University of California, Berkeley
College of Engineering
Computer Science Division EECS

Spring 2023 Kubiatowicz

Midterm II
SOLUTION

March 15th, 2023
CS162: Operating Systems and Systems Programming

Your Name:	
SID:	
TA Name:	
Discussion Section Time:	

General Information:
This is a **closed book** exam. You are allowed 2 pages of notes (both sides). You have 110 minutes to complete as much of the exam as possible. Make sure to read all of the questions first, as some of the questions are substantially more time consuming.

Make your answers as concise as possible. On programming questions, we will be looking for performance as well as correctness, so think through your answers carefully. If there is something about the questions that you believe is open to interpretation, please ask us about it!

Problem	Possible	Score
1	20	
2	20	
3	22	
4	18	
5	20	
Total	100	

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899

Problem 1: True/False and Why[20 pts]

Please *EXPLAIN* your answer in TWO SENTENCES OR LESS (Answers longer than this may not get credit!). Also, answers without an explanation *GET NO CREDIT.*

Problem 1a[2pts]: In systems which provide multiple segments at a time, the segment identifier is always taken from a portion of the virtual address.

□ True ☑ False

Explain: *The x86 instruction set identifies the appropriate segment from the instruction opcode. For instance a "push" instruction operates on the SS segment.*

Problem 1b[2pts]: A system that is in a SAFE state is guaranteed to eventually complete execution of all threads.

□ True ☑ False

Explain: *This statement is false for multiple reasons. (1) the fact that the system is in a SAFE state doesn't prevent a thread from going into an infinite loop, preventing it from completing; (2) the only thing that a SAFE state guarantees is that there exists some scheduling of threads that will be able to complete, assuming the no thread ask for a total number of resources exceeding their specified maximum; other schedules might still deadlock.*

Problem 1c[2pts]: The CFS scheduler makes sure that every runnable thread gets the CPU for a minimum amount of time.

☑ True □ False

Explain: *CFS has a Minimum Granularity value which is the minimum quantum (amount of CPU time) that a scheduled thread gets. This helps to avoid excessive overhead when there are too many threads to schedule normally.*

Problem 1d[2pts]: The `fork()` system call must allocate new physical memory for the child process and copy all of the contents of the parent's address space into this new memory in order ensure that parent and child can subsequently modify memory contents without affecting each other.

□ True ☑ False

Explain: *Instead of copying all of the memory contents, the OS can perform a fork() by copying the page tables of the parent into the child. By setting the permission bits of each PTE to "read-only", the OS can copy pages only when parent or child tries to write (copy on write or COW).*

Problem 1e[2pts]: A user process could generate a page fault while performing a load or store operation to a memory address that it has access to.

☑ True □ False

Explain: *A page fault could occur for a number of reasons even when the process is generating a valid access: (1) a write to a copy-on-write page that was set to read-only after a fork operation. (2) the accessed page could have been temporarily paged out to disk.*

Problem 1f[2pts]: In PintOS, there is a single kernel thread that handles all of the system calls made by user threads in a process (clarification during exam: answer after Project 2 completion).

☐ True ☑ False

Explain: *Each user thread has its own paired kernel thread (i.e. stack) that is used during system calls from that thread. Thus there are multiple kernel threads, not a single one.*

Problem 1g[2pts]: In PintOS, the timer_interrupt function (which handles timer interrupts) does not need to be reentrant (i.e. handle the case in which more than one timer interrupt event is being handled at the same time).

☑ True ☐ False

Explain: *At the time that the timer_interrupt handler is invoked, interrupts (including timer_interrupts) are disabled. Consequently, additional timer_interrupts will not be handled until after the current interrupt is fully processed.*

Problem 1h[2pts]: Priority donation is a mechanism whereby a non-interactive thread gives its priority to an interactive thread, thereby raising the priority of the interactive thread and making it more responsive.

☐ True ☑ False

Explain: *Priority Donation is a process by which a higher-priority thread attempting to acquire a lock "donates" its priority to the lower-priority thread holding the lock in order to guarantee that the lower-priority thread finishes its execution and releases the lock.*

Problem 1i[2pts]: A system which will be used to perform hard realtime scheduling needs to have an accurate measure for the worst-case execution time (WCET) of every task that will be run.

☑ True ☐ False

Explain: *While WCET is not used during scheduler execution, it is an important component of various schedulability criteria to determine if a group of threads can be properly scheduled together while still making deadlines.*

Problem 1j[2pts]: A system with an inverted page-table uses an amount of memory for the page table that scales with the size of the physical memory.

☑ True ☐ False

Explain: *Since an inverted-page table is a hash-table mapping from virtual-memory pages to physical memory pages, it only needs to contain mappings (PTEs) for those virtual-to-physical mappings that are currently valid; the number of such valid mappings with scale with the size of physical memory (more physical memory, more PTEs). Invalid mappings (i.e. "not present") can be represented simply by the* absence *of a PTE mapping for a given virtual address.*

Problem 2: Multiple Choice [20pts]

Problem 2a[2pts]: Consider the following pseudocode for three threads (T1, T2, T3) that try to acquire three locks (A, B, C):

```
   T1 {                    T2 {                    T3 {
L1.   acquire A         L2.   acquire B         L4.   acquire C
L3.   acquire B         L5.   acquire C         L6.   acquire A
      acquire C               acquire A               acquire B
      release C               release A               release B
      release B               release C               release A
      release A               release B               release C
   }                       }                       }
```

Assume the code executes lines L1, L2, L3, L4, L5, L6, in that order. After line L6, the system is deadlocked. After executing which line did the system enter an unsafe state? *(Select one):*

A: ☐ L1
B: ☑ L2
C: ☐ L3
D: ☐ L4
E: ☐ L5

A: Not selected because only one resource has been acquired, so there cannot be lack of safety
B: This is an unsafe state, because T1 and T2 have acquired a lock needed by the other to complete and thus neither thread can complete.
C-E: Not selected because we are already in an unsafe state in B.

