CS2106 Cheatsheet 19/20 S1 Midterms

Memory

- Data: global variables
- Text: raw source code/instructions
- Stack: return address of caller/arguments/parameters/local variables
- Heap: dynamically allocated with `malloc`

Process Abstraction

Fork

- child → `fork() == 0`; parent → `fork() > 0`
- Data is copied; local changes made to child not reflected on parent
- `exit(status)` causes normal process termination and the value of `status` is returned to the parent's call to `wait(&status)`
  - `status`: 8 LSB is exit status, next 8 is PID (`pid = status>>8`)

Zombie Process

- Process that has completed execution (via `exit` system call) but still has an entry in the process table
- This occurs for child processes to allow the parent process to read its child's exit status
- `kill` command has no effect on a zombie process

Orphan Process

- Child process that is still executing, but whose parent has died
- When orphan processes die, they do not remain as zombie processes; they are waited on by init (PID 1)

Process Scheduling

Policies

- Non-preemptive/Cooperative: Process B will wait for process A to finish (or be blocked) before B starts running
- Preemptive: Every process given fixed time quota to run, and after time quota, process is suspended

Definitions

- Timer Interrupt: OS scheduler will trigger every 1-10ms
- Time Quantum: A multiple of the timer interrupt
- 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

First Come First Served (FCFS)

- Basically a FIFO queue; any process will eventually run
- Blocked process gets removed and added back to queue when ready
- Convoy Effect: many short processes waiting for a long process

Shortest Job First (SJF)

- Select task with smallest **total** CPU time
- OS will need to know total 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 (if such a task joins the queue)

Round Robin (RR)

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

Priority Scheduling

- Assign a priority to all tasks and do task with highest priority first
- Low priority tasks may be starved
- Hard to guarantee exact amount of CPU time given to each process
- May lead to priority inversion
  - $P(H) > P(M) > P(L)$
  - L and H shares CS but neither of them share CS 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
- If time quantum is used before finishing task, priority drops
- If task blocks before time slice is used, priority retains

Inter Process Communication (IPC)

Shared Memory

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

- Attaches the System V shared memory segment identified by `shmid` to the address space of the calling process. The attaching address is specified by `shmaddr` (if `NULL`, system chooses a suitable address)
- Region is identified/referred to by the user defined `int shmid` field, and not by the shared region address `shmaddr`
- Efficient: only the initial steps (create/attach) involves OS
- Ease of use: shared memory behaves as per normal memory space
- Hard to synchronise (see Race Condition)

Message Passing

- Message to be passed is stored in kernel memory space
- Easy to synchronise when synchronous primitives are used
- Inefficient: every send/receive 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 (wait when buffer is empty/full)
- read end: `0`, write end: `1`

Signal

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

- `signal()` sets the disposition of the signal `signum` to `handler`, which is either `SIG_IGN`, `SIG_DFL`, or the address of a defined function

Threads

```
int pthread_create(pthread_t *tid, const pthread_attr_t
    *attr, void *(*start_routine) (void *), void *arg);
```

- Returns `0` on success, `!0` on errors

```
int pthread_exit(void *retval);
```

- Terminates thread and sets `retval` to user defined value

```
int pthread_join(pthread_t tid, void **retval)
```

- Waits for `tid` and sets `retval` to `tid`'s return value

Threads in the same process shares

- Memory Context: Variables, Text, Data, Heap
- OS Context: PID, files, etc

Pros:

- Economy: multiple threads in same process requires less resources
- Resource Sharing: no need for IPC to pass messages around
- Responsiveness
- Scalability: take advantage of multiple CPUs

Cons:

- Synchronisation of parallel execution of multiple threads

A single thread exits when:

- Calling `pthread_exit` or `pthread_cancel`
- Returning from `start_routine` (like `pthread_exit` with `retval`)

ALL threads exit when:

- Any thread calls `exit()` or main thread returns from `main()`

Synchronisation

Race Condition

Incorrect execution due to the unsynchronized access to a shared modifiable resource (global variable)

```
int globalVar = 0; //shared among all threads
void* doSum(void* arg)
    int i, localVar = 0;
    for (i = 0; i < 50000; i++)
        localVar++;
    globalVar += localVar;
```

Can be translated to

```
lw  $r1, globalVar
add $r1, $r1, 50000
sw  $r1, globalVar
```

Only 2 possible outcomes, `globalVar = 50000 || 100000`:

1. Both threads do `lw` simultaneously (ie. both loads `0` to their `$r1`) and both does their for loop and `sw 50000` to `globalVar`
2. One thread finishes and `sw 50000`, the other thread then starts working, `lw 50000`, and finishes to `sw 100000` to `globalVar`

Critical Section (CS)

Properties to maintain:

- Mutual Exclusion: exactly one process at any point of time
- Progress: if no processs, one should enter
- Bounded Wait: there exists a bound, or limit, on the number of times other processes are allowed to enter their critical sections after a process has made request to enter its critical section and before that request is granted.
- Independence: process not executing in critical section should never block other process

Incorrect use can lead to:

- Deadlock: All processes blocked
- Livelock: Processes constantly change state to avoid deadlock
- Starvation: Some processes blocked forever

Semaphore

An object that consists of a counter, a waiting list (not necessarily FIFO) of processes, and two functions: `signal` and `wait`

- `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 and remove a process from waiting list

