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

Fall 2026 Kubiatowicz

Midterm II
SOLUTION
October 30th, 2026
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 170 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	16	
2	16	
3	21	
4	27	
5	20	
Total	100	

[This page left for π]

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899

Problem 1: True/False and Why[16 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 Linux CFS jobs with lower nice value (a high priority) are run more often than other jobs.

☐ True ☑ False

Explain: *A lower nice value gives a thread a larger weight, so its vruntime advances more slowly and it receives a proportionally larger time slice (a larger share of the CPU). Every runnable thread still runs once per scheduling period – it runs longer, not more often.*

Problem 1b[2pts]: A system that provides segmentation without paging necessarily puts holes in the physical address space, forcing the operating system to waste physical memory.

☐ True ☑ False

Explain: *Variable-sized segments do cause external fragmentation (holes), but the holes need not be wasted: the OS can place new segments into them, or compact memory by copying segments and updating their base registers.*

Problem 1c[2pts]: Resource Cycles always lead to deadlock.

☐ True ☑ False

Explain: *A cycle in the resource-allocation graph is necessary but not sufficient for deadlock. If a resource type has multiple instances, a thread outside the cycle that holds one of them may finish and release it, breaking the cycle.*

Problem 1d[2pts]: A direct mapped cache can sometimes have a higher hit rate than a fully associative cache with an LRU replacement policy (on the same reference pattern).

☑ True ☐ False

Explain: *Example: repeatedly cycling through N+1 blocks misses on every access in an N-block fully-associative LRU cache, while in an N-block direct-mapped cache only the two blocks that map to the same line miss – all the other blocks hit.*

Problem 1e[2pts]: You can always reduce the number of page faults by increasing the amount of memory available to a process.

☐ True ☑ False

Explain: *Belady's anomaly: with a FIFO replacement policy, adding memory can actually increase the number of page faults. (And once the working set fits, more memory cannot remove the remaining compulsory misses.)*

Problem 1f[2pts]: The lottery scheduler prevents CPU starvation by assigning at least one ticket to each scheduled thread.

☑ True ☐ False

Explain: *Every lottery draw picks a ticket uniformly at random, so a thread holding at least one ticket has a nonzero probability of winning each draw. It therefore (probabilistically) gets the CPU and cannot be starved indefinitely.*

Problem 1g[2pts]: The Rate Monotonic Scheduler (RMS) takes deadlines into account at runtime to help provide real-time guarantees.

☐ True ☑ False

Explain: *RMS assigns fixed priorities offline – the shorter the period, the higher the priority – and always runs the highest-priority ready task. Deadlines are met by checking the schedulability (utilization) bound in advance, not by examining deadlines at runtime.*

Problem 1h[2pts]: The original Martian lander got into a loop of rebooting over and over because the thin Martian atmosphere did not block cosmic rays well enough (causing corruption of the contents of DRAM memory).

☐ True ☑ False

Explain: *The Mars Pathfinder resets were caused by priority inversion, not cosmic rays: a low-priority task holding a mutex needed by the high-priority bus-management task was preempted by medium-priority tasks, the bus task missed its deadline, and the watchdog timer rebooted the system (fixed by enabling priority inheritance).*

Problem 2: Multiple Choice [16pts]

Problem 2a[2pts]: What is true about the SRTF scheduling policy? (*choose all that apply*):

A: ☑ SRTF is impossible to implement.
B: ☑ SRTF is optimal with respect to average response time.
C: ☐ SRFT will not preempt a running task when a task with a short burst time arrives.
D: ☐ SRTF optimizes throughput over responsiveness.
E: ☐ None of the above.

A: TRUE. SRTF needs to know each task's remaining running time – i.e. the future – so it can only be approximated (e.g. by MLFQ), never actually implemented.
B: TRUE. Always running the job that is closest to finishing minimizes the average response (completion) time; SRTF is optimal for this metric.
C: FALSE. SRTF is preemptive: a newly arrived task with a shorter remaining time preempts the running task.
D: FALSE. SRTF optimizes average response time; its frequent preemptions (and possible starvation of long jobs) do nothing for throughput.
E: FALSE. Obviously.

Problem 2b[2pts]: Debugging in Pintos can often be very difficult due to the complexity of the code base and often just staring at the code is an ineffective strategy. Which of the following are good strategies for debugging in Pintos? (*choose all that apply):*

A: ☑ Setting breakpoints in GDB to walk through specific function calls.
B: ☑ Setting memory watchpoints in GDB to watch for variable modifications.
C: ☑ Checking function pre- and post-conditions with ASSERT statements.
D: ☑ Checking execution progress with PANIC statements.
E: ☑ Checking variables with print statements.

A: TRUE. Breakpoints let you stop at a specific function and inspect its arguments, locals, and the backtrace.
B: TRUE. A watchpoint stops execution at the exact instruction that modifies a variable – ideal for finding who is corrupting a value.
C: TRUE. ASSERTs check invariants and pre/post-conditions and give you a backtrace at the first point where something goes wrong.
D: TRUE. PANIC halts the kernel, allowing for a hard-stop check of an incorrect path or progress to a desired destination point.
E: TRUE. Judiciously placed printf() statements are a simple and effective way to see what the kernel is actually doing.

