CS2106 Finals Cheatsheet v1.2 (2019-02-10)
by Julius Putra Tanu Setiaji, page 1 of 4

1 Introduction to OS

Motivation for OS: Manage resources and coordination (process sync, resource sharing), Simplify programming (abstraction of hardware, convenient services), Enforce usage policies, Security and protection, User program portability: across different hardware, Efficiency: Sophisticated implementations optimised for particular usage and hardware.

1.1 OS Structures

1.1.1 Monolithic

- Kernel is one BIG special program, various services and components are integral part
- Good SE principles with modularisation, separation of interfaces and implementation
- **Advantages**: Well understood, Good performance
- **Disadvantages**: Highly coupled components, Usually devolved into very complicated internal structure

1.1.2 Microkernel

- Kernel is very small & clean, only provides basic and essential facilities: IPC, address space & thread management, etc.
- **Higher level services** built on top of the basic facilities, run as server process outside of the OS, using IPC to communicate
- **Advantages**: Kernel is generally more robust & extensible, better isolation & protection between kernel & high level services.
- **Disadvantages**: Lower performance

1.2 Virtual Machine also known as Hypervisor

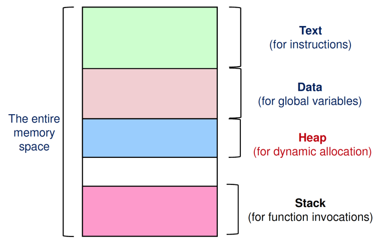
A software emulation of hardware – **virtualisation** of underlying hardware (illusion of complete hardware).

- **Type 1 Hypervisor**:
  Provides individual VMs to guest OS's (e.g. IBM VM/370)
- **Type 2 Hypervisor**:
  Runs in host OS, guest OS runs inside VM (e.g. VMware)

2 Process Abstraction

2.1 Process Abstraction

- **Process** = a dynamic abstraction for executing program
- Information required to describe a running program (Memory context, hardware context, OS context)
- An executable binary consists of two major components: instructions and data
- During execution, more information:
  - **Memory context**: text, data, stack, heap
  - **Hardware context**: General Purpose Registers, Program Counter, Stack Pointer, Stack FP, ...
  - **OS context**: PID, Process state, ...


The entire memory space
Text (for instructions)
Data (for global variables)
Heap (for dynamic allocation)
Stack (for function invocations)

Note: in this case, stack grows upwards, heap grows downwards

2.2 Stack Memory

- New memory region to store information of a function invocation
- Described by a **stack frame**, containing: Return address of the caller (PC, old SP), Arguments for the function, Storage for local variables, Frame Pointer, Saved Registers
- **Stack Pointer** = The top of stack region (first unused location)
- **Frame Pointer** = points to a fixed location in a stack frame
- **Saved Registers** = memory to temporarily hold GPR value during **register spilling**

2.2.1 Function Call Convention

E.g. On executing function call, **Caller**: Pass parameters with registers and/or stack, Save Return PC on stack; **Callee**: Save the old FP, SP, Allocate space for local vars on stack, adjust SP (Stack Pointer)
On returning from function call, **Callee**: Restore saved registers, FP, SP; **Caller**: Continues execution

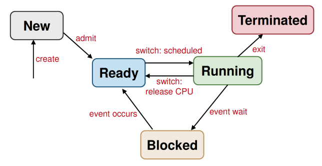
2.3 Dynamically Allocated Memory

Using a separate **heap memory region**

2.4 Process Identification & Process State

- Using process ID **(PID)**, a unique number among the processes.
- OS dependent: Are PID's reused? Are there reserved PID's? Does it limit max number of processes?

5 State Process Model:



- **Process State** = indication of the execution status

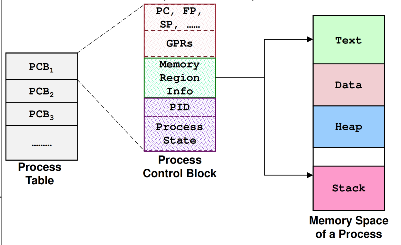
1. **New**: process created, may still be initialising, not yet ready
2. **Ready**: process is waiting to run
3. **Running**: process being executed on CPU
4. **Blocked**: process waiting, can't execute till event is available
5. **Terminated**: process finished execution, may require OS cleanup

Transitions:

- nil -> New (Create)
- New -> Ready (Admit): Process ready to be scheduled
- Ready -> Running (Switch): Process selected to run
- Running -> Ready (Switch): Process gives up CPU voluntarily or preempted by scheduler
- Running -> Blocked (Event wait): e.g. syscall, waiting for I/O, ...
- Blocked -> Ready (Event occurs)

2.5 Process Table & Process Control Block

- **PCB/Process Table Entry** = entire execution context for a process
- **Process Table** = maintains PCB for all processes, stored as one table
- Issues: Scalability, Efficiency



2.6 System Calls

- API to OS – different from normal function call in that have to change from user mode -> kernel mode
- General System Call Mechanism:
  1. User program invokes the library call (using normal function call mechanism)
  2. Library call places the system call number in a designated location (e.g. register)
  3. Library call executes a special instruction to switch user -> kernel mode (commonly known as TRAP)
  4. In kernel mode, the dispatching to the appropriate system call handler by dispatcher
  5. System call handler is executed
  6. System call handler ended, control return to library call, switch kernel -> user mode
  7. Library call return to user program via normal function return mechanism

2.7 Exception & Interrupt

Exception:

- Synchronous, occurring due to program execution
- Effect: have to execute an **exception handler**, similar to a forced function call

Interrupt:

- External events interrupting execution, usually hardware-related
- Asynchronous, occurring independent of program execution
- Effect: execution is suspended, have to execute **interrupt handler**

2.8 Process Abstraction: Unix

- `int fork();` duplicate current executable, returns PID of newly created process (for parent) or 0 (for child)
- `int execl(const char *path, const char *arg0, ..., const char *argN, NULL);` replaces current executing process image, does not return unless error. Will not exit on error.
- `void exit(int status);` `status` is 0 for normal, else problematic. Does not return.
- `int wait(int *status);` returns the PID of terminated child, status stores exit status. Blocking.

