CS 162 Fall 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.

Preliminaries

(a) Full name

(b) Student ID number

(c) Autograder login (e.g. `student123`)

1. (27.0 points)

For the following questions, select whether they are true or false and explain in two sentences or less. Explanations longer than two sentences or lack of explanation will receive no credit.

(a) (3.0 points)

i. A process's address space in kernel space.

True [✓ False]

ii. Explain.

A process's address space is in user space.

(b) (3.0 points)

i. The OS directly schedules processes, not threads.

○ True

[✓ False]

ii. Explain.

Threads encapsulate concurrency and are scheduled by the OS. Processes are instances of programs and encapsulate isolation.

(c) (3.0 points)

i. Threads in the same process share the same stack.

○ True

[✓ False]

ii. Explain.

Each thread has its own stack.

(d) (3.0 points)

i. After forking, open files are preserved, so closing a file descriptor in the child process would make the parent process unable to use its own file descriptor that refers to the same file object.

○ True

[✓ False]

ii. Explain.

File descriptors index into file descriptions which are reference counted, so the parent can still use the same file descriptors even after a child closes them.

(e) (3.0 points)

i. Each user thread has its own separate kernel stack.

[✓ True]

False

ii. Explain.

There is a kernel stack for each thread.

(f) (3.0 points)

i. Programs which use multiple threads are always faster than programs that do not.

○ True

[✓ False]

ii. Explain.

Overhead of multithreading (e.g. context switching, thread creation) synchronization can cause a multithreaded program to run slower than a single threaded program, especially on a single core machine where the threads would concurrently instead of in parallel. If the multithreaded program requires synchronization, a large critical section may result in threads being run sequentially instead of in parallel.

(g) (3.0 points)

i. During a system call, the kernel validates arguments on the user stack then copies them over to the kernel stack.

○ True

[✓ ● False]

ii. Explain.

The kernel copies over the arguments onto the kernel stack before validating them to prevent against time-of-check to time-of-use (TOCTOU) attacks.

(h) (3.0 points)

i. x86 calling convention as talked about in Discussion 0 requires the base pointer to be pushed onto the stack by the callee.

○ True

[✓ False]

ii. Explain.

This is suggested but not strictly required.

(i) (3.0 points)

i. The Linux `exec` system call creates a new process.

○ True

[✓ False]

ii. Explain.

Linux `exec` will override the current process image with a new image. To spawn a new process in Linux, `fork` should be used instead. Pintos `exec` is similar to Linux `exec` and `fork` combined.

2. (18.0 points) Select All

(a) (3.0 pt) Which of the following that are true regarding I/O?

- ■ High level I/O deals with `FILE*` struct.
- ☐ All I/O related syscalls are defined to all forms of I/O under the "everything is a file" uniformity idea from POSIX.
- ☐ There is one file descriptor table per thread.
- ☐ Low-level I/O is buffered in user space.
- ☐ None of the above.

i. seen in lectures and assignments that `FILE*` is used for high level I/O.
ii. The uniformity idea from POSIX covers `open`, `close`, `read`, and `write` but nothing more. For instance, sockets do not support `lseek`.
iii. Each process has a file descriptor table which are shared by all the threads in that process.
iv. Low-level I/O is buffered in kernel space.

(b) (3.0 pt) Which of the following are true regarding x86 calling convention as talked about in Discussion 0?

- ☐ `addl (%eax), %ebx` adds the longword in memory at the address `eax` to the longword in `ebx`, and stores the result in `eax`.
- ■ When the callee returns, the return address is gone but the arguments are still on the stack.
- ☐ Stack alignment means the stack is 16 byte aligned before we push arguments onto the stack.
- ☐ The saved `ebp` memory address is lower than the argument's.
- ■ The arguments are pushed onto the stack in reverse order.

i. Result is stored in the latter operand `%ebx`.
ii. Callee is responsible for popping the return address off the stack per step 4.
iii. Stack must be 16 byte aligned after we push the arguments.
iv. Saved `ebp` memory address refers to the base pointer of the previous stack frame. The arguments being pushed onto the stack are for the next stack frame, meaning the saved `ebp` memory address is above the arguments.
v. Step 1.

