CS 162 Spring 2021 — Operating Systems and Systems Programming

Midterm 1

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.

This exam has 6 questions of varying difficulty and length. 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 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 internet sources in constructing my answers." **Type your full name below to acknowledge that you've read and agreed to this statement.**

1. **(10.0 points) 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) (2.5 points)

   i.

   Performing a syscall requires a transfer from user mode to kernel mode.

   ◯ True

   ◯ False

   ii.

   Explain.

(b) (2.5 points)

i.

A syscall can be successfully completed without a transfer from user mode to kernel mode.

◯ True

◯ False

ii.

Explain.

(c) (2.5 points)

i.

This program will print “Hello World”.

```
void func_name(int special_int) {
  if (special_int == 1) {
    pthread_exit(NULL);
  }
}
void *run(void *arg) {
  pthread_mutex_t *mutex_ptr = (pthread_mutex_t *) arg;
  pthread_mutex_lock(mutex_ptr);
  func_name(1);
  pthread_mutex_unlock(mutex_ptr);
  return NULL;
}
int main() {
  pthread_mutex_t mutex;
  pthread_t thread;
  pthread_mutex_init(&mutex, NULL);
  pthread_create(&thread, NULL, run, &mutex);
  pthread_join(thread, NULL);
  pthread_mutex_lock(&mutex);
  printf("Hello World");
  pthread_mutex_unlock(&mutex);
}
```

○ True

○ False

ii.

Explain.

(d) (2.5 points)

i.

The file descriptor table resides in kernel memory.

◯ True

◯ False

ii.

Explain.

(e) (2.5 points)

i.

The file descriptor table resides in user memory.

◯ True

◯ False

ii.

Explain.

(f) (2.5 points)

i.

Calling `lseek()` on an open socket file descriptor results in an error.

◯ True

◯ False

ii.

Explain.

(g) (2.5 points)

i.

Calling `lseek()` on an open socket file descriptor does not result in an error.

◯ True

◯ False

ii.

Explain.

2. (12.0 points) 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) (2.0 pt)

Which of the following situations correctly describe instances of user mode transfer or kernel mode transfer?

- ☐ The timer interrupt interrupts a user program.
- ☐ A user program attempts to access unmapped memory and triggers a page fault.
- ☐ A bug in the kernel's scheduler algorithm causes a segmentation fault.
- ☐ One process sends a signal to another process.
- ☐ None of the above.

(b) (2.0 pt)

Which of the following correctly describe the instruction to return from an interrupt (`iret`)?

- ☐ Is executable in user mode
- ☐ Atomically modifies multiple registers
- ☐ Is used in kernel mode transfer or user mode transfer
- ☐ May modify data on the user stack
- ☐ None of the above

(c) (2.0 pt)

In the below program, several threads are created. Which of the following statements in `foo` always print the same memory address when evaluated by different threads in the same process?

```
int global;

void* foo(void* arg) {
  printf("%p\n", &foo);
  printf("%p\n", &global);
  printf("%p\n", &arg);
  printf("%p\n", arg);
  return NULL;
}

int main() {
  void* hmem = malloc(1);
  for (int i = 0; i < 3; i++) {
    pthread_t pid;
    pthread_create(&pid, NULL, foo, hmem);
  }
}
```

☐ `printf("%p\n", &foo)`

☐ `printf("%p\n", &global)`

☐ `printf("%p\n", &arg)`

☐ `printf("%p\n", arg)`

☐ None of the above

(d) (2.0 pt)

Let's assume that there are three threads, Threads A, B, and C, running in Process Z. For which of these synchronization scenarios would you utilize a single semaphore initialized with a value of 2 (as opposed a semaphore initialized to some other value)?

☐ Ensuring that Thread A completes before Thread B

☐ Preventing more than 2 of the threads from running function `f()` simultaneously

☐ Ensuring that Thread A runs after Thread B and Thread C have both completed

☐ Preventing Thread A or B from running function `f()` simultaneously

☐ None of the above

(e) (2.0 pt)

Let's assume that there are three threads, Threads A, B, and C, running in Process Z. For which of these synchronization scenarios would you utilize a single semaphore initialized with a value of 2 (as opposed to a semaphore initialized to some other value)?

