CS 162 Summer 2021 Midterm 1 Solutions

Midterm Exam

INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn't work, by emailing course staff with your solutions before the exam deadline.

This exam is intended for the student with email address `<EMAILADDRESS>`. If this is not your email address, notify course staff immediately, as each exam is different. Do not distribute this exam PDF even after the exam ends, as some students may be taking the exam in a different time zone.

For questions with **circular bubbles**, you should select exactly *one* choice.

- ◯ You must choose either this option
- ◯ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

- ☐ You could select this choice.
- ☐ You could select this one too!

You may start your exam now. Your exam is due at <DEADLINE> Pacific Time. Go to the next page to begin.

This is a **proctored, closed-book exam**. During the exam, you may **not** communicate with other people regarding the exam questions or answers in any capacity. If there is something in a question that you believe is open to interpretation, please use the "Clarifications" button to request a clarification. We will issue an announcement if we believe your question merits one.

Make sure you read through the exam completely before starting to work on the exam.

We will overlook minor syntax errors in grading coding questions. You do not have to add necessary `#include` statements.

(a) Name

(b) Student ID

(c) Please read the following honor code: "I understand that this is a closed book exam. I hereby promise that the answers that I give on the following exam are exclusively my own. I understand that I am allowed to use unlimited **handwritten** cheat-sheets of my own making, but otherwise promise not to consult other people, physical resources (e.g. textbooks), or internet sources in constructing my answers." **Type your full name below to acknowledge that you've read and agreed to this statement.**

1. True/False

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

(a) (3.0 points)

i. In Base and Bound address translation, a user program can directly modify the "Base" and "Bound" registers.

○ True

● False

ii. Explain your answer.

False. Base and Bound registers can only be modified by the kernel to ensure isolation and protection. If a user program could modify these registers, it would be able to access any physical address.

(b) (3.0 points)

i. Two threads in the same process share a stack.

- ○ True
- ● False

ii. Explain your answer.

False. Each thread of execution requires its own stack, even if two threads are in the same process.

(c) (3.0 points)

i. In general, IPC using shared memory is faster than IPC using pipes.

- (●) True
- ( ) False

ii. Explain your answer.

True. IPC using shared memory is faster because each read/write only requires a single memory operation, whereas reads/writes to pipes require switching to kernel mode.

(d) (3.0 points)

i. In the client-server model, every client writes requests to the same connection socket.

◯ True

● False

ii. Explain your answer.

False. Each client connects to the same server socket, but reads/writes to a separate connection socket after the connection is accepted.

(e) (3.0 points)

i. Assume that `fd` is a file descriptor for an empty file that was opened with write permissions. If I call `write(fd, "Hello World", 11)`, it is guaranteed that the file was modified by the time `write` returns.

○ True

● False

ii. Explain your answer.

False. Answer A: `write` is not guaranteed to write the requested number of bytes, so 0 bytes may have been written to the file. Answer B: `write` is buffered in the kernel. When `write` returns, the data may still be buffered in the kernel and the file may be unmodified.

(f) (3.0 points)

i. In the context of the Banker's Algorithm, an unsafe state implies that a system is in deadlock.

- ○ True
- ● False

ii. Explain your answer.

False. In Banker's Algorithm, an unsafe state is one in which resources can be requested in an order that causes deadlock.

(g) (3.0 points)

i. In Pintos, a kernel-level stack can grow as large as it needs to in order to perform its functions.

◯ True

● False

ii. Explain your answer.

False. Stacks in the Pintos kernel must fit entirely into a 4K page. In fact, they share the 4K page with the corresponding Thread Control Block (TCB).

2. Multiple Choice

In the following multiple choice questions, **please select all options that apply**. Answering a question instead of leaving it blank will **NOT** lower your score (the minimum score for a single question is 0, not negative).