Problem 2b[2pts]: Select all of the following that are true of page tables (*choose all that apply*):

A: ☐ Each thread within a process has its own page table.

B: ☑ The number of PTEs in a simple, single-level page table is proportional to the size of virtual memory.

C: ☑ Multi-level page tables are more memory-efficient than single-level page tables for sparse address spaces.

D: ☐ In a system with 2^{16} physical pages, a page table entry will consist of exactly 16 bits.

E: ☑ Two different page tables can reference the same physical page.

A: FALSE. All threads within a process share the same page table.
B: TRUE. If there are N total addresses in the virtual address space (for example $N=2^{32}$ for a 32-bit address space) and pages are of size P (for example 2^{12}), the total number of PTEs in a single=level page table is N/P, which is proportional to N.
C: TRUE. A sparse virtual address space has a number of large unmapped areas, e.g. the hole between heap and stack. A single-level page table needs a null (invalid) PTE for every unmapped page (of which there are many). A multi-level page table can represent a large number of invalid mappings with a single invalid PTE in the top-level of the page table.
D: FALSE. The PTE would need not only 16 bits to identify the page frame, but additional bits for the control bits (i.e. "P", "W", etc).
E: TRUE. This is the basic mechanism for shared memory.

Problem 2c[2pts]: Which of the following is true about a multi-level feedback queue (MLFQ)? (*choose all that apply*):

A: ☐ Starvation is never a problem for a MLFQ.

B: ☑ Short-running tasks are given higher priority.

C: ☑ A MLFQ is an approximation of shortest remaining time first.

D: ☑ A MLFQ is less fair than round robin, where "fair" is with respect to CPU time.

E: ☑ One way to keep a job's priority high in a MLFQ is to insert a bunch of short sleep operations.

A: FALSE. The continuous arrival of new threads (which get placed into the highest-priority queue) can prevent longer-running threads in lower queues from running.
B: TRUE. Tasks which run for short periods are placed in the top, highest-priority queue.
C: TRUE. MLFQ provides an approximation to SRTF by trying to sort tasks by runtime length dynamically (longer running tasks get forced to lower priority queues), then give priority to shortest tasks first.
D: TRUE. Since MLFQ prioritizes short-running tasks, it is not as fair as Round-robin, which just cycles through runnable threads over and over.
E: TRUE. The addition of many sleep() calls will fool the scheduler into classifying this job as a short-runtime thread, which will keep it high in a high queue (and thus give it priority).

Problem 2d[2pts]: Which of the following are true about CFS? (*Choose all that apply*):

A: ☑ CFS attempts to equalize the virtual CPU time for all jobs.

B: ☐ Giving a thread a lower nice value results in CFS allocating a lower share of physical CPU time to that thread.

C: ☐ Adding new thread to the run queue in CFS takes *O(1)* time.

D: ☑ The target latency in CFS is the period of time over which every job should get service.

E: ☑ In CFS, higher-weight threads accumulate less virtual CPU time per physical CPU cycle than lower-weight threads.

A: TRUE. This is the basic algorithm of CFS; it keeps threads sorted by virtual time in a queue (heap) of threads. It selects the next thread to run as the thread with the least virtual time.
B: FALSE. Thread weight in CFS=$1024/(1.25)^n$ which means that lower nice=>higher weight. In CFS, higher weight => more physical CPU time. Note that this behavior is completely consistent with Unix Nice values in general, for which smaller (or more negative) nice values are "less nice", meaning that they consume more CPU time.
C: FALSE. CFS uses a red-black tree for the run queue. Red-black queues have O(log n) operations.
D: TRUE. CFS uses the weights of all the threads to split up the target latency into a series of quanta for each of the threads. The sum of all of the quanta add up to the target latency; consequently, each thread (running its quanta) gets a chance to run at least once during the target latency period.
E: TRUE. In CFS, each selected thread runs for its quanta. After the quanta expires, the virtual CPU time of the thread is updated by adding the CPU time it just ran, divided by the quanta. Consequently, higher weight threads accumulate a smaller amount of virtual CPU time per physical CPU time.

Problem 2e[2pts]: One of the following statements about schedulers is false. Which one is it? (*choose the false statement)*:

A: ☐ FCFS maximizes throughput of threads (where throughput represents the average amount of actual work being done by threads).

B: ☐ SRTF minimizes average completion time of threads.

C: ☑ The average throughput of threads running under a RR scheduler (regardless of quanta) is always greater than or equal to the average throughput of threads running with FCFS.

D: ☐ The average throughput of threads running under a RR scheduler with quantum q=2 is always greater than or equal to the average throughput of threads running under RR with q=1.

E: ☐ If threads arrive in increasing order of runtime, then the FCFS and SRTF schedulers will have the same average completion times.

A: Not selected because it is TRUE. Because FCFS runs threads to completion without unnecessary switching, it gives the maximum possibility for the processor to run at full speed (best caching behavior, branch prediction, TLB hits, etc).

