CS2106 Intro. to Operating Systems

AY 25/26 Sem 1 — github/omgeta

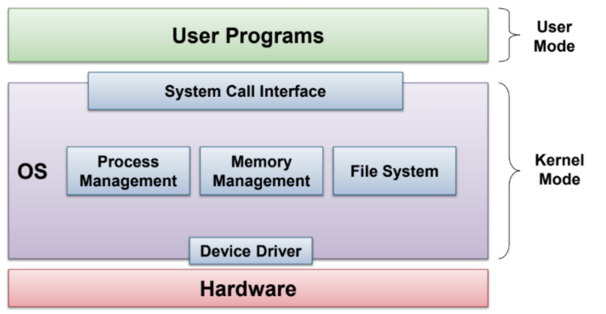
1. Introduction

Operating systems are software intermediaries between users and hardware, allocating resources, providing services and controlling program execution.

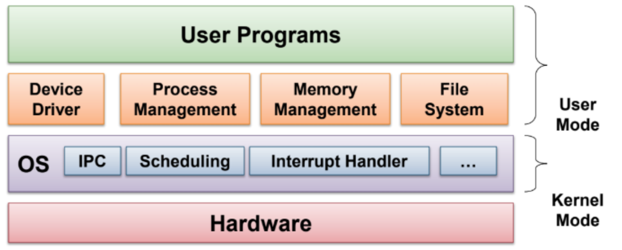
i. Kernel mode: OS access to hardware resources
ii. User mode: Software has limited or controlled access to hardware resources

Structures:

i. **Monolithic**: Kernel is big and highly coupled. Well-understood and performant, but complicated.



ii. **Microkernel**: Kernel is small with only essential features; high-level services are userspace server processes communicating with kernel via IPC. Robust, extensible and isolates kernel from high-level services, but lower performance.



iii. **Layered**
iv. **Client-Server**

Types:

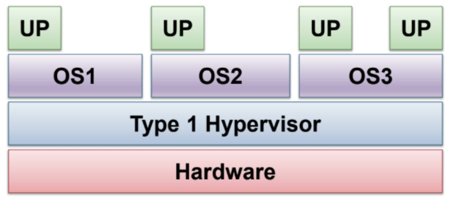
i. **None**: Program directly interact with hardware; minimal overhead but not portable and inefficient.
ii. **Batch**: Execute jobs with minimal user interaction; efficient for many jobs but CPU idles on IO.
iii. **Timesharing**: Multiple users interact using terminals, user job scheduler illudes concurrency.
iv. **Real Time**: Hard real time has timing requirements strictly respected, soft real time allows some time constraints to be missed.
v. **Embedded**: Specialized for devices; considers power, memory and real-time constraints but not general-purpose.
vi. **Distributed**: Manages systems on network; improves resource sharing and fault tolerance but adds complexity and reliability concerns.

Virtual Machines

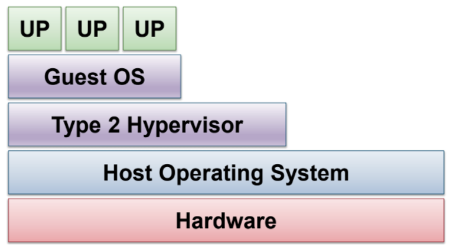
Virtual machines provide a virtualization of underlying hardware that runs an operating system and applications independently on a host system.

Hypervisors are software which manage virtualization.

i. **Type 1 Hypervisor**: Run directly on hardware.



ii. **Type 2 Hypervisor**: Run as an app on host OS.



System Calls

System calls are API calls to the OS for kernel services, changing from user mode to kernel mode via TRAPs.

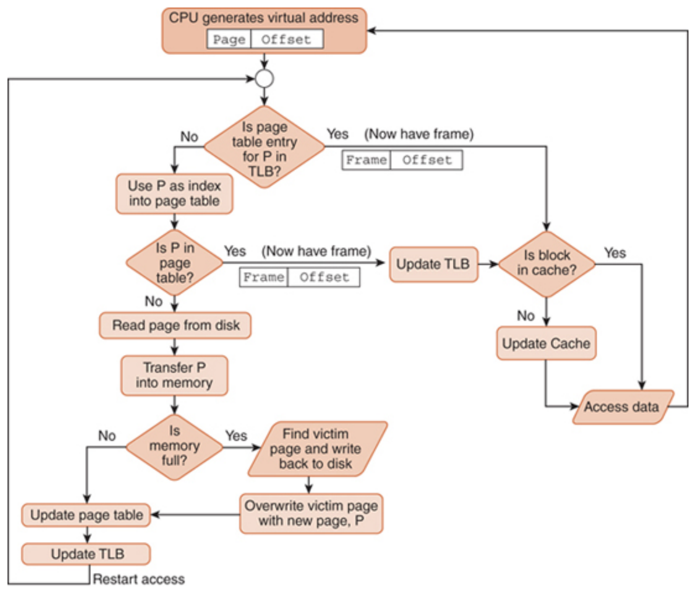
Mechanism:

i. User invokes library call
ii. Library call (usually assembly code) places system call number in special register
iii. Library call executes TRAP to enter kernel mode
iv. System call dispatcher calls appropriate system call handler based on system call number
v. System call handler executed
vi. OS switches to user mode and returns control to library call
vii. Library call returns to user using function return

Exceptions and Interrupts

Exceptions are synchronous, caused by program execution errors, and cause the exception handler to execute.

