CS162 Fall 2025

Ion Stoica, Matei Zaharia Midterm 1

Solutions last updated: Monday, October 13, 2025

Print Your Name: _____

Print Your Student ID: _____

Print the Name and Student ID of the person to your left: _____

Print the Name and Student ID of the person to your right: _____

Print the Name and Student ID of the person in front of you: _____

Print the Name and Student ID of the person behind you: _____

You have 80 minutes. There are 6 questions of varying credit. (75 points total)

Question:	1	2	3	4	5	6	Total
Points:	14	12	16	16	16	1	75

For questions with **circular bubbles**, you may select only one choice.

- ○ Unselected option (Completely unfilled)
- ◎ Don't do this (it will be graded as incorrect)
- ● Only one selected option (completely filled)

For questions with **square checkboxes**, you may select one or more choices.

- ■ You can select
- ■ multiple squares
- ☑ (Don't do this)

Anything you write outside the answer boxes or you ~~cross out~~ will not be graded. If you write multiple answers, your answer is ambiguous, or the bubble/checkbox is not entirely filled in, we will grade the worst interpretation. For coding questions with blanks, you may write at most one statement per blank and you may not use more blanks than provided.

As a member of the UC Berkeley community, I act with honesty, integrity, and respect for others. I will follow the rules of this exam.

Acknowledge that you have read and agree to the honor code above and sign your name below:

_____

This page left intentionally (mostly) blank

The exam begins on the next page.

Q1 *True or False* **(14 points)**

Please explain your answer in **two sentences or fewer**. Select only one of **true or false** and provide an appropriate explanation for the answer of your choice. You must fill out at most one box for each question and provide an explanation in the box below.

Q1.1 (2 points) Threads within the same process can modify each other's local stack variables.

● **True.** Explain a scenario where a thread can access and modify another thread's local variables.

○ **False.** Explain what mechanism within the kernel prevents this.

If the thread is passed a pointer to another threads stack variable it can modify it.

Q1.2 (2 points) Creating a pipe will allocate a file on disk for two processes to communicate.

○ **True.** Explain how the kernel manages the pipe's file.

● **False.** Explain what the kernel uses instead of a file.

The pipe is stored in a memory buffer.

Q1.3 (2 points) When an external device wants to communicate with the OS, it sends an IRQ line directly to the CPU.

○ **True.** Explain how the device sends the IRQ directly to the CPU.

● **False.** Explain what sends the interrupt to the CPU instead.

The PIC sends the interrupt.

Q1.4 (2 points) A user making a syscall in PintOS is required to specify the address of the kernel function.

◯ **True.** Explain where the user finds the addresses of the kernel functions.

● **False.** Explain what instruction and information you need to make a system call.

You use int 30 and put the system call number on the stack as an argument.

Q1.5 (2 points) In Unix, running `execv` keeps the file descriptors from the calling process accessible.

● **True.** Explain what `execv` does to let you keep the open file descriptors.

◯ **False.** Explain why file descriptors are lost when you do `execv`.

exec just replaces the running code, the pcb retains its open file descriptors

Q1.6 (2 points) In PintOS, a kernel-level stack can grow as long as required to accomplish its functions

◯ **True.** Explain how kernel-level stacks grow past their initial size.

● **False.** Explain the maximum size of the kernel's stack in PintOS.

No, it only takes up 1 page

Q1.7 (2 points) For any given program, having multiple threads **will always be strictly faster** than having a single thread if there's zero overhead from context switching and thread creation.

◯ **True.** Explain why multiple threads always make programs faster if there is no overhead.

● **False.** Explain why programs with multiple threads might be the same speed or slower.

If code is purely sequential it can not be optimized by more threads.

Q2 *Multi-Select* **(12 points)**

For this section, **incorrect bubbles** will result in **negative points** (e.g., marking **true** when the correct answer is **false**). Leaving an option blank will **be worth 0 points**. The expected value of guessing is **0.**

For each line, you may only select either true or false; you may not select both.