B: Not selected because it is TRUE. SRTF is provably optimal at minimizing average completion time of threads.

C: Selected because it is FALSE. The RR scheduler (with any quantum) multiplexes threads, interrupting them more than if they were run under FCFS. Consequently, the throughput is higher under FCFS.

D: Not selected because it is TRUE. Because RR with q=2 interrupts threads less than RR with q=1, this is true.

E: Not selected because it is TRUE. SRTF operates by running threads in increasing order of thread runtime. If threads arrive in that order, then FCFS will run them in the same order as SRTF.

Problem 2f[2pts]: Which of the following is true about TLBs? (*choose all that apply):*

A: ☑ For physically addressed caches, TLB access can occur in parallel with cache lookup, even though the physical address is required to complete cache access.

B: ☐ The TLB is always structured as a single-level, fully-associative caching structure for performance reasons.

C: ☐ During a context switch from a thread in one process to a thread in another, the contents of the TLB must saved into the PCB of the first process and loaded from the PCB of the second.

D: ☐ During a TLB miss on an Intel x86 processor, the OS must walk the page table to find the missing translation.

E: ☐ System with inverted page tables cannot benefit from the addition of a TLB.

A: TRUE. We showed a couple of ways to do this in class. The simplest one is to make sure that the cache index (and byte offset) fit entirely in the page offset; this way the cache access can operate on the page offset bits while the TLB translates virtual=>physical page.

B: FALSE. TLBs are caches on translations and can thus be structured in many different ways (just like caches). Multi-level TLBs are certainly used on a number of processors..

C: FALSE. During context switches, TLBs are flushed, not saved and restored.

D: FALSE. As discussed in class, the x86 processor has hardware (MMU) that walks the page table. The OS does not do this.

E: FALSE. An inverted page table is stored in memory, just like a regular (forward) page table. Consequently, table lookup would take time and a system with an inverted page table will thus benefit from the performance improvement of the TLB.

.

Problem 2g[2pts]: Which of the following schedulers attempts to automatically give priority to interactive processes (those getting input from users) over long-running ones and can be actually implemented in a laptop? (*Choose all that apply):*

A: ☐ Lottery Scheduler
B: ☐ SRTF
C: ☑ Linux CFS
D: ☐ Round-Robin
E: ☑ MLFQ

A: FALSE. On average, the Lottery Scheduler provides a fixed fraction of the CPU, where the fraction is determined based on lottery tickets. It does not try to automatically figure out which processes/threads are interactive so that they can be prioritized.
B: FALSE. SRTF cannot be implemented in any form, since it requires precise information about the future execution time of threads.
C: TRUE. Threads with short runtimes (i.e. interactive threads) get behind other (non-interactive) threads in virtual runtime. Consequently, when these interactive threads do run, they get selected with priority over long running threads.
D: FALSE. Round robin doesn't do any prioritizing, it just multiplexes threads.
E: TRUE. The multi-level feedback queue scheduler consists of multiple queues organized in priority order. Further, short-running threads stay in high priority queues, while long-running threads move to lower-priority queues. This process prioritizes interactive threads automatically over long-running threads.

Problem 2h[2pts]: Consider a system with a priority-based scheduler that provides priority donation. Suppose that the following events occur in this order:

1. Thread A is created with priority 3 and calls acquire on locks lock_1 and lock_2.
2. Thread B is created with priority 6 and calls acquire on lock_1.
3. Thread C is created with priority 10 and calls acquire on lock_2.
4. Thread A releases lock_1.
5. Thread A releases lock_2.

Which of the following sequences represent the evolution of Thread A's effective priority? (*Choose one):*

A: ☑ 3, 6, 10, 10, 3
B: ☐ 3, 6, 10, 6, 3
C: ☐ 3, 3, 10, 6, 3
D: ☐ 3, 6, 10, 3, 3
E: ☐ 3, 6, 6, 3, 3

A: This option is correct, since Thread A starts at priority (3) when acquiring both locks. Then, when Thread B tries to acquire lock 1, it donates its priority (6) to Thread A. Next, when Thread C tries to acquire lock 2, it donates its priority (10) to Thread A. Next when thread A releases lock 1, nothing changes because Thread C is still donating its priority (10) in order get Thread A to release lock 2. Finally, when Thread A releases lock 2 (and no longer holds any locks), its priority returns to (3).

Problem 2i[2pts]: Which of the following is *not true* about the EDF scheduler? (*Choose the incorrect statement):*

A: ☐ It is specified for periodic tasks with well-defined worst-case execution time (WCET).

B: ☐ It can allocate up to 100% of processor cycles while still meeting deadlines.

C: ☐ It is a real-time scheduler.

D: ☐ It can run a non-realtime tasks in the background as long as the total utilization of the realtime tasks is < 1.

E: ☑ It operates by selecting the task with the shortest remaining computation time to run next.

A: This one was not selected because it is TRUE (EDF is specified for periodic tasks with WCET.
B: This one was not selected because it is TRUE (EDF can allocate up to 100% of processor cycles for periodic tasks, i.e. utilization == 1)
C: This one was not selected because it is TRUE (EDF is a real-time scheduler)
D: This one was not selected because it is TRUE. The notion of utilization < 1 means that the WCET computations for the realtime tasks would not need 100% of the CPU. Further, since B is true, and EDF is preemptive, there would be no problem with having a task running in the background at lower priority than the realtime tasks, and have it run whenever there is not a realtime-task with a pending deadline.
E: This was selected because it is FALSE. EDF operates by always running the task with the most pressing deadline – independent of the amount of computation it has remaining.

