# CS 162 Summer 2022 Solutions

Midterm Exam

**INSTRUCTIONS**

Please do not open this exam until instructed to do so.

Do not discuss exam questions for at least 24 hours after the exam ends, as some students may be taking the exam at a different time.

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!

**Preliminaries**

This is a proctored, closed-book exam. You are allowed 1 page of notes (both sides). You may not use a calculator. You have 110 minutes to complete as much of the exam as possible. Make sure to read all the questions first, as some are substantially more time-consuming.

If there is something about the questions you believe is open to interpretation, please ask us about it.

We will overlook minor syntax errors when grading coding questions. **There is a reference sheet at the end of the exam that you may find helpful.**

**(a)** Name

**(b)** Student ID

**(c)** Please read the following honor code: "I understand this is a closed-book exam. I promise the answers I give on this exam are my own. I understand that I am allowed to use one 8.5x11, double-sided, handwritten cheat-sheet of my own making, but otherwise promise not to consult other people, physical resources (e.g. textbooks), or the internet in constructing my answers."

**Write your full name below to acknowledge that you've read and agreed to this statement.**

1. **(12 points) True/False**

Please explain your answer in **TWO SENTENCES OR LESS**. Longer explanations may get no credit. Answers without an explanation **GET NO CREDIT**.

**(a) (2 points)**

**i.** In Pintos the virtual address `0xc0000008` refers to the same physical address no matter which process we are in.

[✓ True.]

◯ False.

**ii.** Explain.

**Addresses above `PHYS_BASE` are mapped to the same location in phsyical memory.**

**(b) (2 points)**

**i.** After a multi-threaded process returns from `main`, the process will continue running until all threads call `pthread_exit` (assuming no errors).

◯ True.

[✓ False.]

**ii.** Explain.

**Returning from `main` results in the `exit` syscall being called. `exit` shuts down the entire process immediately, without waiting for threads to finish.**

Aside: When a thread returns from the function it began running due to `pthread_create`, `pthread_exit` is implicitly called. Regardless, the `exit` syscall does not wait for all threads to gracefully exit.

**(c) (2 points)**

**i.** A thread can modify the stack variables of another thread in the same process.

[✓ True.]

False.

**ii.** Explain.

**Threads in the same process share the same address space.**

**(d) (2 points)**

**i.** It is possible to create a lock without trapping into the kernel.

[✓ True.]

False.

**ii.** Explain.

**Locks can be implemented using atomic operations, such as `test&set` or `compare&swap`. These operations generally do not require trapping into the kernel.**

**(e) (2 points)**

**i.** `fork()` is slow in Linux because it has to copy the entire address space of a process.

○ True.

[✓ False.]

**ii.** Explain.

**`fork` can use copy-on-write, so the operating system does not need to copy the entire address space.**

**(f) (2 points)**

**i.** To maximize throughput, one should use FCFS instead of SRTF.

[✓ True.]

○ False.

**ii.** Explain.

**FCFS results in fewer context switches, so the total time to complete all jobs will be lower. SRTF has a better response time, but a worse throughput due to context switching when new jobs arrive.**

Clarification: throughput refers to CPU throughput in this question.

**2. (18 points) Multiple Choice**

**Choose ALL that apply.**

**(a) (2 pt)** Which of the following x86 instructions can you **NOT** execute in user mode?

- ☐ `pushl`
- ☐ `addl $16, %esp`
- ■ `iret`
- ☐ `int $0x30`
- ☐ None of the above.

You can only return from interrupt if you are in kernel mode. The `int $0x30` instruction can be performed from user mode – it is how PintOS user programs issue syscalls.

**(b) (2 pt)** Which of the following are part of the kernel?

- ☐ C standard library
- ■ Device drivers
- ■ Scheduler
- ☐ Shell
- ☐ None of the above.

The shell is a user program; the C standard library is linked to user programs. Device drivers and the scheduler are typically part of the kernel.

