CS2106 Cheatsheet

1. Process Abstraction

Processes

Process: abstraction for information required to describe a *running program.*

- Memory context: text, data, stack, heap
- Hardware context: registers, PC/FP/SP
- OS context: process properties, resources used

Process table: contains all PCBs

- PCB: contains information about the entire execution context for a process

UNIX Process Abstraction

Process state diagram

- Ready
- Running
- Stopped (paused, really): on STOP signal
- Suspend: blocked and waiting for I/O
- Zombie

System calls: `fork()`, `execl()`, `wait()`, etc.

2. Process Scheduling

Scheduler: part of OS that decides what and when to run processes

Scheduling policies either *preemptive* or *non-preemptive*

- Preemptive: scheduler can pick another process even if running process isn't done
- Non-preemptive: stays scheduled until it ends/gives up CPU voluntarily

Batch Processing

Criteria

- Low *turnaround time* (i.e. finish time - arrival time)
- High *throughput*
- High *CPU utilisation*

First-Come First-Served (FCFS)

- No starvation
- Problem: *convoy effect* – short I/O processes and long CPU processes block one another => lead to either I/O or CPU idling

Shortest Job First (SJF)

- Starvation possible
- Can predict CPU time using exponential moving average

$Predicted_{n+1} = \alpha Actual_n + (1 - \alpha) Predicted_n$

Shortest Remaining Time (SRT)

- *Preemptive*
- Starvation possible
- Takes care of short jobs quickly, even if they arrive late

Interactive Processing

Criteria

- Low *response time*
- High *predictability* (low variation in response time)

Periodic scheduler

- Can be implemented using *timer interrupt* handler, which invokes the scheduler
- Interval of timer interrupt (ITI): ~1-10ms
- Time quantum: multiple of ITI that determines execution duration given to process

Round Robin (RR)

- Queue of tasks, pick first and execute. When quantum elapses, put at back of queue
- No starvation

Priority Scheduling

- Assign priority to each task, pick task with highest priority
- Starvation possible
- Problem: priority inversion

Multi-Level Feedback Queue (MLFQ)

- Reduce priority if job uses up entire time slice, retain priority otherwise
- Adaptive because it learns process behaviour
- Minimises both response time for I/O-bound processes, turnaround time for CPU bound processes

Lottery Scheduling

- Each process gets tickets. Randomly pick a ticket, grant winner the resource
- Responsive
- Good level of control: process can distribute its tickets to children
- Simple implementation

3. Inter-Process Communication

Methods

- Shared memory: easy but synchronisation problems
- Message passing harder to use but no need synchronise
- UNIX pipes
- UNIX signals: signals are asynchronous notifications about an event. Recipients of signals must handle it using handlers

4. Threads

Threads: lightweight alternative to processes — can have multiple threads in a single processes, no need for entire memory duplication/IPC

Shared resources

- Memory context: text, data, heap
- OS context: PID, other resources

Not shared resources

- Stack
- Registers
- Identification e.g. thread ID

Processes vs. Threads

Unlike process switches that require changing OS and hardware and memory context, thread switches require changing only registers and stack (i.e. FP/SP registers)

Which is faster? Not always true that one is faster than the other – e.g. `malloc` on threads could make it slower. **Measure**!

5. Synchronisation

Race condition: when execution outcome depends on order in which shared resources are accessed/modified

Synchronisation problems only arise when:

- Shared
- Mutable
- Access

Deadlock: when all processes are blocked, so no progress

Critical Section

Properties of correct critical sections

- Mutual exclusion
- Progress
- Bounded wait
- Independence: process NOT executing in critical section should NOT block other processes

Test and Set

`TestAndSet <reg>, <memory location>` — atomic instruction!

- Loads `<memory location>` into `<register>`
- Basically returns value inside, and sets it to 1 regardless (locked)
- Returns 1 if it's locked, 0 if it's unlocked

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

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

Peterson's Algorithm

```
Want[0] = 1;
Turn = 1;
// wait only if it isn't your turn
while (Want[1] && Turn == 1);
// ...
Want[0] = 0;

Want[1] = 1;
Turn = 0;
while (Want[0] && Turn == 0);
// ...
Want[1] = 0;
```

Semaphore

Semaphore: High-level abstraction. Contains a non-negative integer value, 2 atomic operations:

- `Wait(S)` — if S ≤ 0, blocks (goes to sleep); S– on proceeding
- `Signal(S)` — S++, wakes up 1 sleeping process (if any)

Mutex/binary semaphore: Semaphore where S = 1

6. Synchronisation Problems

Producer Consumer

Producer: produce items only when buffer is not full

Consumer: consume items only when buffer is not empty

Initially, `notFull` = size of buffer, `notEmpty` = 0

```
// PRODUCER
while (true) {
  wait(notFull);
  wait(mutex);
  // ...
  signal(mutex);
  signal(notEmpty);
}
```

```
// CONSUMER
while (true) {
  wait(notEmpty);
  wait(mutex);
  // ...
  signal(mutex);
  signal(notFull);
}
```

Readers Writers

Can have multiple readers at once, but not multiple writers

- This solution can starve your writers

```
// READER
while (true) {
  wait(mutex);
  nReader++;
  if (nReader == 1)
    wait(roomEmpty);
  signal(mutex);
  // ...
  wait(mutex);
  nReader--;
  if (nReader == 0)
    signal(roomEmpty);
  signal(mutex);
}

// WRITER
while (true) {
  wait(roomEmpty);
  // ...
  signal(roomEmpty);
}
```

Dining Philsophers

Limited eater solution: initially, seats = 4

```
while (true) {
  wait(seats);
  wait(chopstick[LEFT]);
  wait(chopstick[RIGHT]);
  // ...
  signal(chopstick[LEFT]);
  signal(chopstick[RIGHT]);
  signal(seats);
}
```

Tanenbaum solution

```
void philosopher(int i) {
  while (true) {
    takeChopsticks(i);
    putChopsticks(i);
  }
}

void takeChopsticks(int i) {
  wait(mutex);
  state[i] = HUNGRY;
  safeToEat(i);
  signal(mutex);
  wait(s[i]); // hmm
}

void putChopsticks(int i) {
  wait(mutex);
  state[i] = THINKING;
  safeToEat(LEFT);
  safeToEat(RIGHT);
  signal(mutex);
}

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

General Semaphore from Mutex

```
int count = N;

GeneralWait() {
  wait(mutex);
  count--;
  if (count < 0) {
    signal(mutex);
    wait(queue);
  }
  signal(mutex);
}

GeneralSignal() {
  wait(mutex);
  count++;
  if (count <= 0) signal(queue);
  else signal(mutex);
}
```