Q2.1 (2 points) Select all the roles of the Operating System

T	F	
■	☐	Giving the illusion of infinite memory for each process.
■	☐	Provide a common set of services.
■	☐	Manage fair access to system resources
■	☐	Managing communication between hardware and applications.
■	☐	Preventing processes from accessing each other's memory

Q2.2 (2 points) Select all options that are true about processes.

T	F	
☐	■	When a process is forked, new **file descriptions** for each open file are allocated for the child process.
☐	■	A process can override the signal handler corresponding to `SIGSTOP` by using `sigaction`.
■	☐	Base and Bound (B&B) provides protection by preventing one process from accessing or modifying the address space owned by another process.
☐	■	If a process calls `fork` and then calls `exit`, its child process should immediately exit
☐	■	Process control blocks (PCBs) are allocated on the kernel stack.

Q2.3 (2 points) Select all that is true about atomics

T	F	
☐	■	Atomic operations **can not** be performed in user mode for security concerns
■	☐	Semaphores can be used to enforce mutual exclusion
■	☐	On multiprocessor systems, it is better to implement locks using atomic instructions instead of disabling and re-enabling interrupts
☐	■	User-level code can implement a lock by disabling and re-enabling interrupts
■	☐	User-level code can implement a lock by using atomic instructions

Q2.4 (2 points) Select all that is true about a variety of topics

T	F	
☐	■	If a process opens a file `fd`, calls `fork()`, and then closes `fd`, the child's file corresponding to `fd` will be closed
■	☐	If one thread closes a **file descriptor**, then it will be closed for all threads in the same process
■	☐	In PintOS, after a call to `fork()`, you need to `close()` in both the parent and child process to close the **file description**
☐	■	In `x86`, the stack must be aligned upon entry to the callee function
■	☐	`fork()` must copy over the stack pointer to the new process

Q2.5 (4 points)

```
1  void* helper(void *_) {
2    // 0644 = owner can read/write
3    int f = open("out.txt",
4                 O_WRONLY|O_APPEND|O_CREAT, 0644);
5    write(f, "ROHAN", 1);
6    close(f);
7    return NULL;
8  }
9
10 int main() {
11   int f = open("out.txt",
12                O_WRONLY|O_APPEND|O_CREAT, 0644);
13   write(f, "ION", 2);
14   pthread_t t;
15   pthread_create(&t, 0, helper, 0);
16   sched_yield();
17
18   write(f, "MATEI", 5);
19   lseek(f, 0, SEEK_SET);
20   write(f, "DATA", 1);
21   close(f);
22 }
```

Assume all calls succeed and calls to `write` writes the maximum number of bytes possible.

Select all possible contents of `out.txt` with the choices below:

T	F	
■	☐	`DOMATEI`
☐	■	`DORMATEI`
☐	■	`IOMATEIDR`
■	☐	`DOMATEIR`
☐	■	`DIONMATEI`

Q3 *Short Answers* **(16 points)**

Q3.1 (2 points) Explain how you would implement `pthread_join()` via semaphores

Semaphore in a shared data between threads. In join just down the semaphore and in thread exit up the semaphore.

Q3.2 (2 points) On a `fork()` system call, what changes (if any) are made to the open **file description**?

Ref count goes up.

Q3.3 (4 points) Give 1 benefit and 1 drawback for both Mesa and Hoare semantics.

Mesa semantics:

Easy to implement since we just add a thread to the ready queue. Does not provide any gauntrees about when that thread will be scheduled.

Hoare Semantics:

Harder to implement since we need to send make the thread own the lock and transfer control to it immediately. Easier to reason about since the thread that gets signaled gets control.

Q3.4 (2 points) On Unix, how many new processes are created by the following code?

```
1 int main(void) {
2   execv('./new_program');
3   execv('./new_program');
4 }
```

Number of processes

0

Explain:

On Unix, exec does not create a new process, it just replaces the current process.