**(c) (2 pt)** How does the Pintos kernel find the `struct thread` of the currently running kernel thread?

- ☐ The address is stored in an `intr_frame`.
- ☐ The kernel scans a global array of `struct threads`.
- ☐ The address is stored in the upper 12 bytes of `%eax`.
- ■ The kernel rounds down `%esp` to the nearest page.
- ☐ None of the above.

Each `struct thread` is at the bottom of the page that its corresponding kernel stack is on.

**(d) (2 pt)** Which of the following will ALWAYS lead to a transfer from user mode to kernel mode?

- ■ A process opens a file.
- ☐ A process adds two large numbers, leading to integer overflow.
- ■ A process tries to dereference a `NULL` pointer.
- ☐ A process reads a file with `fread`.
- ☐ None of the above.

A: The OS must check permission when opening a file, so the kernel must be involved. B: Integer overflow need not trap to the kernel. C: Dereferencing a `NULL` pointer will result in a memory access violation, which typically results in the kernel shutting down the process. D: `fread` doesn't always go to the kernel. See https://stackoverflow.com/a/9587245.

**(e) (2 pt)** Which of the following are TRUE about syscalls?

- ☐ Forked processes share a single address space.
- ☐ Pipes are two-way communication channels between processes on the same physical machine.
- ■ Open file descriptors are preserved across the (Linux) `exec` syscall.
- ☐ The `dup2` syscall allows a file descriptor to point to multiple entries in the kernel file table.
- ☐ None of the above.

A: Processes do not share an address space. B: Pipes are one-way; there is a read-end and a write-end. C is true: That's why we call `dup2` before `fork` in the shell homework. D: A file descriptor only ever points to 1 entry in the kernel file table. It is possible, however, to have multiple file descriptors refer to a single file.

**(f) (2 pt)** What are the disadvantages of disabling interrupts to serialize access to a critical section?

- ■ This technique does not work with a multiprocessor.
- ☐ Disabling interrupts will turn off the hardware timer.
- ■ The computer can miss critical events when interrupts are disabled.
- ■ User code cannot disable interrupts.
- ☐ None of the above.

A: False, disabling interrupts does not help prevent multiple processors from accessing the same critical section. B: False, the hardware timer usually generates non-maskable interrupts, which are not ignored even if other interrupts are disabled. C: True, important interrupts may be ignored if interrupts are disabled. D: True, it would be dangerous if user programs had the ability to disable interrupts.

**(g) (2 pt)** Which of the following MUST occur during a context switch between two threads of different processes?

- ☐ Computational registers are saved in user memory.
- ☐ The user stack is saved to the kernel stack.
- ☐ The file descriptor table is saved to the kernel heap.
- ☐ Buffered data in memory is flushed to disk.
- ■ None of the above.

A: Registers are saved in **kernel** memory. B: The page table base register would be set to point to the second process's page tables, not a default value. C: The file descriptor table is already in memory; nothing extra needs to be saved. D: Buffered data is not necessarily flushed to disk during a context switch.

**(h) (2 pt)** Which of the following are a result of the "everything is a file" paradigm?

- ☐ All kernel data structures are stored as files.
- ■ User processes interact with I/O devices as if they are files.
- ☐ Everything stored on disk is part of a file.
- ■ One can read and write to a pipe using file syscalls.
- ☐ None of the above.

A: Kernel data structures may be stored in memory. B: True; you can use `read` and `write` to read from disk/keyboard/mouse. C: Not everything on disk is a file. The boot loader and free map are examples in PintOS. D: True; you operate on pipes using `read` and `write`.

**(i) (2 pt)** Which of the following scheduling algorithms may cause starvation?

- ☐ Linux CFS
- [✓ FCFS]
- ☐ Round robin
- [✓ MLFQS]
- ☐ None of the above.

A: CFS always runs the thread that has run for the least amount of time, guaranteeing that all threads run for approximately equal amounts of time. B: FCFS can cause starvation since it does not pre-empt threads that run for an indefinite amount of time. C: Round robin scheduling guarantees that a new thread runs in each quanta. D: MLFQS can cause starvation for threads in lower priority queues since threads in higher priority queues can remain there for indefinite amounts of time.