(c) (3.0 pt) Which of the following are true about synchronization primitives?

[✓ Semaphores can be used to implement monitors.]

[✓ Monitors can be used to implement semaphores.]

☐ Semaphores can be used to achieve mutual exclusion but not scheduling constraints.

☐ Calls to signal a condition variable can always be replaced with broadcast when using Hoare semantics assuming behavior of Hoare semantics broadcast is the same as the behavior of Mesa semantics broadcast.

☐ None of the above.

i. Semaphores can be used to implement both locks and condition variables (see Fall 2019 Midterm 1 Question 4(c) for a thorough implementation).
ii. Keep track of a counter variable in our monitor as our shared data. When downing a semaphore, we wait using the condition variable until counter becomes a positive integer. When upping a semaphore, we increment the counter and broadcast using the condition variable. Any time the counter variable is accessed, the lock of the monitor should be used.
iii. Scheduling constraints can be accomplished as well with semaphores (e.g. shared data from Discussion 2).
iv. Mesa semantics broadcast doesn't guarantee a thread will run right away. The `if` statements used to check the condition for Hoare semantics will no longer be valid.

(d) (3.0 pt) Which of the following are true about dual mode operation?

[✓ A thread can execute entirely in user mode.]

[✓ A thread can execute entirely in kernel mode.]

☐ Starting a new process is the only way to switch from kernel to user mode.

[✓ Page table entries cannot be modified in user mode.]

☐ None of the above.

i. Thread creation and exiting are ignored per Clarifications.
ii. Some threads are designated to do purely kernel operations.
iii. Resuming after an interrupt, exception, or system call switches from kernel to user mode.
iv. This privilege is limited to kernel mode for security reasons.

(e) (3.0 pt) Which of the following must be saved during a context-switch?

☐ Page table

[✓ Base pointer]

☐ Current CPU privilege mode

☐ Ready list

[✓ Program counter]

i. Threads share the same address space, so there's no need to save the page table. Moreover, even in a process context switch, the page table wouldn't need to be saved since the memory management unit handles the page table.
ii. Each thread has its own base pointer.
iii. Current privilege level (CPL) is stored as part of the code segment (CS) register in x86. Full credit was given for selecting and not selecting due to not covering this during class.
iv. Ready list is stored in the memory.
v. Each thread has its own program counter.

(f) (3.0 pt) Which of the following that are true about interrupts?

- [✓] The hardware saves the values for the stack pointer and program counter before jumping to the interrupt handler.
- ☐ Interrupts are synchronous events independent of the process.
- ☐ Disabling interrupts is guaranteed to implement mutual exclusion.
- [✓] Lower priority interrupts may be put on hold by a higher priority interrupt.
- ☐ None of the above

i. These values need to be saved when trapping from user mode to kernel mode.
ii. Interrupts are asynchronous.
iii. A multiprocessor could allow other CPUs to enter critical section.
iv. Higher priority interrupts are able to preempt lower priority interrupts.

3. Short Answer

(a) (12.0 points)

Examine the following function `copy_parts(const char *src, const char *dest)`, which copies certain bytes from the `src` file into the `dest` file.

```
void copy_parts(const char* src, const char* dest) {
  char* buffer = malloc(1000 * sizeof(char));

  int read_fd = open(src, O_RDONLY);
  lseek(read_fd, 500, SEEK_SET);
  read(read_fd, buffer, 150 * sizeof(char));

  int write_fd = open(dest, O_WRONLY);
  lseek(write_fd, 200, SEEK_SET);
  write(write_fd, buffer, 100 * sizeof(char));

  /* breakpoint */

  read(read_fd, buffer, 100 * sizeof(char));
  write(write_fd, buffer + 50, 100 * sizeof(char));

  close(read_fd);
  close(write_fd);
}
```

i. (4.0 pt) Is the following function's behavior deterministic? If not, what is the issue and how do we fix it?