(a) (3.0 pt) What will this program print (select all possible output sequences)? You can assume that all syscalls do not error.

```
void *run(void *arg) {
  sched_yield();
  printf("I am a different thread! ");
  return NULL;
}
int main() {
  pthread_t thread;
  pthread_create(&thread, NULL, run, NULL);
  printf("I am the main thread! ");
  pthread_join(thread, NULL);
  printf("Oh no I'm about to get terminated. ");
}
```

- [x] I am the main thread! I am a different thread! Oh no I'm about to get terminated.
- [x] I am a different thread! I am the main thread! Oh no I'm about to get terminated.
- [ ] I am the main thread! Oh no I'm about to get terminated.
- [ ] I am the main thread! Oh no I'm about to get terminated. I am a different thread!
- [ ] None of the Above

Even though `sched_yield` yields to the scheduler, it does not guarantee that the current thread will not be immediately rescheduled. As a result, "I am the main thread!" and "I am a different thread!" can be printed in any order. `pthread_join` ensures that "Oh no I'm about to get terminated." is printed after the other statements (because the child thread must exit before `pthread_join` returns).

(b) (3.0 pt) Which of the following scheduling algorithms can lead to some form of starvation (select all that apply)?

- [x] FIFO
- [ ] Round Robin
- [x] SRTF
- [x] MLFQS (w/ Fixed Priority Scheduling)
- [ ] None of the Above

In FIFO, one thread can starve all other threads by running in an infinite loop. In SRTF and MLFQS w/ Fixed Priority Scheduling, short jobs can starve long jobs. In Round Robin, every thread is guaranteed an equal quantum.

(c) (3.0 pt) Assume that the OS is using a Strict Priority Scheduler. Thread A (priority 30) is currently holding Lock 1, and Thread B (priority 60) is waiting to acquire Lock 1. If a third thread (Thread C) is introduced to the system, which of the following priorities for Thread C will lead to priority inversion (select all that apply)?

- ☐ 0
- ☐ 20
- ■ 40
- ■ 50
- ☐ 70
- ☐ 80
- ☐ None of the Above

If Thread C has a priority larger than Thread A, then it will be scheduled before Thread A. This will prevent Thread A from releasing Lock 1, and Thread B will continue to be blocked. If Thread C's priority is greater than Thread B's, then this is the correct behavior (i.e. Thread C should be scheduled before Thread B). However, if Thread C's priority is less than Thread B's, then this is priority inversion because Thread C's execution is being prioritized over Thread B's.

(d) (3.0 pt) Assume that our OS uses Banker's Algorithm to allocate resources in order to avoid deadlock. Suppose our system contains resources A, B, and C. In addition, there are threads 1, 2, and 3. The total amount of resources, the current allocation of resources, and the maximum amount of resources needed for each thread to complete are as follows:

Maximum Resources Needed to Complete

	Resource A	Resource B	Resource C
Thread 1	5	3	3
Thread 2	4	9	6
Thread 3	7	3	7

Current Resources Allocated

	Resource A	Resource B	Resource C
Thread 1	1	1	1
Thread 2	1	1	1
Thread 3	1	1	1

Total Resources

Resource A	Resource B	Resource C
8	9	10

At this point, Thread 1 requests all the remaining resources it needs, and Banker's Algorithm needs to determine if this request should be granted.

However, the Banker's Algorithm implementation is buggy. Recall that Banker's algorithm runs a simulation of requests. Unfortunately, everytime the algorithm simulates Thread 3 requesting its remaining resources, it assumes that Thread 3 will fail to release `X` units of Resource C after completing.

For what value of `X` will Thread 1's original request not be granted (select all that apply)?

- ☐ 0
- ☐ 1
- ☐ 2
- ☐ 3
- ☐ 4
- ■ 5
- ■ 6
- ■ 7
- ☐ None of the Above

We start by calculating the unallocated resources that are currently available by subtracting current resources allocated from total resources. This gives us 5A/6B/7C. Since Thread 1 has requested all its remaining resources, we assume that the request is granted and Thread 1 completes, giving 1A/1B/1C back to the system. This gives us 6A/7B/8C in unallocated resources. At this point, only Thread 3 can grab its remaining resources, because it needs an additional 6A/2B/6C, whereas Thread 2 needs 3A/8B/5C. When we simulate Thread 3 acquiring its remaining resources and then completing, the remaining available resources will be 7A/8B/(9-X)C. This is because the buggy Banker's Algorithm assumes that Thread 3 will not release X units of Resource C. Since Thread 2 needs 3A/8B/5C to complete, (9-X) must be less than 5 for Banker's Algorithm to deny this resource request. This is true when X=5, 6, or 7.