☐ Ensuring that Thread B completes before Thread A

☐ Preventing more than 2 of the threads from running function `f()` simultaneously

☐ Ensuring that Thread C runs after Thread A and Thread B have both completed

☐ Preventing Thread B or C from running function `f()` simultaneously

☐ None of the above

(f) (2.0 pt)

Assume we have an empty file called `red.txt`. Consider the following code:

```
int main() {
  int red = open("red.txt", O_RDWR);
  if (fork() == 0) {
    close(red);
  } else {
    int status;
    char* str = "blue";
    wait(&status);
    write(red, str, 4);
  }
}
```

What could be the contents of red.txt after we run this code?

☐ blue

☐ blu

☐ The file is still empty

☐ The program is guaranteed to error

☐ None of the above

(g) (2.0 pt)

Which of the following statements about files are true?

☐ Within a single process, calling `open()` twice on the same file with the same permissions will return two different file descriptors.

☐ `fwrite` uses a buffer in kernel space to store data before data is written to disk.

☐ A child process created using `fork()` will have the same set of open file descriptors as its parent process immediately after `fork()`.

☐ Threads in the same process will always share the same file descriptors.

☐ None of the above.

3. (28.0 points) Short Answer

(a) (4.0 pt)

Explain dual-mode operation and describe its main purpose.

(b) (4.0 pt)

Is it possible for busy waiting to perform better than conventional blocking synchronization in certain cases? Explain the conditions when this would occur, or explain why this impossible.

(c) (4.0 pt)

Explain the difference between the `fork()` syscall in Linux and the `exec()` syscall in PintOS.

(d) (4.0 pt)

When implementing the Project 1 syscalls, Monty Mole forgot to check for invalid memory accesses. Given this vulnerability, a malicious program wants to view kernel data at physical address `0xC0A1E5CE`. Explain how the malicious program could accomplish this. *Hint: Think about which syscall(s) would be useful here.*

(e) (4.0 pt)

Process A and B each have open file descriptors equal to 162. Is it possible that these file descriptors correspond to different files? Explain the conditions when this would occur, or explain why this impossible.

(f) (4.0 pt)

In general, why is it faster to have inter-process communication using a pipe over using a file?

(g) (4.0 pt)

Fawful is implementing locks using `futex`'s. The implementations of `lock_acquire` and `lock_release` are given below (we will use the signatures for `test_and_set` and `futex` given in lecture):

```
int futex(int *uaddr, int futex_op, int val, const struct timespec *timeout);
int test_and_set(int *val);

void lock_acquire(int *lock) {
  while (test_and_set(lock)) {
    futex(lock, FUTEX_WAIT, 1, NULL);
  }
}

void lock_release(int *lock) {
  *lock = 0;
  futex(lock, FUTEX_WAKE, 1, NULL);
}
```

Assuming we have two threads A and B in a single process Z, let's assume that the `futex(lock, FUTEX_WAIT, 1, NULL)` function call in `lock_acquire` **ALWAYS** puts the thread to sleep rather than **conditionally** putting the thread to sleep. Explain why this would not yield a correct implementation of a lock.

4. (10.0 points) Koopalings Castle

Eek! There's only 1 castle left in the Mushroom Kingdom for the Koopalings and the Goombas to use, as the rest have been freed from their rule. The Koopalings (Larry, Morton, Wendy, Iggy, Roy, Lemmy, and Ludwig) each want to use the castle to practice their own stomping, while the Goombas want to use the castle to hold a party together. We've attempted to write a program to manage this resource.

Constraints:

- If either Koopalings or Goombas can enter the castle, Koopalings should be let in.
- Koopalings cannot be in the castle at the same time as Goombas.
- There is a maximum of 1 Koopaling in the castle at any given time.
- There is a maximum of 24 Goombas in the castle at any given time.

```
#define LIMIT 24

pthread_cond_t koopaling_cv, goomba_cv;
pthread_mutex_t lock;
int activeKoopalings, activeGoombas;
int waitingKoopalings, waitingGoombas;

void koopaling_stomp(void) {
  pthread_mutex_lock(&lock);
  waitingKoopalings++;
  if (activeKoopalings + activeGoombas > 0) {
    pthread_cond_wait(&koopaling_cv, &lock);
  }
  waitingKoopalings--;
  activeKoopalings++;
  pthread_mutex_unlock(&lock);
  STOMP_MARIO();
  pthread_mutex_lock(&lock);
  activeKoopalings--;
  if (waitingGoombas == 0 && waitingKoopalings > 0) {
    pthread_cond_signal(&koopaling_cv);
  } else if (waitingKoopalings > 0) {
    pthread_cond_signal(&koopaling_cv);
  } else if (waitingGoombas > 0) {
    pthread_cond_broadcast(&goomba_cv);
  }
  pthread_mutex_unlock(&lock);
}

void goomba_party(void) {
  pthread_mutex_lock(&lock);
  waitingGoombas++;
  if (activeKoopalings > 0 || activeGoombas == LIMIT) {
    pthread_cond_wait(&goomba_cv, &lock);
  }
  waitingGoombas--;
  activeGoombas++;
  pthread_mutex_unlock(&lock);
  SUPER_GOOMBA_PARTY();
  pthread_mutex_lock(&lock);
  activeGoombas--;
  if (activeGoombas == 0 && waitingKoopalings > 0) {
    pthread_cond_signal(&koopaling_cv);
  } else if (waitingGoombas > 0) {
    pthread_cond_signal(&goomba_cv);
  }
  pthread_mutex_unlock(&lock);
}
```