Problem 2c[2pts]: What is true about priority inversion? (*choose all that apply):*

A: ☐ Priority inversion could not occur if there were only two possible priorities for threads.
B: ☑ Priority donation is a solution to priority inversion.
C: ☑ Priority inversion could result in starvation of high priority threads.
D: ☐ Priority inversion is only a problem on a multi-core system.
E: ☐ None of the above

A: FALSE. As discussed in class, imagine a low-priority thread grabs a lock that a high-priority thread subsequently needs. If the high-priority thread is doing a non-blocking poll of the lock, it will never let the low-priority thread run to release the lock. Alternatively, with only two priorities, a high-priority thread blocked on a lock held by a low-priority thread still waits while the lock holder shares the CPU (round-robin) with other low-priority threads – the inversion is merely bounded rather than unbounded.

B: TRUE. Donating the waiter's priority to the lock holder lets the holder finish its critical section and release the lock promptly.

C: TRUE. A high-priority thread can wait indefinitely while medium-priority threads keep preempting the low-priority lock holder (the Mars Pathfinder scenario).

D: FALSE. Priority inversion happens on a single core (Mars Pathfinder was a uniprocessor); it is a scheduling/locking problem, not a multicore one.

E: FALSE. Obviously.

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

A: ☑ Acquiring resources in a predetermined order is one way to avoid deadlock.

B: ☑ Deadlock is a form of starvation.

C: ☑ Rolling back thread state is one way to recover from deadlock once it occurs.

D: ☐ Deadlock means that no threads are able to make progress.

E: ☐ The Banker's algorithm is one way to recover from deadlock once it occurs.

A: TRUE. A global ordering on lock/resource acquisition prevents the circular-wait condition, so deadlock cannot occur.

B: TRUE. Deadlocked threads never make progress, so deadlock is a (permanent) form of starvation – although starvation does not imply deadlock.

C: TRUE. Recovery can roll one of the deadlocked threads back to a checkpoint (releasing its resources) and restart it, as with database transactions.

D: FALSE. Only the threads in the wait cycle are stuck; other threads in the system can continue running.

E: FALSE. The Banker's algorithm is a deadlock avoidance scheme (it refuses requests that would lead to an unsafe state); it does nothing once a deadlock has occurred.

Problem 2e[2pts]: Which of the following statements about memory management are true? (*choose all that apply):*

A: ☐ In general, MIN is the best page replacement policy that can be implemented.

B: ☐ Systems with segment registers cannot allocate physical memory in fixed-sized chunks since segments are variable sized.

C: ☐ Adding more physical memory to a system will always reduce the page miss rate.

D: ☑ The Clock algorithm is an approximation of LRU.

E: ☐ A hardware implemented modified bit is required in order to implement a page replacement policy without losing information.

A: FALSE. MIN is optimal, but it requires knowing future accesses, so it cannot be implemented – it is only a yardstick for other policies.

B: FALSE. Segmentation can be combined with paging (as on the x86): each segment is mapped through page tables, so physical memory is allocated in fixed-size pages.

C: FALSE. Belady's anomaly: with FIFO replacement, more memory can increase the miss rate (and once the working set fits, more memory does not help at all).

D: TRUE. Clock uses the hardware use bit to approximate LRU: it evicts a page that has not been referenced since the hand last passed it.

E: FALSE. The OS can emulate a dirty bit in software (map the page read-only; on the first write fault, mark it dirty and make it writable) – or simply always write pages back.

Problem 2f[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: ☐ EDF

C: ☑ Linux CFS

D: ☐ Round-Robin

E: ☑ MLFQ

A: FALSE. The lottery scheduler just picks proportionally to tickets; it has no mechanism of its own to detect interactive jobs – someone else must hand them more tickets.

B: FALSE. EDF is a real-time scheduler that orders tasks by deadline; it knows nothing about interactivity.

C: TRUE. Threads that sleep waiting for input accumulate no vruntime, so they have the smallest vruntime and run first when they wake – and CFS runs on every Linux laptop.

D: FALSE. Round-robin treats every thread identically.

E: TRUE. Jobs that block before using up their quantum stay in the high-priority queues while CPU hogs are demoted – and MLFQ is implemented in many real OSes.

Problem 2g[2pts]: What is true about the EDF scheduler? (*Choose all that apply*)

A: ☐ It can be used to enforce realtime deadlines when the worst-case execution time (WCET) is unknown.

B: ☐ It sorts periodic tasks into categories, based on their period, with the smallest period being assigned the highest priority.

C: ☑ It can allocate up to 100% of processor cycles to real-time periodic tasks while still providing a guarantee of meeting real-time deadlines.

D: ☐ It is simpler to operate than Rate-Monotonic-Scheduling (RMS), since it can operate non-preemptively.

