manually (eg. malloc, new)

Problems:

- Allocated only at runtime
  - Size not known at program compilation time
  - Cannot specify a region in data
- No definite deallocation timing
  - Must be freed explicitly by the program
  - Cannot place in stack region

Solution:

Add Heap region for dynamic allocation

- BUT: Generates holes btw data due to variable deallocation timing

## Stack vs Heap

- Objects are stored in heap and stack stores references to it

- Objects in heap are globally accessible
- Data are primitive variables eg. int, char.. even when not initialized

## OS context

### Process identification

Features:

- Distinguish processes from each other (Unique)
- Communicated to the hardware

### Process state

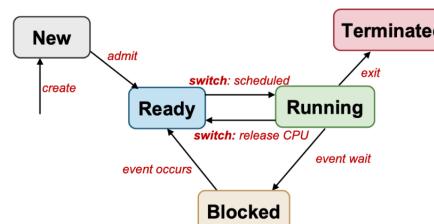
**New** New process that may still be under initialization (not schedulable)

**Ready**

**Running** Executed on CPU

**Blocked** Sleeping/waiting for event - cannot execute until event available

**Terminated** Finished execution and requires OS cleanup



## Process control block (PCB)

Table representing all processes containing entire execution context

- Maintained by OS in OS memory for every process
- PC, FP, SP and other GPR (only updated when swapped out)
- Memory region info (Text, data, heap and stack)
- It is a "point" to real memory space **NOT** actual memory space used by process

Top: Top of stack, PID, privileges, process state

- Pointer to parent process

### Context switching

1. Interrupt or syscalls
2. Save state of old process into PCB
3. Reload state of new process from PCB
- Involves a change to kernel mode and back

## Exceptions and interrupts

Exceptions

- Synchronous (due to program execution)
- Machine level instructions arise errors
- Exception handler executed automatically in software (can be handled by programmer)
- Handled by OS which sends a signal to application to implement try-catch mechanism

Interrupts

- Asynchronous (Can happen anytime)
- External events that cause execution to fail (hardware related errors)
- Program execution suspended and **interrupt handler** executed automatically

## Instruction execution

1. Read byte from PC and decode instruction
2. Read 2 bytes to get the address/operands
3. Perform ALU operations
4. Store result into destination
5. Check if any interruptions

Interrupts can happen at anytime, and will remain pending until step 5 where it is handled

## Interrupt handling

1. Push PC and status register into hardware
  2. Disable interrupts
  3. Read **Interrupt Vector Table** - Table where the OS stores address of all interrupt handlers
  4. Switch to kernel mode
  5. Set PC to handler address and execute the instructions
- OS populates the IVT table with address of interrupt routines
  - Hardware reads IVT to locate the handler

## System calls

**Application Program Interface** - Provides way of calling facilities/services in kernel

Synchronous function only done in kernel mode providing system services

Typically more expensive than library calls due to context switching

OS dependent and cannot be avoided by any application

## Functions

**Process control** Direct processes like creating, terminating or synchronisation

## File/device manipulation

**Information maintenance** Get system data, time, etc

**Communication** Create, delete connections and send/receive messages

## Method (Programming language specific)

- Library version with the same name and same arguments
- User friendly library version
- Using func `long syscall(long number);`

## Mechanism

1. User invoke library call
2. Place call number in the designated location
3. Library call executes a special instruction (**TRAP/syscall**) to change user to kernel mode
4. (in kernel) syscall handler is determined (by a **dispatcher**)
5. syscall handler is executed
6. Syscall handler ends and control returned to the library call
7. Return to user mode and continue normal function mechanism

## 03. Process abstraction in Unix

Process information

**Pid**

**Proc state** Running, sleeping, stopped, zombie

**Parent pid**

**Cumulative CPU time** For scheduling

## fork()

Process creation  
Package unistd.h and sys/types.h  
return PID of newly created process(parent) and 0 (child process)

## Behaviour

- Creates a child process
- Copy data of parent (Independent memory space)
- Unless explicitly modifying value at address, parent and child variables are distinct
- Same code, same address space
- Differs by pid, ppid and fork() return value

## Implementation

Clone the parent process

- Create address space of child process
- allocate new pid to child and pass to parent
- Create kernel process data structures
- Copy kernel environment of parent process
- Initialize child process context (pid, ppid, cpu\_time = 0)
- Copy memory regions from parent
  - Very expensive operation
  - Code, data, stack
- Acquire shared resources
- Initialize hardware context for child process (copy parent registers)  
Problem: Memcopy is very expensive operation

Solution: Copy on write

- Only duplicate a memory location when it is written to

## exec(@param)

Replace current executing process image

- Code and data replaced
- PID/PPID intact
- If fail, program just continues to run the next instruction

## Format

param char \*path, char \*arg0...

- Note that last term MUST be **NULL** indicating end of argument list

## header unistd.h

## exit(status)

return Does not return anything  
param int status - returned to the wait call

- Most system resource used by process are released on exit
- return from main() implicitly calls exit(0)
- Basic processes are not reusable

- Pid and status
- Process accounting info

## wait(&status)

Parent child synchronisation

param &status - address to put return value

## header sys/types.h and sys/wait.h

return pid of terminated process

- Call is blocking - suspend operation until at least one child terminates
- Cleans up remainder of child system resources (PID, status)
- waitpid - used for waiting for specific child process

## Orphan and zombie process

**Zombie** - All processes that has exited but parent did not call wait  
**Orphan** - Child process whose parent has been terminated

- Parenthood will be propagated up to init which may use wait() to clean up

## 04. Inter Process Communication

### Mechanism

- Shared memory

- Message passing

- Pipes

- Signal

### Shared memory

Communication through read/write to shared memory

Advantages

**Efficient** Only require OS to setup shared region once

**Ease of use** Simple reads and write to memory

- Implicit communication

- Can store any type of information

Disadvantages

**Limited to single machines** Less efficient over different system

**Need Sync** Might have data races without synchronisation

**Race condition** - System behaviour is dependent on the context/interleaving of process → unpredictable outcome

### Steps

- Create/locate a shared memory region M
- Attach M to process memory space
- Read from/write to M
- Detach M from memory space after use
- Destroy M
  - Only one process need to do this
  - Can only destroy if M is not attached to any process