**3. (21 points) Short Answer**

Please answer in **THREE SENTENCES OR LESS**. Longer explanations may get no credit.

**(a) (3 pt)** In Pintos, what does `_start` do besides setting up arguments and calling `main`?

**`_start` ensures that `exit` is called with the return code from `main`.**

**(b) (3 pt)** Describe one important difference between `read` and `fread`. How are they related?

**Important differences (only one needed): `read` is a syscall and `fread` is a high-level C library function. `fread` maintains a user-level buffer, while `read` does not. `read` may not read the desired number of bytes while `fread` does (assuming EOF is not reached).**
**How they are related: `read` is called by `fread`.**

**(c) (3 pt)** Explain the steps needed to establish a pipe between a parent and a child process.

**Call the `pipe` syscall, then call the `fork` syscall. Finally, close one end of the pipe in the parent process and close the other end in the child process.**

**(d) (3 pt)** Describe a key difference between Hoare and Mesa semantics.

**With Hoare semantics, the signaling thread immediately transfers the lock and control of the CPU to the waiting thread. With Mesa semantics, the waiting thread will be woken up after being signaled, but may only be scheduled to run at a later time.**

**(e) (3 pt)** What role does the Interrupt Vector Table play in protecting the kernel?

**The interrupt vector table (IVT) ensures that we only start executing kernel code at predefined entry points.**

**(f) (3 pt)** Why might the lottery scheduler be a bad choice for real-time scheduling?

**Real-time schedulers must be predictable; lottery scheduling is not predictable. With lottery scheduling, we cannot guarantee that a task will be run by its deadline.**

**(g)** **(3 pt)** Rahul plays StarCraft (a computer game). He discovers that inserting `printf` statements into StarCraft code makes StarCraft run faster on his computer. Explain why this happens.

**Rahul's computer may use a MLFQS. Inserting print statements causes StarCraft to give up the CPU earlier (due to I/O), which will cause StarCraft to run in a higher priority queue.**

**4. (18 points) Parallel File Copy**

*For online exam takers: you may find it useful to read this problem in a separate PDF instead of scrolling back and forth.*

You are making a backup copy of your entire hard drive, but you find that copying a single file at a time takes too long. You decide to put your CS 162 knowledge to good use, and copy multiple files in parallel.

Your program takes in a non-negative integer `n` and two lists of strings, both of length `n`. The first list, `in_files`, contains a list of source files; the second list, `out_files`, contains the corresponding destination file names. Write a program to copy each source file to the corresponding destination. You must create a separate **process** to copy each file. The files should all be copied concurrently.

The main (parent) process must wait for all the files to be copied, and all child processes to exit, before exiting.

Fill in the blanks below to implement your parallel file copy program. You must use the **low-level I/O API** shown in the reference sheet. You may assume calls to `read` and `write` never error. Do **NOT** assume they read/write the full amount.

**NOTE: You should only use AT MOST one line per blank. Extra lines will not be graded. Your solution does not need to use all the blanks. You may use loops, conditional statements, and other control structures in your solution as you see fit.**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>

#define BUFSIZE 2048

void copy_file(char* src, char* dst) {
  char buf[BUFSIZE];

  int srcfd = open(src, O_RDONLY);
  int dstfd = open(dst, O_WRONLY | O_CREAT | O_TRUNC, 0644);

  int chunk;
  int total;

  while (______________[A]_______________) {
    ______________[B]_______________
    ______________[B]_______________
    ______________[B]_______________
    ______________[B]_______________
    ______________[B]_______________
    ______________[B]_______________
  }

  close(srcfd);
  close(dstfd);
}

void copy(int n, char** in_files, char** out_files) {
  pid_t children[n];
  int i;

  for (i = 0; i < n; i++) {
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
  }