Interrupts are asynchronous, caused by external events, and cause the interrupt handler to execute.

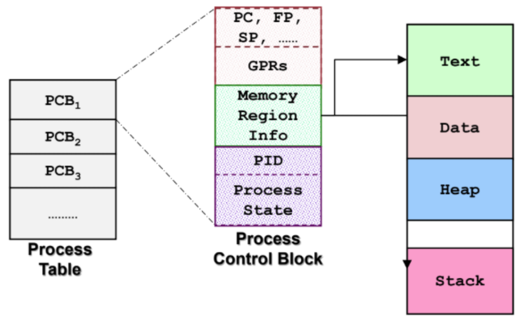


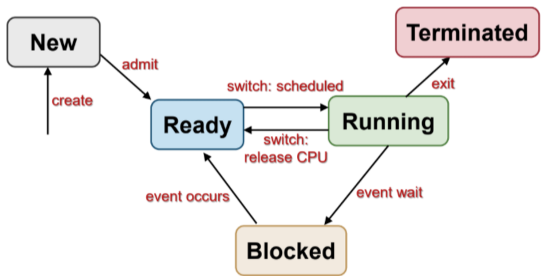
2. Processes and Threads

Processes

Process is an abstraction for a running program.

i. Process ID (PID): uniquely identifies a process
ii. Process State: indicates execution status
iii. Process Control Block (PCB): stores process execution context (registers, address space, etc.)
iv. Process Table: maintains PCBs in kernel
v. UNIX processes stores parent PID, CPU time, etc. with `init` as root process





POSIX System Calls:

i. `fork()`: duplicate process and PCB to new PID; memory is copy-on-write (shared until modified)
ii. `execl()`: replace core image of process
iii. `exit()`: end process and return exit status
iv. `wait()`: block until a child exits, then cleanup; `waitpid()`: wait for specific process
   - Zombie: child exited without cleanup
   - Orphan: parent exited before child; `init` becomes parent and cleans up

Scheduling

Schedulers order process execution with policy types:

i. Non-Preemptive: processes run until block or yield.
ii. Preemptive: processes have fixed time quota.
   - Interval of Timer Interrupt (ITI): Scheduler trigger time ($1 - 10ms$)
   - Time Quantum: Max time quota; must be multiple of ITI ($5 - 100ms$)

and algorithms dependent on environment and criteria:

i. Batch Processing (no user interaction) vs Interactive (responsive)
ii. Fairness: processes have a fair share of CPU time (per process or per user basis) with no starvation.
iii. Utilization: all parts of computer should be used.

Batch Scheduling Algorithms

Criteria:

i. Throughput: Jobs completed per unit time
ii. CPU Utilization: Percentage of time CPU busy
iii. Turnaround Time: finish time − arrival time
iv. Waiting Time: time in ready queue

First-Come First-Serve (FCFS) uses FIFO queue for jobs based on arrival time.

+ Starvation-Free (assuming all tasks complete)
− Convoy Effect: Long task at start can greatly increase turnaround time for shorter tasks later

Shortest Job First (SJF) selects job with smallest total CPU time (guess if info not available).

+ Minimises Average Waiting Time
− Starvation Possible: Long jobs may never run

i. CPU time: $pred_{n+1} = \alpha \cdot actual_n + (1 - \alpha) \cdot pred_n$; where α is weight placed on recent event

Shortest Remaining Time (SRT) selects job with smallest remaining time.

+ Preemptive: Short jobs can preempt current job; provides good service for short jobs even when late

Interactive Scheduling Algorithms

Criteria:

i. Response time: time between request and response
ii. Predictability: variation in response time

Round Robin (RR) uses FIFO queue for jobs, each run with fixed time quantum then moved to end of queue if ready. Blocked tasks are queued elsewhere until ready.

i. Response Time Guarantee: n^{th} task by $(n-1)q$
ii. Choice of Time Quantum: If big, better CPU utilization but longer waiting time. If small, more context switch overhead but shorter waiting time.

Priority Scheduling chooses task with highest priority. If preemptive, higher priority tasks preempt lower priority.

+ Handles Priority
− Starvation Possible: Low priority tasks may never run, especially if preemptive. Solutions:
   - Give current process a time quantum
− Priority Inversion: Medium priority job M makes progress despite higher priority job H waiting. H unable to preempt M because it is blocked waiting (for a resource) for a low priority job L, which also cannot preempt M to release the resource.

Multi-Level Feedback Queue (MLFQ) uses priority scheduling, with RR for same priorities. New tasks have highest priority. Tasks fully using time quantum reduce priority; while yield/block early retains priority.

+ Adaptive: Adjusts to actual behaviour
+ Minimises: Response time for IO bound tasks, and turnaround time for CPU bound tasks
− Abuse Possible: If process keeps yielding early

Lottery Scheduling gives out "lottery tickets" to tasks, with a random task chosen during scheduling.

+ Responsive: New tasks can participate immediately
+ Level of Control: Tickets can be distributed to child processes; more tickets for higher priority; each resource can have its own ticket distribution
+ Simple to Implement

Inter-Process Communication

Shared-Memory involves a process creating a shared memory region which is attached to by another process.