Problem 2j[2pts]: Which of the following are potential ways to avoid (reduce) the total number of cache misses in a particular situation? *(Choose all that apply):*

A: ☑ Increase cache size.
B: ☑ Increase cache associativity.
C: ☑ Decrease cache associativity.
D: ☑ Prefetching.
E: ☑ Rearrange the layout of data structures.

Cache miss rates are VERY application dependent. This means that the rate of cache misses depends on the actual access patterns in the application. In general, all of these are true, as discussed in class. I will discuss each of these, but perhaps C is the most counter-intuitive:

A: If your application is getting a lot of cache misses, it is likely because your cache is too small and an increased cache size will reduce the number of misses. This result is almost always true – unless you have a FIFO replacement policy (which you wouldn't on a cache, but might for demand paging).
B: If your application is getting a lot of conflict misses (accessing successive addresses that map to the same cache line) then you can reduce misses by increasing associativity (e.g. from direct mapped to 2-way set associative).
C: We talked about this in class. If you have a fully associative cache of size N and you repeatedly cycle through N+1 addresses, every access will be a miss. By going to a direct-mapped cache, you reduce it to 2 misses for every N+1 accesses.
D: By prefetching, you can reduce the number of compulsory misses.
E: By rearranging the in-memory layout of data structures, you can avoid misses in a cache which has conflict misses.

Problem 3: Deadlock and the Cephalopod Banquet [22pts]

Problem 3a[4pts]: Name and define the four conditions for deadlock ***(One sentence each):***

- *Mutual exclusion – Only one thread at a time can hold a given resource*
- *Hold and Wait – Thread holding at least one resource is waiting for another one*
- *No Preemption – Resources are released only voluntarily by the thread holding the resource, after thread is finished with it*
- *Circular Wait – There exists a set {T1, …, Tn} of waiting threads, with T1 waiting for a resource held by T2, T2 waiting for a resource held by T3, …, Tn waiting for a resource held by T1*

Problem 3b[2pts]: Suppose that we utilize the Banker's algorithm to determine whether or not to grant resource requests to threads. The job of the Banker's algorithm is to keep the system in a "SAFE" state. What is a SAFE state? ***(No more than two sentences):***

In a safe state, there is some ordering of the threads in the system such that threads can complete, one after another, without deadlocking and without requiring threads to give up resources that they already have.

Problem 3c[2pts]: Explain how the Banker's algorithm prevents deadlock by removing one or more of the conditions of deadlock from (3a). Be explicit. ***(No more than two sentences):***

Answer: The Banker's algorithm directs execution to keep the system in a SAFE state by avoiding any combinations of the four conditions of deadlock (i.e. your answer from 3a) that might lead to deadlock.

Note: This question is a bit of a trick question. The four conditions of deadlock from (3a) are necessary but not sufficient for deadlock. Thus, while the four conditions will continue to exist individually, the Banker's algorithm avoids exactly those configurations of the resource graph that exhibit non-resolvable cycles consisting of all four of the deadlock conditions from (3a).

Those of you that said that the Banker's algorithm will eliminate "circular waiting" or "hold and wait" did not get full credit, since threads may still be put to sleep (by Banker's algorithm) while holding resources; since they will be woken up by the Banker's algorithm when other threads finally release their resources, the sleeping threads are effectively waiting on these other threads. However, we did give credit to people who said that the Banker's algorithm prevents indefinite *circular waiting or hold and wait (or some variant of this statement that mentions the fact that the conditions do not persist indefinitely).*

Problem 3d[4pts]: Suppose that we wish to evaluate the current state of the system and declare whether or not it is in a SAFE state. If there were only two types of resource, we could describe the state of the system with the following data structures:

```
typedef struct FreeRes {      // Track system-wide resources
   int FreeResA, FreeResB;    // Number of copies ResA and ResB free
} FRes_t;

typedef struct ThreadRes {    // Track resources for one thread
   int MaxNeedA, MaxNeedB;    // Max number each resource needed
   int CurHeldA, CurHeldB;    // Current number resources held
} TRes_t;

FRes_t curFree;               // Structure of all free resources
TRes_t curThreadRes[];        // Array of all thread resources
```

Assume that curFreeRes and curThreadRes[] have been initialized to reflect the current state of the system. Fill in the missing lines to complete our check for safety. ***Only one expression per line, no comma expressions or additional semicolons.***

```
 1. bool IsSAFE(FRes_t *curFreeRes, TRes_t curThreadRes[], int numThreads) {
 2.    int FreeA = curFreeRes->FreeResA;
 3.    int FreeB = curFreeRes->FreeResB;
 4.    bool ThreadDone[numThreads] = {false}; // init array to all false
 5.    int Remaining = numThreads; // Number threads not finished
 6.    bool needPass = true;       // Completion flag
 7.    while (needPass) {
 8.       needPass = false;  // No threads completed yet this iteration
 9.       for (int i = 0; i < numThreads; i++) {
10.          if (!ThreadDone[i]) { // Thread hasn't completed, thus check

11.             if ((curThreadRes[i].MaxNeedA-curThreadRes[i].CurHeldA <= FreeA) &&

12.                 (curThreadRes[i].MaxNeedB-curThreadRes[i].CurHeldB <= FreeB)) {

13.                _FreeA += curThreadRes[i].CurHeldA_____________________;

14.                _FreeB += curThreadRes[i].CurHeldB_____________________;

15.                ThreadDone[i] = _true__________________;

16.                needPass = _ true_____________________;

17.                Remaining = _Remaining - 1_____________;
                }
12.          }
13.       }
14.    }
15.    return (_Remaining == 0_____________________________________);
16. }
```