No. Since we are using low-level IO, we have no guarantee that we are able to read or write the amount desired in one function call to `read` or `write`. Therefore, we need to call `read` and `write` in a while loop to ensure that we can finish reading and writing the same number of bytes each time we call copy_parts. Alternatively, we can use the high-level IO (`fread` and `fwrite`) to guarantee deterministic behavior (assuming all calls succeed).

ii. (4.0 pt) If there was an issue with the function, suppose we have addressed the issue and the function behavior is now guaranteed to be deterministic. What is the current file position of the src and dest files at the breakpoint? Explain.

We are 650 bytes into the `src` file and 300 bytes into the `dest` file.
`lseek(read_fd, 500, SEEK_SET)` brings us to position 500 in the `src` file.
`read(read_fd, buffer, 150 * sizeof(char)` brings us to position 650 in the `src` file.
`lseek(write_fd, 200, SEEK_SET)` brings us to position 200 in the `dest` file.
`write(write_fd, buffer, 100 * sizeof(char))` brings us to position 300 in the `dest` file.

iii. **(4.0 pt)** If there was an issue with the function, suppose we have addressed the issue and the function behavior is now guaranteed to be deterministic. Suppose a file `src.txt` has 500 `a`'s followed by 200 `b`'s followed by 300 `c`'s followed by EOF. Suppose a file `dest.txt` has 100 `x`'s followed by EOF. What is the result of calling `copy_parts(“src.txt”, “dest.txt”)`? Explain. You may use up to ten sentences for this question to explain if necessary.

`lseek(read_fd, 500, SEEK_SET)` brings us to position 500 in `src.txt`. The function call `read(read_fd, buffer, 150 * sizeof(char))` brings us to position 650 in `src.txt`, and reads 150 `b`s into buffer.
The function call `lseek(write_fd, 200, SEEK_SET)` brings us to position 200 in `dest.txt`, so `dest.txt` starts off with 100 `x`s followed by 100 null characters. The `write(write_fd, buffer, 100 * sizeof(char))` copies 100 b's into dest.txt, so now “dest.txt” consists of 100 `x`s followed by 100 null characters followed by 100 `b`s.
The `read(read_fd, buffer, 100 * sizeof(char))` starts at position 650 in `src.txt` and copies 50 `b`s and then 50 `c`s into the first 100 entries of buffer. The final `write(write_fd, buffer + 50, 100 * sizeof(char))` copies 50 `c`s and then 50 `b`s into `dest.txt`. The `b`s from `buffer[100]` to `buffer[149]` were not overwritten by the previous `read` call.
In the end, `dest.txt` contains 100 `x`s followed by 100 null characters followed by 100 `b`s followed by 50 `c`s followed by 50 `b`s.

(b) (4.0 pt) Can a user thread acquire a lock without entering the kernel? Explain.

Yes. For example, you can use an atomic read-modify-write instruction.

(c) (4.0 pt) Explain the purpose of system calls, and why they cannot be executed in user mode.

We use system calls to request services from the OS. They can't be executed in user mode because that may violate process isolation / cause the kernel to crash.

(d) (4.0 pt) Explain two reasons why high-level I/O uses a user space buffer.

System calls to low-level I/O are much more expensive (25x) compared to ordinary function calls. Byte-oriented kernel buffers exacerbate this problem and greatly reduces throughput.
Moreover, low-level I/O lack functionality (e.g. reading until a newline using `fgets` or `getline`) that are useful to users.

(e) (4.0 pt) What happens when a multithreaded program calls fork?

The child process inherits the single thread where `fork` was called. All other threads are not copied and therefore do not exist in the new process.

(f) (4.0 pt) In Pintos, is there a need to switch page directory base register when switching from user to kernel mode? Why or why not?

There is no need since kernel virtual memory is global.

(g) (4.0 pt) What does the `$0x30` represent in `int $0x30` in Pintos?