+ Efficient: only create and attach involve OS
+ Ease of Use: shared region behaves as normal
− Synchronization Necessary
− Difficult to Implement

i. `shmget()`: create/get shared memory segment
ii. `shmat()`: attach shared memory to address space
iii. `shmdt()`: detach shared memory
iv. `shmctl()`: control shared memory (e.g. delete)

```
#include <stdio.h>
#include <sys/shm.h>

// Create (shmid == -1 on err)
int shmid = shmget(IPC_PRIVATE,
   sizeof(int), IPC_CREAT | 6000);

// Attach (shm == (int*) -1 on err)
int *shm = (int*) shmat(shmid, NULL, 0);

// Read/Write
int x = shm[0];
shm[0] = 10;

// Detach and delete
shmdt((void*) shm);
shmctl(shmid, IPC_RMID, 0);
```

Message Passing involves a process sending a message to a receiving process. Messages can be explicitly named to sender/receiver or to shared mailbox. Primitives can be blocking (synchronous) or non-blocking (asynchronous).

+ Portable
+ Implicit Synchronization (if blocking primitives)
− Inefficient: usually require OS intervention
− Hard to Use: messages limited in size/format

i. `SEND(r, msg)` and `RECV(s, msg)`

UNIX Pipes provide a half-duplex (unidirectional)/ full-duplex (bidirectional) byte stream from writing process to reading process(es).

+ Implicit Synchronization: writer blocks if full, reader blocks if empty

i. `pipe()`: scan read and write fds into `int[2]`
ii. `dup()`: duplicate argument fd into lowest unused fd
iii. `dup2()`: duplicate argument fd into specified fd
iv. `read()`, `write()`: read/write to fd with data
v. `close()`: close a file descriptor

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define READ_END 0
#define WRITE_END 1

int fd[2];
char buffer[20];
char *str = "test";

pipe(fd);

if (fork() > 0) { // Parent
  close(fd[READ_END]);
  write(fd[WRITE_END],
        str,
        strlen(str) + 1);
  close(fd[WRITE_END]);
} else {          // Child
  close(fd[WRITE_END]);
  read(fd[READ_END],
       buffer,
       sizeof(buffer));
  printf("Received: %s\n", buffer);
  close(fd[READ_END]);
}
```

UNIX Signals are asynchronous notifications sent to processes to inform of events. Signals must be explicitly handled by default handler or user defined handler*.

i. Common Signals:
   - SIGINT: interrupt (Ctrl-C)
   - SIGTERM: terminate request
   - SIGSEGV: segmentation fault
   - SIGKILL: force termination (no handling*)
   - SIGSTOP: pause execution (no handling*)
   - SIGCONT: resume a stopped process

ii. `signal()`: assign a handler to a signal
iii. `kill()`: send SIGKILL
iv. `raise(sig)`: send signal to current process

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void seg_handler(int signo) {
  if (signo == SIGSEGV) {
    printf("Caught SIGSEGV\n");
    exit(1);
  }
}

int main() {
  int *test = NULL;
  if (signal(SIGSEGV, seg_handler) ==
   SIG_ERR)
    printf("Registration failed\n");
  *test = 10;
}
```

Threads

Threads are lightweight processes that share memory with all threads in the same process, managed by thread tables.

- + Economical: cheaper creation and context switching
- + Resource Sharing: process resources can be shared
- + Concurrent: multithreading improves responsiveness and can benefit from multiple CPUs
- – Parallel System Calls may be called and must be checked for correctness
- – Process Operations (e.g. `fork()`) must be consistent when executed with threads

Resource Distribution:

1. Shared: Text, Data, Heap, Open Files
2. Private: Thread ID, Registers, "Stack" (i.e. just changing SP, FP)

POSIX System Calls:

1. `pthread_create(pthread_t* tid, const pthread_attr_t*, func, void* args)`
2. `pthread_exit(void* exitValue)`
3. `pthread_join(pthread_t tid, void** status)`

Implementations

User Threads implements threads per-process as library.

- + Portable: implementable on all OSes
- + Ease of Use: just library calls
- + Configurable: user can self-manage synchronisation
- – Threads blocking will block whole process

Kernel Threads implements threads in the OS.

- + Parallel: threads can run across many CPUs
- – Slower: thread operations are now system calls
- – Less Flexible: not customizable thread policies per-process

Hybrid Threads map a number of user threads to a number of kernel threads.

- + Maintains flexibility, efficiency and concurrency

Synchronization

Synchronization is vital at critical sections (CS): memory regions modified concurrently by processes that are subject to errors caused by race conditions.

Correctness Properties:

1. Mutual Exclusion: at most 1 process in CS
2. Progress: if no process is in CS, a waiting process should be granted access
3. Bounded Wait: after process P_i requests to enter CS, there exists upper bound of times other processes can enter before P_i
4. Independence: process not executing in CS should not block other process

Symptoms of Incorrect Synchronization:

1. Deadlock: all processes are blocked
2. Livelock: due to deadlock avoidance, processes are not blocked but not making progress
3. Starvation: a process is unable to access CS

Implementations

Test and Set Lock (TSL): atomic assembly instruction that loads value at memory location into the register and changes the value at memory location to 1.
Variants: Compare and Swap, Atomic Swap (XCHG), Load Link / Store Conditional

- + Atomic
- – Busy Waiting: wastes processing power

```
enter_region: TSL REG, LOCK
              JNE REG, $0, enter_region
              RET

leave_region: MOV LOCK, $0
              RET
```

Peterson's Algorithm: 2-process software lock using flags and a turn variable.