The Cephalopod Diners Problem: Consider a large table with ***identical*** multi-even-armed cephalopods (e.g. octopuses). ***In the center is a pile of forks and knives.*** Before eating, each diner must have an equal number of forks and knives, one in each arm (e.g. if octopuses are eating, they would each need four forks and four knives). The creatures can only grab one utensil at a time. They grab utensils in a random order until they have enough utensils to eat. After they finish eating, they return all of their utensils at once. Diners are implemented as threads. Consider the following sketch for a Solution to the Cephalopod Diners problem using ***Mesa monitor synchronization:***

```
 1. typedef struct DinerUtensils {
 2.    int forks,knives;       // utensils held by creature
 3. } Diner_t;                 // clearly forks+Knives <= NumArms
 4. typedef struct CephTable {
 5.    struct lock lock;    // lock_init, lock_acquire, lock_release
 6.    struct condition CV; // cond_init, cond_wait, cond_signal, cond_broadcast
 7.    Diner_t *diners;     // Utensils for each diner
 8.    int numDiners, numArms;    // Number of diners and arms for each diner
 9.    int idleForks, idleKnives; // Number of forks/knives on table
10. } Table_t;
11. Table_t myTable;           // the current table!

12. // Initialize table. Arms must be even, both Forks and Knives >= Arms/2
13. void Init(Table_t *myTab, int Diners, int Arms, int Forks, int Knives) {
14.    lock_init(&(myTab->lock));
15.    cond_init(&(myTab->CV));
16.    // void *calloc(int n, size_t size) allocates num*size bytes set to 0
17.    myTab->diners = (Diner_t *)calloc(Diners, sizeof(Diner_t));
18.    myTab->numDiners = Diners;       // Number of Diners
18.    myTab->numArms = Arms;           // Number of arms per Diner
19.    myTab->idleForks = Forks;        // Number Forks on table initially
20.    myTab->idleKnives = Knives;      // Number Knives on table initially
21. }

22. // Called by diner "CephID" to return all their utensils to table
22. void DoneEating(Table_t *myTab, int CephID) {
23.    /* Return all utensils to the pile */
24.    lock_acquire(&(myTab->lock));
25.    myTab->idleForks += (myTab->diners)[CephID].forks;
26.    myTab->idleKnives += (myTab->diners)[CephID].knives;
27.    (myTab->diners)[CephID].forks = 0;
28.    (myTab->diners)[CephID].knives = 0;
29.    cond_broadcast(&(myTab->CV),&(myTab->lock));
30.    lock_release(&(myTab->lock));
31. }

32. // Called by diner "CephID" to grab a single utensil (fork or knife)
33. void GrabUtensil(Table_t *myTab, int CephID, boolean wantFork) {
33.    /* Implementation in Problem (3e) */
34. }

35. // Safety Check: Ok to give numforks forks and numknives knives to caller?
36. bool CephCheck(Table_t *myTab, int CephID, int numforks, int numknives) {
37.    /* Implementation in Problem (3g) */
38. }
```

Problem 3e[4pts]: Implement the code for the `GrabUtensil()` routine. It should return only after the request has been satisfied and sleep until it is ok. Assume that a Cephalopod will never request more resources than it needs. You can also assume that the `CephCheck()` routine returns true only if the requested number of forks and/or knives can be given to the particular Cephalopod without taking the system out of a safe state. `GrabUtensil()` should be able to deal with multiple threads accessing the state; implement this routine as a monitor and assume Mesa scheduling. Make sure your implementation is compatible with the provided `DoneEating()` routine. However, it is up to the Cephalopod thread to eat and subsequently call `DoneEating()`; you should not do that in `GrabUtensil()`. ***Only one expression per line, no comma expressions or additional semicolons. Do not worry about error handling code here.***

```
1.  // Called by diner "CephID" to grab a single utensil (fork or knife)
2.  void GrabUtensil(Table_t *myTab, int CephID, boolean wantFork) {
3.     int numforks = (wantFork?1:0);
4.     int numKnives = (wantFork?0:1);

5.     _Lock_acquire(&(myTab->lock));_____________________________;

6.     while (_!CephCheck(myTab, CephID, numforks, numknives)_____) {

7.        _cond_wait(&(myTab->CV), &(myTab->lock))_____________;
8.     }

9.     _myTab->diners[CephID].forks += numforks_____________;

10.    _myTab->idleforks -= numforks_____________;

11.    _myTab->diners[CephID].knives += numknives_____________;

12.    _myTab->idleknives -= numknives_____________;

13.    _Lock_release(&(myTab->lock))_____________;
14. }
```

Problem 3f[2pts]: In its general form, the Banker's algorithm makes a decision about whether or not to allow an allocation request by making multiple passes through the set of resource holders (threads). See, for instance, the nested loops in (3d). Explain why a Banker's algorithm *dedicated* to the Cephalopod Diners problem, namely the CephCheck() routine, could operate with a single pass through the resources holders. ***Provide two sentences or less.***

Since every diner (Cephalopod) at a given table has an identical number of arms (and thus resource requirements), once the Banker's algorithm has determined that a single Cephalopod can complete, then we know that they all can complete [simply put all of its resources back on the table, then there will be enough on the table for anyone to complete]. Thus, we need only a single pass through the diners to see if any of them can complete.