(a) (3.0 pt)

Does the above code satisfy the given constraints assuming Mesa semantics? Explain.

(b) (3.0 pt)

Does the above code satisfy the given constraints assuming Hoare semantics? Explain.

(c) (4.0 pt)

Eek! The Koopalings and Goombas aren't cooperating, and they've decided to each use their own lock. In other words, every `lock` in `koopaling_stomp` will be replaced with `koopaling_lock`, and every `lock` in `goomba_party` will be replaced with `goomba_lock`. Describe a possible race condition that might result from this.

5. (20.0 points) Beaver Bother Logs

You are writing a program for the beavers of Beaver Bother that will automatically sort log entries into separate log files based on a simple protocol: the first character of each log entry will be a digit representing the index of the destination log file. For example, the entry `2 - Beaver Bother` will be saved to `log2.txt`, while the entry `0 - Bothering Beavers` will be saved to `log0.txt`. Log entries will be collected from standard input. You've written a function, `get_entry`, to get one log entry from `stdin` and save it to a null-terminated buffer. **IMPORTANT: Even though each entry is null-terminated, the null terminator should not be written to the log file.**

You also decide that the main process of the program should spawn a child process for each log file (to handle file IO), and you plan to use pipes to forward log entries to each child process. Fill in the missing code below to finish your program! **We recommend writing down your answers on scratch paper first, then filling in the input boxes.**

NOTE: You should only use one line per blank. Extra lines will not be graded.

```
#define BUFFER_LEN 1024

int get_entry(FILE* infile, char* buf, size_t buflen) {
  size_t index = 0;
  char ch = fgetc(infile);
  int log_index = ch - '0';
  buf[0] = '\0';
  while ((index < buflen - 2) && (ch != '\n')) {
    buf[index++] = ch;
    buf[index] = '\0';
    ch = fgetc(infile);
  }
  buf[index++] = '\n';
  buf[index] = '\0';
  return log_index;
}

typedef struct pipe_fds {
  int fds[2];
} pipe_fds;

int main(int argc, char** argv) {
  int num_logs = atoi(argv[1]);
  pipe_fds log_fds[num_logs];
  pid_t pid;
  int log_index;
  for (log_index = 0; log_index < num_logs; log_index++) {
    ______[A]______;
    pid = fork();
    if (pid == 0)
      ______[B]______;
  }
  if (pid != 0) {
    char str[BUFFER_LEN];
    while (1) {
      log_index = get_entry(stdin, str, BUFFER_LEN);
      size_t bytes_written;
      size_t total = 0;
      size_t str_length = strlen(str);
      int wfd = log_fds[log_index].fds[1];
      while (bytes_written = write(______[C]______)) {
        ______[D]______;
      }
    }
  } else {
    char filename[BUFFER_LEN];
    sprintf(filename, "log%d.txt", log_index);
    int file_fd = open(filename, O_RDWR | O_CREAT, 0644);
    char read_buf[BUFFER_LEN];
    size_t bytes_read;
    int rfd = ______[E]______;
    while (bytes_read = read(rfd, read_buf, BUFFER_LEN)) {
      size_t bytes_written;
      size_t total = 0;
      while (bytes_written = write(______[F]______)) {
        ______[G]______;
      }
    }
  }
  return 0;
}
```

(a) (3.0 pt)

[A]

(b) (3.0 pt)

[B]

(c) (2.0 pt)

[C]

(d) (2.0 pt)

[D]

(e) (2.0 pt)

[E]

(f) (2.0 pt)

[F]

(g) (2.0 pt)

[G]

(h) (4.0 pt)