### Message Passing

Explicit communication through exchange of message

**Naming** Have to identify the parties in the communication

**Synchronisation** Behaviour of sending/receiving operations

- Messages have to be stored in kernel memory space
- All sending/receiving operations have to be done through syscalls

### Direct communication

Sender/receiver explicitly name parties in communication

- One link per pair of communicating processes
- Need to know identity of other party

### Indirect communication

Message storage (Mailbox/Port)

- Can be shared among a number of processes

## Synchronisation behaviour

### Blocking

- send and receive is blocked until a message has received/sent (other party is ready)

### Non-blocking (asynchronous)

- execute immediately and sends information to somebody OR returns empty handed

Typically, receive is synchronous and send is asynchronous  
BUT send just buffers the message and only sends the message when the receiver is receiving (no loss)  
Message buffers

- Under OS control

- Decouples sender and receiver - less sensitive to variation in execution

- Mailbox capacity declaration in advance

### Rendezvous

- Synchronous message passing

- Sender is blocked until receiver sends matching receive

- No buffering needed

### Pros

- Applicable beyond a single machine

- Portable** - Easily implemented on many platforms and processing environments

- Easier synchronisation

- Implicit via send/receive behaviour

- Communication and synchronisation is done simultaneously

### Cons

- Inefficient** Usually requires OS intervention every operation

- Difficult to use** Requires information packing into specific format

### Unix Pipes

3 different communication channels to user

- stdin - standard in bounded to keyboard input

- stderr - used for error messages

- stdout - linked to screen

”|” symbol for linking input/output channels of each process to each other  
Create communication channel with 2 ends (one reading, one writing)

- Producer-consumer relationship

- Like anonymous file reading information FIFO

### Synchronisation

- Circular bounded byte buffer with implicit synchronization

- Writer wait when buffer is full

- Reader wait when buffer is empty

- Unidirectional (one write one read exclusively) vs bidirectional (any end for reading and writing)

### pipe(@param)

param array of file descriptors

return 0 for success, != for errors

## Unix signals

Asynchronous notification about an event sent to process/threads

- Handle signal via default or user defined/supplied handlers(not all signals)
- Can only call async safe functions within handler (signal/wait cannot be called in POSIX)
- e.g. Kill, stop, continue, error

### Standard signals

First the signals described in the original POSIX.1-1990 standard.

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from <a href="#">abort(3)</a>
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from <a href="#">alarm(2)</a>
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

## 05. Process scheduling

### Concurrency

- Multiple process progress at the same time
- Virtual parallelism (threading, context switching)
- Physical parallelism (multicore, multi CPU)

### Scheduling

More ready processes vs limited CPU

- Each process diff CPU usage

- Scheduling algorithms** - Deciding the process to run based on environment and process behaviour

- Processes have phases of io-bound and cpu intensive activities

Processing environment (sorted by increasing user interaction and response times):

- Batch
- Interactive
- Real time

### Objectives

Fairness Should get fair share of CPU time n prevent **starvation** - processes never have access to CPU

Utilization Hardware is used all the time

**non-preemptive** - Process gives up resources voluntarily **preemptive** - Process given a fixed quota to run and is suspended for other processes

### Timeline

- Scheduler is triggered (OS take over)
- Context switching can happen and current process information stored and placed back in queue
- Pick process to run using algorithm and setup its context

### Batch

Non-preemptive used dominantly cos minimal user interaction needed

### Objectives

**Turnaround time** Total time till finish running **including waiting** (finish time - arrival time)

**Throughput** Rate of task completion

**Makespan** Total time to complete **ALL** tasks

**CPU utilization** Percentage of **time** CPU is used

### Algorithms

#### 1. First Come First Serve (FCFS)

- Scheduled using FIFO queue based on arrival time (bad turnaround time)
- Guarantee no starvation - number of tasks in front of task is strictly decreasing
- Reordering can reduce waiting time
- Convoy Effect** - Not all hardware resources are used concurrently because a process blocks it

#### 2. Shortest Job First (SJF)

- Choose tasks with smallest total CPU time
- Estimate CPU time for tasks in advance using the previous CPU-bound phases
- $Estimated_{n+1} = \alpha Actual_n + (1 - \alpha) Predicted_n$
- Starvation is possible as long tasks end up continuously waiting

#### 3. Shortest Remaining Time (SRT)

### Interactive system

### Objectives

**Response time** Time between request and response by system

**Predictability** Decreased variation in response time

### Premptive scheduling

- Periodic scheduler with time interrupts
- Time quantum** - Execution duration given to a process
  - Constant(regardless whether the process has given up processing halfway) OR Variable(full time quantum given to a process)

### Algorithms

#### 1. Round Robin (RR)

- Store tasks in FIFO queue and pick tasks to run until
  - Fixed time slice elapsed
  - Task gives up CPU voluntarily
  - Task blocks
- Task placed at end of queue to wait for next turn
- Response time guaranteed bounded by num task time quantum
- Timer interrupt needed
- Choice of time quantum duration is important
  - Longer quantum: Better utilization but greater wait time
  - Shorter quantum: Bigger overhead/ lower utilization but shorter wait time
- Poor performance when there are many jobs all exceeding time quantum
  - high overhead from context switching

#### 2. Priority scheduling

- Assign tasks processes priorities based on impt and select them
- Starvation for low priority tasks

### Solution:

- Lower priority every time the process have executed
- Give processes a time quantum

- Priority Inversion** - Lower priority tasks locks the resources used by a higher priority task leading to deadlock OR allowing lower priority tasks to run before

### Solution:

- Temporarily increase priority of tasks that lock resources until it unlocks
- Low priority task inherit the priority of high priority tasks which is restored once unlocked