Problem 3g[4pts]: Finally, implement the `CephCheck()` method. This method should implement the Banker's algorithm: return true if the given Cephalopod can be granted '`numforks`' forks and '`numknives`' knives without taking the system out of a SAFE state. Assume that a Cephalopod will never request more resources than it needs. Do not worry about making this routing threadsafe; it will be called with a lock held by `GrabUtensil()`.

Do not blindly implement the Banker's algorithm: this method only needs to have the same external behavior as the Banker's algorithm for this application. Your answer to **(3f)** is relevant here. ***This code should not permanently alter the local variables of the Table_t (although it can do so temporarily). Only one expression per line, no comma expressions, and no additional semicolons.*** *Hint: check the requesting Cephalopod, then the rest.*

```
1.  // Safety Check: Ok to give numforks forks and numknives knives to caller?
2.  bool CephCheck(Table_t *myTab, int CephID, int numforks, int numknives) {
3.     // Easy case first
4.     if (numforks > myTab->idleForks || numknives > myTab->idleKnives)
5.        return false;

6.     if (myTab->numArms/2 - myTab->diners[CephID].forks <= myTab->idleforks &&

7.         myTab->numArms/2 - myTab->diners[CephID].knives <= myTab->idleknives ) {

8.        return _true_____________________________;

9.     }
10.    for (_int i = 0_________; _i < myTab->numDiners__; _i++_____________) {

11.       if (myTab->numArms/2-myTab->diners[i].forks <= myTab->idleforks-numforks &&

12.           myTab->numArms/2-myTab->diners[i].knives <= myTab->idleknives-numknives){

13.             return _true_____________________;
14.       }
15.    }
16.    return _false_____________________________;
17. }
```

Problem 4: Short Answer Potpourri [18pts]

Problem 4a[3pts]: Rohan finished implementing priority scheduling, but he isn't sure if his code handles priority scheduling for waiters on a condition variable correctly. Specifically, condition variables in the starter code wake up waiters in FIFO order, and he forgot if he modified them to wake up the highest priority waiter first. Since he is too lazy to look at his code, he instead decides to write a test to prove that he made the necessary changes. Fill in the following code such that it prints a different output in the following two scenarios:

1. The condition variable wakes up waiters in FIFO order.
2. The condition variable wakes up the highest priority waiter first.

In both cases, you should assume that the preemptive priority scheduler is implemented correctly: at any given time, the highest priority thread that is awake is running. You should also assume that other synchronization primitives (i.e. locks and semaphores) correctly wake up waiters in priority order. You do not need to test priority donation. ***While you do not have to fill in every line, place no more than one expression per line, no comma expressions or additional semicolons:***

```
1. static thread_func priority_condvar_thread;
2. static struct lock lock;
3. static struct condition condition;
4. void test_priority_condvar(void) {
5.   lock_init(&lock);
6.   cond_init(&condition);
7.   thread_set_priority(0);
8.   for (int i = 0; i < 3; i++) {
9.     // Priority varies from 0 to 63, higher value=>higher priority

10.    int priority = _i+1_____________________________________;
11.    char name[16];
12.    snprintf(name, sizeof name, "%d", i);
13.    // NOTE: `thread_create` yields if new thread has higher priority.
14.    thread_create(name, priority, priority_condvar_thread, NULL);
15.  }
16.  for (int i = 0; i < 3; i++) {
17.    lock_acquire(&lock);
18.    cond_signal(&condition, &lock);
19.    lock_release(&lock);
20.  }
21. }
22. static void priority_condvar_thread(void* aux UNUSED) {

23.  _lock_acquire(&lock)_____________________;

24.  _cond_wait(&condition,&lock)_____________;
25.  // thread_name() returns name-string from thread_create()
26.  msg("Thread %s woke up.", thread_name());

27.  _lock_release(&lock)_____________________;

28.  _____________________________________________;
29. }
```

Problem 4b[2pts]: Assuming the code of (**4a**), show how the two scenarios would differ: What would be the output in Scenario 1?

Thread 0 woke up.
Thread 1 woke up.
Thread 2 woke up.

What would be the output in Scenario 2?

Thread 2 woke up.
Thread 1 woke up.
Thread 0 woke up.

Problem 4c[2pts]: Rohan thinks he can get away with removing the "`for`" loop for calling `cond_signal` (lines 16-20 of code in (**4a**)) and instead acquire the lock, call `cond_broadcast` once, then release the lock. Would the test still work? *Explain in two sentences or less.*

No, the test would not work because all of the waiters would wake up before the initial thread releases the lock, and thus would run in priority order for both scenarios (i.e. the test wouldn't be able to distinguish).