- – Busy Waiting: wastes processing power
- – Low Level: error prone
- – Not General: limited to just mutual exclusion

```
// Process i
flag[i] = 1;
turn = j;
while (flag[j] && turn == j);
// critical section
flag[i] = false;
```

Semaphores: protected integer accessed only via two atomic operations `WAIT(S)` and `SIGNAL(S)`.
Binary semaphore or mutex has values 0 or 1.

1. `WAIT(S)`: if $S \leq 0$ block; else $S \leftarrow S - 1$
2. `SIGNAL(S)`: $S \leftarrow S + 1$, unblock one process if any
3. Invariant: $S_{curr} = S_{init} + \#SIGNAL(S) - \#WAIT(S)$; s.t. $\#SIGNAL(S)$ is number of `SIGNAL(S)` executed and $\#WAIT(S)$ is number of `WAIT(S)` completed
4. Number of processes in critical section: $N_{CS} = \#WAIT(S) - \#SIGNAL(S) \implies S_{curr} + N_{CS} = S_{init}$ (= 1 if binary semaphore)
5. `pthread_mutex_lock()`, `pthread_mutex_unlock()`

- – Deadlock possible with incorrect usage (e.g. acquiring locks in wrong order)

Conditional Variables: allow a process to wait for a condition and be signaled when condition occurs.

1. `pthread_cond_wait()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`

Producer-Consumer

Processes share a bounded buffer of fixed size K, where producers add items until buffer full and consumers remove items when not empty.

Busy Waiting Solution:

```
// Producer
while (1) {
 while(!can_prod);
  x = produce();

  wait(mutex);
  if (K > count) {
    buf[in] = x;
    in = (in+1)%K;
    count++;
    can_cons = 1;
  } else {
    can_prod = 0;
  }
  signal(mutex);
}
```

```
// Consumer
while (1) {
  while(!can_cons);

  wait(mutex);
  if (count > 0) {
    x = buf[out];
    out = (out+1)%K;
    count--;
    can_prod = 1;
  } else {
    can_cons = 0;
  }
  signal(mutex);

  consume(x);
}
```

Blocking Solution (init `not_full = K, not_emp = 0`):

```
// Producer
while (1) {
  x = produce();

  wait(not_full);
  wait(mutex);
  buf[in] = x;
  in = (in+1)%K;
  count++;
  signal(mutex);
  signal(not_emp);
}
```

```
// Consumer
while (1) {
  wait(not_emp);
  wait(mutex);
  x = buf[out];
  out = (out+1)%K;
  count--;
  signal(mutex);
  signal(not_full);

  consume(item);
}
```

Reader-Writer

Processes share a critical region, where readers can read simultaneously but writers must have exclusive access.

```
// Writer
while (1) {
  wait(empty);
  write();
  signal(empty);
}
```

```
// Reader
while (1) {
  wait(mutex);
  readers++;
  if (readers == 1)
    wait(empty);
  signal(mutex);

  read();

  wait(mutex);
  readers--;
  if (readers == 0)
    signal(empty);
  signal(mutex);
}
```

i. `mutex`: syncs readers, `empty`: sync readers, writers
ii. Only the first and last reader will update `empty`

– Writer Starvation: possible if readers keep entering and `empty` is never signalled. Solutions:
  - Queue: all pass through `wait(queue)`
  - Writer-Priority: readers entering must `wait(readTry)`; but first waiting writer blocks new readers with `wait(readTry)` and does `signal(readTry)` when done

Barrier

```
int i_am_last = 0;
wait(mutex);
count++;
if (count == N) {
  i_am_last = 1;
  signal(barrier);
}
signal(mutex);

wait(barrier);
if (!i_am_last) signal(barrier);
```

Dining Philosophers

N philosophers around a circular table, with a single chopstick between. 2 chopsticks are needed to eat.

Tanenbaum's Solution:

```
// Philosopher
void p(int i) {
  while (1) {
    // think
    take(i);
    eat();
    put(i);
  }
}

void safe_to_eat
    (int i) {
  if (state[i] ==
   HUNGRY &&
   state[LEFT] !=
   EATING &&
   state[RIGHT] !=
   EATING) {
    state[i] =
   EATING;
    signal(s[i]);
  }
}
```

```
// Take Chopstick
void take(int i) {
  wait(mutex);
  state[i] =
   HUNGRY;
  safe_to_eat(i);
  signal(mutex);
  wait(s[i]);
}

// Put Chopstick
void put(int i) {
  wait(mutex);
  state[i] =
   THINKING;

   safe_to_eat(LEFT);

   safe_to_eat(RIGHT)
  signal(mutex);
}
```

Limited Eater Solution, when at most $N-1$ philosophers eat concurrently, it's guaranteed at least one can eat:

```
void philosopher(int i) {
  while (1) {
    // think
    wait(seats);
    wait(chopstick[LEFT]);
    wait(chopstick[RIGHT]);
    eat();
    signal(chopstick[RIGHT]);
    signal(chopstick[LEFT]);
    signal(seats);
  }
}
```

3. Memory Management

Memory management ensures that processes in main memory can run concurrently, safely, and efficiently.

i. **Static Relocation**: all memory references modified by offset at load time.
   – Slow loading time
   – Difficult to identify memory references

ii. **Dynamic Relocation**: special base and limit registers store start and size of process.
   – Slow runtime access (add base + limit check)

iii. **Logical Addresses**: programs have independent virtual address space which Memory Management Unit (MMU) maps to physical addresses.