`int $0x30` is used to trap to the kernel to make a system call. `$0x30` represents the index number in the interrupt vector table that stores the handler for system calls (i.e. `syscall_handler`).

4. (18.0 points) Shared Data Revisited

The following is a modified version of the shared data question from Discussion 2, and is related to the implementation of the `wait` syscall in Project 1. For this problem, we're going to implement a wait function that allows one thread (the main thread) to wait for another thread to finish writing some data.

We're going to assume we don't have access to `pthread_join` for this problem. Instead, we're going to use synchronization primitives (locks and semaphores). We need to design a struct for sharing information between the two threads.

We also need to implement three functions to initialize the shared struct and synchronize the 2 threads. `initialize_shared_data` will initialize our shared struct. `wait_for_data` (called by the main thread) will block until the data is available. `save_data` (called by the child thread) will write 162 to the struct. Another requirement is that the shared data needs to be freed once as soon as it is no longer in use. When using synchronization primitives, the critical section should be as few lines as possible while still ensuring a race free environment.

Fill in the blanks so that main will always print "Parent: Data is 162". You may assume the program compiles without error and all library calls will succeed.

```
typedef struct shared_data {
  sem_t sem;
  int data;
  _______________________
} shared_data_t;

void initialize_shared_data(shared_data_t* shared_data) {
  sem_init(&shared_data->sem, 0, 0);
  shared_data->data = -1;
  _______________________
}

int wait_for_data(shared_data_t* shared_data) {
  sem_wait(&shared_data->sem);
  int data = shared_data->data;
  _______________________
  return data;
}

void* save_data(void* shared_pg) {
  shared_data_t* shared_data = (shared_data_t*)shared_pg;
  shared_data->data = 162;
  sem_post(&shared_data->sem);
  _______________________
  return NULL;
}

int main() {
  shared_data_t* shared_data = malloc(sizeof(shared_data_t));
  initialize_shared_data(shared_data);
  pthread_t tid;
  int error = pthread_create(&tid, NULL, &save_data, (void*)shared_data);
  int data = wait_for_data(shared_data);
  printf("Parent : Data is % d\n", data);
  _______________________
  return 0;
}
```

(a) **(3.0 pt)** Fill in the missing code (if any) for `shared_data_t`. You may use up to five lines.

```
int ref_cnt;
pthread_mutex_t lock;
```

`ref_cnt` is necessary to keep track of how many threads have access to the shared data and when the shared data can be freed. The `lock` provides mutual exclusion for `ref_cnt` since `ref_cnt` may be modified by both threads concurrently (see `wait_for_data` and `save_data` below). Alternatively, another semaphore or equally valid synchronization mechanism could have been used to achieve mutual exclusion.

(b) **(3.0 pt)** Fill in the missing code (if any) for `initialize_shared_data`. You may use up to five lines.

```
shared_data->ref_cnt = 2;
pthread_mutex_init(&lock, NULL);
```

`shared_data->ref_cnt` needs to be initialized to 2 since there are two threads who will have access to `shared_data`. The lock also needs to be initialized with `pthread_mutex_init`.

(c) **(4.0 pt)** Fill in the missing code (if any) for `wait_for_data`. You may use up to 10 lines.

```
pthread_mutex_lock(&shared_data->lock);
int ref_cnt = --shared_data->ref_cnt;
pthread_mutex_unlock(&shared_data->lock);
if (ref_cnt == 0) {
    free(shared_data);
}
```

The reference count needs to be decremented. The lock needs to be acquired before and released after decrementing the reference count. Finally, we need to check if the reference count is 0 and free `shared_data` if so.