#### 3. Multi-level Feedback Queue (MLFQ)

- Runs processes with higher priorities but lower priority when the job fully utilise its time quantum
- Processes that voluntarily gives up/blocks before time quantum retains its priority
- New processes have highest priority
- Processes with same priority run in RR
- Long CPU processes are starved as its priority becomes very low and cannot complete
  - Solution: Occassionally reset tasks to full priority
- Processes can exploit and voluntarily give up to retain its priority
  - Solution: Peg priority based on total CPU usage instead

#### 4. Lottery scheduling

- Give out tickets to processes for various system resources and randomly pick winners to be granted the resources
- Responsive** - Newly created processes can immediately join the lottery
- Good control - Process can be assigned quantity of tickets and resources which can be shared among its children/threads
- Simple implementation

## 06. Synchronisation primitives

### Race condition

- Execution of sequential processes should be **deterministic** - Repeated execution returns the same result... however,
- Process sharing modifiable resources AND execute concurrently by interleaving may cause synchronization problems - non deterministic outcomes
- Order determines the execution outcome

### Critical section

Region where unsynchronised access could lead to incorrectly interleaving scenarios

Only **ONE** process should be in the critical section at one time

### Properties

**Mutual exclusion/Mutex** All other processes should be blocked from entering critical section if occupied

**Progress** If no process occupying section, waiting process should be granted access

**Bounded wait** Upper bound on number of times other processes can enter critical section before a waiting process

**Independence** Process not in critical section should not block other process

### Incorrect synchronization

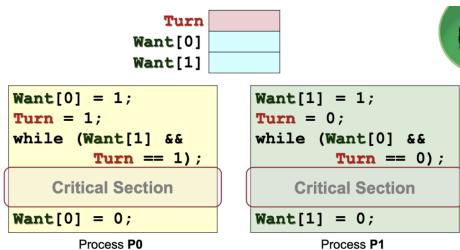
**Deadlock** All processes blocked

**Livelock** Processes keep changing states to avoid deadlock resulting in no progress (Deadlock avoidance mechanism)

**Starvation**

## Implementations

- High level language implementation (Peterson Algorithm)
  - Busy waiting - Repeatedly tests while-loop condition instead of going into blocked state (Wastes CPU power)
  - Low level - Error prone and complicated
  - Not general - cannot extend to allowing multiple access instead of Mutex



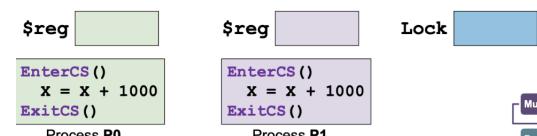
### Assembly implementation

- Atomic instruction to load value and replace value in memory location (**TestAndSet**)
- **Atomicity** - Prevents race condition of process viewing and upon modification, the value changes
- No bounded wait unless fair scheduling algorithm employed
- Employs busy waiting still

```

void EnterCS( int* Lock )
{
  while( TestAndSet( Lock ) == 1 );
}
  
```

- returns the current content  
- sets content of M to 1



```

void ExitCS( int* Lock )
{
  *Lock = 0;
}
  
```

### High level synchronization mechanism (Semaphore)

- Generalised mechanism to block a number of processes (sleeping), **S**, and unblock sleeping processes
- **wait()** - Blocks if  $S \leq 0$ , decrement **S**
- **signal()** - Increment **S**, wake up sleeping processes, operation **never** blocks
- **Invariant** -  $S_{current} = S_{initial} + N_{signal} - N_{wait}$

### Conditional variables

- Tasks wait for certain events to happen
- Event completion/begin is broadcasted

### POSIX semaphore

header `semaphore.h`  
`wait` `pthread_mutex_unlock`  
`signal` `pthread_mutex_lock`

### Midterm

- **Tail-call optimization** - Stack frame can be replaced in iterative recursion since all information is retained in the 'return' function call at every iteration

## 07. Threads

### Motivation

- Processes are expensive
  - Process creation via `fork()` duplicates memory space and process context
  - Requires context switching and saving/restoring process information repeatedly
- Hard to communicate with each other
  - Independent memory space (IPC communication only)
- Need easy way to execute some instructions simultaneously (more threads of control to same process)

### Features

- **Multithreaded process** - Single process with multiple threads
- Threads share the same **Memory context** (Text, data, heap) and **OS context** (PID)
- Contains unique information
  - Identification (thread id)
  - Registers (GPR and special)
  - Stack

### Benefits

**Economy** Less resources to manage

**Resource sharing**

**Responsiveness**

**Scalability** Can take advantage of multiple CPUs

### Problems

- Syscall concurrency
  - Parallel execution of threads may result in parallel syscalls
- Process behaviour
  - Unable to impact process operations

### Thread implementation

1. User thread
  - Implemented as a user library - More flexible and configurable
  - Runs on any OS
  - Kernel unaware of threads in process (scheduling at process level)
  - If process is blocked, all the threads in the process cannot run
2. Kernel thread
  - Implemented in the OS through syscalls (slower and more resource intensive)
  - Scheduling among threads instead of via process
  - Use threads in kernel operations
  - Less flexible
    - Used by all multithreaded programs so kernel must cater to all
    - Balance btw many and less features
3. Hybrid thread
  - OS schedules on kernel threads
  - User threads can be bound to kernel threads
  - If the user thread is blocked, the kernel thread that it is assigned to is blocked and the OS will execute another kernel thread
  - Lead to greater flexibility

## Posix Threads (pthreads)

### `pthread_create`

`param` `tid`, `attributes`, `&function`, `function_params`

`return` 0 success, != 0 fail

### `pthread_exit`

Automatically called after function finishes

`param` `exit_value`

`behaviour` return value of function will be the "exit value"

