142 Midterm Cheat Sheet

**Chapter 1 Introduction**

When the running process performs a blocking I/O operation *before* the expiration of it’s time splice, it would be in placed in: **Device queue for specific I/O device**.

When a running process blocks on an I/O\*: **The device queue for the specific I/O device.**

When time splice for a running process expires, the process will be placed in:  **Ready queue as another one is scheduled. \*\*difference is that this is expired.**

Programming the interrupt vector table used to implement system calls is done by: **Kernel level- privileged**

CPU-bound process: **Spends more time doing computations than I/O. IO < #’s**

I/O bound process: spends more time doing I/O than computations.

A trap is: **A software interrupt**

In general, the fastest way to pass parameters to sys call would be: **using CPU registers**

The principle of time-sharing ensures that: **The OS loads multiple programs in memory & switches the CPU among them in frequent manner.**

Many hardware architectures allow the OS kernel to protect itself from the user programs it runs by providing: **User & kernel execution modes**

**Chapter 2 Operating System Structures**

OS Kernel to protect itself: **2 separate modes: user & kernel execution modes.**

An interrupt transfer control to the interrupt service routine by: **Function pointers in IVT.**

When many parameters must be passed (more than regs) the best way to pass: **the stack.**

**Chapter 3 Processes**

Two processes are using an IPC communication implemented ***in the kernel***, what is the minimum overhead required from the moment the sender does a post to the moment the receiver process gets access to the communicated data: **2 system calls & 1 context switch**

**\*\***If ***shared memory*** -0 system calls & 1 context switch.

\*\*if **one of the process does send a zero buffer:** Process doing send blocks in the kernel until the receiver invokes the receive for the IPC in question.

\*\*if not in context of amount: **the process doing the send blocks in the kernel until the receiver invokes the receive for the ipc in question.**

Starvation of a low priority processes (priority scheduling) can be prevented: **Periodically increasing the priority of low level processes that has not been ran yet.**

In unix, a process’s PCB remains used until: **Process invokes exit() or terminates and the parent invokes wait() for it.** When using user-level threads (single kernel-level thread)

Right Ater returning of a successful fork system call a parent & child share: **heap, stack, text; all Above**

The long term scheduler ensures that the mix of processes loaded in memory is: **Even mix of CPU & I/O**

In unix, a parent process obtains the exit status of it child process by using the following system call: **wait(); \*\***a process which invokes the wait(): **block until the child exits/terminates.**

After returning a successful fork system call, parent & child memory is marked as the following by the OS: **Cow memory**

Right after an ***asynchronous*** I/O operation, the process performaing the I/O: **keeps executing other tasks while the I/O is in progress.**

One of the main issues with caching is: **Cache consistency**

A process is currently running in the CPU when it blocks on an I/O, the state of the process will change to: **From running to waiting \*\***time splice expire: **From running to ready**

**Chapter 4 Threads**

When using a ***user*** level threads, a thread context switch within the same address space is similar to: **A function calls \*\*if it’s kernel then it’s System call.**

When using user-level threads (single kernel-level thread) you can assure user threads won’t block with**: using non-blocking system calls**

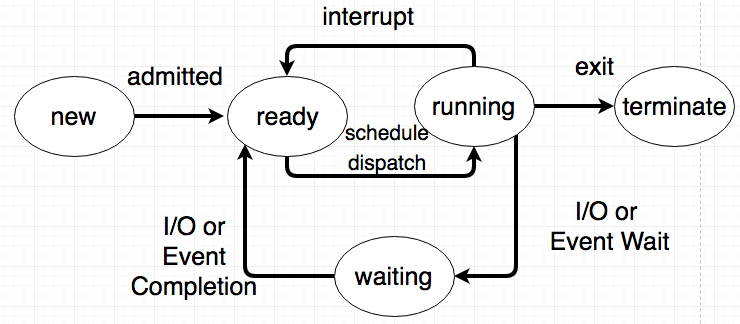
The ptrthread interface to wait or synchronize with a child threasds is: **join\*\* pthread\_join()**

**\*\***To terminate: **pthread\_exit(); \*\***To create a new: **pthread\_create();**

**Short Question:**

a. What is a micro-kernel & adv vs disadv of model:**Type of kernel where code is moved from kernel to user space. Msg passing is how communications work. Easier to port the OS to a new architecture. More secure. Performance overhead of user space to kernel space communication is bad.**

b. What is an up-call & what are the the main disadv: **Up-call is a call from the kernel to user level. It is not secure and not reliable. Users code can be run by the kernel. Might not share resources or crash.**

c. 5 basic process states \*\*or explain briefly the various states of a process to make a diagram illustrating the actions/events: **New:** created, **t:** currently executed by CPU, **Waiting:** Waiting for event, **Ready:** in ready queue, ready to execute and waiting for OS to allocate CPU to it, **Terminate:** Process has finished execution. **Each transition:**

d. Many scheduling algorithms rely on remaining execution time (RET) as a parameter to decide on what process to schedule next. Actual RET is hard or impossible to compute efficiently. Explain how real schedulers approximate RET: **Estimate RET by sampling the execution time of the process every time it runs and maintain an exponential average for it:**