Problem 4d[4pts]: Consider a computer system with a two-level hardware cache. Ignore any virtual to physical translation or concerns about cache size or transfer time. Also, assume there are no page faults. Possibly useful parameters are as follows:

Variable	Measurement	Value
P_{L1H}	Probability of cache hit when going to the first level of the cache (L1).	90%
P_{L2H}	Probability of a cache hit when going to second level of the cache (L2).	95%
S_{L1}	Size of L1 cache.	64 KB
S_{L2}	Size of L2 cache.	2 MB
S_M	Size of DRAM	4 GB
S_D	Size of Disk	2 TB
T_{L1}	Time to access L1 cache (hit)	5ns
T_{L2}	Time to access L2 cache (hit)	20ns
T_M	Time to access DRAM	100ns
T_D	Time to transfer a page to/from disk	10 ms

Compute the Average Memory Access time (AMAT) and show your work. Give a symbolic expression followed by an actual value for AMAT. You should be able to compute this value without needing a calculator, however you can leave an unsimplified expression if necessary (Show your work!):

Most of the information in the table is irrelevant. We just use AMAT for a two-level cache:

$$AMAT_{L1} = T_{L1} + (1 - P_{L1H}) \times AMAT_{L2} = T_{L1} + (1 - P_{L1H}) \times (T_{L2} + (1 - P_{L2H}) \times T_M)$$

$$AMAT_{L1} = 5ns + (1 - 0.90) \times (20ns + (1 - 0.95) \times 100ns) = 5ns + 2.5ns = 7.5ns$$

Problem 4e[2pts]: Provide one difference between how PintOS handles the main thread of a process calling pthread_exit() vs exit(). *Explain in two sentences or less.*

`pthread_exit()` waits for all threads to exit, while `exit()` forces all threads to exit.

Problem 4f[2pts]: In PintOS, the `sema_init()` system call returns an integer. Why can't the `sema_init()` system call return the actual initialized `struct semaphore` to the user? *Explain in two sentences or less.*

The OS cannot return a pointer to the actual structure because this structure is in kernel memory. Thus, the system call returns a handle in the form of an integer.

Problem 4g[3pts]:
Five jobs are waiting to be run. Their expected running times are 10, 8, 3, 1, and X. In what order should they be run to minimize average completion time? State the scheduling algorithm that should be used AND the order in which the jobs should be run. *HINT: Your answer will explicitly depend on X.*

To minimize average completion time, we should use SRTF (which is optimal). We can do this (even though SRTF is technically not implementable) since we explicitly know all the future runtimes of the threads. Thus, our scheduling order depends on the value of X in the following way:

```
X, 1, 3, 8, 10   if X < 1
1, X, 3, 8, 10   if 1 ≤  X < 3
1, 3, X, 8, 10   if 3 ≤  X < 8
1, 3, 8, X, 10   if 8 ≤  X <10
1, 3, 8, 10, X   if 10 ≤  X
```

[This Page Intentionally Left Blank]

Problem 5: Address Translation [20 pts]

In class, we discussed the "magic" address format for a multi-level page table on a 32-bit machine, namely one that divided the address as follows:

Virtual Page # (10 bits)	Virtual Page # (10 bits)	Offset (12 bits)

You can assume that Page Table Entries (PTEs) are 32-bits in size in the following format:

Physical Page # (20 bits)	OS Defined (3 bits)	0	Large Page	Dirty	Accessed	Nocache	Write Through	User	Writeable	Valid

Problem 5a[2pts]: What is the size of a page in this machine? *Show your work.*

Since the Offset is 12 bits, pages are 2^{12} = 4KB in size, i.e. 4096 Bytes.

Problem 5b[2pts]: What is "magic" about this configuration? Make sure that your answer involves the size of the page table and explains why this configuration is helpful for an operating system attempting to deal with limited physical memory. *Explain in two sentences or less.*

This is "magic" because each level of the 2-level page table takes exactly 1 page, making it easy to "page out" individual pieces of the page table. You can see property because there are 2^{10} ==1024 entries x 4 bytes per entry == 4096 bytes/page table level.

Problem 5c[2pts]: Assume that we have a 64-bit processor which has the same page size as you gave in problem (5a) and the same 12 access control bits as given in the above PTE. Now, if we reserve 8-bytes for each PTE in the page table (whether or not they need all 8 bytes), how would the virtual address be divided for a 64-bit address space? Make sure that your resulting scheme has a similar "magic" property as in (5b) and that all levels of the page table are the same size—with the possible exception of the top-level. How many levels of page table would this imply? *Explain in two sentences or less.*

Following the same property as above, we want each level of the page table to be one page in size, so would take $2^{12}/8 == 2^9$ => 9 bits per level. So total number of levels would be (64-12)=52/9, i.e. a 6-level page table divides like this: [7 bits][9 bits][9 bits][9 bits][9 bits][9 bits][12 bit offset]

Problem 5d[2pts]: Returning to a 32-bit processor with the virtual address format and PTE as shown above (in **5a)**, suppose that we want an address space with one physical page at the top of the address space and one physical page at the bottom of the address space. How much memory would be devoted to the page table for this mapping (in bytes)? *Explain in two sentences or less.*

The page table of interest has two non-null pointers for the first level, pointing at 2 second-level elements of the page table. Each element is a page in size. Thus, there are 3 pages = 3 × 4096 = 12288 bytes in the page table.

Problem 5e[2pts]: Given a 32-bit processor with the virtual address format and PTE as shown above (in **5a)**, and assuming that you can have as many processes (and page tables) as you like, what is the maximum amount of physical memory that this machine could divide up among these processes? *Explain in two sentences or less.*

The PTE format has 20 bits devoted to the physical page frame, so maximum amount of physical memory would be 2^{20} pages x 4KB=4GB of total memory (i.e. 2^{32}). The number of processes does not matter here.