Zombie process = (1) parent terminates before child – `init` becomes pseudo-parent, who will call `wait` on children (2) child process terminates but parent did not call `wait` – child becomes zombie, can fill up processs table

3 Process Scheduling

3 categories of **processing environment**: (1) **Batch Processing**: no user, no interaction, no need to be responsive, (2) **Interactive**: with active user interacting, need to be responsive, consistent in response time, (3) **Real-time Processing**: deadline to meet, usually periodic process

3.1 Criteria for Scheduling Algorithms

- **Fairness**: fair share of CPU time, no starvation
- **Balance**: all parts of the computing system should be utilised

3.2 Types of scheduling policies

- **Non-preemptive (cooperative)** – a process stays scheduled until it blocks/gives up the CPU voluntarily
- **Preemptive**: A process is given a fixed time quota to run (possible to block or yield early), at the end of the time quota, the running process is suspended.

3.3 Scheduling a process

1. Scheduler is triggered (OS takes over)
2. If context switch is needed: context of current running process is saved, placed on blocked/ready queue
3. Pick a suitable process **P** to run based on scheduling algorithm
4. Setup the context for **P**
5. Let process **P** run

3.4 Scheduling for Batch Processing

Criteria:

- **Turnaround time**: Total time taken
- **Throughput**: Rate of task completion
- **CPU Utilisation**: % of time when CPU is working on a task

3.4.1 First-Come First-Served (FCFS)

- Tasks are stored on a FIFO queue based on arrival time. Pick the head of queue to run until (task is done OR task is blocked). Blocked task removed from queue, when it is ready again, placed at back of queue like a newly arrived task.
- **Guaranteed** to have no starvation: no of tasks in front of task X in FIFO is always decreasing -> task X will get its chance eventually.
- Shortcoming: **Convoy Effect** – due to non-preemptiveness, one slow process (CPU intensive) slows down the performance of the entire set of processes.

3.4.2 Shortest Job First (SJF)

- Select the task with the smallest total CPU time, thus **guaranteeing** smallest average waiting time.
- Shortcomings: Need to know total CPU time for a task in advance (have to guess if not available), starvation is possible (biased towards short jobs, long jobs may never get a chance)
- Predicting CPU Time, common approach (**Exponential Average**): $\text{Predicted}_{n+1} = \alpha\text{Actual}_n + (1-\alpha)\text{Predicted}_n$, where α = degree of weighting decrease, higher α discounts older observations faster

3.4.3 Shortest Remaining Time (SRT)

- Select job with shortest remaining (or expected) time.
- Variation of SJF that is preemptive and uses remaining time.
- New job with shorter remaining time can preempt currently running job
- Provide good service for short jobs even when they arrive late

3.5 Scheduling for Interactive Systems

- Criteria:
  - **Response time**: Time between request and response by system
  - **Predictability**: Lesser variation in response time
- Preemptive scheduling algorithms are used to ensure good response time, thus scheduler needs to run periodically.
- **Timer interrupt** = interrupt that goes off periodically based on hardware clock
- Timer interrupt handler **invokes OS scheduler.**
- **Interval of Timer Interrupt (ITI)** typically 1-10ms
- **Time Quantum** = execution duration given to a process, can be constant/variable, must be multiple of ITI (commonly 5-100ms)

3.5.1 Round Robin (RR)

- Tasks stored in a FIFO queue, pick task from head of queue until (time quantum elapsed OR task gives up CPU voluntarily OR task blocks)
- Basically a preemptive version of FCFS
- Response time guarantee: given n tasks and quantum q, time before a task get CPU is bounded by $(n-1)q$
- Choice of time quantum: big = better CPU util, longer waiting time; small = bigger overhead (worse CPU util) but shorter waiting time

3.5.2 Priority Scheduling

- Assign a priority value to all tasks, select task with highest priority value.
- Preemptive: highest priority process can preempt running process with lower priority
- Non-preemptive: late coming high priority process has to wait for next round of scheduling
- **Shortcomings**: Low priority process can starve, worse in preemptive variant
- **Possible solutions**: Decrease the prioty of currently running process after every time quantum, Given the current running process a time quantum – this process not considered in the next round of scheduling
- Generally hard to guarantee/control exact amount of CPU time given to a process
- **Priority Inversion**: 3 processes, priorities Hi, Mi, Lo. L locks resource, M pre-empts L, A arrives and tries to lock same resource as L. Then M continues executing although H has higher priority.

3.5.3 Multi-level Feedback Queue (MLFQ)

- Adaptive, minimising both response time for IO-bound and turnaround time for CPU-bound
- Rules:
  - Priority(A) > Priority(B) -> A runs
  - Priority(A) == Priority(B) -> A and B in RR
  - New job -> highest priority
  - If a job fully utilised its time slice -> priority reduced
  - If a job gives up/blocks before it finishes the time slice -> priority retained
- **Shortcomings**: (1) Starvation – if there are too many interactive jobs, long-running jobs will starve, (2) gaming the scheduler by running for 99% of time quantum, then relinquish the CPU, (3) a program may change its behaviour CPU-bound -> interactive
- Possible solution:
  - **Priority boost**: after some time period S, move all jobs to the highest priority. Guaranteeing no starvation as highest priority -> RR, and the case when CPU-bound job has become interactive
  - **Better accounting**: Once a job uses up its time allotment at a given level, its priority is reduced

3.5.4 Lottery Scheduling

- Give out "lottery tickets" to processes. When a scheduling decision is needed, a ticket is chosen randomly among eligible tickets.
- In the long run, a process holding X% of tickets can win X% of the lottery held and use the resource X% of the time.
- Reponsive: newly created process can participate in next lottery
- Good level of control: A process can be given lottery tickets to be distributed to its child process, an important process can be given more lottery tickets, each resource can have its own set of tickets (different proportion of usage per resource per task)
- Simple implementation

4 Process Alternative – Threads

- Motivation:
  - Process is expensive: under `fork()` model – duplicate memory space and process context, context switch requires saving/restoration of process information
  - Hard for independent processes to communicate with each other: independent memory space – no easy way to pass information, requires Inter-Process Communication (IPC)
- A traditional process has a single thread of control – only one instruction of the whole program is executing at any one time. Instead, we add more threads of control such that multiple parts of the program are executing simultaneously conceptually.