### `pthread_join`

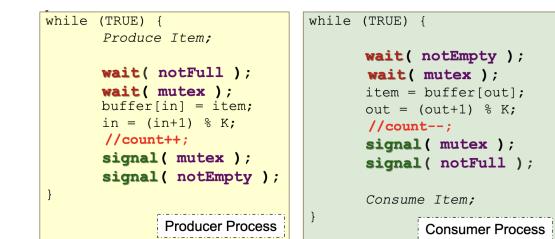
`param` `tid`, `&status` - Exit value returned by target pthread

## 08. Classic synchronisation problems

Use cases of synchronisation (mutexes)

### Producer Consumer

- Processes share a bounded buffer of size **K**
  - Producers produce items to insert in buffer when not full
  - Consumers remove items from buffer when non-empty
- Busy waiting vs blocking version and message passing

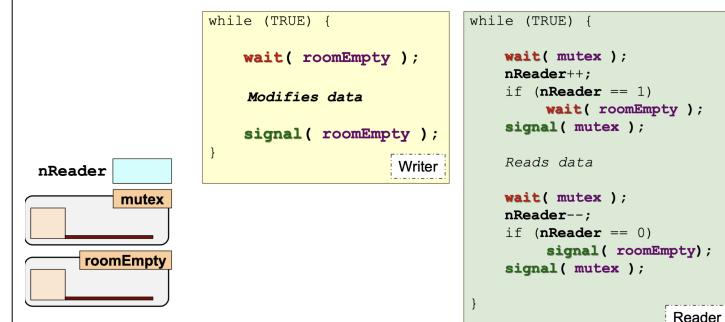


Initial Values:

□ `in = out = 0`  
 □ `mutex = S(1)`, `notFull = S(K)`, `notEmpty = S(0)`

### Reader writer/Barbershop

- Processes share data structure **D**
  - Reader retrieves information from **D** (multiple concurrent accesses)
  - Writer modifies information from **D** exclusively
- Starvation of the writer as readers can continue to occupy the resource continuously



Initial Values: `roomEmpty = S(1)`, `mutex = S(1)`, `nReader = 0`

## Dining philosophers



- Single chopsticks between philosophers sitting around a table
- Philosophers need pair to eat without deadlocks and starvation
- Philosophers makes the same decision (takes the same side of chopsticks everytime)

## Livelock

- To prevent deadlock, philosophers put down chopstick if cannot get the other side
- Livelock because all philosophers take chopstick, put down ....

## Limited eater

- One of the philosophers delay their eating temporarily until either side finishes
- All philosophers can complete their execution pigeonhole theorem (more chopsticks than philos)
- Not fair to the philosopher not eating (How to decide who stops operation?)

## Tanenbaum Solution

- Randomly decide which side chopstick the philosophers take first

## 09. Memory abstraction

### Hardware

- uses Random Access Memory
  - Access latency approximately constant regardless of location
- Can be treated as a 1D array of **AU** - Addressable unit
- **Contiguous** - Consecutive memory address with unique index (physical address)

### Content

- Text (for instructions), Data (global variables), Heap (dynamic allocation)
- Stack (for function invocation)
- Btw heap and stack is region of unallocated memory

### Memory usage of process

#### Types of data

1. **Transient data** - Valid only for a specific duration (parameters, local var)
2. **Persistent data** - Valid for complete duration of prgm until explicitly deallocated

### Role of OS

1. Allocate and manage memory space for each process
2. Protect memory space of distinct processes
3. Provide memory-related system calls
4. Manage memory space for internal OS use

### Memory abstractions

#### Motivation

- Processes assume memory starts at 0 and occupy the same memory location
- Difficult to protect memory space

## Solutions (?)

- Address reallocation
  - Offset all memory references when process is loaded to actual memory address
  - Problems: Slow runtime because have to run through all the instructions
  - Difficulty in distinguishing memory references
- Base and limit registers
  - All memory references compiled as offset from base register
  - When loading, base register initialised to starting address of process memory space
  - Limit register as the range of memory space of current process (protect memory space integrity)
  - Memory access checked against limit

## Logical address

- How each process views its memory space
- Mapping between physical and logical address needed
- Each process has a self contained, independent logical memory space

## Contiguous Memory allocation

### Assumptions

- Processes occupies contiguous memory region
- Physical memory must accommodate 1..\* processes with complete memory space

### Multitasking/context switching

- Multiple concurrent processes in memory
- Free memory via removing terminated processes/swapping to secondary storage

### Fixed size partitions

- Fixed number of partitions of equal size
- Larger min addressable units of partition
- **Internal fragmentation** - More memory allocated to program than used
  - Pros: Easy to manage and fast to allocate
  - Cons: Partition size need to be large enough to contain the largest process

### Variable size partitions

- **External fragmentation** - Unallocated contiguous memory surrounding partitions insufficient to satisfy memory request of process
  - Pros: Flexible, no internal fragmentation
  - Cons: Need more info in OS and takes longer time to locate appropriate region

### Allocation algorithms

- First fit
  - Take the first hole large enough
  - Fastest runtime (don't have to iterate through all the free partitions)
  - May be inefficient (high external fragmentation)
- Best fit
  - Choose the smallest partition that fits the information
  - Slower runtime but more efficient
- Worst fit
  - Choose the largest partition
  - Try to ensure that there is enough free space in the large partition for other process

## Merging and Compaction

- **Merging** Freed partitions can join surrounding holes

- **Compaction** Mv occupied partitions to create bigger consolidated holes
  - Time consuming (can be amortised)

## Multiple free lists

- Dictionary of list of free holes
  - Key = size, value = addresses
- Find smallest hole that accommodates req size
- Partition size increases exponentially
- Faster allocation (vs iterating all holes)

## Buddy system

- Free block split into half repeatedly to meet request size
- Merge buddy blocks when both free
- Efficient partition splitting and dealloc
- Locate good match for holes

## Allocation algorithm

- Free block split half repeatedly to meet req size
- Buddy block pair merge when free