  ______________[D]_______________
  ______________[D]_______________
  ______________[D]_______________
  ______________[D]_______________
  ______________[D]_______________
}

int main(int argc, char** argv) {
  // Example usage:
  int n = 2;
  char* in_files[] = { "test1.txt", "test2.txt" };
  char* out_files[] = { "copy1.txt", "copy2.txt" };
  // test1.txt should be copied to copy1.txt
  // test2.txt should be copied to copy2.txt
  copy(n, in_files, out_files);
  return 0;
}
```

**(a)** **(2 pt)** [A] (max 1 line)

```
(chunk = read(srcfd, buf, BUFSIZE)) > 0
```

Other solutions are possible.

**(b)** **(6 pt)** [B] (max 6 lines)

```
total = 0;
while (total < chunk) {
  total += write(dstfd, &buf[total], chunk - total);
}
```

**(c) (7 pt)** [C] (max 11 lines)

```
pid_t child = fork();
if (child > 0) {
  children[i] = child;
} else if (child == 0) {
  copy_file(infiles[i], outfiles[i]);
  exit(0);
} else {
  perror("fork failed");
}
```

**(d) (3 pt)** [D] (max 5 lines)

```
for (i = 0; i < n; i++) {
  waitpid(children[i], NULL, 0);
}
```

**5. (25 points) Channels**

*For online exam takers: you may find it useful to read this problem in a separate PDF instead of scrolling back and forth.*

We've seen how locks can be used to synchronize access to shared data. Channels are an alternative model to synchronization. When using channels, one or more threads produce data that is sent through the channel. Other threads then read the data that was sent to the channel.

A buffered channel is one that maintains an internal buffer of a given size. As long as the buffer has space, calls to send should complete quickly (they should just put the new value in the buffer, possibly acquiring some locks if necessary). If the buffer is full, send should wait until space becomes available, unless the caller indicates they do not want to wait (in which case the send operation should do nothing). The `recv` method reads values from the channel in the order in which they were put in the buffer.

In this problem, you will implement a buffered channel that transports integers between multiple threads in the same process. This is the interface you must implement:

```
#include <stdlib.h>
#include <stdbool.h>
#include <pthread.h>

typedef struct chan {
  /* TODO: add fields here. */
} chan_t;

// Initializes a channel with buffer size `size`.
void chan_init(chan_t* chan, size_t size);

// Sends a value through the channel.
// If wait is true, wait for space in the channel if necessary.
// If wait is false, no waiting is permitted. It is OK to acquire
// locks, but you must not wait for some other thread to call send/recv.
// Returns true if the send was successful; false otherwise.
bool send(chan_t* chan, int value, bool wait);

// Receive a value from the channel, storing the result in *dst.
// If wait is true, wait for a value to become available.
// If wait is false, no waiting is permitted. It is OK to acquire
// locks, but you must not wait for some other thread to call send/recv.
// Returns true if the receive was successful; false otherwise.
bool recv(chan_t* chan, int* dst, bool wait);

// Frees all memory allocated by chan_init.
// You can assume that this will only be called by a single thread,
// and that no threads will be waiting due to calls to send/recv.
void chan_free(chan_t* chan);
```

Please be sure to read the descriptions of each method, as they contain important details.

Your implementation should be thread-safe. That is, it should behave correctly regardless of the number of threads concurrently calling `send` and `recv`. You can assume that `chan_init` is called and completes before any concurrent use of the channel.

To make your implementation simpler, you may assume you have access to a circular buffer that stores integers. The interface presented by the buffer is shown below:

```
// A circular buffer of fixed size.
//
// The provided methods are NOT thread-safe.
```

```
typedef struct circ_buf {
  /* FIELDS OMITTED */
} circ_buf_t;


// Initializes a circular buffer of the given size.
void circ_buf_init(circ_buf_t* buf, size_t size);

// Puts a value into the buffer.
// Panics if the buffer is already full.
void circ_buf_put(circ_buf_t* buf, int value);

// Gets and removes a value from the buffer.
// Panics if the buffer is already empty.
int circ_buf_get(circ_buf_t* buf);

// Returns the number of entries in the buffer.
size_t circ_buf_len(circ_buf_t* buf);