4.1 Process and Thread

- A single process can have multiple threads
- Threads in the same process shares: **Memory Context** (text, data, heap), and **OS Context** (PID, other resources like files, etc.)
- Unique information needed for each thread: Identification (usually thread id), Registers (general purpose & special), "stack"
- Process context switch involves: OS Context, Hardware Context, Memory Context
- Thread switch within the same process involves: Hardware context (registers, "stack" – actually just changing FP and SP)

4.2 Benefits

- **Economy**: requires much less resources
- **Resource sharing**: no need for additional information passing mechanism
- **Responsiveness**: multithreaded programs can appear much more responsive
- **Scalability**: Multithreaded program can take advantage of multiple CPU's

4.3 Problems

- **System call concurrency** – have to guarantee correctness and determine the correct behaviour
- **Process behaviour** – impact on process operations, e.g. does `fork()` duplicate threads? If single thread executes `exit()`, hwo abut the whole process, etc.

4.4 Thread Models

- **User Thread**
  - Implemented as a user library, a runtime system in the process handles thread operations
  - Kernel is not aware of threads in the process.
  - **Advantages**: Multithreaded program on ANY OS, thread operations are just library calls, more conigurable and flexible (such as customised thread scheduling policy)
  - **Disadvantages**: OS is not aware of threads, scheduling is performed at process level. One thread blocked -> process blocked -> all threads blocked, cannot exploit multiple CPUs
- **Kernel Thread**
  - Implemented in the OS, thread operation as system calls.
  - Thread-level scheduling is possible
  - Kernel may make use of threads for its own execution
  - **Advantages**: Kernel can schedule on thread level
  - **Disadvantages**: Thread operation is a syscall (slower and more resource intensive), generally less flexible (used by all multithreaded programs – many features: expensive, overkill for simple program, few features: not flexible enough for some)
- **Hybrid Thread Model**:
  - Have both kernel and user threads, OS schedule on kernel threads only, user thread can bind to a kernel thread.
  - Great flexibility (can limit concurrency of any process/user)

4.5 Threads on Modern Processor (Intel Hyperthreading)

- Threads started off as software mechanism: Userspace lib -> OS aware mechanism
- Hardware support on modern processors, supplying multiple sets of registers to allow threads to run natively and parallelly on the same core: **Simultaneous Multi-Threading (SMT)**

4.6 POSIX Threads: `pthread`

- Standard by IEEE, defining API and behaviour.
- `int pthread_create(pthread_t* tidCreated, const pthread_attr_t* threadAttributes, void* (*startRoutine) (void*), void* argForStartRoutine);`
- `int pthread_exit(void* exitValue)`
- `int pthread_join(pthread_t threadID, void **status);`
- except for `pthread_exit`, return 0 = success

5 Inter-Process Communication

- 2 common IPC mechanisms: Shared-Memory & Message Passing
- 2 Unix-specific IPC mechanisms: Pipe and Signal

5.1 Shared-Memory

- General idea: Process p_1 creates a shared memory region M, process p_2 attaches m to its own memory space. p_1 and p_2 can now commmunicate suing memory region M
- OS involved only in creating and attaching shared memory region
- **Advantages**: Efficient (only initial steps involves OS), Ease of use (information of any type or size can be written easily)
- **Disadvantages**: Synchronisation (shared resource -> need to synchronise access), Implementation is usually harder
- In Unix: (1) create/locate shared memory region M, (2) Attach M to process memory space, (3) Read/Write M, (4) Detach M from memory space after use, (5) Destroy M (only 1 process, can only destroy if M is not attached)

5.2 Message Passing

- General idea: process p_1 prepares a message M and send it to process p_2, p_2 receives the message M
- Message has to be stored in kernel memory space, every send/receive operation is a syscall
- **Advantages**: Portable (can be easily implemented on different processing environment), Easier synchronisation (using synchronous primitive)
- **Disadvantages**: Inefficient (usually requiring OS intervention), Harder to use (message usually limited in size and/or format)

5.2.1 Naming (how to identify the other party in the comm):

- **Direct Communication**
  - Sender/receiver explicitly name the other party
  - Characteristics: 1 link/pair of communicating processes, need to know the identity of the other party
- **Indirect Communication**
  - Message are sent to/received from message storage (known as mailbox or port)
  - Characteristic: 1 mailbox can be shared among a number of processes

5.2.2 Synchronisation (behaviour of the sending/receiving ops)

- **Blocking primitives** (synchronous): sender/receiver is blocked until message is received/has arrived
- **Non-blocking Primitive** (asynchronous): sender resume operation immediately, receiver either receive message if available or some indication that message is not ready yet.

5.3 Unix Pipes

- A communication channel with 2 ends, for reading and writing.
- A pipe can be shared between 2 processes (producer-consumer)
- Behaviour: like an anonymous file, FIFO (in-order access)
- Pipe functions as **circular bounded byte buffer with implicit synchronisation**: writers wait when buffer full, readers wait when buffer empty
- Variants: Multiple readers/writers, half-duplex (unidirectional) or full-duplex (bidirectional)
- `int pipe(int fd[]);` returns 0 = success. `fd[0]` reading end, `fd[1]` writing end

5.4 Unix Signal

- An async notification regarding an event sent to a process/thread
- Recipient of signal handle by a default set of handlers OR user-supplied handler
- Common signals in Unix: SIGKILL, SIGSTOP, SIGCONT, etc.

6 Synchronisation

6.1 Race Condition

- When 2/more processes execute concurrently in interleaving fashion AND share a modifiable resource resulting in non-deterministic execution.
- Solution: designate code segment with race condition as **critical section** where at any point in time only 1 process can execute.