E: ☐ None of the above

A: FALSE. EDF's guarantee rests on an admission test ($\Sigma C/P \leq 1$), which needs each task's worst-case computation time C.
B: FALSE. That describes Rate-Monotonic Scheduling; EDF uses dynamic priorities – the task with the earliest deadline runs first.
C: TRUE. EDF meets all deadlines for any periodic task set with utilization up to 100% (RMS only guarantees up to ~69%).
D: FALSE. EDF is preemptive: a newly released task with an earlier deadline preempts the running task.
E: FALSE. Obviously.

Problem 2h[2pts]: The Meltdown security flaw (made public in 2018) allowed user code to gain access to protected information in the kernel because (*choose one)*:

A: ○ There was a wide-spread bug in the x86 architecture that caused certain loads to *ignore* kernel/user distinctions in PTEs under the right circumstances, allowing user code to access kernel memory.

B: ○ User code could observe the *time* it took to complete certain system calls and therefore infer the contents of kernel memory.

C: ⊗ For certain processor pipelines, after making an illegal access to kernel memory (which would fail), user code could make timing measurements while accessing the cache and *deduce* the value stored in the kernel at that address.

D: ○ A POSIX-compatible system call implemented by multiple operating systems neglected to properly check arguments and could thus be *fooled* into returning the contents of protected memory to users.

E: ○ The Meltdown flaw allowed a user code sequence to disable the thermal protection of modern multicore processors and caused chips to literally melt the vulnerable circuits that interpreted the User bit of the PTE, allowing uncontrolled access to kernel data.

C: TRUE. The exploit loads from a kernel address that will fault; before the fault is taken, out-of-order execution speculatively uses the loaded value to touch a user cache line, so the kernel byte can be deduced afterwards by timing which line is now cached (a cache side channel).

[This Page Intentionally Left Blank]

Problem 3: Short Answer Potpourri [21pts]

Problem 3a[2pts]: Consider a computer system with a two-level hardware cache. Ignore any virtual to physical translation time and, assume there are no page faults. Possibly useful parameters are:

Variable	Measurement	Value
P_{L1H}	Probability of cache hit when going to the first level of the cache (L1).	80%
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	64 GB
S_D	Size of Disk	2 TB
T_{L1}	Time to access L1 cache (hit)	2ns
T_{L2}	Time to access L2 cache (hit)	10ns
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:

$$AMAT = T_{L1} + (1 - P_{L1H}) \times [\, T_{L2} + (1 - P_{L2H}) \times T_{L2} \,]$$
$$= 2ns + 0.2 \times [\, 10ns + 0.05 \times 100ns \,]$$
$$= 5ns$$

Problem 3b[3pts]: Overlapping TLB lookup and cache access: A byte-addressable machine uses 32-bit virtual and physical addresses and 4 KiB pages. Its *physically tagged cache* has 16 KiB of data capacity, 64-byte blocks, and 4-way associativity. Metadata does not count toward data capacity. The processor attempts to perform the cache lookup in parallel with TLB lookup. After translation completes, it compares physical tags and checks access permissions.

(a) [1 point]: How many index bits and offset bits does the cache require? Show work:

64-byte blocks ⇒ offset = $\log_2 64$ = 6 bits
4-way associativity ⇒ 4 banks, 16KiB/4 ⇒ 4KiB bank⇒12bits-6bits = 6 bit index

(b) [1 point]: Can the processor determine the correct cache index before TLB lookup completes? Explain using the number of cache index bits, cache offset bits, and page offset bits.

Yes: With 4KiB pages⇒ 12 bit page offset ≥ cache offset + index = 12
So, cache lookup bits fit within page offset bits.

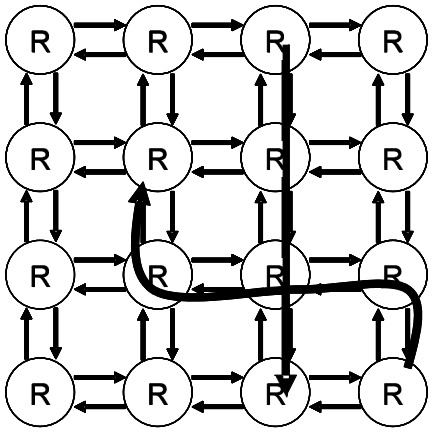
(c) [1 point]: Suppose the cache's capacity increases to 32 KiB, while retaining the 64-byte blocks. What is the *minimum* power-of-two associativity that allows overlapping TLB and cache lookup?

From (b), we see that the cache index can't get any larger if we leave page size and cache block size the same. This means that the size of the banks must stay the same (4 KiB). Thus we need to have 32KiB/4KiB = 8-way associativity.

Problem 3c[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. It denies resource requests by putting the requesting thread to sleep if granting the request would cause the system to enter an "UNSAFE" state, waking it only when the request could be granted safely. What is a SAFE state? *[Define in 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. If no such sequence exists the state is UNSAFE and a deadlock may (but need not) occur.