For the following code, assume that all calls to `pthread_create` succeed. (Questions are on the next page)

```
1  typedef struct {
2      int counter;
3      int limit;
4  } shared_t;
5
6  void* helper(void *arg) {
7      shared_t *shared = (shared_t*) arg;
8
9      while (1) {
10         int val = shared->counter;
11         if (val >= shared->limit) {
12             break;
13         }
14         shared->counter += 1;
15         printf("incremented!");
16     }
17     return NULL;
18 }
19
20 int main(void) {
21     pthread_t t1, t2;
22     shared_t shared = {.counter = 0, .limit = 5};
23
24     pthread_create(&t1, NULL, helper, &shared);
25     pthread_create(&t2, NULL, helper, &shared);
26
27     pthread_join(t1, NULL);
28     pthread_join(t2, NULL);
29
30     printf("Final counter value: %d\n", shared.counter);
31     return 0;
32 }
```

Q3.5 (3 points) What is the maximum number of times "`incremented!`" can be printed? Explain how you reached your answer.

20

Q3.6 (3 points) What is the maximum value of `counter`? Explain how you reached your answer.

6

Reason: Both threads can read val=4 and both increment shared-counter++.

Q4 *You Shall Not Pass* **(16 points)**

A common synchronization primitive used is a barrier. A barrier will block all threads from passing it until every thread has gotten to the location. Think about a barrier placed before F1 drivers to ensure they all reach the starting line before the race begins.

We want to implement a barrier which **does not busy wait** and **is able to be used multiple times**.

A barrier can be described by the following pseudo code:

```
structure Barrier {
  count        // number of threads that must arrive
  waiting      // counter of how many threads have arrived
}

procedure barrier_init(B, num_threads):
  B.count = num_threads
  B.waiting = 0

procedure barrier_wait(B):
  B.waiting += 1
  if B.waiting == B.count:          // last thread arrives
    B.waiting = 0                   // reset for next use
    wake_waiting_threads()          // wake all other threads
  else:
    wait_for_all_threads()          // wait until everyone gets here
```

For reference, here is the C code for compare and swap

```
1 bool CAS(int *mem, int cmp, int swp) {
2   if (*mem == cmp) {
3     *mem = swp;
4     return true;
5   } else {
6     return false;
7   }
8 }
```

Also, recall `futex` is a syscall that allows a thread to put itself to sleep on a queue associated with a user-level address. It also allows a thread to ask the kernel to wake up threads that might be sleeping on the same queue. Here is also the function signature for `futex`:

```
int futex(int *uaddr, int futex_op, int val);
```

If `futex_op == FUTEX_WAIT`, the kernel checks atomically as follows:

1. If `(*uaddr == val)`: The calling thread will be put to sleep.
2. If `(*uaddr != val)`: The calling thread will keep running (i.e. `futex` returns immediately)

If `futex_op == FUTEX_WAKE`, this function will wake up to `val` waiting threads.

Q4.1 (12 points) Implement the code below:

```
 1 struct barrier {
 2   int threads;
 3   int waiting;
 4   bool direction;
 5 }
 6
 7 void barrier_init(struct barrier *b, int threads) {
 8   b->threads = threads;
 9   b->waiting = 0;
10   b->direction = 0;
11 }
12
13 void barrier_wait(struct barrier *b) {
14   int wanted_dir = !b->direction;
15   int w;
16
17   do {
18     w = b->waiting;
19   } while (!CAS(&b->waiting, w, w + 1));
20
21   if (w + 1 == b->threads) {
22     b->waiting = 0;
23     b->direction = wanted_dir;
24     futex(&b->state, FUTEX_WAKE, b->threads - 1);
25   } else {
26     while (wanted_dir != b->direction)
27       futex(&b->state, FUTEX_WAIT, !wanted_dir);
28   }
29 }
```

Q4.2 (4 points) If the code on lines 22 and 23 were swapped, would there be any issues? (Assume the rest of the code is filled in correctly).

For reference, lines 22-23 would look like this after being swapped:

```
1 // rest of the code
2 b->direction = wanted_dir;
3 b->waiting = 0;
4 // rest of the code
```

● **True.** Give an example scenario where an issue arises with lines swapped.

○ **False.** Explain why the barrier will work without issues if the lines were swapped.

If direction switches then there could be a thread that passed the atomic add but doesn't end up calling futex wait and then calls barrier_wait again before the thread responsible for resetting waiting sets it back to 0.

This page left intentionally (mostly) blank

The exam continues on the next page.