Contiguous Memory Allocation

Processes allocated contiguous blocks of physical memory.

Fixed-Size Partition:

+ Easy to manage
+ Fast allocation (all free partitions are the same)
– Internal fragmentation
– Size must be large enough for largest process

Variable-Size Partition have process-dependent partitions. Allocation requires OS to maintain a list/bitmap of partitions and holes.

+ No internal fragmentation
– Complex to manage
– External fragmentation, requires time-consuming compaction to consolidate holes
– Slower allocation to find space:
  - First Fit: use first hole large enough
  - Best Fit: find smallest hole large enough
  - Worst Fit: find largest hole

Buddy Allocation splits memory into blocks of size 2^k.

+ Easy coalescing (flip bit k for buddy)
– Internal fragmentation
– Requires bookkeeping to track block sizes

Disjoint Memory Allocation

Processes can occupy non-contiguous memory regions. OS maintains maps between logical and physical memory.

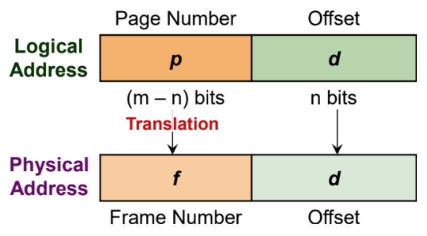
Paging

Splits logical memory into fixed-size pages mapped by page table into physical memory with equal-sized frames.

+ No external fragmentation
+ Protection with page table entry bits:
  - Access Rights: write, read, exec
  - Valid: if process active; memory access checked against this bit to catch out-of-range access
– Internal fragmentation (last page may not be full)
– Require two memory accesses per memory reference (lookup page table entry + access physical address)

Logical addresses are grouped into pages by their higher-order page bits and mapped to a frame:

i. 2^N byte page size $\implies$ N offset bits



Translation Lookaside Buffer (TLB) caches page table entries to avoid 2 slow memory lookups per reference (get frame number + access actual item).
On context switch, TLB is flushed and new PT brought in.

i. Hit: frame number used to find physical address
ii. Miss: retrieve page table entry from page table, then use frame number to find physical address

Page sharing involves multiple processes's entries to map to the same frame (e.g. shared libraries, `fork()`)

i. Copy-on-Write: initially `write` bit = 0; on write, OS duplicates frame and updates page table with new frame number, `write` = 1, and modifies frame

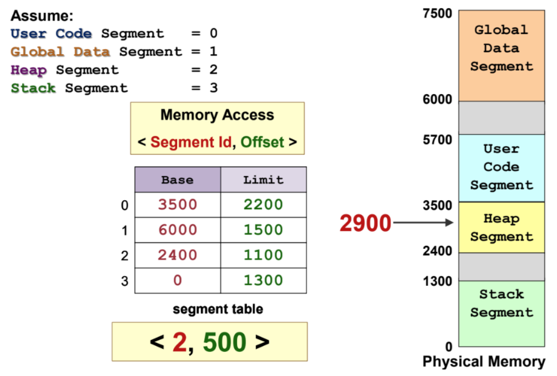
Segmentation

Splits logical memory into contiguous variable-sized segments, each with base and limit in a segment table.

+ Segments can grow/shrink independently
+ Segments can be protected/shared independently
– External fragmentation possible

Logical addresses are grouped by segment ID and mapped to a segment:

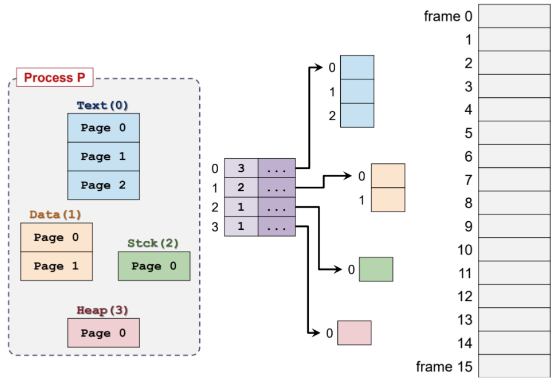
i. Logical Addr. = `<SegID, Offset>`; where `SegID` maps a segment's `<Base, Limit>` in segment table
ii. Physical Addr. = `Base + Offset`; where `Offset` < `Limit` if valid



	Base	Limit
0	3500	2200
1	6000	1500
2	2400	1100
3	0	1300

Segmentation with Paging

Segments are divided into fixed-size pages instead of being stored contiguously. Each segment has its own page table. Segments grow/shrink by adding or removing pages.



Virtual Memory

Virtual memory enables larger programs and more programs to run on limited main memory.

i. Pages are stored in secondary storage (e.g. disk)
ii. Pages are loaded to main memory on-demand
iii. Page table uses `isResident` bit to track if pages are memory resident (in physical memory) or non-memory resident (in secondary storage)

CPU can only access memory resident pages.
Attempt to access non-memory resident causes page fault:

i. Hardware traps to OS (page fault interrupt)
ii. OS locates page on disk
iii. Run page replacement and update page table
iv. Restart instruction that caused page fault

Page Table Structures

Direct Paging keeps all page table entries in a single table.