Problem 3d[2pts]: The figure at the right illustrates a 2D mesh of network routers. Each router is connected to each of its neighbors by two network links (small arrows), one in each direction. Messages are routed from a source router to a destination router and can stretch through the network (i.e. consume links along the route from source to destination). Messages can cross inside routers.



Assume that no network link can service more than one message at a time, and that each message must consume a continuous set of channels (like a snake). Messages always make progress to the destination and never wrap back on themselves. The figure shows two messages (thick arrows).

Define a routing policy that avoids deadlocks in this network by restricting the routes that messages take. Sketch a proof that deadlocks cannot happen when using this routing policy. *Hint: assume that a deadlock occurs and show why that leads to a contradiction in the presence of your new routing policy.*

The simplest policy was given in class: every message must route along the X direction first, then along the Y direction. (The is called "dimension order routing" but you weren't required to name it).

Proof that this is deadlock free by contradiction: assume there is a deadlock cycle. That would mean that there is a set of messages deadlocked in a cycle. However, such a cycle would involve at least one message that routed from the Y direction to the X direction, which is disallowed by our policy. Consequently, there is no deadlock.

Problem 3e[2pts]: One possible problem with your answer to (3d) is that your restrictions can cause serious congestion under some access patterns. Suppose that we were to double the total number of network channels (i.e. by duplicating every channel in (3d)). Call the new channels the "adaptive" channels. Explain how you could route without restriction on the adaptive channels while still avoiding deadlock.

Keep the original links as deadlock-free "escape" links that still obey X–Y routing, and let messages route freely (any direction, any turn) on the adaptive links. A message that is blocked waiting for an adaptive link may always instead take the escape link that X–Y routing dictates from its current router. Because any blocked message can always drain into the escape network, which is deadlock-free by (3d), a cycle of messages waiting on adaptive links cannot persist – so there is no deadlock.

Problem 3f[3pts]:
Here is a table of processes and their associated arrival and running times.

Process ID	Arrival Time	CPU Running Time
Process A	0	5
Process B	4	5
Process C	2	1
Process D	7	2
Process E	8	3

Show the scheduling order for these processes under 3 policies: First Come First Serve (FCFS), Shortest-Remaining-Time-First (SRTF), Round-Robin (RR) with timeslice quantum = 1. *Assume that context switch overhead is 0 and that new RR processes are added to the **head** of the queue and new FCFS processes are added to the **tail** of the queue.*

Time Slot	FCFS	SRTF	RR
0	***A***	***A***	***A***
1	***A***	***A***	***A***
2	***A***	***C***	***C***
3	***A***	***A***	***A***
4	***A***	***A***	***B***
5	***C***	***A***	***A***
6	***B***	***B***	***B***
7	***B***	***D***	***D***
8	***B***	***D***	***E***
9	***B***	***E***	***A***
10	***B***	***E***	***B***
11	***D***	***E***	***D***
12	***D***	***B***	***E***
13	***E***	***B***	***B***
14	***E***	***B***	***E***
15	***E***	***B***	***B***

Problem 3g[3pts]: Suppose that a system uses an N^th-chance clock algorithm for demand paging. Also assume that physical memory has 8 pages and each PTE has a dirty bit, a use bit, and a counter. Assume that the clock algorithm evicts dirty pages when its counter reaches 2, and evicts clean pages when its counter reaches 1. Assume that the clock hand starts at PPN 7 (wrapping around to PPN 0) and that the algorithm checks control bits *first,* before incrementing clock hand. Assume that the physical pages are allocated to virtual pages as follows:

PPN	Contains VPN	Dirty	Use	Counter
0	1	1	~~1~~ 0	0
1	3	0	~~1~~ 0	0
2	~~5~~ 6	0	0	~~0~~ ~~1~~ 0
3	2	0	~~0~~ ~~1~~ 0	0
4	8	0	~~1~~ 0	0
5	~~12~~ 4	~~1~~ 0	0	~~1~~ ~~2~~ 0
6	9	1	0	~~0~~ 1
→ 7	~~7~~ 10	0	~~1~~ 0	0

For the following access pattern, provide the PPN of the page frame at the time that the access succeeds. Assume all accesses are reads. We have filled the first entry for you:

Accessed VPN 3: PPN =? **1**

Accessed VPN 2: PPN =? ***3 (hit: set use ⇒ 1)***

Accessed VPN 6: PPN =? ***2 (set use ⇒ 0 on 7, 0, 1; count on 2 from 0⇒1⇒0 [clean])***

Accessed VPN 8: PPN =? ***4***

Accessed VPN 4: PPN =? ***5 (set use ⇒ 0 on 3, 4; count on 5 from 1⇒2⇒0 [dirty]; dirty⇒0)***

Accessed VPN 10: PPN =? ***7 (count on 6⇒1)***

Problem 3h[4pts]: For the following problem, assume a hypothetical machine with 4 pages of physical memory and 7 pages of virtual memory. Given the access pattern:

A B C D E A F A E C F G B A G D C A

Indicate in the following table which pages are mapped to which physical pages for each of the following policies. Assume that a blank box matches the element to the left. We have given the FIFO policy as an example. *Break ties by choosing PPN with smallest ID!*