$N = \text{req size}$ ,  $2^K = \text{largest allocable block size}$

$A[J] = \text{linked list of free blocks with size } 2^J$

1. Find smallest s such that  $2^s \geq N$
2. Access  $A[S]$  for free block
  - Free block exists
    - Remove block from free block list
  - Find smallest R from  $S+1 - K$  such that  $A[R]$  has a free block B
  - Split B and restart from step 2

## Deallocation algorithm

1. Check if buddy (sizeof= $2^S$ ) is free in  $A[S]$ 
  - If buddy is free
    - Remove both blocks from list
    - Add merged block, B' to  $A[S+1]$
    - Repeat steps for B'

2 blocks B and C of size  $2^S$  are buddies if

- Right S bits (0..S-1) of B and C are identical
- S+1th bit of B and C is different

## 10. Disjoint memory schemes

- Processes informations can be split into disjoint physical memory chunks

- **Address space** - Amount of memory allocated for all possible addresses for system
  - "32-bit system  $\rightarrow 2^{32}$  address space

## Paging scheme

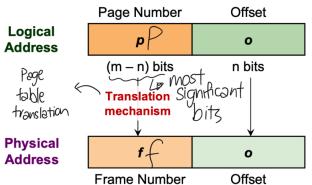
- **Physical frames** - Phy mem split into regions of fixed size
- Frames sizes are decided by hardware
  - Reduce number of access to memory
  - Dedicated component to support paging by speeding address translation
- **Logical page** - Logical mem split into regions of same size
- Pages of process loaded into any available mem frame at execution time
- Logical mem remains contiguous

## Page table - Lookup mechanism

- Map logical pages to physical frames
- Frame sizes should be power of 2
- $size_{page\_table} * s_{page} = s_{mem\_address} * s_{addressing}$

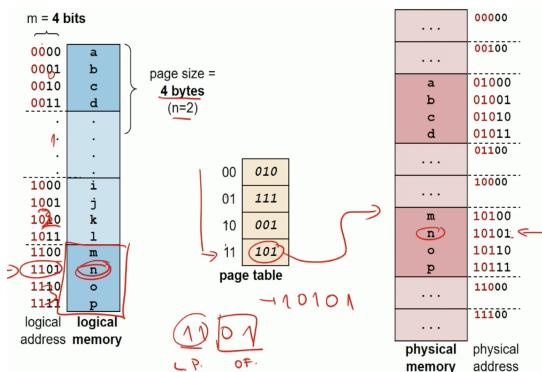
## Translation

$$PA = F \times \text{sizeof(physical\_frame)} + \text{offset}$$



## Benefits

- No external fragmentation
- Every single free frame can be used
- Insignificant internal fragmentation
- Separation of logical and physical address
  - Greater flexibility
  - Simple translation



## Implementation

- Pure software solution
- Page table info in Process Control Block (PCB)
  - 2 mem access per reference
    - Read the indexed page table entry to get the frame number
    - Access the actual memory item
  - Memory context of process includes page table/pointers to it
- Hardware support
  - Contain **Translation Look-aside Buffer** - Specialized on-chip component to cache page table entries
  - Use page numbers to search TLB associatively
  - Possible to have multiple TLB per core
    - TLB-hit** Frame number retrieved to generate physical address
    - TLB-miss** Memory access to page table → frame number used to generate physical address and update TLB
  - Very small and fast (reducing Average Memory Access Time)

- $AMAT = P(TLB \ hit) * \text{latency}(TLB \ hit) + P(TLB \ miss) * \text{latency}(TLB \ miss)$
- Latency of filling in TLB hidden by hardware
- Part of hardware context, so entries will be flushed when context switch occurs
  - Save pointer to prev process' page table in PCB
  - Ensure process does not get incorrect translation

## Protection

- Access-right bits
  - Enforce protection like read, write, execute permissions
  - Memory access checked against access-right bits in hardware
- Valid Bit
  - Exclude pages that are out-of-range for particular process from mapping
  - Check against bits in hardware whether the page can be accessed by process

## Page sharing

- Several processes' logical pages mapped to same physical frame
  - Used in shared code pages - code used by many processes
  - Implementing copy on write - pages shared until child changes its value

## Segmentation scheme

### Motivation

- Process contains different types of memory regions (data, text, heap, stack)
  - Order of memory regions does not matter
- Regions are variable sized and grow/shrink at execution time
  - Gaps btw regions shld be provisioned early
  - Difficult to check the permissions of memory access (which range it is)

### Implementation

- Process broken into named segments of variable sizes
- Manage memory at level of memory segments
- Segments itself are mapped to contiguous physical partitions
  - base address and limit (range of segment)
- Logical address  $\langle \text{SegID}/\text{name}, \text{Offset} \rangle$ 
  - SegID used to find  $\langle \text{Base add}, \text{Limit} \rangle$  in segment table

### Evaluation (Pros and Cons)

- Independent contiguous memory space matches programmers view of memory
- More efficient bookkeeping
- Segments can grow/shrink and be protected/shared independently
  - Cause external fragmentation due to variable sized contiguous memory regions

### Segmentation with paging

- Each segment contains fixed sized pages with its own page table
- Segment grows by allocating new page and adding to its page table

## 11. Virtual Memory Management

### Motivation

- Logical memory space may be  $\gg$  than phy mem
- Not portable to different phy memory sizes

## Advantages

- Security: Cannot be corrupted by other processes
- Efficient for caching virtual mem pages (locality)
- Simplify memory management, programming and protection
- Memory sharing

## Extension of paging scheme

- Split logical address space into fixed size pages
  - May reside in phy mem or secondary storage
- 2 new page types to denote phy location of information
  - Memory resident** Pages in phy memory stored in kernel space
  - Non-memory resident** Pages in secondary mem
- Resident bits** - Used to denote whether resident in mem
- Page fault** - CPU access non-memory resident page
  - CPU can only access mem resident pages - blocks process
  - OS needs to bring pages from secondary storage to phy mem

## Execution

- Check page table if page is mem resident
  - Access phy mem loc directly
  - Throw page fault **exception**
- Locate page in secondary memory
- Load page into phy memory
- Update page table
- Re-execute step 1 with mem-resident page