6.2 Critical Section

Properties of correct implementation:

- **Mutual Exclusion**: if a process is executing in critical section, all other processes are prevented from entering it
- **Progress**: If no process is in critical section, one of the waiting processes should be granted access
- **Bounded Wait**: After a process p_i requests to enter the critical section, $\exists$ an upper-bound of number of times other processes can enter the critical section before p_i
- **Independence**: process not executing in critical section should never block other processes

Symptoms of incorrect synchronisation:

- **Deadlock**: all processes blocked -> no progress
- **Livelock**: processes keep changing state to avoid deadlock and make no other progress, typically processes are not blocked
- **Starvation**: some processes are blocked forever

6.3 Implementations of Critical Section

6.3.1 Test-and-set: an atomic instruction

- Load the current content at `MemoryLocation` into `Register`, Stores a 1 into `MemoryLocation`
- Disadvantage: busy waiting – wasteful use of processing power

6.3.2 Peterson's Algorithms

```
bool flag[2] = {false, false};
int turn;
```

```
flag[0] = true;
turn = 1;
while (flag[1] && turn == 1)
↪ {
  // busy wait
}
// critical section
flag[0] = false;
```

```
flag[1] = true;
turn = 0;
while (flag[1] && turn == 0)
↪ {
  // busy wait
}
// critical section
flag[1] = false;
```

Disadvantages:

- **Busy Waiting**, wasteful use of processing power
- **Low level**: higher-level programming construct desirable to simplify mutex and less error prone
- **Not general**: general synchronisation mechanism is desirable, not just mutex

6.3.3 Semaphore

A generalised synchronisation mechanism, providing a way to block a number of processes and a way to unblock one/more sleeping process(es)

- `wait(S)`: if S is (+)-ve, decrement. If S is now (-)ve, go to sleep
- `signal(S)`: increment S, if pre-increment S negative, wakes up 1 sleeping process

Properties

- Given $S_{\text{initial}} \geq 0$, where #signal(S) = no of `signal()` executed, #wait(S) = no of `wait()` completed
- **Invariant**: $S_{\text{current}} = S_{\text{initial}} +$ #signal(S) – #wait(S)

Binary semaphore, S = 0 or 1 known as mutex (mutual exclusion)
Deadlock still possible

6.4 Classical Synchronisation Problems

- **Producer-Consumer**: produce only if buffer not full, consume only if buffer not empty
- **Reader-Writers**: writer exclusive access, reader can share
- **Dining Philosophers**: assign partial order to the resources, establishing convention that all resources will be requested in order. E.g. label forks 1-5, and always pick up lower-numbered fork first.

6.5 Synchronisation Implementations

- POSIX semaphores
- pthread_mutex_t: `pthread_mutex_lock`, `pthread_mutex_unlock`
- pthread_cond_t: `pthread_cond_wait`, `pthread_cond_signal`, `pthread_cond_broadcast`

7 Memory Management

- **Logical address**: how the process views its memory space
- **Base register**: start offset
- **Limit register**: range of memory space of current process

7.1 Contiguous Memory Management

- 2 schemes of allocating memory partition:
  - **Fixed-size**
    Pros: Easy to manage, fast to allocate (all free partitions are the same, no need to choose), **Cons**: partition size need to be large enough to contain the largest process (internal fragmentation)
  - **Variable-size**
    Pros: Flexible, removes internal fragmentation, **Cons:** Need to maintain more information in OS, takes more time to locate appropriate region, external fragmentation
- **Merging and Compaction**: consolidate holes (time-consuming)
- **Allocation algo**: **First-Fit**: first large enough hole, **Best-Fit**: smallest large enough hole, **Worst-Fit**: largest hole

7.1.1 Allocation Algo: Buddy System

- Provides efficient: (1) Partition splitting, (2) locating good match of free partition, (3) Partition de-allocation and coalescing
- Free block is split into half repeatedly to meet request, two halves form buddy blocks. When buddy blocks are both free, merge.
- Keep array `A[0..K]` where 2^K = largest allocatable block size.
- `A[J]` is a linked list, keeps tracks of free block(s) of size 2^J
- Blocks B and C are buddy of size S, if the Sth bit of B and C are complements (0 and 1) and the leading bits up to the Sth bit are the same (e.g. 10100 and 10000)
- Allocating a block of size N:

```
S = Math.log2(N).ceil # smallest S s.t. 2^S <= N
while true
  # If got free block, remove from list and allocate
  allocate(A[S].shift) and return if A[S].size > 0
  # Else, find smallest R s.t. A[R] has a free block
  # repeatedly split s.t. A[S..R-1] has a new free block
  R = (S+1).upto(K).find { |i| A[i].size > 0 }
  (R+1).downto(S).each{ |i| split A[i], from: A[i+1] }
end
```

- To free a block B:

```
def free(B)
  S = Math.log2(B.size).ceil
  C = A[S].find_free_buddy_of(B)
  A[S].append(B) and return if C.nil? # buddy not free
  A[S].delete(C)
  B2 = merge(B, C) # merge buddies
  free(B2)
end
```

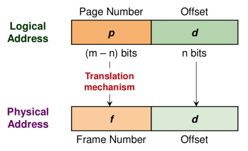
8 Disjoint Memory Schemes

8.1 Paging Scheme

- Physical mem is split into fixed-size regions: **Physical Frame**
- Logical mem of a process is split into regions of the same size: **Logical Page**
- Logical memory space is contiguous but occupied physical memory region can be disjointed.
- Paging removes external but **not internal** fragmentation.