− Too large for big address spaces (e.g. 2^{20} entries for 32-bit addresses, 4KB pages)

i. M-bit address and 2^N byte page size $\implies N$ offset bits and $M-N$ page number bits

2-Level Paging divides page table into smaller page tables indexed via a page directory.

+ Space-efficient: only allocates page subtables for used regions

i. 2^P direct paged table entries and 2^K subtables $\implies K$ index bits and 2^{P-K} entries per subtable

Inverted Page Table keeps one entry per frame, each entry mapped to the current to the current occupying process and page's <PID, PageNum>.

+ Space-efficient: only allocate for smaller number of physical page frames
+ Frame management is easier and faster
− Slower lookup; requires searching or hashing

Page Replacement Algorithms

Page replacement occurs when there is no free physical frame during a page fault, and page eviction is required.

Evicted Page Write Policy:

i. Clean Page: no need to write back
ii. Dirty Page: write back to secondary memory

Time for memory access, with probability of page fault p:

$$T_{access} = (1-p)\cdot T_{mem} + p \times T_{pagefault}$$

Optimal (OPT) evicts page not reused for longest time.

+ Guarantees minimal page fault rate
− Not implementable (needs future knowledge)

FIFO evicts oldest loaded page.

+ Easy implementation with queue
− Belady's Anomaly: more frames can increase faults (because not exploiting temporal locality)

LRU evicts least recently used page.

+ Approximates OPT, giving good results
+ Does not suffer from Belady's Anomaly
− Hard to implement: keep track of last access
  - Counter: maintain logical time counter, incremented on every memory reference; store time into PTE on every page reference.
    − Need to search through all pages
    − "Time of use" always increases; can overflow
  - Stack: maintain stack of page numbers, remove page from stack and push it to top on reference, then replace page at bottom of stack
    − Not pure stack; hard to implement hardware

Second Chance (CLOCK) evists oldest loaded page after giving a second chance to referenced page.

i. Store pages in a circular queue with reference bits
ii. On page load, set reference bit to 0
iii. On page access, set reference bit to 1
iv. During page replacement, if oldest page reference bit is 1, unset and requeue, else replace it

Working Set Model

Working Set Model reduces page faults by ensuring each process has enough frames to hold its active pages.

i. $W(t,\Delta)$ = active pages in last Δ references at time t
ii. Allocate $\geq |W(t,\Delta)|$ frames to avoid excessive faults

Choosing Δ:

i. Too small $\Rightarrow$ working set misses some active pages
ii. Too large $\Rightarrow$ includes pages from different locality

Thrashing is when a process causes frequent page faults (e.g. every few instructions), typically because its working set exceeds its allocated frame count.

Frame Allocation

Proportion:

i. Equal: all processes get equal share of frames
ii. Proportional: frames distributed by process size

Scope:

i. Local: replace page of same process on page fault
   + Frames per process are constant every run
   − If frames per process are not enough, progress of process is hindered
   − Thrashing can be limited to a process, but it will hog I/O and degrade others' performance
ii. Global: replace page of any process
   + Allow self-adjustment between processes
   − Badly behaving processes can affect others
   − Frames per process can be different every run
   − Cascading Thrashing: thrashing process steals pages, causing other processes to thrash

Common Sizes

Byte Sizes:

i. 1 GiB = 2^{30} Bytes
ii. 1 MiB = 2^{20} Bytes
iii. 1 KiB = 2^{10} Bytes

4. File Systems

File systems manage data on secondary storage in a safe, persistent, self-contained and efficient manner.

Files

Files represent data created by process, and its metadata:

i. Name: human-readable reference to the file; extensions denote filetype (magic numbers can too)
ii. Identifier: unique ID used by file system
iii. Type:
   - Regular File: ASCII/ Binary data
   - Directory
   - Special File: UNIX Character/ Block files
iv. Size: file size
v. Protection: access rights (e.g. read/write/execute)
   - UNIX Permission Bits: `rwx` for owner, group, all
   - Access Control List: granular permissions on per-user or per-group basis
vi. Time, Data and Owner Information
vii. Table of Content: determines access method

Data Structures:

i. Byte Sequence: just raw bytes
   + Most flexible
ii. Fixed-Length Records: array of records
   + Easy to jump to any record:
      - Offset of n^{th} = size of record * $(n-1)$
iii. Variable-Length Records
   + Flexible
   – Harder to find records

Access Methods:

i. Sequential: read in order, cannot skip but can rewind
ii. Random: read in any order using either `read(offset)` or `seek(offset)`
iii. Direct: random access for records

Opened File Two-Table Approach:

i. System-wide: one entry to file/ inode per open file
ii. Per-process: one file descriptor in PCB per open file by process, each to the system-wide table
   - Point to two entries if opened by two processes
   - Point to same entry if after `fork()` or `dup()`

POSIX System Calls:

i. `open()`: open a file and return file descriptor
ii. `read()`, `write()`: read/write with file descriptor
iii. `lseek()`: move pointer of file descriptor
iv. `close()`: release file descriptor

Implementations

Contiguous Allocation allocates consecutive blocks.

+ Easy tracking: files just need start + length
+ Fast access: just need to seek first block
– External fragmentation: gaps after deletion
– File size must be specified in advance

Linked List (LL) stores each disk block in nodes (next block number, data), with first, last block number per file.

+ No external fragmentation
– Slow random access: must traverse linked list
– Extra pointer per block; unreliable if pointer wrong

File Allocation Table (FAT) is same as LL but stores all pointers in a FAT in memory.