## Page Fault

- Page transfer is I/O operation done by DMA controller
  - Direct Access Media - abstract I/O from CPU
- Faulted process is in **blocked** state
- May send SIGSEGV if accessing mem region without access rights
  - Sig thrown when offset  $>$  page size
- Thrashing** - System slow down due to multiple page faults

## Locality

**Temporal** Mem address likely used multiple times

**Spatial** Adjacent mem address likely accessed soon

## Paging scheme

### Objectives

- Reduce startup cost when allocating pages during initialization
- Reduce footprint of processes in phy mem

## Demand Paging

- Only allocate mem resident pages when page fault
- Processes start with no mem resident pages
- Fast startup time and small mem footprint
  - Sluggish proc initially due to multiple page faults
  - Thrash other processes during page fault

## Page table structure

### Consideration

- Large logical and phy mem space → Large page size
- Every proc needs a page table
- Minimize page table size
  - High overhead
  - Page table may occupy several frames (Difficult to keep track locations)

### 2-level paging

- Split page table into regions with page table num
- Regions allocated when mem usage grows
- Page directory in PCB keeps track of smaller page tables
- Memory context of process only stores the page directory

### Problems

- 2 serialized memory access to get frame number
  - MMU cache frequent page directory entries
  - Speed up page-table walk upon TLB miss
  - Usual: log → page directory → page table → phy frame
  - MMU: log page → page-table → phy frame
  - TLB: log page → phy frame

### Inverted Page Table

- Map physical frames to  $\langle$ pid, page number $\rangle$
- Upper bound on table size (limited to num phy frames)
- One table for all processes but slow translation
  - Need search whole table to find the phy frame that has the specific page
- Does not support memory page sharing

## Page Replacement Algorithms

- Deciding the phy memory frame to evict during page fault
- $T_{mem\ access} = (1 - p)T_{mem} + p * T_{page\ fault}$ 
  - p = probability page fault,  $T_{mem}$  = access time for mem resident page

### Optimal Page Replacement

- Replace page that will not be needed again for longest period of time
- Impossible to know future memory references

### FIFO Page Replacement

- Evict oldest loaded memory page
- Maintain queue of resident page numbers and update queue during page faults
- **Belady's Anomaly** - More physical frames the higher likelihood of page faults
  - FIFO does not exploit temporal locality (commonly used frames evicted)

### Least Recently Used (LRU)

- Replace page that has not been used in the longest time
- OS keeps track of last access time
- Logical Time counter incremented for every mem reference
  - Page table entry has a time-of-use field which increases when referenced
  - Replace pages that have the smallest TOU
  - O(n) operation and overflow possible (TOU strictly increasing)
- Maintain stack of page numbers
  - Page removed from stack and pushed to top when referenced
  - Replace page at bottom of stack during replacement
  - Hard to implement in hardware (entries removed randomly)
- Second-Change Page Replacement/CLOCK
  - Modified circular FIFO to give second chance to pages accessed
  - Every PTE maintains **Reference bit** - 1 = accessed since last reset
    - 1. Oldest FIFO page is selected
    - 2. If ref bit == 0: page replaced, else..
    - 3. Page is skipped
      - Ref bit cleared to 0
      - **NOTE!!** Page does not refresh its position in queue
      - Next FIFO page selected/given chance

### Page Replacement

#### Victim page

Page that will be replaced

#### Global

Victim page chosen among all physical frames

- **Self adjustment** - Processes that need more frames can get more
- Badly behaved processes affect others
- Non-deterministic number of frames allocated to processes during each run

#### Local

Victim page chosen from pages of process that caused page fault

- More stable - number of frames allocated to each process remain constant
- Frame allocation matters - progress of process may be hindered

### Frame Allocation

- How to distribute N physical memory frames among M processes

### Approach

#### Equal

Every process gets equal num frames  $N/M$

#### Proportional

Each process, P gets  $\frac{\text{size}_p}{\text{size}_{\text{total}}} * N$

### Problems

- Difficulty in finding right number of frames to prevent thrashing
- **Cascading thrashing** - Process steals page alloc from other proc
- I/O bandwidth costly and may degrade performance

### Working Set Model

- **Working set** - Set of pages referenced by process during a period of time
- Working set may changes continuously but is relatively constant
- Model aims to find a good window to reduce possibility of page faults
- $W(t, \Delta)$  = pages in interval at time t,  $\Delta$  = window size
- **Transient region** - Working set continuously changing in size
- **Stable region** - Working set stays the same for a long time

## 12. File System

### Introduction

#### Motivation

- Volatile physical memory so external storage needed for longer term storage
- Storage media is not portable
  - Dependent on hardware specification and organisation

### Services

- Provides abstraction on top of physical media
- High level resource management scheme
- Protection between processes and users
- Sharing between processes and users

### Objectives

- Self-contained** Information stored on media is enough to describe organisation (plug and play)
- System knows how to read and write information on the storage system automatically

**Persistent** Retain and be retrieved beyond the lifetime of OS and processes

**Efficient** Good management of free and used space while having low overhead for bookkeeping information

### Memory vs File System management

	Memory Management	File System Management
Underlying Storage	Ram	Disk
Access speed	Constant	Variable Disk I/O time
Unit of addressing	Phy mem address	Disk sector
Usage	Address space for process, Implicit when process runs	Non-volatile data, Explicit access
Organization	Paging/Segmentation determined by OS and hardware	Different FS:ext(Linux), FAT(windows), HFS(mac)

### File

#### Objective

- Abstraction for logical unit of information by processes
- Contains information structured in specific styles

### Metadata

- Additional information associated with the file

#### Name

Human readable reference to the file

- Consistent naming rule
- Case sensitivity, length of name, allowed symbols
- File extension (sometimes used to indicate file type)