**T\_n+1 = at\_n + (1-a)T\_n ::t\_n is currrrent exec time. A = constant <= 1. T = tau. Or current avg exectime.**

e. Define the PCB & mention 4 items: **It is a data structure kept in the kernel memory used by the OS to keep individual state for specific processes. PCB has: Process state, program counter, CPU registers, CPU scheduling information, Memory Management information, Accounting Information, I/O status Information (choose 4).**

a. Explain the idea of thread-pools & how they are used in multithreaded programs: **Thread pool-** Group of threads that is created @ initialization & maintained through the life of the program. Main goal is to reuse each thread multiple times to save in overhead of creating. It can be adapted dynamically to match current load.

b. Define write-through & write-back cache- **Write through-** Write cache & memory on all writes. **Write back: write to cache only & use a cache bit to indicate line is dirty. So ti can be de-staged to memory when evicted.**

c. Write high level pseudo code for the scheduler of an OS:

scheduler(){

nextPCB = getNextProcessPCBfromReadyQueue();

if(!nextPCB) return;

Save CurrentCPUState(currentPCB);

SaveCurrentMemoryContext(currentPCB);

flushCache();

RestoreNextmemoryContextFromPCB(nextPCB);

currentPCB=nextPCB;

REstoreNextCPUstatefromPCB(nextPCB); }

d. Explain the concept of aging in priority scheduling systems: **Aging:** is technique by which low priority processes get their priority increased periodically if CPU attention is not received in the recent past. The once the process gets some CPU attention, it’s priority must be restored to low

a. Pthread\_exit(void \*)n): used to complete a thread execution & pass a parameter to parent thread. w/ join();

b. pthread\_create(&threads[jj],attributes,threadRoutine,(void\*): **Creates a new thread exec threadRoutine function**

c. pthread\_join(threads[jk]): **Used by a thread to synchronize with the exit of another thread.**

d. write code of a program that catches SIGINT signal and toggles a globale variable called *debug\_enabled*:

int signal\_handler(intsignnumber){

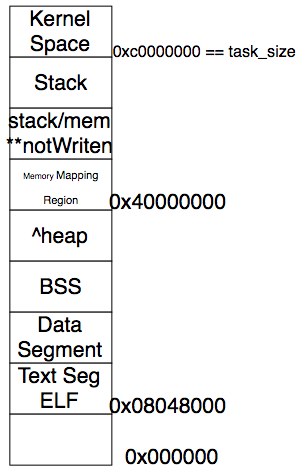
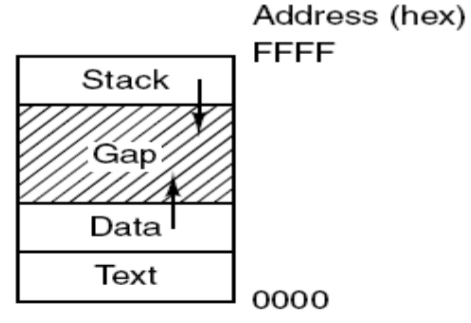
debug\_enabled= !debug\_enabled;}

int main(){

signal(SIGINT, signal\_handler);

MainLoop(); }

e. For unix-like system.. Draw a diagram of the layout of a program in memory. Enemurate the segments involved. Explain which segments can grow & in what direction:

\*\*nothing grows other than heap \*\*draw a diagram of the memory layout of a prog

(grow up) & stack (grows down) …this is data & heap are the same. Growing up

a. Threads: Assume a multithreaded application using user level threads mapped to a single kernel level thread in one process (many to one model). Describe the details of what happens when a thread (ie thread 1) executes a blocking system call: **Thread 1 blocking system call:** The full process wil be block & no other user level hreads will be allowed to run while thread 1 is performing the blocking system call. Precise address of where thread 1 blocked & information about the CPU status will be stored into the PCB (in kernel), for the process hosting thread 1. Pretty much state gets stored & used later when the system call is unblocked by the OS (other process gives back control. **Thread 1 to thread 2 ctsw:** Context switch is handled by the user thread library and without intervention of he kernel. The librry will save the state of thread 1 in a structure kept by the library to store the state of thread 1. The library then proceeds to restore the state of thread 2 and control is given to thread 2 so it can continue to execute wherever it left & it was last scheduled by the user-level thread library.