+ Faster access than LL: traversal now in memory
– Takes up space: FAT tracks all disk blocks

Indexed Allocation (inode) uses one ”index” block per file to store pointers to all other blocks.

+ Less memory overhead: only index block of opened files need to be in memory
+ Fast direct access
– Limited max file size: number of blocks bounded by max array size of index block
– Index block overhead

Directories

Directories organise and manage files in a system.

Structures:

i. Single-Level: all files in one root directory.
ii. Tree: hierarchical tree of directories and files.
iii. DAG: possible when files are shared/linked
   - Hard Link: directory entries share same i-node
      + Low overhead: just represent with pointers
      – Deletion problems: when to delete file; resolve with storing count of references
   - Symbolic Link: symbolic link entry is a special link to target filepath which must be followed
      + Simple deletion: only delete file when owner deletes; symbolic links stay but don’t work
      – Larger overhead: special link file takes space
iv. General: possible when symbolic links create cycle
   – Hard to traverse (need stop infinite loop)
   – Hard to know when to remove a file/directory

POSIX System Calls:

i. `mkdir()`,`rmdir()`: create/remove a directory
ii. `chdir()`,`getcwd()`: change/ get current working dir
iii. `link()`,`symlink()`: create hard/ symbolic link
iv. `unlink()`: remove link from the filesystem

Implementations

Linked List stores directory entries sequentially.

+ Simple to implement
– Slow search: must traverse entries linearly

Hash Table maps filenames to files using hash function.

+ Fast lookup
– Hash table has limited size
– Depends on good hash function

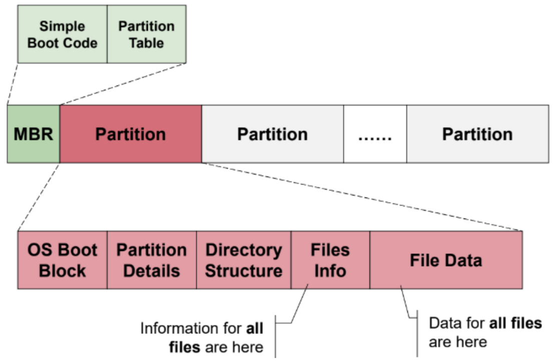
File Information Approaches:

i. Store everything in directory entry
ii. Store name in directory entry, and point to a different structure for other data

Disk Organisation

In BIOS:

i. Master Boot Record (MBR) at sector 0 contains boot code and partition table.
ii. Partitions can contain independent file systems



Free Space Management

Tracks free disk blocks on each partition to support file allocation and frees.

Bitmap represents each disk block by 1 bit.

+ Easy to manipulate: use bit operations to find first free block or n-consecutive free blocks.
− Need to keep in memory for efficiency reasons

Linked List maintains a list of free blocks by their indices.

+ Easy to find free block: take any block
+ Only first pointer needed (but others can be cached for efficiency)
− High overhead: mitigate by storing free block list itself in free blocks

I/O

Disk I/O has high latency due to seek and rotation delays; scheduling improves performance.

Disk Structure:

i. Track: one ring around the disk has many tracks of different radii
ii. Sector: sector of a track
iii. Disk head: point moving above the disk which transforms its magnetic field into electric current or vice versa
iv. By rotating, we change sector
v. By shifting the disk head between the center and the circumference, we change track

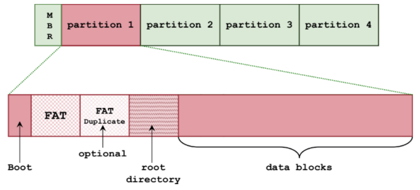
Scheduling Algorithms:

i. FCFS (First Come First Serve)
ii. SSTF (Shortest Seek Time First)
iii. SCAN (Elevator): services in one direction, then reverses
iv. C-SCAN (Circular SCAN): only services in one direction
v. Deadline: ensures deadline-based read/write ordering
vi. CFQ (Completely Fair Queuing): assigns time slices to processes
vii. BFQ (Budget Fair Queuing): shares I/O based on number of sectors requested

Case Studies

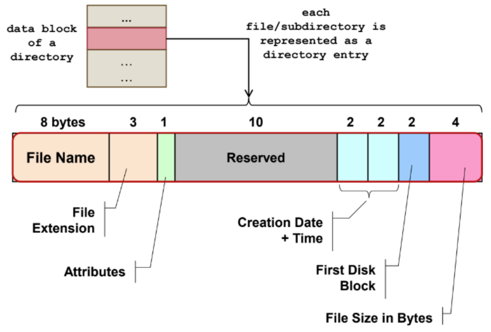
Microsoft File Allocation Table (FAT)

FAT is used in early DOS and removable devices.

