CS2106 Cheatsheet 19/20 S1 Finals

Introduction

- Data: global variables
- Text: raw source code/instructions
- Stack: args/params/local vars/caller return addr
- Heap: dynamically allocated with `malloc`

Stack Frame Setup/Teardown

On executing function call:

- Caller: Pass arguments with Registers/Stack
- Caller: Save return PC on Stack
- Transfer control from caller to callee
- Callee: Save registers used by callee and old FP/SP
- Callee: Alloc space for local vars of callee on stack
- Callee: Adjust SP to point to new stack top

On returning from function call:

- Callee: Restore saved Registers, FP, SP
- Transfer control from callee to caller
- Caller: Continues execution in caller

Process Abstraction

- Exceptions: synchronous; due to program
- Interrupts: asynchronous; independent of program
- Memory Context: text, data, stack, heap
- Hardware Context: registers, PC, SP, FP
- OS Context: PID, process state, opened files
- PCB: stores all contexts of an executing process
- `exec()`: replaces current process with new process

Fork

- child → `fork() == 0`; parent → `fork() > 0`
- Data is copied, not shared
- `exit(status)` causes value of `status` to be returned to the parent's call to `wait(&status)`. 8 LSB is exit status (`pid = status>>8`)

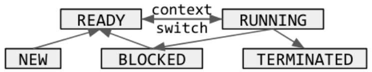
Zombie Process

- Child process that has completed execution (`exit` system call) but still has an entry in the process table (so parent process can still read exit status)
- `kill` command has no effect on a zombie process

Orphan Process

- Parent terminated, but child still executing
- Dead orphan processes waited on by init (PID 1)

Process Scheduling



- Non-preemptive/Cooperative: B will wait for A to finish (or be blocked) before B starts running
- Preemptive: suspend process after time quantum
- Burst Time: time spent actually using CPU
- Turnaround Time: end time - enqueue time
- Response Time: start time - enqueue time
- Waiting Time: turnaround time - burst time
- Throughput: # tasks finished per unit time

Batch Processing

- No user interaction, no need to be responsive
- Prioritises turnaround/burst time, throughput

First Come First Served (FCFS)

- FIFO queue; any process will eventually run
- Blocked process is added back to queue when ready
- Convoy Effect: short tasks waiting for a long task

Shortest Job First (SJF)

- Select task with smallest **total** CPU time
- OS needs to know CPU time for a task in advance

Shortest Remaining Time First (SRT)

- Select task with smallest **remaining** CPU time
- Will preemptively stop another process to allow the shorter remaining time task to run

Interactive Processing

- Prioritises response time, and predictability

Round Robin (RR)

- Preemptive version of FCFS
- Each process given a pre-defined time quantum
- After using up time quantum, if task is not yet finished, task gets put back in back of queue

Priority Scheduling

- Do tasks with highest priority first
- Hard to guarantee exact amount of CPU time given to each process; low priority tasks may be starved
- May lead to priority inversion: $P(H) > P(M) > P(L)$. L, H shares CS but neither with M. L is running in CS. H also needs to run in CS. H waits for L to come out of CS. M able to interrupt L and starts running. M runs till completion. L resumes and starts running till the end of CS. H finally enters CS and starts running.

Multi-Level Feedback Queue (MLFQ)

- If $P(A) > P(B)$, then A runs
- If $P(A) = P(B)$, then A and B runs in RR

Settings:

- New tasks will have highest priority
- Time slice is used before finishing task ⇒ $P\downarrow$
- Task blocks before time slice is used ⇒ P same

Inter Process Communication (IPC)

Shared Memory

```
shmat(int shmid, void *shmaddr, int flg);
```

- Attaches the shared memory segment (`shmid`) to the address space (`shmaddr`) of the calling process. If `shmaddr` is `null`, attach to first available addr
- Region is referred to by `shmid`, not by `shmaddr`
- Efficient: only the initial attach step involves OS
- Ease of use: shared memory behaves as per normal
- Hard to synchronise (see Race Condition)

Message Passing

- Passed message is stored in kernel memory space
- Easy to synchronise for synchronous primitives
- Inefficient: every send/recv operation involves OS

UNIX Pipe

- `A | B` means pipe output of `A` to input of `B`
- Data must be accessed in order (FIFO)
- Implicit synchronisation (waits when empty/full)
- Read end: `0`, write end: `1`

Signal

```
signal(int signum, sighandler_t handler);
```

- `signal()` sets the disposition of the signal `signum` to `handler`: either `SIG_IGN` (ignore), `SIG_DFL` (default handler), or the address of a defined function
- Asynchronous → vulnerable to race conditions
- Use `sigwait` to synchronously block till signal recv