Q5 *Luca's Pool Party* **(16 points)**

In modern multi-core systems, parallelism is essential to improve performance. However, creating and destroying threads on demand is expensive: each thread requires system resources (stack space, OS bookkeeping, etc.), and scheduling decisions about what work is prioritized first is out of the programmer's control.

A thread pool addresses this problem by creating a fixed number of worker threads only once and reusing them over a program's lifetime. Once created, tasks are submitted to the pool and the thread pool's workers will continuously fetch and execute tasks.

Here we will represent the list of tasks as a queue in which any thread can submit work to and the worker threads will take a task from and then execute the task.

For this question, assume tasks have no return values and assume all tasks succeed without crashing.

Here are the structures we have access to for this class. **Make sure to read and understand all of the code before starting this problem. Please read through all parts of the question before you start coding.**

```
1  struct task {
2    struct list_elem elem;
3    void (*fn)(void *);
4    void *arg;
5  };
6
7  struct pool {
8    int num_threads;
9    struct list tasks;
10   struct lock lock;
11   struct condition cond;
12 };
```

```
1  // global pool
2  struct pool global_pool;
3
4  // workers should execute
5  // this
6  static void worker();
7  // use this to add a task to
8  // the pool
9  void pool_submit(struct
   *task);
```

```
1  struct task* get_task() {
2    // return NULL if list is empty
3    // pop and return the start of the list otherwise
4    if (list_empty(&global_pool.tasks))
5        return NULL;
6    struct list_elem *e = list_pop_front(&global_pool.tasks);
7    struct task *t = list_entry(e, struct task, elem);
8    return t;
9  }
```

Q5.1 (2 points) First, we need to initialize the thread pool and create all of the workers.

```
 1 void pool_init(int num_threads) {
 2   // initialize the global thread pool
 3   // assume this is only called once
 4   global_pool.num_threads = num_threads;
 5   list_init(&global_pool.tasks);
 6   lock_init(&global_pool.lock);
 7   cond_init(&global_pool.cond);
 8
 9   // assume name will never need more than 16 bytes
10   char *name = malloc(16 * sizeof(char));
11
12   for (int i = 0; i < num_threads; i++) {
13     // give it the name "worker i"
14     sprintf(name, "worker %d", i);
15     // thread_create from PintOS, not pthreads
16     thread_create(
17       name, PRI_DEFAULT,
18       worker,
19       NULL
20     );
21   }
```

Q5.2 (8 points) Fill in the blanks. **Not all lines will be used**. Implement a solution without busy waiting. **You are allowed to use `if` statements and loops**.

Different staff solutions uses 4-7 lines, feel free to use more or less.

```
1  void worker() {
2    // executed by every worker in the thread pool
3    while (1) {
4      struct task *t = NULL;
5      lock_acquire(&global_pool.lock);
6      t = get_task();
7      while (t == NULL) {
8        cond_wait(&global_pool.cond, &global_pool.lock);
9        t = get_task();
10     }
11     lock_release(&global_pool.lock);
12     // this line is not used
13     // this line is not used
14     // this line is not used
15
16     t->fn(t->arg);  // execute the task
17     free(t);        // we no longer need it after this
18   }
19 }
```

Q5.3 (3 points) Fill in the blanks for submitting tasks to the queue.

```
1 void pool_submit(struct task *t) {
2   lock_acquire(&pool.lock);
3   list_push_back(&pool.tasks, &t->elem);
4   cond_signal(&pool.cond, &pool.lock);
5   lock_release(&pool.lock);
6 }
```

Q5.4 (3 points) Assume the previous 2 parts have been implemented correctly. If tasks are very short and submitted often, what problems will `pool_submit` and `worker` have?

Very high contention on the list of tasks.

Q6 *(Most) Important Question* **(1 points)**

Questions about your Professors (any non-blank answer will get credit)

Q6.1 (0.5 points) What schools has Matei taught at (Ranked in decreasing order of excellence)?

1. Berkeley, 2. MIT, 3. TreeFurd

Q6.2 (0.5 points) What universities has Ion been a student at?

CMU, Politehnica Bucharest