Access→		**A**	B	C	D	E	A	F	A	E	C	F	G	B	A	G	D	C	A
FIFO	1	A				E							G					C	
	2		B				A							B					
	3			C				F							A				
	4				D						C						D		
MIN	1	***A***																	
	2		***B***					***F***					***G***				***D***		
	3			***C***															
	4				***D***	***E***								***B***					
LRU	1	***A***				***E***								***B***				***C***	
	2		***B***				***A***						***G***						
	3			***C***				***F***									***D***		
	4				***D***						***C***				***A***				

Problem 4: Pintos Scheduling and Debugging[27pts]

As you might recall, in the early part of the term, we introduced the Linux `futex()` system call, short for "fast userspace mutex". Now that you have seen the internals of the kernel scheduler as part of preparing to do Project 2, we will implement the kernel-side of a simplified version of the `futex()` system call for Pintos. The signature for our simplified `futex` syscall is as follows:

```
int32_t futex(int32_t *uaddr, int32_t futex_op, int32_t val) {
  /* uaddr: a pointer to an int in user space.
     futex_op: a code indicating which futex operation to perform
     (e.g. FUTEX_WAIT or FUTEX_WAKE).
     val: a value that is compared against *uaddr.
  */
}
```

Normally, this system call is buried in `libc` and used to implement various locking primitives such as locks, semaphores, and monitors. While the full version of futex() defines a number of futex_op values, we will focus on only two of them:

1. For `FUTEX_WAIT`, if `*uaddr == val`, the calling thread will go to sleep. The loading of the value stored at uaddr, the comparison of that value with val, and the blocking of the thread will happen atomically. After waking up, the calling thread will **return 0**. If, instead, `*uaddr != val`, the function will return immediately with a **return value of -1**.
2. For `FUTEX_WAKE`, this function will wake up to val waiting threads. You can assume in this problem that val will always be <= the actual number of waiting threads. **This will return the number of threads actually awoken.**

Consequently, `futex()` provides a unique sleep queue in the kernel associated with each uaddr.

As you might recall, in lecture, we showed one possible implementation of a user-level lock using the `futex()` system call as a primitive. It looked like this:

```
acquire(int32_t *thelock) {
   while (test&set(thelock)) {
      futex(thelock, FUTEX_WAIT, 1);
   }
}
```

```
release(int32_t *thelock) {
   *thelock = 0;
   futex(thelock, FUTEX_WAKE, 1);
}
```

Problem 4a[2pts]: Explain the behavior for a `futex()` call of type `FUTEX_WAIT`, given above. Why must the `futex()` system call check if `*uaddr == val`, given that, presumably, this property was true at the time that we decided to call `futex()`? (look at the code for `acquire()`, given above)? What must be true at the time that we make the check?

Answer: *The kernel must recheck the condition, since it might have changed between the time that it was shown to be true (in the* while() *condition) and when we get into the kernel [if we don't, we might miss a wakeup from* release()*]. We must check this condition atomically against any other threads that might be running, i.e. in Pintos with interrupts disabled.*

In Linux, the `futex()` system call works across processes when you use `mmap()` to share the memory that contains the `uaddr` value. However, in this problem, we are going to implement a more restricted version of `futex()` that is for synchronizing threads within a single multithreaded process. To implement the `futex` system call, we will need the following structures and functions, whose declarations we will place in `src/synch/synch.h`:

```
// Structure to track a thread while it is waiting
struct futex_waiter {
   int32_t *uaddr;
   struct thread *thread;
   struct list_elem elem;
};

// Sleeps if *uaddr == val. Otherwise, returns immediately.
int32_t sleep_on_addr(int32_t *uaddr, int32_t val);

// Wakes up to val threads sleeping on uaddr.
// Returns the number of threads woken.
int32_t wake_on_addr(int32_t *uaddr, int32_t val);
```

To anchor our list of waiting threads, we will add the following to src/userprog/process.h and assume that the list is correctly initialized on PCB creation:

```
struct process {
   /* Owned by process.c */

   . . .

   struct list futex;

};
```

In addition to the Pintos list functions given in the Function Reference Sheet at the end of the exam, some other functions that might be helpful are as follows:

```
// Interrupt control functions
enum inter_level intr_disable();
void intr_set_level(enum inter_level level);

// Scheduling functions
struct thread *thread_current();
void thread_block();
void thread_unblock(struct thread *thread);

// User address validation similar to Project 1
// Checks validity of numbytes bytes starting at address uaddr
// If invalid, will kill process automatically
void valid_uaddr(void* uaddr, uint32_t numbytes);
```