(e) (3.0 pt) In general, locks implemented using futexes are faster because (select all that apply):

- ■ Locks are not always contended
- ☐ Locks are almost always contended
- ■ Syscalls are slow compared to operations in userspace
- ☐ Userspace memory accesses are faster than kernelspace memory accesses
- ☐ None of the Above

A futex lock remains in userspace (i.e. doesn't need to make a syscall) if the lock is uncontended. If locks are almost always contended, a lock implemented using futexes would need to make roughly the same number of syscalls as a lock implemented without futexes. This affects lock performance because syscalls are slow compared to operations in userspace.

(f) (3.0 pt) Which of the following are differences between kernel mode and user mode (select all that apply)?

- ■ In kernel mode, additional kernel-mode instructions can be executed.
- ■ Control of I/O devices is typically only available in kernel mode.
- ☐ The MMU (Memory Management Unit) is only active in kernel mode.
- ☐ Interrupts are always disabled in kernel mode.
- ☐ None of the Above

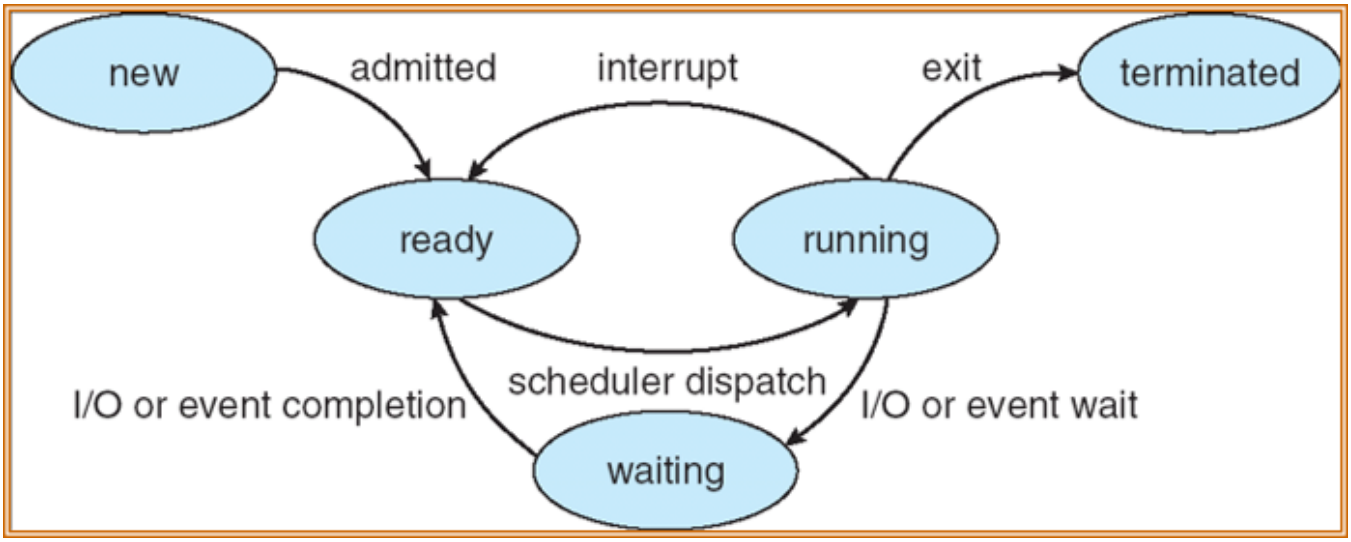
The MMU is always active and used to translate virtual addresses to physical addresses. Interrupts may be selectively disabled in kernel mode.