**Definitions**

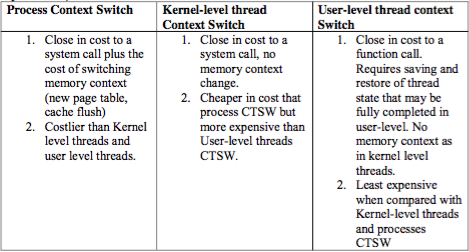
1. Hypervisors: It is a software, hardware, or firmware that virtualizes a computer system creating abstractions, namely VM’s. Individual OS may run in native hardware.

2. Virtual Machine: It is a virtualized abstraction of hardware including virtual cpu, memory, & i/o. Hypervisor has a data struct to track processes running on them.

3. Process Scheduling: **Short term (CPU scheduler**): slects which process should be executed net from those in the ready queue & allocates CPU. Very frequent & has to be fast and efficient. The goal of this is to be able to efficiently distribute the CPU cycles between various processes loaded in memory fairly. **Medium:** this type of scheduler will remove partially executed programs from memory (swap out) & brign them back in (swap in) @ a later time. Purpose is to improve process mix held in memory or resolve memory overcommitment issues. **Long-term** (job scheduler): selects which processes should be brought into the ready queue. Invoked infrequently, may be slow. It creates a balanced mix of CPU- bound & IO-bound in the system. Controls the degree of multiprogramming.

4. Threads: **User level:** Implemented in user space. Most thread operations do not require kernel intervention and hence they can be implemented more efficiently than in kernel level threads. Context switch & thred creation happen @ user space & can be implemented efficiently. Adv:Most thread oeprations, such as context switch. Do not require traps or system calls. DisAdv: IF user level thread performs a blocking operation or access memory that has been paged out all the user-level threads of the associated process are blocked. Cannot exploit multiprocessor CPU’s directly. **Kernel Level:** Implemented in kernel space. Most operation require kernel intervention & are costly compared to user level. Context switch & thread creation requires traps or system calls. Adv: IF a kernel level thread performs a blocking operation or access memory another ready thread for that process can be selected to run. Can leverage multiprocessor architectures. DisAdv: Operations involve kernel code execution and needs a system call or trap.

\*\*Compare the cost of user-level threads, kernel level threads, & process context switch:

****

5. Process Scheduling: **(A, 0, 4); (B, 1, 3), (C,2,1), (D,4,2). (process name, arrival time, time units needed to complete). SJF (shortest job first) -> non-preemptive. Shortest remaining time next Schedule-> preemptive.**

1. write code for a program that forks 4 children.

int rr[4];

const char \* child\_path[4]= {“binarypath1”,”binarypath2”,…etc};

for(int i=0; i<4; i++){  
 rr[i]= fork();

if(rr[i]== 0){

exec(child\_path[i]);

} for(int j=0; j<4; j++){

int result = wait(rr[j]);

}

2. Process & thread scheduling:

Process CTSW:

nextPCB = getNextProcessPCBFromReadyQueue(); if (!nextPCB) return; // Same process continues to run! SaveCurrentCPUState(currentPCB); SaveCurrentMemoryContext(currentPCB); flushCache(); RestoreNextMemoryContextFromPCB(nextPCB); currentPCB=nextPCB; RestoreNextCPUStateFromPCB(nextPCB);

Kernel Level CTSW:

SaveCurrentCPUState(currentThreadStruct-kernel);

RestoreNextCPUStateFromPCB(nextThreadStruct-kernel);

User Level CTSW:

SaveCurrentThreadState(currentThreadStruct-user);

RestoreNextThreadStateFromPCB(nextThreadStruct-user);

//below is the comparison

**Process CTSW:** Close in cost to a system call + the cost of switching. Costlier than kernel & user.

**Kernel CTSW:** Close in cost to a system call. No memory context change. Process <kernel <usr

**User CTSW:**  Close to a function call. Requires saving & restoreing of thread states. No memory context. Least expensive

**3.** Process- Write code for a program that forks 2 children, each to execute specific binary…

int main(){

int pid[2];

for(int n=0; n<2; n++){

if((pid[n] =fork) == -1){

printf("error");

}else if(pid[n] == 0{

exec(binaryPath[n]);}}

sleep(60);

for(n=0; n<2; n++){

wait(pid[n]);}}

**Process synchronization:**

a. A cmpe 142 student named Smarty… readers/writers solution given in the textbook & decides he wants to implement a readers code… (asynchronous reader) code is provided below

Semaphore mutex=1, db=1; int rc=0;

void reader(void) { while(TRUE) {

down(&mutex);

if( wc > 0) { signal(&mutex); yield(); continue; }

rc = rc + 1; if(rc==1) {

down(&db); }

up(&mutex); read\_database(); down(&mutex); rc = rc – 1;

if (rc == 0) up(&db); up(&mutex); use\_data\_read();