Problem 4b[3pts]: First, let's implement the syscall routing to the appropriate functions. We are guaranteed that the op is either FUTEX_WAIT or FUTEX_WAKE when the syscall number is SYS_FUTEX. Complete the following code (*Hint: don't forget validation or the return value*):

```
1.  // System call handler
2.  static void syscall_handler(struct intr_frame* f) {
3.     uint32_t* args = ((uint32_t*)f->esp);
4.     valid_uaddr(args, 4);  // Check syscall number argument
5.     switch (args[0]) {
6.        case SYS_FUTEX:
7.           valid_uaddr(args, 16); // checking rest of args                ;
8.           int32_t* uaddr = (int32_t*)args[1];                            ;
9.           valid_uaddr(uaddr, 4); // sizeof(int32_t)                      ;
10.          int32_t op = (int32_t)args[2];                                 ;
11.          int32_t val =(int32_t)args[3];                                 ;
12.          f->eax = sys_futex(uaddr, op, val);  // Return result         ;
13.          break;
14.       // Other system calls
15.       . . .
16.    }
17. }

18. // futex system call handling
19. int32_t sys_futex(int32_t *uaddr, int32_t op, int32_t val) {
20.    switch (op) {
21.       case FUTEX_WAIT:
22.          return sleep_on_addr(uaddr, val);
23.       case FUTEX_WAKE:
24.          return wake_on_addr(uaddr, val);
25.       default:
26.          // Unsupported operation, return error
27.          return -1;
28.    }
29. }
```

Problem 4c[4pts]: Next, let's write the function that implements the FUTEX_WAIT functionality. Remember that we must potentially put the thread to sleep and switch to another running thread. *Hint: Be careful to synchronize in a way that avoids conflict with other threads!*

```
30. /*
31.  * Check *uaddr to see if still == val
32.  *    if so, sleep and later return 0
33.  *    if not, return error (-1) immediately
34.  */
35. int32_t sleep_on_addr(int32_t *uaddr, int32_t val) {

36.     enum inter_level old_level = intr_disable();                    ;

37.     if (*uaddr != val                                               ) {

38.         intr_set_level(old_level);  // restore inter level          ;
39.         return -1;  // Woke immediately

40.     } else {
41.         // Prepare waiter structure for later access by FUTEX_WAKE
42.         struct futex_waiter waiter; // uninitialized

43.         waiter.uaddr = uaddr;                                       ;

44.         waiter.thread = thread_current();                           ;
45.         // Enqueue for later

46.         list_push_back(&thread_current()->pcb->futex, &waiter.elem);    ;
47.         // Schedule next thread

48.         thread_block();  // Unload thread and resume something else    ;

49.         intr_set_level(old_level); // Running again, restore inter   ;
50.         return 0;
51.     }
52. }
```

Problem 4d[5pts]: Finally, let's write the function that implements the FUTEX_WAKE. Once again, remember to synchronize against other threads.

```
53. /*
54.  * Wake up to val threads sleeping on uaddr.  Return number awoken.
55.  */
56. int32_t wake_on_addr(int32_t *uaddr, int32_t val) {

57.     enum inter_level old_level = intr_disable();                    ;
58.     int32_t num_woken = 0;

59.     struct list* futex = &thread_current()->pcb-futex;              ;

60.     struct list_elem *e = list_begin(futex);                        ;

61.     while (e != list_end(futex)  && woken < val                     ) {

62.         struct list_elem *next = list_next(e);                      ;

63.         struct futex_waiter *waiter =list_entry(e,struct futex_waiter, elem);
64.         // Check if this waiter is waiting on the address we are waking

65.         if (waiter->uaddr == uaddr                                  ) {

66.             list_remove(e);                                         ;

67.             thread_unblock(waiter->thread);                         ;
68.             num_woken++;
69.         }
70.         e = next;
71.     }

72.     intr_set_level(old_level); // Restore inter level               ;
73.     return num_woken;
74. }
```

Problem 4e[2pts]: Why doesn't `sleep_on_addr()` use some form of `malloc()` to allocate a `futex_waiter` structure when the thread is put to sleep and why, correspondingly, doesn't `wake_on_addr()` need to later free the `futex_waiter` structure. Where is the `futex_waiter` stored while the thread is blocked and how is it freed when the thread is unblocked?

Answer: *The `sleep_on_addr()` function doesn't need to allocate the `futex_waiter` structure on the heap, because it can allocate it as a local variable on the stack. This works because the stack is retained even as the thread is put to sleep (line 48). When the thread is later awoken, `sleep_on_addr()` continues where it left off (at line 49) and automatically release the stack memory holding the `futex_waiter` when it returns to the caller.*

[This Page Intentionally Left Blank]

Problems 4f-4i[11pts]: Pintos Debugging: For the next few questions, we are going to examine an implementation of `process_wait()` for Project 1 (User Programs). Unlike Project 1, this code was written to support multiple user threads. Note that this code, shown on the next page, is known to be buggy!

Each parent keeps a list of struct `child_status`, one per child it has created, which is shared between the parent and the child. Its `ref_cnt` counts the live holders (the parent and the child), and the struct is freed when the count reaches 0.

All of the code we will be looking at is on the next page. You may assume the following about code that is not shown:

- `process_execute()` initializes the fields as commented below, and once the child has loaded successfully it sets `cs->pid` and pushes cs onto the parent's `pcb->children` list while holding `pcb->lock`.
- The child stores its child_status in `pcb->self_status`.
- `pcb->lock` protects `pcb->children`. A process may have several user threads, and any of them may call `wait()`.
- You may assume code marked (correct) is correct, and `process_exit()` runs in only one thread per process.