3. Short Answer

(a) Suppose the same laptop connects to google.com on port 80 twice. How are the two connections to Google disambiguated (i.e. distinguished from each other) by Google's servers?

Each connection is uniquely identified by a 5-tuple: Source IP, Destination IP, Source Port, Destination Port, and Protocol. In this case, the laptop will assign a unique source port to each client, allowing Google's server to distinguish between the two connections.

(b) Explain how and why you would modify the diagram below if you had an Operating System that supported Hoare semantics?



Thread State

We would add an additional arrow directly from "waiting" to "running". This represents the fact that a thread waiting on a condition variable, when signalled, is immediately given the CPU.

(c) Assume that Thread A is the only thread running on my system (i.e. there are no other threads vying for CPU resources). Describe a scenario in which the idle thread would still be scheduled and run by the OS instead of Thread A.

Whenever Thread A enters a "waiting" state (because, for example, it is blocked on I/O), the OS must schedule the idle thread since there are no other threads that are ready to run.

(d) Analyze the following code:

```
sema_t semA;
sema_t semB;

void initialize() {
  // Initialize semaphores to value of 2.
  sema_init(&semA, 0, 2);
  sema_init(&semB, 0, 2);
}

void functionA() {
  sema_down(&semA);
  sema_down(&semB);
  sema_up(&semB);
  sema_up(&semA);
}

void functionB() {
  sema_down(&semB);
  sema_down(&semA);
  sema_up(&semA);
  sema_up(&semB);
}
```

Assume that `initialize()` has already been called. Now let's say that two threads are executing `functionA()`, and two threads are executing `functionB()`. All four threads are running concurrently. Explain how this scenario can lead to deadlock.

If each thread runs the first line of their respective function, this will lead to deadlock. This is because both semaphores will be equal to 0, and all threads will be blocked trying to down a semaphore that was already downed by another thread.

(e) Ron is trying to implement syscalls in PintOS, and he wants to make sure that he safely reads and writes from memory. As a result, he makes sure to *correctly* validate that every argument on the stack is in the user process's virtual address space. In addition, when a buffer pointer is passed in as a syscall argument, he makes sure to check that the first byte and last byte of the buffer are valid. Explain how you can exploit Ron's implementation of buffer pointer validation to cause a kernel crash. Be specific (e.g. explain how you would select a buffer pointer and buffer size for your exploit).

By setting the buffer pointer equal to a valid address in the user's heap, and setting the size of the buffer such that the end of the buffer is a valid location in the user's stack, you can cause the kernel to crash. This is because the kernel will attempt to access addresses in between the heap and the stack, which are not allocated.

(f) Is it possible for an interrupt handler (code triggered by a hardware interrupt) to sleep while waiting for another event? If so, explain how. If not, explain why not.

Answer A: NO. Strictly speaking, an interrupt handler must not sleep while waiting for another event, since it doesn't have a thread-control block (context) to put onto a wait queue and is operating with interrupts disabled. Answer B: YES. an interrupt handler that wants to sleep must allocate a new kernel thread to finish its work, place the thread on a wait queue, then return from the interrupt (reenabling interrupts in the process).

4. Spell Finder

Harry has a big test coming up at LogPorts Castle, and he hasn't read the textbook yet! He needs to be able to perform a specific spell, but he doesn't know where it is in the textbook.

Harry has asked you, a computer magic major at LogPorts, to write a multiprocessing program for him that can rapidly find the spell in the textbook. Your solution must create `num_processes` processes, and each process should search an equal portion of the textbook using the `search_for_spell` function.

You can assume that `num_processes` is a power of 2, and that `textbook_size` is a multiple of `num_processes`.