8.1.1 Logical Address Translation

- **Page Table**: lookup table for logical address translation (logical page <-> physical frame). Need to know 2 things: F = physical frame number, Offset = from start of physical frame
- Physical address = $F\times$ physical frame size + Offset
- Given: frame size = 2^n, m bits of logical address



Logical Address LA:
p = most significant $m-n$ bits of LA,
d = remaining n bits of LA

Physical Address
$PA = f \times 2^n + d$

8.1.2 Implementation

- **Pure-software**: OS stores page table information in the PCB
  Issues: require 2 mem access for every mem ref (1) read indexed page table entry to get frame number, (2) access actual mem item
- **Hardware support**: Translation Look-aside Buffer (TLB) = cache of a few page table entries
  - Use page number to search TLB, either TLB-Hit or TLB-Miss
  - TLB is part of hardware context. Thus, context switch -> TLB entries are flushed

1. **(HW)** LA is decomposed into `<Page#, Offset>`
2. **(HW)** Search TLB for Page#:
   (a) TLB-Hit: Use `<Frame#, Offset>` to access physical mem. Done.
   (b) TLB-Miss: Trap to OS (TLB Fault)
3. **(OS - TLB Fault)** Access full table of the process (in PCB), `<Page#>` is the index:
   (a) If memory resident, replace a TLB entry, return from trap, retry step 1
   (b) If non-memory-resident, trap to OS (Page Fault)
4. **(OS - Page Fault)** Locate the page in secondary swap pages. Load the page into a physical frame (applying replacement algo if needed). Update PTE, return from trap, retry step 3

8.1.3 Protection

- **Access Right Bits**: each page table entries has several bits to indicate rwx, mem access is checked against the access right bits
- **Valid bit**: each page table entries has a bit to indicate if the page is valid for the process to access. OOB access caught by OS

8.1.4 Page sharing

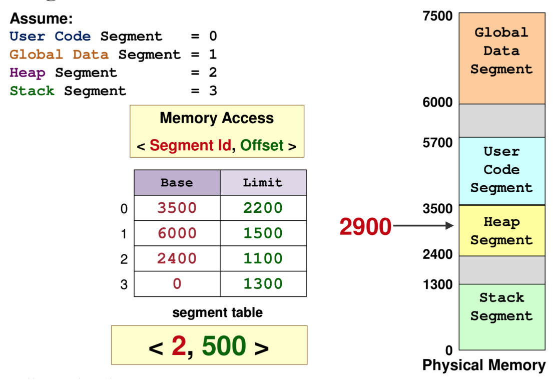
Deduplication. Possible usage:

- **Shared code page**: used by many processes, e.g. C stdlib, syscalls
- **Implement CoW**: upon forking, before any mem value mutation

8.2 Segmentation Scheme

- Some regions may grow/shrink at exec time. Thus putting them in non-contiguous mem space to allow them to grow/shrink freely and easier check whether access is in-range.
- Mem space of a process separated into different segments: user code, global vars, heap, stack, lib code, etc.
- Logical mem space of a process is a collection of segments
- Each segment: has a name and a limit
- All mem ref is `Segment Name + Offset` (e.g. Heap + 245)
- Each segment mapped to a contiguous physical mem region (base address + limit)
- **segment id** = single number representing segment name
- Logical address: `<SegID, Offset>` (SegID used to look up `<Base, Limit>` in a segment table)
- PA = Base + Offset, Offset < Limit for valid access
- **Pros**: Each segment is an independent contiguous mem space (can grow/shrink, protected/shared independently)
- **Cons**: Requires variable-size contiguous memory regions, external fragmentation

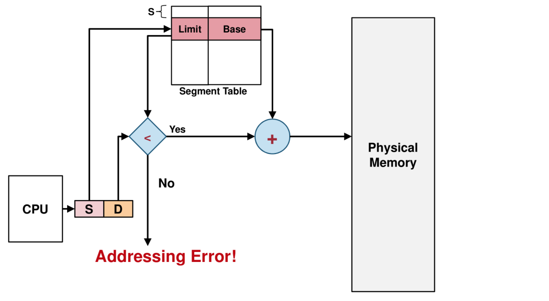
CS2106 Finals Cheatsheet v1.2 (2019-02-10)
by Julius Putra Tanu Setiaji, page 3 of 4

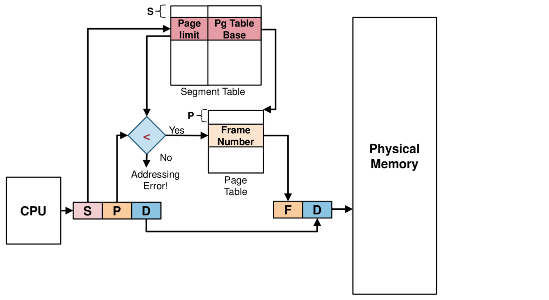


	Base	Limit
0	3500	2200
1	6000	1500
2	2400	1100
3	0	1300

8.3 Segmentation: Hardware Support and combined with Paging

Each segment is now composed of several pages instead of a contiguous memory region (each segment has a page table)
Table: (Segment#, Page Limit, Page Table Base)





9 Virtual Memory Management

- Use page table to translate **virtual** -> physical address
- Extend paging scheme: Some pages may be in physical memory, others in secondary storage.
- Use a `memory_resident?` bit to distinguish between 2 page-types: **memory-resident** and **non-memory-resident**
- CPU can only access memory-resident pages. **Page Fault** (PF) when CPU tries to access non-memory-resident page. OS needs to bring them into physical memory.
- Just like cache, exploit **Temporal** & **Spatial** Locality
- Accessing page X:
  1. If page X **memory resident**: Access physical mem. Done.
  2. Else **PF**: Trap to OS
  3. Locate page X in secondary storage
  4. Load page X into a physical memory frame
  5. Update page table
  6. Go to step 1

9.1 Page Table Structure

9.1.1 Direct Paging

- Keep all entries in a single table
- With 2^p pages in logical mem space, p bits to specify one unique page, 2^p PTE (each containing physical frame number + additional info bits), e.g. Virtual Address: 32 bits, page size = 4 KB, $p = 32 - 12 = 20$, size of PTE = 2 B, PT Size = $2^{20} \times 2$ bytes = 2MB

9.1.2 2-Level Paging

- Split page table into smaller page tables, each with a **PT number**
- Total 2^P entries, with 2^M smaller PTs, M bits to identify 1 PT, each smaller PT contains $2^{(P-M)}$ entries
- Use **page directory** to keep track, containing 2^M indices. Ideally, size of page dir = page size
- **Advantages**: corresponding PT to empty entries in page dir need not be allocated