#### Identifier

Unique id for the file used internally by FS

#### Type

eg. directory, text, object, executable

- May have a predefined internal structure that must be processed by specific program (jar, pdf)
- Distinguishing Type is OS dependent (some use file extensions and some **magic numbers** - Embedded info in the file stored at the beginning)

### Size

**Protection** Classified under reading writing and execution rights for owner, group, and everyone

- Types of access: Read, write, execute, append, delete, list (read metadata of file)

- **access control list** - List of user identity and allowed access types (very customisable but too much overhead)

- **permission bits** - Classify users into 3 classes (owner, group and all) and define permissions RWX for the classes

– Command: getfacl (get file acl)

```
$ getfacl exampleDir
# file: exampleDir
# owner: ccris
# group: compsc
user::rwx
user:sooyj:rwx
group::r-x
group:cohort21:rwx
mask::rwx
other::---
```

"getfacl" is the command to get ACL information

Permission for Specific User

Permission for Specific Group

Permission "upperbound"

```
void main() {
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[10];
    in_fd = open("test_in.txt", O_RDONLY);
    if (in_fd < 0)
        printf("In Error in opening file to read");
    out_fd = open("test_out.txt", O_WRONLY | O_CREAT, 0600);
    if (out_fd < 0)
        printf("Out Error in opening file to write");
    rd_count = read(in_fd, buffer, 1);
    while (rd_count != 0) {
        wt_count = write(out_fd, buffer, 1);
        rd_count = read(in_fd, buffer, 1);
    }
    close(in_fd);
    close(out_fd);
}
```

Opens 'test\_in.txt' for reading and return a file descriptor

Opens 'test\_out.txt' for writing and return a file descriptor

Read 1byte of data from file

Writes 1byte of data to file

Close the files

## Implementation in OS

### Information

**File pointer** Keeps track of current position within a file

**File descriptor** Unique identifier of file (int) Used internally in software rather than the name

**Disk location** Actual file location on disk

**Open count/reference count** How many process opened this file

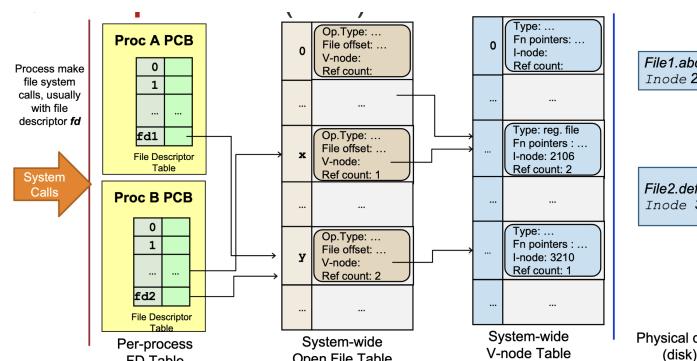
### Criteria

- Multiple processes can open same file
- Multiple files can be opened concurrently

### Approach

Using 3 tables to store information

1. File is opened twice (can be same process)
  - IO occurs at independent offsets
  - 2 per-proc and 2 sys-wide w diff FD
2. Multiple proc using same open file table entry
  - Forks duplicate file descriptor (share offset)
  - Only one offset (IO change offset for both proc)
  - 2 per proc, same FD and sys-wide table



### Mmap

- Map a shared memory region or share files to multiple process
- Both process can access shared data on RAM without system calls

### Directory

#### Motivation

- Provides logical grouping of file
- Keep track of files (system usage of directory)

## Structure

- Single level
  - All files in one directory (CD etc)
- Tree structure
  - Recursively embedded in other directory
  - Addressed by absolute or relative pathnames from curr working dir
- DAG
  - File can be shared using hard links (dir not allowed)
  - Shared file appearing in multiple directory referring to the same content using pointers
  - Low overhead (only pointers added) but deletion problems (synchronisation)
  - Hard link "ln" command (dir not allowed)
- General graph
  - Undesirable structure - hard to traverse (cycles)
  - Hard to determine when to remove file
  - Implementation via symbolic links (dir allowed)
  - Sym link is a special link file that contains the path name of file F - find out where F is and access it
  - Deletion is easy as only the special link file is deleted OR dangling link created (link without file)
  - Larger overhead as special link file is taking up actual disk space
  - Command: ln -s (cat and vim acts on the target file)

## File system implementation

- Stored as 1-D logical blocks mapped into disk sector organised with unique ID
- Storage contains Master Boot Record (MBR) and  $\geq 1$  partitions

### MBR

1. **Simple boot code** - Defines the partition to be initialised first
2. **Partition table** - Starting and ending address of each partition

### Partitions

Partitions can contain independent file systems and OS

**OS boot block** All information needed to boot OS from the partition into memory

**Partition details** Metadata of partition, loc and num blocks used/free etc

**Directory structure, files info and file data**

## 13. File System Implementation

### Objectives

- Keep track of logical blocks
- Allow efficient access and utilise disk space effectively

### Allocation Policies

#### Contiguous

- Allocate consecutive disk blocks to a file
- Simple to keep track and fast (only need to keep track of the first block)
- External fragmentation because file size must be specified in advance

#### Linked list of disk blocks

- Stores pointer to the next disk block number and actual file data
- Need to keep track of the first and last disk block number
- Solves fragmentation but random access of file is very slow
- Overhead with storing next block number
- Less reliable (if one of the pointer breaks, all break)

### Access methods

**Sequential** Data read in order and cannot be skipped

**Random access** Read bytes in any order

- Can be read to specific (absolute) offset
- Seek to relative offset from current pointer location (used by unix and windows)

**Direct access** Random access to any record directly (Fixed length records)

### Operations

**Create, read, write**

**Open** Performed before any operation to prepare necessary info for file operations

**Repositioning/Seek** Move current position to new location

**Truncate** Removes data from specified position to end of file

- Done as system calls to OS for protection, concurrent efficient access

- Commands: 'open' 'read' 'write' 'lseek' 'close'