If the parent process unexpectedly crashes, what will happen to the child processes? Explain. *Hint: Child processes are not automatically terminated when the parent process is terminated.*

6. (20.0 points) Coroutines

Coroutines are functions whose execution can be suspended and resumed. You're likely familiar with them by a different name in the form of Python's generators. In this problem, we'll implement a simple version of coroutines in C. Our coroutines are *lazy,* meaning that they do not execute/continue executing until a value has been requested from them (via the `next` function). You'll implement coroutines according to the following API, defined in `coroutine.h`:

```
coroutine.h

typedef struct coroutine coroutine_t;

/* Launches a new coroutine using `function`, passing it `aux` as its second
 * argument.  Returns a new coroutine_t* which can be used to request values
 * from the coroutine using the `next` function. */
coroutine_t* launch_coroutine(void (*function)(coroutine_t*, void*), void* aux);

/* Used inside a coroutine to pass `value` to the corresponding caller of
 * `next`.  As our coroutines are lazy, `yield` should block until another call
 * to `next` is made before continuing execution. */
void yield(coroutine_t* coroutine, void* value);

/* Obtains the next `yield`ed value from the coroutine, if there is one, and
 * stores it in `dest`.  If and only if there is no value to be obtained
 * (because the coroutine has finished executing), the return value will be
 * `false`. */
bool next(coroutine_t* coroutine, void** dest);
```

An example of using this API is shown below:

```
// Finite coroutine iterating over the characters of a string
static void iterate(coroutine_t* coroutine, void* _string) {
  char* string = (void*)_string;
  while (*string) {
    yield(coroutine, (void*)(*string++));
  }
}

// Used to demonstrate laziness
static void say_hello(coroutine_t* coroutine, void* _) { printf("Hello world!\n"); }

int main() {
  char* alphabet = "ABCDEFG";
  coroutine_t* iterate_coroutine = launch_coroutine(iterate, alphabet);
  void* value;
  while (next(iterate_coroutine, &value)) {
    printf("%c ", (char)value);
  }
  printf("\n");
  coroutine_t* say_hello_coroutine = launch_coroutine(say_hello, NULL);
}
```

The output from the above example should be:

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

Notice that `Hello world!` is **not** part of the output. This is because as mentioned before, our coroutines are *lazy,* and there was no call to `next` made with `say_hello_coroutine`, therefore the function `say_hello` never began running.

Fill in the blanks below to complete the implementation of coroutines. You can assume that all syscalls with not error, and you can use as many lines as necessary. Your implementation must not be subject to race-conditions, must not busy-wait, and must not leak resources (namely memory). Furthermore, we emphasize that this is a **coding** question: all solutions must be given in the C programming language. No credit will be awarded for pseudocode, the contents of comments, code which calls "helper functions" which do not exist, etc.

`coroutine.c`

```
struct coroutine {
  void* value;
  bool finished;
  void (*function)(coroutine_t*, void*);
  void* aux;
  sem_t has_yielded;
  sem_t next_requested;
  pthread_t thread;
};

static void* coroutine_stub(void* _coroutine) {
  coroutine_t* coroutine = (coroutine_t*)_coroutine;
  sem_wait(&coroutine->next_requested);
  coroutine->function(coroutine, coroutine->aux);
  coroutine->finished = true;
  sem_post(&coroutine->has_yielded);
  return NULL;
}

coroutine_t* launch_coroutine(void (*function)(coroutine_t*, void*), void* aux) {
  coroutine_t* coroutine = malloc(sizeof(coroutine_t));
  coroutine->value = NULL;
  coroutine->finished = false;
  coroutine->function = function;
  coroutine->aux = aux;
  sem_init(&coroutine->has_yielded, 0, 0);
  sem_init(&coroutine->next_requested, 0, 0);
  pthread_create(&coroutine->thread, NULL, coroutine_stub, coroutine);
  return coroutine;
}

void yield(coroutine_t* coroutine, void* value) {
  --------------------------------------------
  --------------------------------------------
  --------------------------------------------
}

bool next(coroutine_t* coroutine, void** dest) {
  --------------------------------------------
  --------------------------------------------
  --------------------------------------------
  --------------------------------------------
  --------------------------------------------
  --------------------------------------------
  --------------------------------------------
}
```

(a) (6.0 pt)

```
void yield(coroutine_t* coroutine, void* value)
```

(b) (14.0 pt)

```
bool next(coroutine_t* coroutine, void** dest)
```

7. 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.