9.1.3 Inverted Page Table

- Instead of keeping per-process PT, only a **single** mapping of physical frame to `<pid, page#>` (ordered by frame# instead of page#)
- **Advantage**: Memory saving, 1 table for all processes
- **Disadvantage**: Slow translation ($O(n)$ to look up a page#)

9.2 Page Replacement Algorithms

- When a page is evicted: **clean page** (not modified), **dirty page** (modified, must write back)
- Memory access time: $T_{access} = (1-p) \times T_{mem} + p \times T_{page\_fault}$ where p = prob of PF, T_{mem} = access time for mem-resident page, T_{page_fault} = access time if PF occurs
- Since $T_{page\_fault} >> T_{mem}$, need to reduce p to keep T_{access} reasonable (reduce total no of PF)

9.2.1 Optimal Page Replacement (OPT)

- Replace the page that will not be used again for the longest period of time. Guarantees min no of PF
- However, not realisable, need **future knowledge** of mem refs
- Base comparison for other algorithms

9.2.2 FIFO

- Mem pages are evicted based on their loading time
- Implementation: OS maintain a queue of resident page numbers. Remove the first page in queue if replacement is needed. Update the queue during PF trap.
- **Advantage**: simple to implement (no HW support needed)
- **Problems**: Belady's Anomaly (more frames, more PF) because FIFO does not exploit temporal locality

9.2.3 Least Recently Used (LRU)

- Make use of temporal locality. Does not suffer from Belady's Anomaly
- **Impl 1**: A logical time counter, incremented for every mem ref. PTE has a time of use field, replace page with smallest time of use.
- However, need to search through all pages, and counter is forever increasing (overflow)
- **Impl 2**: Use a stack. If page X is referenced, remove from stack, push on top of stack. Replace the page at bottom of stack.
- However, not a pure stack. Entries can be removed from anywhere in the stack. Hard to implement in hardware.

9.2.4 Second-Chance Page Replacement (CLOCK)

- Modified FIFO to give a second chance to pages that are accessed.
- Each PTE maintains a reference bit: **1** = accessed, **0** = not accessed
- Algo:
  1. Select oldest FIFO page.
  2. If ref_bit == 0, page is replaced.
  3. If ref_bit == 1, page is given 2nd chance: ref_bit cleared to 0. Arrival time reset (as if newly loaded). Next FIFO page is selected, go to step 2
- Impl: use a **circular queue** to maintain the pages, with a pointer to the oldest page. To find a page to be replaced, advance until a page with ref_bit == 0, and clear reference bit as pointer passes.
- When all ref_bit == 1, degenerate into FIFO

9.3 Frame Allocation

- N physical mem frames, M processes competing for frames.
- **Equal allocation** (each process gets $\frac{N}{M}$ frames), **Proportional Allocation** (Let $size_p$ = size of process p, $size_{total}$ = total size of all processes, each process gets $\frac{size_p}{size_{total}} \times N$ frames)
- Insufficient physical frame -> Thrashing (heavy IO to bring non-resident pages into RAM)

9.4 Page Replacement

- **Local Replacement**: victim page selected among pages of the process that causes PF
  Pros: Frames allocated to a process remain constant, perf is stable between runs. **Cons**: if frame allocated not enough, hinder progress of a process
  A thrashing process steals page from other process causing other process to thrash (**cascading thrashing**)
- **Global Replacement**: victim page chosen among all physical frames
  Pros: Allow self-adjustment between processes (process that needs more can get from other). **Cons**: badly-behaved process can affect others, frames allocated to a process can be different from run to run (thrashing can be limited to one process, but that process can hog the IO and degrade perf of other processes)

9.5 Working Set Model

- In a new locality, a process will cause PF for the set of pages. Afterwards no PF until process transits to a new locality
- Defines **Working Set Windows** Δ = time interval
- $W(t, \Delta)$ = active pages in the interval at time t
- interval = looking back
- Allocate enough frames for pages in $W(t, \Delta)$ to reduce prob of PF
- Accuracy of working set model directly affected by choice of Δ. **Too small**: may miss pages in the current locality, **Too big**: may contain pages from different locality

10 File System

General Criteria: **Self-contained**, **persistent**, **efficient**

10.1 File System Abstraction

10.1.1 File

- logical unit created by process, contains data and metadata (name, identifier, type [regular {ASCII, Binary}, directories, special files], size, access rights, timedate, owner, table of content)
- File **protection**: permission bits (owner, group, universe) (rwx), Access Control List (minimal ACL – same as permission bits/extended ACL – added named users/group)
- **Operations** on metadata: Rename, Change/Read attributes
- **Structure**: **Array of bytes**, **Fixed-length records** (array of records, can grow/shrink/jump to any record easily), **Variable length records** (flexible, but harder to locate a record)
- **Access methods: Sequential** (data read in order from beginning, cannot skip but can rewind), **Random** (data can be read in any order, **Direct access** (for file containing fixed-length records, allow random access to any record directly)
- **Generic operations**: create, open, r/w, repositioning, truncate
- OS provides file ops as syscalls. Information kept for opened file: **File pointer** current loc in file, **disk location**: actual file loc on disk, **open count**: how many process opens this file
- Organise open file information: **System-wide open-file table** (1 entry/unique file), **Per-process open-file table** (1 entry/file used in the process, each entry points to the system-wide table)
- Processes sharing file in unix: (1) diff file descriptors, I/O can occur at indep offsets, (2) same file descriptor, only 1 offset, I/O changes the offset for the other process (e.g. `fork` after file is opened)

10.1.2 Directory

- Used to (1) provide logical grouping of files, keep track of files
- Ways to structure directory:
  - **Single-level**
  - **Tree-structured**
    Dirs can be recursively embedded in other directories. 2 ways of referring to the file: **absolute/relative pathname**
  - **DAG**
    A file can be shared, only 1 copy of actual content appearing in multiple directories with diff path names. In Unix: hard link (pointers to the same actual file on disk, pros: low overhead, cons: deltion problems)/symbolic link (special link file containing pathanme, pros: simple deletion, cons: larger overhead)
  - **General Graph**
    Generally not desirable: hard to traverse (need to prevent infinite looping), hard to determine when to remove a file/dir.