1. If *counter* < 0, |*counter*| is the number of waiting processes
2. `wait()` may be blocked, but `signal()` is never blocked
3. `wait()` and `signal()` must be **executed atomically**
4. Deadlock is still possible with incorrect use of Semaphore

```
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.

Stack Frame Setup/Teardown

- An executable (binary) consists of Text and Data
- When under execution, memory context (text and data) and hardware context (general purpose registers, program counter) has to be known

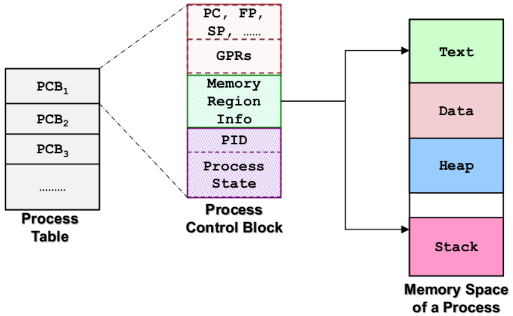
On executing function call:

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

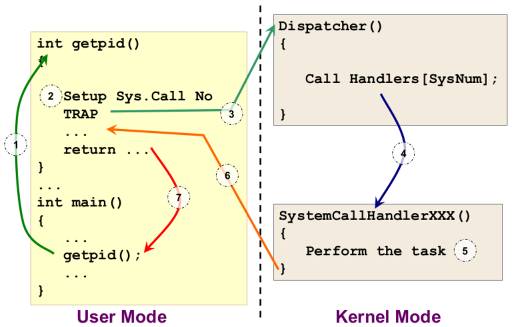
On returning from function call:

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

Process Table



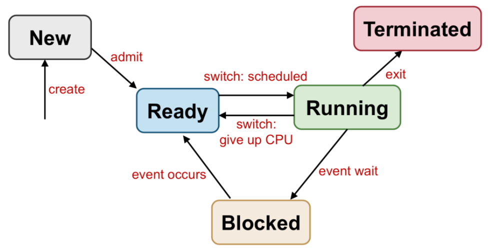
System Calls



(3) Library call executes TRAP to switch from user mode to kernel mode

- Introduces overheads. Possibly require context switch

State Processes

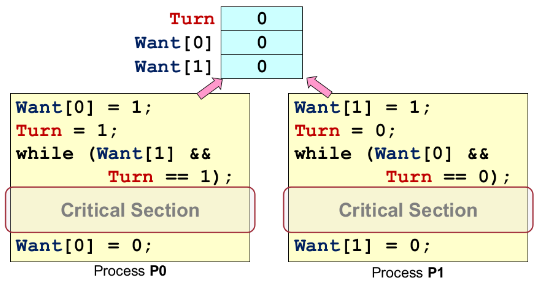



```
// before main starts: New -> Ready -> Running
int main() {
    int input, result;

    // Assuming I/O blocks
    // Running -> Blocked -> Ready -> Running
    printf("Give input below:\n");

    // Running -> Blocked ->
    // [user input] -> Ready -> Running
    scanf("%d", &input);

    // assuming it takes a long time
    // Running -> Blocked -> Ready -> Running -> ...
    result = ComplexFunc( input );

    // write result to disk (takes long time)
    // Running -> Blocked -> Ready -> Running -> ...
    saveToDisk( result );

    return 0;
} // after main ends: Running -> Terminated
```

Multithread vs Multiprocess

1. Memory
   - Threads share same memory space
   - Processes each have independent memory space
   - Multiprocess is used if memory usage is heavy
   - Multiprocess is expensive if tasks share large amount of data
2. Overhead
   - Thread creation is cheaper than forking
3. Protection
   - Since processes have independent memory space, child processes can potentially hang, deface memory, without much negative behaviours to other processes
4. Hardware
   - Whether hardware is capable of exploiting multiple threads
5. OS
   - Some OSes favour one over the other

Peterson's Algorithm



- Assumes writing to Turn is an atomic operation
- Busy Waiting: OS have to wait for time quantum of busy waiting task to finish to reschedule the other task → less efficient than sleep/block

TestAndSet

- Write 1 to a memory location and return old value (atomic operation)
- Similar to a binary semaphore to maintain Mutual Exclusion

```
volatile int lock = 0;
void Critical()
    while (TestAndSet(&lock) == 1); // busy wait
    critical section // only one process can enter
    lock = 0 // release lock when finished with the CS
```

Producer Consumer

no. of slots

```
semaphore NotFull=n, NotEmpty=0, Mutex=1;
```

producer

```
while (1) {
  NotFull.wait();
  Mutex.wait();
    Buf[in] = x;
    in = (in+1)%n;
  Mutex.signal();
  NotEmpty.signal();
}
```

consumer

```
while (1) {
  NotEmpty.wait();
  Mutex.wait();
    x = Buf[out];
    out = (out+1)%n;
  Mutex.signal();
  NotFull.signal();
}
```

notifications

critical section

Dining Philosophers

```
semaphore C[5]= 1;
semaphore Chair = 4;

while (1) {
  // thinking
  Chair.wait();
    C[i].wait();
    C[(i+1)%5].wait();
    // eating
    C[(i+1)%5].signal();
    C[i].signal();
  Chair.signal();
}
```

get a chair

this is a count-down lock that only allows 4 to go!

this is our old friend

release my chair

Readers Writers

```
semaphore Mutex = 1, WrtMutex = 1;
int       RdrCount;
```

reader

```
while (1) {
  Mutex.wait();
    RdrCount++;
    if (RdrCount == 1)
      WrtMutex.wait();
  Mutex.signal();
  // read data
  Mutex.wait();
    RdrCount--;
    if (RdrCount == 0)
      WrtMutex.signal();
  Mutex.signal();
}
```

blocks both readers and writers

writer

```
while (1) {
  WrtMutex.wait();
  // write data
  WrtMutex.signal();
}
```