Threads

- `pthread_create()`: `0` on success, `!0` on errors
- `pthread_exit(void *retval)`: terminates the calling thread and returns a value via `retval`
- `pthread_join(pthread_t t, void **ret)`: waits for `t` and sets `ret` to `t`'s `retval`
- A single thread exits when:
  - Calling `pthread_exit` or `pthread_cancel`
  - Returning from `start_routine`
- ALL threads exit when:
  - Any thread calls `exit()`
  - Main thread returns from `main()`
- Threads in the same process shares
  1. Memory Context: variables, text, data, heap
  2. OS Context: PID, opened files (`pd`), signals, etc

  ***Not shared:** Thread ID, registers, and **stack**

Kernel Threads

- Maintains kernel-level thread table
- Thread operations are system calls (slower)
- Can Thread Switch if threads from **same** process

User Threads

- Each process maintains its own private thread table
- Thread operations are runtime procedures (faster)
- Only Process Switch: threads **transparent to OS**

Hybrid Threads

- OS only aware of (and schedules) kernel threads
- Multiple user threads can bind to a kernel thread

Process Switch	Thread Switch
Switches full VM space	Switches GPR/PC/stack
Flushes TLB cache	Keeps TLB cache
More overhead	Less overhead

Threads vs Processes

1. Memory
   - Threads share same memory space
   - Processes each have independent memory space
   - Multiprocess is used if memory usage is heavy
   - Multithreading is cheaper if sharing large data
2. Overhead: thread creation is cheaper than forking
3. Protection: processes have independent memory space, child processes can potentially hang, deface memory, without really affecting other processes
4. Hardware: some do not support multiple threads
5. OS: some OSes favour one over the other

Synchronisation

Race Condition

```
lw   $r1, var    # init var as 0
addi $r1, $r1, 3
sw   $r1, var
```

Only 2 possible outcomes, `var` is `3` or `6`:

1. Both threads `lw` from `var` simultaneously (ie. both loads `0` to `$r1`) and eventually `sw 3` to `var`
2. One thread finishes and `sw 3`. The other thread then starts executing, and finishes to `sw 6` to `var`

Critical Section (CS)

Properties to maintain:

- Mutual Exclusion: exactly one process in CS
- Progress: if no processs, one should enter
- Bounded Wait: process must eventually enter
- Independence: process not executing in CS should never block other processes waiting to enter

Incorrect use can lead to deadlock/livelock/starvation

Test & Set

1. Atomically set `1` to memory and return old value
2. If old value is `1`, repeat step 1 (busy wait)
3. When exiting CS, set memory/lock to `0`

Semaphore

Counter & waiting list (may not be FIFO) of processes

- `wait(S)`: decrement *counter* by 1. If *counter* < 0, add process to waiting list and block self
- `signal(S)`: increment *counter* by 1. If *counter* ≤ 0, resume/remove a process from waiting list

1. $-$*counter* is the number of waiting processes
2. `signal()` is never blocked, but `wait()` may
3. `wait()` and `signal()` must be **atomic**
4. Deadlock is still possible with incorrect use

```
semaphore S = 3;
while (1)
    S.wait();
    // CS
    S.signal();
```

At most 3 processes can be in CS before CS is locked. Only when a process in CS finishes the work and calls `signal()` can another process enter CS.

Memory Management

Buddy System Allocation (RAM)

To allocate memory:

1. Look for smallest 2^k block $\geq$ requested memory, R
2. If found, allocate it
3. Else, split a free memory slot $> R$ by half
   1. If lower limit is not reached, repeat step 3
   2. Else, allocate it

To deallocate memory:

1. Free the block
2. Determine if neighbour blocks are free
3. If free, combine the two and repeat step 2 till upper limit reached or non-free neighbour encountered

Problems With Physical Memory (PM)

If program can address full 32-bit PM address space,

1. Program can crash if RAM $< 2^{32}$B $=$ 4GB
2. Can run out of space if running multiple programs
3. Can access and corrupt other programs' data

Virtual Memory (VM)

- Without VM: program address = RAM address
- Maps program address to RAM address
- `printf("%p")` prints VM address (not PM addr)

This solves each problem of PM by,

1. Allowing writing to/fro disk when RAM is full
2. Allowing program data to be anywhere in RAM
3. Making each program have a different mapping

Paging Scheme

- Physical Frame: split regions of physical memory
- Virtual Page: split regions of logical memory

Memory Management Unit (MMU)
- Maps virtual addr to physical memory addr
- Transfers betw RAM/disk always in whole pages

Page Table
- Maps virtual page num to physical frame num
- Virtual page number is used as index to find PTE
- Every process stores in RAM its own page table
- Size of PTE increases as page/frame size decreases
- Size of page table dependent on total VM space