NOTE: For each blank, you can use as many lines as you need to solve the problem. You may be able to leave some blanks empty. You can assume that all syscalls will not error.

```
/* This function will read the textbook file into array `arr`. You can assume
that the textbook will be exactly `textbook_size` characters long. Assume that
this is already implemented. */
void read_textbook_into_array(char *arr, int textbook_size);

/* This function will search for Harry's spell in array `arr`, and print the
spell if found. The function will only search for the spell in the subarray
between `start_index` (inclusive) and `end_index` (exclusive). Assume that this
is already implemented. */
search_for_spell(char *arr, int start_index, int end_index);

void process_array(int num_processes, int textbook_size) {
  char *arr = malloc(sizeof(char) * textbook_size);
  read_textbook_into_array(arr, textbook_size);

  int start_index = 0;
  int end_index = textbook_size;

  for (__[A]__) {
    pid_t pid = fork();
    if (pid == 0) {
      __[B]__;
    } else {
      __[C]__;
    }
  }

  search_for_spell(arr, start_index, end_index);
}
```

(a) [A]

> **Option A:** `int i = 1; i <= num_processes; i *= 2` **Option B:** `int i = 0; i < num_processes; i++`

(b) [B]

> **Option A:** `end_index = end_index / 2` **Option B:** **“‘ start_index = i * textbook_size / num_processes; start_index = (i+1) * textbook_size / num_processes; break;**

(c) [C]

```
Option A: \texttt{start\_index\ =\ end\_index\ /\ 2} Option B:


start_index = 0;
end_index = 0;
```

(d) Why is this program not memory-efficient with respect to the size of the textbook (assume that the Operating System does not use copy-on-write when `fork`ing)?

The first thing that the function does is allocate space for the textbook in memory and read the entire textbook into the heap. However, each time the process is `fork`ed, because there is no copy-on-write mechanism, the entire address space (including the in-memory copy of the textbook) will be copied into the child process's address space. As a result, there will be `num_processes` copies of the textbook in-memory by the time the program completes.

(e) Neville, a computer magic minor, recommends that you use multithreading instead. Justify his recommendation.

Option A: Multithreading would allow each of the threads to share the same copy in-memory copy of the textbook. Option B: Context switching between threads is faster than context switching between processes, leading to better performance.

5. Video Games

Recall our Crowded Video Games implementation from discussion, which allowed a limited number of players, `max_players`, to connect to a video game server:

```
struct server {
  // ...
  sema_t sem;
};

struct server *initialize_server(int max_players) {
  struct server *s = malloc(sizeof(struct server));
  // ...

  // Initialize semaphore to value of `max_players`.
  sema_init(&s->sem, max_players);
  return s;
}

void play_session(struct server *s) {
  sema_down(&s->sem);
  connect(s);
  play();
  disconnect(s);
  sema_up(&s->sem);
}
```

We now need to add new functionality to our game. Specifically, we want to add two new functions that can be called by the server administrator, `add_server_capacity` and `remove_server_capacity`. `void increase_server_capacity(struct server *s)` will be called when the server is upgraded and the maximum number of connections increases by one. `void decrease_server_capacity(struct server *s)` will be called when the server is overloaded and the maximum number of connections needs to decrease by one.

You can assume that, when `increase_server_capacity` or `decrease_server_capacity` are called, no player is connected to the server (i.e. the server is down for maintenance).

NOTE: For each blank, you can use as many lines as you need to solve the problem. You may be able to leave some blanks empty. When you see `// ...`, assume that there is other server logic there that you don't need to worry about modifying.

```
struct server {
  // ...
  sema_t sem;
  _____[A]_____
};

struct server *initialize_game(int num_servers) {
  struct server *s = malloc(sizeof(struct server));
  // ...
  sema_init(&s->sem, 0, num_servers);
  _____[B]_____
  return s;
}

void play_session(struct server *s) {
  sema_down(&s->sem);
  connect(s);
  play();
```

```
  disconnect(s);
  sema_up(&s->sem);
}

void increase_server_capacity(struct server *s) {
  _____[C]_____
}

void decrease_server_capacity(struct server *s) {
  _____[D]_____
}
```

(a) [A]

```
// NO CHANGE
```

(b) [B]

```
// NO CHANGE
```

(c) [C]

```
sema_up(&s->sem);
```

(d) [D]

```
sema_down(&s->sem);
```

No more questions.