// Returns the capacity of the buffer.
// Will always equal the size used to initialize the buffer.
size_t circ_buf_capacity(circ_buf_t* buf);

// Frees all resources allocated by `circ_buf_init`.
// You must not use the buffer after calling this method.
void circ_buf_free(circ_buf_t* buf);
```

The buffer implementation is NOT thread-safe; it is your job to synchronize access to the buffer.

Using the provided buffer API, as well as `pthread` locks, semaphores, and/or condition variables, implement the required channel methods by filling in the blanks below. You may find the reference sheet useful.

**NOTE: You should only use AT MOST one line per blank. Extra lines will not be graded. Your solution does not need to use all the blanks. You may use loops, conditional statements, and other control structures in your solution as you see fit.**

```
#include "channel.h"
#include <stdio.h>

typedef struct chan {
    circ_buf_t buf;
    ______________[A]_______________
    ______________[A]_______________
    ______________[A]_______________
    ______________[A]_______________
    ______________[A]_______________
    ______________[A]_______________
    ______________[A]_______________
    ______________[A]_______________
    ______________[A]_______________
    ______________[A]_______________
} chan_t;

void chan_init(chan_t* chan, size_t size) {
    ______________[B]_______________
    ______________[B]_______________
    ______________[B]_______________
    ______________[B]_______________
    ______________[B]_______________
    ______________[B]_______________
    ______________[B]_______________
    ______________[B]_______________
    ______________[B]_______________
    ______________[B]_______________
}

bool send(chan_t* chan, int value, bool wait) {
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
    ______________[C]_______________
}

bool recv(chan_t* chan, int *dst, bool wait) {
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
    ______________[D]_______________
}
```

```
void chan_free(chan_t* chan) {
    ______________[E]______________
    ______________[E]______________
    ______________[E]______________
    ______________[E]______________
    ______________[E]______________
}
```

**(a)** **(5 pt)** [A] (max **10** lines)

```
pthread_mutex_t lock;
pthread_cond_t cond_recv;
pthread_cond_t cond_send;
```

**(b)** **(3 pt)** [B] (max **10** lines)

```
circ_buf_init(&chan->buf, size);
pthread_mutex_init(&chan->lock, NULL);
pthread_cond_init(&chan->cond_recv, NULL);
pthread_cond_init(&chan->cond_send, NULL);
```

**(c)** **(8 pt)** [C] (max **20** lines)

```
pthread_mutex_lock(&chan->lock);
while (circ_buf_len(&chan->buf) == circ_buf_capacity(&chan->buf)) {
  if (wait) {
    pthread_cond_wait(&chan->cond_send, &chan->lock);
  } else {
    pthread_mutex_unlock(&chan->lock);
    return false;
  }
}

circ_buf_put(&chan->buf, value);

pthread_cond_signal(&chan->cond_recv);
pthread_mutex_unlock(&chan->lock);
return true;
```

**(d) (8 pt)** [D] (max **20** lines)

```
pthread_mutex_lock(&chan->lock);
while (circ_buf_len(&chan->buf) == 0) {
  if (wait) {
    pthread_cond_wait(&chan->cond_recv, &chan->lock);
  } else {
    pthread_mutex_unlock(&chan->lock);
    return false;
  }
}

*dst = circ_buf_get(&chan->buf);

pthread_cond_signal(&chan->cond_send);
pthread_mutex_unlock(&chan->lock);
return true;
```

**(e) (1 pt)** [E] (max **5** lines)

```
circ_buf_free(&chan->buf);
```

## 6. Reference Sheet

```
/*********************************** Threads ***********************************/
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_post(sem_t *sem); // up
int sem_wait(sem_t *sem); // down
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

/********************************** Processes **********************************/
pid_t fork(void);
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int execv(const char *path, char *const argv[]);

/******************************* High-Level I/O *******************************/
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fd, const char *mode);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int fclose(FILE *stream);

/******************************* Low-Level I/O ********************************/
int open(const char *pathname, int flags);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int pipe(int pipefd[2]);
int close(int fd);
```

**No more questions.**