The buggy code is on the next page, and questions are following. Place your answers in the provided boxes!

Problem 4f[3pts]: Consider the code for `child_status_release()` on the next page (lines 9–16). **It has a race condition**. Explain the problem, provide an interleaving of the parent and the child that causes a problem, and explain how to fix it. *BE Brief in your answer!*

Problem: *ref_cnt is read on line 13 after the lock is released, so the check is not atomic with the decrement.*

Example Failure: *The child runs lines 11–13 (2 → 1). Before it reaches line 14, the parent runs lines 11–13 (1 → 0). Now both threads see `ref_cnt == 0` on line 13, and both call `free(cs)`: a double free. (Also accept: the second reader reads `cs->ref_cnt` after the first has already freed it, which is a use-after-free.)*

The FIX: *Save the decremented value in a local while holding the lock, and test the local:*

```
lock_acquire(&cs->lock);
int remaining = --cs->ref_cnt;
lock_release(&cs->lock);
if (remaining == 0)
  free(cs);
```

---------- Begin Code for Pintos Debugging Questions ----------

```
 1. struct child_status {
 2.    pid_t pid;
 3.    int32_t exit_code;           /* initialized to -1 */
 4.    struct semaphore exited;  /* initialized to 0 */
 5.    struct lock lock;         /* protects ref_cnt */
 6.    int32_t ref_cnt;             /* initialized to 2 */
 7.    struct list_elem elem;    /* in parent's children list */
 8. };
 9. // Used in both process_wait and process_exit
10. static void child_status_release(struct child_status* cs) {
11.    lock_acquire(&cs->lock);
12.    cs->ref_cnt--;
13.    lock_release(&cs->lock);
14.    if (cs->ref_cnt == 0)
15.       free(cs);
16. }
17. int32_t process_wait(pid_t child_pid) {
18.    struct process* pcb = thread_current()->pcb;
19.    struct child_status* cs = NULL;
20.    struct list_elem* e;
21.    lock_acquire(&pcb->lock);
22.    for (e = list_begin(&pcb->children); e != list_end(&pcb->children);
23.         e = list_next(e))
24.       if (list_entry(e, struct child_status, elem)->pid == child_pid)
25.          cs = list_entry(e, struct child_status, elem);
26.    lock_release(&pcb->lock);
27.    if (cs == NULL)
28.       return -1;
29.    sema_down(&cs->exited);
30.    child_status_release(cs);
31.    return cs->exit_code;
32. }
33. void process_exit(int32_t status) {
34.    struct process* pcb = thread_current()->pcb;
35.    struct list_elem* e;
36.    /* ... kill other threads, close files (correct) ... */
37.    if (pcb->self_status != NULL) {
38.       pcb->self_status->exit_code = status;
39.       sema_up(&pcb->self_status->exited);
40.       child_status_release(pcb->self_status);
41.    }
42.    for (e = list_begin(&pcb->children); e != list_end(&pcb->children);
43.         e = list_next(e))
44.       child_status_release(list_entry(e, struct child_status, elem));
45.    /* ... destroy pagedir, free PCB (correct) ... */
46. }
```

---------- End Code for Pintos Debugging Questions ----------

Problem 4g[4pts]: There are **2 bugs** in `process_wait()` (lines 17–32). For each bug, give the line number(s), describe a scenario in which it causes incorrect behavior, and explain how to fix it.

Bug #1 (lines/failure scenario): *the entry is never removed from `pcb->children` (lines 22–25). If the parent calls `wait(pid)` twice, the second call should return -1 right away. Instead it finds cs again. If both references have already been released, cs has been freed and this is a use-after-free. Otherwise the parent blocks forever on exited, which was only upped once. The stale entry is also released a second time by the parent's process_exit(), which causes a double free. (Also accept: two threads of the same process call wait(pid) at the same time, and both claim the child because it stays in the list.)*

The FIX: *Call `list_remove(e)` (and break) when the match is found, while still holding `pub->lock`.*

Bug #2 (lines/failure scenario): `cs->exit_code` is read after `child_status_release(cs)` (lines 30–31). If the child has already released its reference, the parent's release frees cs, so line 30 reads freed memory (use-after-free).

The FIX: *Copy `cs->exit_code` into a local variable before calling `child_status_release(cs)`, and return the local copy.*

Problem 4h[2pts]: There is **1 bug** in `process_exit()` (lines 33-46). Give the line number(s), describe a scenario in which it causes incorrect behavior, and explain how to fix it.

Bug (lines/failure scenario): *`list_next(e)` is called after `e` may have been freed (lines 42–44). If a child has already exited and released its reference, `child_status_release()` on line 44 frees the struct that contains `e`. The loop's increment then reads `e->next` from freed memory (use-after-free).*

The FIX: *Pop each entry before releasing it:*

```
while (!list_empty(&pcb->children))
  child_status_release(list_entry(list_pop_front(&pcb->children),
                                  struct child_status, elem));
```

(Also accept saving e's successor before releasing it.)