11 File System Implementations

11.1 Disk Organisation

- **Master Boot Record** at sector 0 with partition table, followed by 1/more partitions, each containing an independent file system.
- Logical view, file = a collection of logical blocks
- When file size ≠ multiple of logical blocks, last block may contain wasted space (**internal fragmentation**)
- Good file implementation: keep track of the logical blocks, allow efficient access, disk space is utilised effectively. (focus on how to allocate file data on disk)

11.2 File Block Allocation

11.2.1 Contiguous

- Allocate consecutive disk blocks to a file
- **Pros**: simple to keep track (each file only needs startinb block number + length), fast access (only need to seek to first block)
- **Cons**: External frag, file size needs to be specified in advance

11.2.2 Linked List

- Keep a linked list of disk blocks, each disk block stores the next disk block numbers (pointer) and actual file data
- File information stores first and last disk block number
- **Pros**: solve fragmentation
- **Cons**: slow random access in a file, part of disk block is used for pointer, less reliable

11.2.3 Linked List Variant: FAT

- All block pointers in a single table: **File Allocation Table** (FAT)
- **Pros**: Faster Random Access (linked list done in memory)
- **Cons**: FAT can be huge when disk is large

11.2.4 Indexed Allocation

- Each file has an **index block** (an array of disk block addresses where `indexBlock[N]` = Nth block address)
- **Pros**: Lesser memory overhead (only index block of opened file needs to be in memory), fast direct access
- Cons: Limited maximum file size (max no of blocks == no of index block entries), index block overhead

11.2.5 Indexed Allocation Variations

- Schemes to allow larger file size
- **Linked scheme**: keep a linked list of index blocks
- **Multilevel index**: 1st level index block points to a number of 2nd level index blocks. Each 2nd level index blocks point to actual disk block. This can be generalised to any number of levels.
- **Combined scheme**: combination of direct indexing and multi-level index scheme, e.g. Unix I-node has 12 direct pointers, 1 single indirect, 1 double indirect, 1 triple indirect

11.3 Free Space Management

Free space list to know which disk block is free during file alloc

11.3.1 Bitmap

- Each disk block represented by a bit, 1 = free, 0 = occupied
- **Pros**: provide a good set of manipulations using bit-level ops
- **Cons**: need to keep in memory for efficiency reason

11.3.2 Linked List

- A linked list of disk blocks, each disk block contains a number of free disk block numbers; a pointer to next free space disk block
- **Pros**: Easy to locate block, only 1st pointer is needed in memory
- **Cons**: High overhead (mitigated by storing the list in free blocks)

11.4 Directory Structure

- Given a full path name, recursively search the directories along the path to arrive at the file information
- Each dir entry stores file name, metadata, (pointer to) file information, disk blocks information

11.4.1 Linear List

- Directory = a list of files
- Locate a file using list requires a linear search (inefficient for large directories or deep tree traversal). **Common solution**: use cache to remember the latest few searches

11.4.2 Hash Table

- Directory = hash table of size N
- Locate a file by file name: File name is hashed into index $K \in [0, N)$. **HashTable[K]** inspected to match file name, using chained collision resolution
- **Pros**: fast lookup
- **Cons**: hash table has limited size, depends on good hash function

11.5 File System in Action

11.5.1 Create /.../parent/F

1. Use full path name to locate parent dir, search for filename F to avoid duplicates. Search could be on cached dir structure
2. Use free space list to find free disk block(s)
3. Add an entry to parent directory with relevant file information

11.5.2 Process P open file /.../F

1. Search system-wide table for existing entry E. Not found, use full pathname to locate file F and load its file information to a new entry E in system-wide table. If not found terminate
2. Create an entry in the P's table, pointing to E
3. Return pointer to this entry.

11.6 Disk I/O Scheduling Algorithms

Time taken = Seek Time + Rotational Latency + Transfer Time

- Seek time = position disk head over proper track (average 2-10ms, approaches $\frac{1}{3}N$, where N = time for max seek distance)
- Rotational latency = wait for desired sector to rotate under the head (4800-15000 RPM, 12.5ms to 4ms, average 6.25ms at 4800 RPM, 2ms at 15000 RPM)
- Transfer Time = $\frac{\text{xfer size}}{\text{xfer rate}}$ (least sig factor) (xfer rate 70-125 MB/s)
- Problem: due to significant seek & rotational latency, OS should schedule disk I/O requests to reduce overall waiting time.
- Rotational latency hard to mitigate, focus on reducing seek time

11.6.1 FCFS

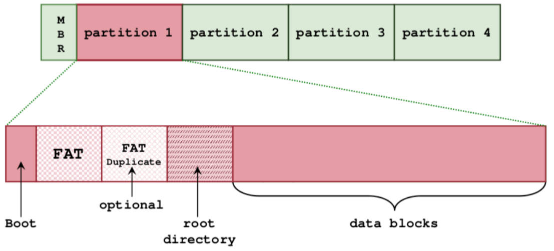
11.6.2 Shortest Seek First (SSF)

11.6.3 SCAN family (aka Elevator)

- Bi-directional (innermost <-> outermost) (SCAN)
- unidirectional (outermost -> innermost) (C-SCAN – Circular SCAN)

12 File System Case Studies

12.1 Microsoft FAT



- FAT12 -> FAT16 -> FAT32
- File data allocated to a number of data blocks/data block clusters
- Allocation info kept as a linked list, kept separately in the File Allocation Table (FAT) which is cached in RAM for linked list traversal: 1 entry per data block/cluster, store disk block information (FREE/EOF/BAD/next block number)
- Dir represented as special type of file
- Root dir is stored in a special location, other dirs are stored in the data blocks.
- Each file/subdir within the dir represented as **directory entry**
- Directory Entry 32-bytes, 8B+3B for file name+ext (first byte of file name may have special meaning: deleted, end of DE, parent dir, etc.), 1B for attributes, 10B reserved, 2B+2B for creation date+time (Year 1980-2107, time resolution ±2 seconds), for the first disk block index (different variant diff number of bits, 12, 16, 32 for FAT12, FAT16, FAT32 respectively), 4B for file size in bytes.