Assuming M-bit machine with page size pB:
- Page size $= p$B $\implies$ bits for page offset $= \log_2(p)$
- Bits to index PTE $= M - \log_2(p)$
- Number of PTEs $= 2^{M-\log_2(p)}$
- Since PTE = 4B, page table size $= 4(2^{M-\log_2(p)})$B

Page Fault
- Only occurs if virtual addr space > physical addr space (ie. program can address out of size of RAM)

Eg: trap to OS, trying to access unmapped page u:
1. Choose physical frame x to evict from RAM, if x is dirty, write x back to disk first
2. Load required frame y from disk to RAM
3. Update page table to reflect changes: unmap PTE mapped to x and update PTE of u to map to y
4. OS returns to instruction that caused the fault

Translation Lookaside Buffer (TLB)
Steps taken when accessing VM:
1. Access TLB ($1ns$) for PTE
2. If PTE not found → TLB-Miss: access RAM ($30ns$) for full page table and update TLB ($1ns$)
3. If PTE has unmapped page → Page Fault (5^+ms)
4. Else, use PTE to access RAM ($30ns$) for data

2-Level Paging
- 1^{st} level page table points to 2^{nd} level page tables
- 1^{st} level page table always kept in RAM
- 2^{nd} level page tables can be swapped in/out RAM
- Minimum overhead is one 1^{st} (to point to 2^{nd}) and one 2^{nd} level page table (to point to useful data)

Let M-bit virtual address be partitioned into an n-bit `PT1` field, an n-bit `PT2` field and a p-bit `Offset` field:
- Bits for page offset $= p \implies$ page size $= 2^p$B
- Number of PTEs per page table $= 2^n$
- Number of 2^{nd} level page tables $= 2^n$
- 2^{nd} level page table can address $2^n 2^p$B of RAM
- Total memory that can be addressed $= 2^n(2^n 2^p)$B

Inverted Page Table
- One global page table for ALL processes
- Number of PTEs = number of physical frames
- Maps physical frame to `<pid, virtual page>`
- `virtual page` is not unique, `pid` & `virtual page` is needed to uniquely identify a physical frame
- Entries ordered by physical frame number, whole table needs to be searched when given page num
- A physical frame cannot map to multiple pages

Segmentation Scheme
- Maps segment to contiguous memory region
- Logical address partitioned to `SegID` and `Offset`
  - `SegID`: look up `<Base,Limit>` in seg table
  - Physical Address, PA = `Base` + `Offset`
  - `Offset` < `Limit` for valid access
- Can still cause external fragmentation

Second-Chance Page Replacement (CLOCK)
$T_{access} = (1 - P_{fault}) \times T_{mem} + P_{fault} \times T_{fault}$
1. Maintain pointer at oldest page
2. If `ref bit == 0`, replace page
3. Else, set `ref bit` of page to `0`
4. Reset page arrival time (page taken as new)
5. Search for next oldest page
6. Go back to step 1

Working Set Model (WS)
- Working Set Window, Δ = interval of time
- $W(t, \Delta)$ = active pages in the interval at time t
- Allocate enough pages to match current locality, so page faults only occur when locality changes
- High page faults, low CPU usage $\implies$ Δ too small
- Low page faults, low CPU usage $\implies$ Δ too large

Local Replacement
- Chosen victim page belongs to **same** process
- Stable performance between multiple runs but may hinder if initial frame allocation is not enough
- eg. Working Set (no WS for whole memory)

Global Replacement
- Chosen victim page can be from **any** process
- Must continually decide how many page frames to assign to each process; allocated in proportion
- Badly behaved process can affect others
- eg. FIFO, LRU can be global

File System (FS)
1. Self-contained: plug-and-play on any system
2. Persistent: lifetime beyond OS and processes
3. Efficient: minimum overhead for book-keeping

Disk Organisation
- 1D array of **logical blocks** → smallest accessible unit, usually 512B to 4KB
- MBR (sector 0) contains partition table, followed by ≥ 1 partitions, each with an independent FS
- Each file system contains: OS boot block, partition details, directory structure, files info, and file data

FAT (File Allocation Table)
- A table stored in RAM containing all block pointers
- Maps current disk block to next disk block
- Can be huge if disk is large (consumes RAM)

Indexed Allocation
- Each file has an index block containing an array with all used block indices: eg. `[4,5,2,-1,-1]`
- Only index blocks of opened file needed in RAM
- Max file size limited by number of index blocks
- Disk memory overhead for the extra index blocks

Free Space Management
- Bitmap: `1` → free, `0` → occupied
  - Easy bit level operations
- Linked List: each disk block points to next free
  - Only first pointer needed in RAM