Problem 5f[12pts]: Assume the translation scheme from (**5a**). Use the Physical Memory table given on the next page to predict what will happen with the following byte-oriented memory instructions. Assume a big-endian processor (i.e. the most significant byte of a 4-byte integer is stored first in memory). Assume that the base table pointer for the current ***user level process*** is `0x00200000`.

Fill in the missing entries in the following table. Addresses in the "Instruction" column are virtual. You should translate these addresses to physical address (i.e. in middle column), then attempt to execute the specified instruction on the resulting address. The return value for a load is an 8-bit data value or an error, while the return value for a store is either "**ok**" or an error. Possible errors are: **invalid, read-only, kernel-only.** *Hints: (1) Don't forget that hexadecimal digits contain 4 bits, so 10-bits is slightly more than two hexadecimal bytes! (2) PTEs are 4 bytes! (3) Make sure to look for duplicate lookup ~~at~~ (say) the first-level PTE!*

Instruction	First-Level PTE	Second-Level PTE	Physical Address	Result
Load [0x00001047]	0x00100007	0x00002067	0x00002047	*0x50*
Store [0x00C07665]	0x00103007	0xEEFF0067	0xEEFF0665	*Ok*
Store [0x00C005FF]	0x00103007	0x11220005	*0x112205FF*	ERROR: read-only
Load [0x00003012]	*0x00100007*	*0x00004007*	0x00004012	~~0x36~~ 0x84
Store [0x02001345]	*0x00100007*	*0x00002067*	0x00002345	ok
Load [0xFF80078F]	*0x001FE007*	0x04150000	*0x0415078F*	*ERROR: invalid*
Test-And-Set [0xFFFFF005]	0x001FF007	*0x00103067*	0x00103005	*0x66*

Physical Memory [All Values are in Hexidecimal]

Address	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
00000000	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
00000010	1E	1F	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D
….																
00001010	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
00001020	40	03	41	01	30	01	31	03	00	03	00	00	00	00	00	00
00001030	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF
00001040	10	01	11	03	31	03	13	00	14	01	15	03	16	01	17	00
….																
00002030	10	01	11	00	12	03	67	03	11	03	00	00	00	00	00	00
00002040	02	20	03	30	04	40	05	50	01	60	03	70	08	80	09	90
00002050	10	00	31	01	10	03	31	01	12	03	30	00	10	00	10	01
….																
00004000	30	00	31	01	11	01	33	03	34	01	35	00	43	38	32	79
00004010	50	28	36	19	71	69	39	93	75	10	58	20	97	49	44	59
00004020	23	03	20	03	00	01	62	08	99	86	28	03	48	25	34	21
….																
00100000	00	00	10	67	00	00	20	67	00	00	30	00	00	00	40	07
00100010	00	00	50	03	00	00	00	00	00	00	00	00	00	00	00	00
…																
00103000	11	22	00	05	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00
00103010	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00	67
…																
001FE000	04	15	00	00	48	59	70	7B	8C	9D	AE	BF	D0	E1	F2	03
001FE010	10	15	00	67	10	15	10	67	10	15	20	67	10	15	30	67
…																
001FF000	00	00	00	00	00	00	00	65	00	00	10	67	00	00	00	00
001FF010	00	00	20	67	00	00	30	67	00	00	40	65	00	00	50	07
…																
001FFFF0	00	00	00	00	00	00	00	00	10	00	00	67	00	10	30	67
…																
00200000	00	10	00	07	00	10	10	07	00	10	20	07	00	10	30	07
00200010	00	10	40	07	00	10	50	07	00	10	60	07	00	10	70	07
00200020	00	10	00	07	00	00	00	00	00	00	00	00	00	00	00	00
…																
00200FF0	00	00	00	00	00	00	00	00	00	1F	E0	07	00	1F	F0	07
…																

Function Reference Sheet

Here are a few function signatures for you

```
/* Pintos locks */
void lock_init(struct lock *lock);
void lock_acquire(struct lock *lock);
void lock_release(struct lock *lock);

/* Pintos semaphore interface */
void sema_init(struct semaphore *sema, unsigned value);
void sema_down(struct semaphore *sema);
void sema_up(struct semaphore *sema);

/* Pintos condition variables */
void cond_init(struct condition *cond);
void cond_wait(struct condition *cond, struct lock *lock);
void cond_signal(struct condition *cond, struct lock *lock);
void cond_broadcase(struct condition *cond, struct lock *lock);

/* Pintos Readers/Writers Locks */
void rw_lock_init(struct rw_lock*);
void rw_lock_acquire(struct rw_lock*, bool reader);
void rw_lock_release(struct rw_lock*, bool reader);

/* Pintos List */
void list_init(struct list *list);
struct list_elem *list_head(struct list *list);
struct list_elem *list_tail(struct list *list);
struct list_elem *list_begin(struct list *list);
struct list_elem *list_next(struct list_elem *elem);
struct list_elem *list_end(struct list *list);
struct list_elem *list_remove(struct list_elem *elem);
bool list_empty(struct list *list);
#define list_entry(LIST_ELEM, STRUCT, MEMBER) ...
void list_insert(struct list_elem *before, struct list_elem *elem);
void list_push_front(struct list *list, struct list_elem *elem);
void list_push_back(struct list *list, struct list_elem *elem);
```

[Scratch Page: Do not put answers here!]

[Scratch Page: Do not put answers here!]