} }

write code is:

while (TRUE){

wait(&mutex); wc++; signal(&mutex); wait(&db);

DoWrite();

signal(&db);

wait(&mutex); wc--; signal(&mutex);

}

b. name & explain three solutions o the critical section problem. Refer to adv & disav of each approach in the explaination…

**TestAndSet Instruction:** the test and set instruction takes a Boolean lock parameter initializes to false & return the original value of lock as the instruction is executed. This instruction can be used in a while loop just before the critical section as follows:

While(testandSet(lock)); critical Section lock = false;

boolean TestAndSet (boolean \*target) { boolean rv = \*target; \*target = TRUE; return rv: }

**Semaphores:** Involves atomic operations on a counter, can be implemented as a blocking entity and hence used to avoid busy waits.

**Advantages:** 1. No special hardware Required, no busy wait. 2. Applies to multiple processes

**Disadvantage:** 1. May require a system call

**Monitor:** Program construction where functions defined within the monitor construct are guaranteed to be executed in mutually exclusive maner. **Adv:** Preferred to semaphores b/c they may prevent deadlocks ez

a. Mention the three requirements that must be satisfied by a solution to the critical-section and briefly explain: **mutual exclusion:** if process P\_i is executing in its critical section, then no other process can be executing in their cirtical section. **Progress:** If no process is executing in tis critical section & there exist some process that wish to enter their CS (critical section), then the selection of the process that will enter the CS next cannot be postponed indefinite. **Bound waiting:** A bound must exist on the # of tiems that other processes are allowed to enter their critical section after a process has made a request to enter its critical section & before that request is granted: 1. Assume that each process executes @ a nonzero speed. 2. No assumption concerning relative speed of N process

b. What are system call & how is it usually implemented: **A system call is a call from user-leve code into kernel-level code; it is usally implemented via trap. Under control of the interrupt vect which is created by OS in boot.**

c. Mention 5 system calls used for process management & briefly explain: **fork():** creates copy of process. **Getpid():** gets the PID of the program **getppid():** gets the parent process id. **Exec():** invoked by a process to change the code that is exec. **Kill():** sends signals o a process.

d. Write the software equivalent of the test\_and\_set() hardware instruction & show how to use it controlling entry to a CS:

boolean TestAndSet (boolean \*target) { boolean rv = \*target;

\*target = TRUE;

return rv:}

do {while ( TestAndSet (&lock ));// critical section lock = FALSE;} while ( TRUE);

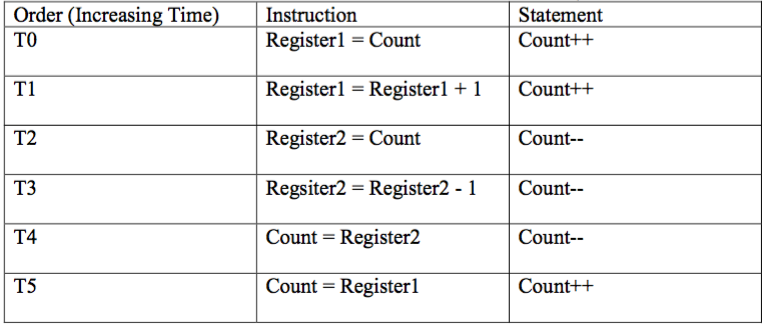
e. Process Synchronization (Race conditions): Assume you have two process incrementing & decrementing a variable about the same time using the statements count++ & count --…

Count++: Register1 = Count Register1 = Register1 + 1

Count = Register1

Count--: Register2 = Count Regsiter2 = Register2 - 1

Count = Register2



The assembly code corresponding to the increment pick the original value of count (i.e. 1) in Register 1 and carries the addition resulting in a count of 2 but the decrement code fully runs just before the increment code gets to deposit the final result in the memory when count is being stored. This is most likely the result of a context switch from the process incrementing the count to the process performing the decrement.

**Sturct Semaphore{**

**int count;**

**mutex lock;**

**queue\_t Q;**

**}Semaphore\_t;**

**Signal(Semaphore\_t \*S){**

**S->lock.acquire();**

**if(S->Q.isEmpty()){**

**S->count++;**

**} else { InsertReadyQueue(S->Q.dequeue());}**

**S->lock.release();}**

**wait(Semaphore \*S)**

**{S->lock.acquire();**

**if(S->count <= 0)**

**{S->Q.enqueue(CurrentProcess);**

**S->lock.release();**

**Scheduler();}**

**else {S->count--;**

**S->lock.release(); }}}**

**Definitions:**

Api – application programming interfacw

Interrupt vector- array of pointers to interrupts handler functions

Thread:the unit of scheduling in a kernel that implements threads, or a userlevel hreead library that implement thrd