12.1.1 Accessing File/Dir

For a dir, disk blocks contain DE for files/subdirs within that dir

1. Use first disk block number stored in DE to find starting point
2. Use FAT to find out subsequent disk block numbers
3. Use disk block number to perform actual disk access on the data blocks.

12.1.2 File Deletion

1. "Delete" the DE (set first letter in filename to 0xE5)
2. Free data blocks: set FAT entries in linked list to FREE

Do not keep track of free space information – must be calculated by going through the FAT

12.1.3 Variants: FAT12, FAT16, FAT32

Support larger hard disk as a single partition

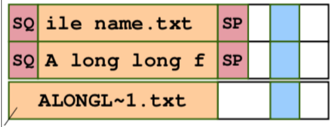
- **Disk cluster**: instead of using a single disk block as smallest alloc unit, use a number of contiguous disk blocks
- **Pros**: Larger cluster size -> larger usable partition
- **Cons**: Larger cluster size -> larger internal fragmentation
- On FAT32, further limitation: 32-bit sector count, only 28-bit used in disk block/cluster index.
- **Bigger FAT Size**
- e.g. 4KB cluster

FAT12	FAT16	FAT32
2^{12} clusters 4KB $\times 2^{12}$ = 16MB	2^{16} clusters 4KB $\times 2^{16}$ = 256MB	2^{28} clusters 4KB $\times 2^{28}$ = 1TB

- Actual size is a little lesser, special values (EOF, FREE, etc) reduces total number of valid data block/cluster indices.

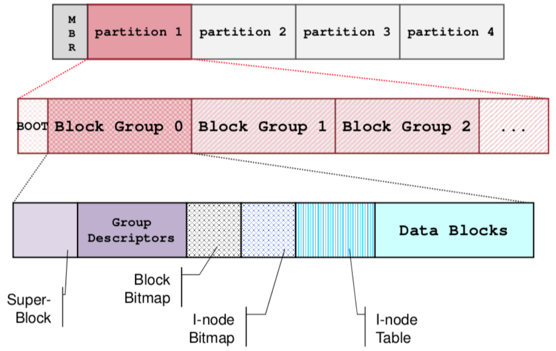
12.1.4 Long File Name Support

- Virtual FAT (VFAT) supports long file names up to 255 chars
- Use multiple DE for a file with long filename.
- Use a previously invalid file attribute so non-VFAT will ignore these additional entries
- Use the first byte in the filename to indicate sequence
- Keep the 8+3 short version for backward compatibility



12.2 Extended-2 File System

- Disk space split into **blocks**, blocks grouped into **Block Groups**
- Each file/dir is described by **I-Node** (Index Node) containing file metadata & data block addresses
- For all bitmaps, 1 = Occupied, 0 = Free



12.2.1 Superblock

- Describes the whole filesystem
- Duplicated in each block group for redundancy
- Includes total I-Nodes number, I-Nodes/group, total disk blocks, disk blocks/group, etc.

12.2.2 Group Descriptors

- Describes each of the block group – no of free disk blocks, free I-nodes, location of the bitmaps
- Duplicated in each block group

12.2.3 Block Bitmap

Keep track of the usage status of blocks of this block group

12.2.4 I-Node Bitmap

Keep track of the usage status of I-Nodes of this block group

12.2.5 I-Node Table

An array of I-Nodes, containing only I-Nodes of this block group

12.2.6 I-Node Structure (128 Bytes)

- 2B for Mode: Filetype (regular, dir, special, etc) + File permission
- 2B user id, 2B group id
- 4B for file size in bytes, 8B for regular files
- 4B each for access, creation, modification, deletion timestamps
- 4B each for 15 data block pointers: 12 direct, 1 single indirect, 1 double indirect, 1 triple indirect
- 2B for refcount (no of times this I-Node is references by DE)

12.2.7 I-Node Data Block

- Allows fast access to small file, flexibility in handling huge file
- e.g. disk block address = 4B, each disk block is 1KB, so each indirect block can store $\frac{1\text{KB}}{4}$ = 256 addresses
- Max file size = direct + single indirect + double indirect + triple indirect = $12 \times 1\text{KB} + 256 \times 1\text{KB} + 256^2 \times 1\text{KB} + 256^3 \times 1\text{KB}$ = 12KB + 256KB + 64MB + 16GB = 16843020 KB

12.2.8 Directory Structure

Data blocks of a dir store a linked list of DE for files/subdirs information within this dir

- I-Node number for that file/subdir (0 indicates unused DE)
- Size of this DE (to locate next DE)
- Length of the file/subdir name
- Type: File/subdir (or other special file type)
- File/subdir name (up to 255 chars)

12.2.9 Accessing file/dir

1. Root directory has a fixed I-Node (e.g. 2), read the actual I-Node
2. Look at the next part in pathname, locate DE in current dir, retrieve I-Node number, read actual I-Node
3. If dir, set current dir to this dir (read DE in pointed by I-Node)
4. Else if a file, read content from blocks pointed by I-Node

12.2.10 File deletion

1. Remove DE from parent dir (point prev entry to the next entry/end, if first entry, DE is replaced with blank record)
2. Update I-Node bitmap (mark as free)
3. Update block bitmap (mark as free)

12.2.11 Hard/Symbolic Link with I-Node

Dir A contains a file X, with I-Node# N, dir B wants to share X

- **Hard Link**: creates a DE in B, using same I-Node# N, can have diff filename. Increase refcount of I-Node# N
- **Symbolic Link**: creates a new file Y in B, Y contains the pathname of X

13 Additional Notes

13.1 Internal vs External Fragmentation

Internal fragmentation is the wasted space within each allocated block because of rounding up from the actual requested allocation to the allocation granularity.

External fragmentation is the various free spaced holes that are generated in either your memory or disk space. External fragmented blocks are available for allocation, but may be too small to be of any use.