Open File Table
- System-wide table of open-file info (file offset, disk location, ref count); Every process stores location of a sys-wide entry per file opened in that process
- Sys-wide entry is deleted only if ref count is 0
- Each `open` call creates new entry with different `fd`
- `fork`/`dup` duplicates process table with **same** `fd`

File Data
- Record: a collection/group of bytes/words/data
- Fixed Length Records: array of fixed length records
- Sequential: only can rewind; Random: any order
- Direct: random access to a fixed length record

`Create()` System Call
To create file `/.../foo/F`:
1. Use full pathname to locate `foo` directory
2. If `F` already exists, return error
3. Else, use free space list to find free disk blocks
4. Add entry with `F`'s file info to `foo` directory

`Open()` System Call
Process `P` wants to open file `/.../.../F`:
1. Use full pathname to locate file `F`
2. If `F` not found, terminate with error
3. Else, create entry `E` in sys-wide table with `F`'s info
4. Store `fd` (index of `E` in sys-wide table) in `P`'s table
5. Return `fd` for further read/write operations

Disk Scheduling
- FCFS, SSF (Shortest Seek First; like SJF)
- SCAN (bi-direction), C-SCAN (outer→inner)

I/O Scheduling
- Deadline: Sorted/Read FIFO/Write FIFO
- Noop: no operation, no sorting
- CFQ: time slice and per-process sorted queues
- BFQ: fair sharing based on no. of sectors requested

Microsoft FAT File System

MBR	Partition 1	Partition 2	...	
BOOT	FAT	FAT Duplicate (Optional)	Root Dir	Data Blocks

- Size of FAT table: $2^{12}/2^{16}/2^{32}$ entries respectively
- FAT entry: either `FREE`, `EOF`, `BAD`, or block no.
- Larger cluster size → more internal fragmentation

Largest Partition: max cluster size = 32KB

FAT12	FAT16	FAT32
2^{12} clusters	2^{16} clusters	2^{28} clusters
$2^{12}2^{15}$=128MB	$2^{16}2^{15}$=2GB	$2^{28}2^{15}$=8TB

Directory Structure
- Directories store in its data block a directory table of directory entries, each describing a file/subdir
- Free space must be calculated by gg through FAT

Directory Entry (32 Bytes)
- Name (8) & Extension (3): limited to 8+3 chars
- Attributes (1): indicate rdonly/hidden/archived
- Reserved (10): not actually used
- Creation Date (4): limited to 1980 to 2107
- First Block (2): 12/16/32 bits respectively
- File Size (4): theoretical 2^{32} = 4GB max file size; FAT16 limited to 2GB as max cluster size = 32KB

File Deletion
1. Set first letter in name of directory entry to `0xE5`
2. Set FAT to `FREE`; leave actual data blocks intact

Linux Ext2 File System

MBR	Partition 1	Partition 2	...		
BOOT	Block Group 0	Block Group 1	...		
Super-block	Group Descriptor	Block Bitmap	I-Node Bitmap	I-Node Table	Data Blocks

- `BOOT`: not used by linux, reserved for boot code

Block Group Layout
- Superblock: layout of file system; number of inodes/disk blocks; start of free disk blocks
- Group Descriptor: location of bitmaps; number of free blocks/inodes; number of directories
- Bitmaps: 1-block long; tracks data block/inode usage; `1` = occupied, `0` = free; With 1KB data blocks, limited to 8192 number of data blocks/inodes
- Inode Table: array size limited by inode bitmap
- Data Blocks: non contiguous, stores all files/dir

Inode (128 Bytes)
- Describes a file/dir's size, permission, ownership
- 12/1/1/1 data block pointers; an indirect block can contain $\frac{\text{size(data block)}}{\text{size(block ptr)}}$ number of pointers
- Inode is deleted only if ref count is 0

Given mB disk blocks and 32 bit disk block address:
1. A disk block address uses $\frac{32}{8} = 4$ bytes
2. Indirect block can store $\frac{m}{4}$ addresses
3. Max file size $= 12m + m\frac{m}{4} + m(\frac{m}{4})^2 + m(\frac{m}{4})^3$B

Directory Entry (Linked-List)
- Stored in data blocks of inodes describing a dir

An entry describes a file or a subdir, each with:
1. Inode Number: `0` → unused; `2` → root dir
2. Size of Entry: to locate next entry
3. Length: of name of file/subdirectory
4. Type: file or subdirectory or others
5. Name: up to 255 characters; OS linearly searches this field to get corresponding inode number (slow)

Symbolic Link	Hard Link
A new inode/file	A directory entry
Contain pathname of TF	*Point to TF's inode
Search for TF's inode	Just use inode ref
Can link to dir	Cannot link to dir
Can link across FS	Cannot link across FS

*If TF is deleted, HL will still work, but SL will not