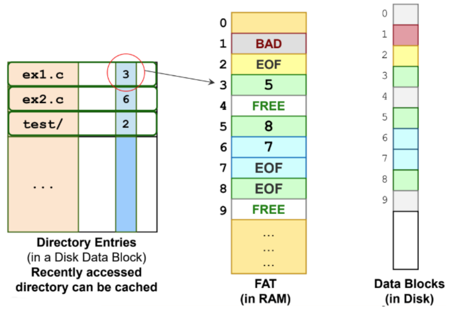


Directory is represented as a special file, where each directory entry is fixed 32-bytes storing:

i. Filename: max. 8+3 characters; first byte may have special meaning
ii. Attributes (e.g. read-only, hidden)
iii. Creation Time: years 1980 – 2017, time ±2s
iv. First Disk Block Index: X-byte length for FATX
v. File Size

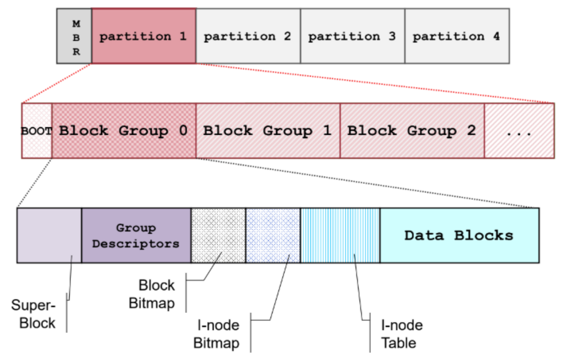


FAT entries contains FREE (block unused), Block number of next block, EOF (NULL ptr) or BAD (unusable)



Extended-2 File System (Ext2)

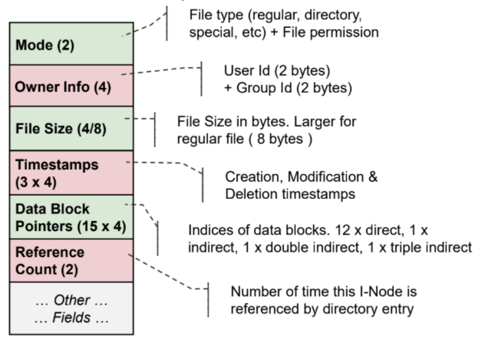
Ext2 is a UNIX file system used in early Linux distros.



Block Group Structure:

i. Superblock: describe whole file system (e.g. total inodes, disk blocks) and duplicated in each block group for redundance
ii. Group Descriptor Table: describe each block group (e.g. number of free disk bocks, free inodes, location of bitmaps) and duplicated in each block group
iii. Block Bitmap: tracks used/free blocks
iv. inode Bitmap: tracks used/free inodes
v. inode Table: stores inodes
vi. Data Blocks

inode Structure of 128 bytes:



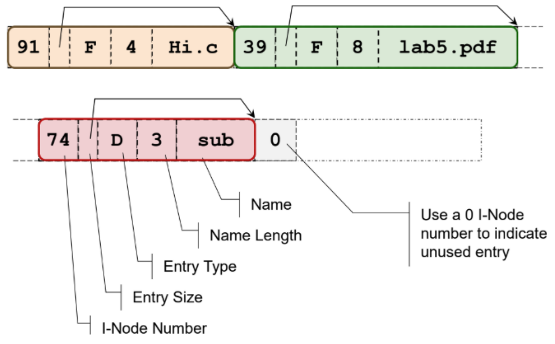
Mode (2)
Owner Info (4)
File Size (4/8)
Timestamps (3 x 4)
Data Block Pointers (15 x 4)
Reference Count (2)
… Other … … Fields …
File type (regular, directory, special, etc) + File permission
User Id (2 bytes) + Group Id (2 bytes)
File Size in bytes. Larger for regular file (8 bytes)
Creation, Modification & Deletion timestamps
Indices of data blocks. 12 x direct, 1 x indirect, 1 x double indirect, 1 x triple indirect
Number of time this I-Node is referenced by directory entry

Multilevel Data Blocks:

i. 12 direct: point to actual data blocks
ii. 1 single indirect: points to block with pointers to data blocks
iii. 1 double indirect: points to block that points to blocks of pointers
iv. 1 triple indirect: points to block that points to blocks of blocks of pointers

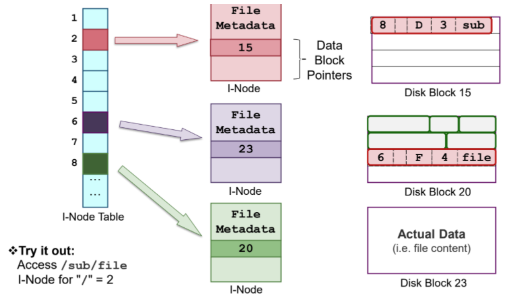
Directories are files containing directory entries:

i. inode Number: 0 indicates unused entry
ii. Size of Entry: for traversal to next entry
iii. Length of Name
iv. Type: regular file, directory or special
v. Name of File: max. 255 characters



Accessing a file (e.g. `/sub/file`):

i. Get root directory inode (usually fixed at 2)
ii. If next slug is subdirectory, locate directory entry, read the inode, change working directory and repeat after step 1
iii. Else if its a file, locate directory entry and read the inode terminating



Open:

i. Trace. If file is not found, throw error.
ii. Load file information with new entry in the system-wide table.
iii. Create a file descriptor in process's table to point to the new entry
iv. Return fd of this entry.

Delete:

i. Remove its directory entry by pointing previous entry to next entry. If its the first entry, make it a blank record.
ii. Update inode bitmap and mark the inode as free.
iii. Update block bitmap and mark blocks as free.

Hard Link:

i. Create new directory entry with same inode target.
ii. Update inode reference count.
iii. For deletion, decrement reference count; if it hits 0 perform actual deletion.

Symbolic Link:

i. Create new file and corresponding directory entry
ii. Store pathname of target file in new file