## File Allocation Table (FAT)

- All "next block" pointers stored in a single table rather than at each data block
- Table always in memory so simple and efficient (faster random access)
- Keeps track of all disk blocks in partition which may waste a lot of memory (valuable)

## Indexed allocation

- Each block has index block
- Index block records pointers (and index) of blocks in file
- Lower memory overhead (only index block of opened file needs to be in memory)
- Faster direct access
- Have maximum file size (limited by number of indexes in index block)
- Index block overhead (waste one block for indexing)

## Modified Index Allocation

**Linked scheme** Linked list of index nodes (normal)

**Multilevel index** Hierachial linked list (entries point to another linked list of nodes..)

**Combined** Direct and multi-level index schemes

- Fast access for small files and accomodates larger files

## Partition Details

### Free Space Management

- Maintain free space info (where and how much, remove/add when file created/deleted)
- **Bitmap** - Disk block represented by bits (1 if free)
  - Mem overhead - complete data structure has to be kept in memory
- **Linked List** - Each disk block contains some free disk blocks/pointer to next free space
  - Easy to locate free block and only one pointer needed in memory
  - BUT high overhead

## Directory Info

### Objectives

- Keep track of files in directory and metadata
- Map file name to file information
  - Locate the file information using pathname and file name to open for use
  - Parse the full path name by recursively searching directories along path

### Linear List

- Each entry represents a file incl name + other metadata + file info/pointer to file info
- Locating file requires a linear search
  - Inefficient for large directories requiring deep tree traversals
  - Latest searches can be cached to improve latency

### Hash tables

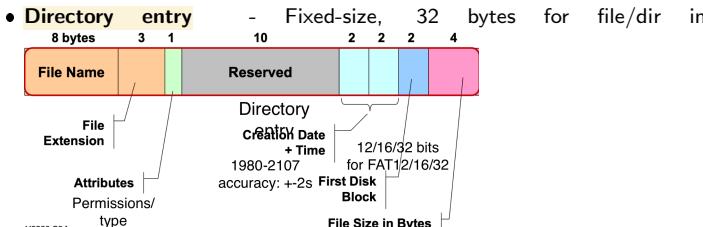
- Every directory contains hash table indexed by filename (with chained collision resolution)
- Fast lookup but may be limited in size and dependent on hash functions (to reduce collisions)

## FAT

- Used by Microsoft
- File data allocated to blocks with allocation info kept as linked list
- Data block pointers in File Allocation Table with block info (free/next occupied block/EOF/unusable)
- OS cache table in Ram to facilitate linked list traversal
- Each partition contains the boot, FAT, root directory and multiple data blocks

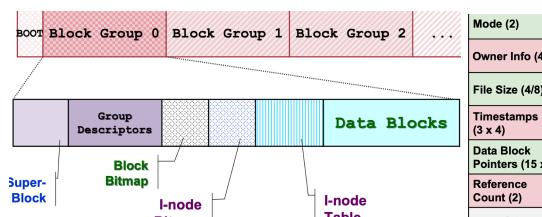
## Directory Structure

- Special file with linear list of directory entries



## Extended-2 FS (Ext2)

- Split into blocks (corresponding to 1 or more disk sectors) and grouped into Block Groups
  - Data blks for same file should be in the same blk grp to reduce data fragmentation (spatial locality)
- **I-Node** - Special structure describing each file/directory
  - Contains file metadata (access right, creation time etc) and data block add
- **Superblock** - Describes whole FS including total num I-Node, I-Nodes per grp, total disk blocks + num in grps
- **Group descriptors** - Describes block grp: num free blocks, num free I-Nodes, bitmap locations
  - Superblock and descriptors duplicated in each block group for redundancy
- **Block/I-Node Bitmap** - Keep track usage status free(0)/occupied(1)
- **I-Node Table** - Array of I-nodes in particular block group accessible by unique index



## I-Node Data Block Pointers (15)

**Direct Block** 1st 12 pointers to actual data

**Single Indirect block** 13th pointer to a disk block full of direct pointers

**Double Indirect Block** 14th pointer to disk block full of single indirect blocks

**Triple Indirect Block** 15th pointer to block containing double indirect blocks

## Directory Structure

- Data block of directory store linked list of directory entry
- **Directory entry** - Variable sized to store file/subdirectory info
  - I-Node number for file
  - Size of this directory entry (to find next directory entry)
  - Length of file/subdirectory name
  - Type of file/subdirectory
  - File/subdirectory name (up to 255 chars)

## Open

1. Process P opens file /.../.../F which is located
  - Split by "dir/" - If dir, locate directory entry in CurDir
  - Retrieve I-Node number, read actual I-Node and continue from NewDir
  - Else directly retrieve I-Node number and read actual I-Node
2. File info loaded into entry F in sys-wide table and V in I-Node table
3. Create entry in P table to point to E (file descriptor) and pointer from E to V
4. Return file descriptor to caller

## Deletion

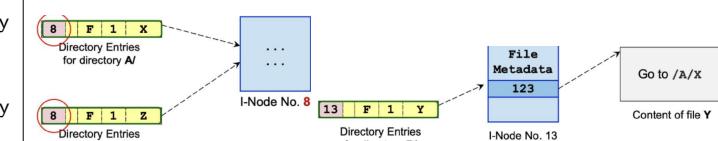
1. Remove its directory entry from parent directory
  - Point prev entry to next entry/end
  - Blank record may be needed
2. Mark specific I-Node in bitmap as free
3. Mark corresponding block in bitmap as free

## Hard link

- New directory entry using same I-Node number as file
- Filename can be different

**Multiple references of I-Node** Hard to determine when to delete I-Node

- Maintain I-Node reference count
- Decrement for every deletion



## Sym Link

- File created with I-Node pointing to memory location of symlink
- Content of symlink == pathname of file

**Only pathname stored** Link easily invalidated (during file name changes/deletion)

## Finals

- Mapped file to SWAP will be dropped reread from file on disk
- Process opens 3 FD (STDIN, STDOUT, STDERR) auto upon initialization