Problem 4i[2pts]: A classmate suggests getting rid of struct child_status and the reference count entirely: "Just put exit_code and the exited semaphore in the child's struct process, and have the parent keep a pointer to the child's PCB." Explain why this does not work. Would putting them in the parent's struct process instead work? Why or why not?

Answer: ☐ Yes ☑ No

Explain: *The child's PCB is freed at the end of* `process_exit()`*. If the child exits before the parent calls* `wait()` *(which is allowed, and* `wait()` *must still return its exit code), the parent's pointer refers to freed memory, so reading* `exit_code` *or calling* `sema_down` *is a use-after-free.*

Parent's PCB: No. The parent may exit before the child (an orphaned child). The child would then write its exit code and `sema_up` *into the parent's freed PCB. The status must live in memory owned by neither process alone and be freed only when both are done with it, which is exactly what the reference count on a separately allocated child_status provides.*

Problem 5: Address Translation [20 pts]

Consider the x86 multi-level memory management scheme for 32-bit mode. The top-level of the translation table is called a "page directory" (PD) and the second level is a "page table" (PT). Virtual addresses are formatted as:

Directory (10 bits)	Page Table (10 bits)	Offset (12 bits)

The table-entries at the top level (Page Directory Entries or PDEs) and second level (Page Table Entries or PTEs) *are formatted identically*. They are 32-bits, ***stored in big-endian form*** in memory (i.e. the MSB is first byte in memory), interpreted as follows:

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

Here, "Present" means that a translation is valid, "Writeable" means that the page is writeable, "User" means that the page is accessible by the User (rather than only by the Kernel).

Problem 5a[2pts]: Explain why this particular "magic" page table format allows the OS to place unused parts of the page table out on disk to free up memory (although they can be brought back when needed). Are there any parts of the page table that must always remain in memory? Explain.

The special property held by this structure for a page table is that every component of the page table takes up exactly one (1) page, including the top-level page directory and second-level page table node. The second-level nodes can be written to disk and their memory freed by marking the top-level PTE as "not present". This behaves exactly like the individual pages (at leaves of structure). The top-level page directory cannot be paged out in this way, since no memory references can occur without it.

Problem 5b[1pts]: Suppose that we have a page table that maps 2000 pages at the bottom of the virtual address space and 5 pages at the top of the address space. What is the total size (in bytes) of the page table? Show your work.

*The 2000 pages would take up two second-level page table nodes (since each can point at 1024 pages. The top 5 pages would fit in one second-level page table node. Thus we have the top-level page directory (1 page in size) + 3 second-level page table nodes = (1 + 3) * 4096 = 16384 bytes.*

Problem 5c[1pts]: Assume that, in contrast to the layout of pages discussed in (5b), we allow the 2005 pages to be spread arbitrarily in the virtual address space. What is the largest size (in bytes) that the page table could be? Show your work.

*We can spread the 2005 pages across all 1024 second-level nodes (at least one page every 1024*4096 bytes). As a result, this page table would be of maximum size = (1+1024) * 4096 = 4198400 bytes.*

Problem 5d[2pts]: Can you make an argument why an Inverted Page Table for the layout of (5b) would be the same size (in bytes) as an Inverted Page Table for your maximized layout of (5c)

The inverted page table only needs entries for page mappings that are "Present", i.e. one entry per physical frame. So, in both layouts, there would be 2005 entries, so that Inverted Page Table would be same size.

Problem 5e[2pts]: The x86 architecture includes six segment registers. Assume that we have an instruction that loads from an address that is referenced to a segment such as ES:

```
mov eax, es:[edi]     ; move contents of es:[edi] into eax
```

How is the final physical address computed and how does this computation relate to the two-level page table described above? Be brief in your description:

The offset in `edi` would be added to the base address in the `es` segment register. The resulting address would be a 32-bit virtual address that would then be turned into a physical address by walking the 2-level page table.

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. If an address is invalid, write INVALID. 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	*0xAA*
Store [0x010005FF]	0x00104007	0x11220005	*0x112205FF*	ERROR: read-only
Load [0x00003028]	*0x00100007*	*0x00004007*	0x00004028	0x99
Store [0x02001ABC]	*0x00100007*	*0x00002067*	0x00002ABC	ok
Load [0xFF80178F]	*0x001FE007*	0x48597000	*INVALID*	*ERROR: invalid*
Store [0x01003644]	0x00104007	0xDDEEF067	0xDDEEF644	*ok*
Test-And-Set [0xFFFFF035]	0x001FF007	*0x00002067*	0x00002035	*0x03*

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	AA	AB	AC	AD	AE	AF	B0	B1	B2
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
…																
00104000	11	22	00	05	55	66	77	88	99	AA	BB	CC	DD	EE	F0	67
00104010	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF	00	00
…																
001FE000	04	15	00	00	48	59	70	00	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	00	20	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!]