We have to store the decremented `shared_data->ref_cnt` in a separate variable `ref_cnt` because a race condition could occur in a multitude of ways. For instance, let's say both the child and parent thread have decremented `shared_data->ref_cnt` and are waiting to execute the if statement. The parent thread checks the condition of `shared_data->ref_cnt == 0` and goes into the body of the if statement ready to free `shared_data`.

i. However, before it frees, we context switch into the child thread. The condition of `shared_data->ref_cnt == 0` is satisfied as well, so the child thread goes into the body of the if statement ready to free `shared_data`. As a result, only one thread will truly get to free `shared_data`, while the other thread would be attempting to free a freed pointer. This results in undefined behavior.

ii. If the parent thread frees the `shared_data`, the child thread still needs to execute this check. As a result, the child thread will attempt to access `shared_data->ref_cnt` which is an illegal behavior since `shared_data` is freed.

Alternatively, you might propose to check `shared_data->ref_cnt == 0` in the critical section. However, this holds on to the lock longer than necessary which the question specifically asked not to do.

(d) **(2.0 pt)** Fill in the missing code (if any) for `save_data`. You may use up to 10 lines.

```
pthread_mutex_lock(&shared_data->lock);
int ref_cnt = --shared_data->ref_cnt;
pthread_mutex_unlock(&shared_data->lock);
if (ref_cnt == 0) {
    free(shared_data);
}
```

(e) **(2.0 pt)** Fill in the missing code (if any) for `main`. You may use up to 10 lines.

No code is missing from `main` since `shared_data` should have been freed within `wait_for_data` or `save_data`.

(f) **(4.0 pt)** What kind of synchronization, if any, is needed for `shared_data->data` in both `wait_for_data` and `save_data`? Explain.

No synchronization is necessary on `shared_data->data` because the data will never be read before it is written to. This is accomplished using `shared_data->sem`which is first initialized to 0. This semaphore is only upped after the child thread sets the data, so the parent thread would never read before then.

5. (22.0 points) Santa's List

Santa is digitizing his accounting department and would like to track naughty and nice children on his computer. He has a file for each person in the form of `childX.txt` (e.g. `child0.txt`) that contains string records representing the good and bad deeds someone has done all year. For example,

```
child0.txt

GOOD attended live lecture and asks questions.
GOOD finished assignments in a timely manner.
BAD didn't study for midterm.

child1.txt

BAD didn't attend any discussion.
BAD didn't help his teammates during projects.

child2.txt

GOOD answered a lot of questions on Piazza.
BAD didn't fill out course evaluations.
```

Fill in the code below that determines whether a child has performed more good deeds than bad deeds. As there are many children, create a new thread for each file for concurrency. When all threads are done counting all files, the program should write the filenames with positive `karma` to `nice.txt` and others to `naughty.txt`. Separate each filename by a newline with a trailing newline at the end of the file.

```
typedef struct thread_data {
  int tid;
  pthread_t thread;
  char filename[128];
  int karma;
} thread_data_t;

void* count_good_bad(void* threadarg) {
  __________[A]__________;
  __________[B]__________;
  char* line = NULL;
  size_t len = 0;
  ssize_t read;
  while ((read = getline(&line, &len, fp)) != -1) {
    if (strncmp(line, "GOOD", 4) == 0) {
      tdata->karma++;
    } else if (strncmp(line, "BAD", 3) == 0) {
      tdata->karma--;
    }
  }
  pthread_exit(NULL);
}

int main(int argc, char* argv[]) {
  int num_children = atoi(argv[1]);
  thread_data_t thread_data_array[num_children];

  for (int i = 0; i < num_children; i++) {
    thread_data_array[i].tid = i;
    /* Stores child<i>.txt (e.g. child0 for i = 0) in to thread_data_array[i].filename */
    sprintf(thread_data_array[i].filename, "child%d.txt", i);
    thread_data_array[i].karma = 0;
    __________[C]__________;
  }

  for (int i = 0; i < num_children; i++) {
    pthread_join(thread_data_array[i].thread, NULL);
  }

  FILE* fp_nice = fopen("nice.txt", "w");
  FILE* fp_naughty = fopen("naughty.txt", "w");

  for (__________[D]__________) {
    if (__________[E]__________) {
      __________[F]__________;
    } else {
      __________[G]__________;
    }
  }

  fclose(fp_nice);
  fclose(fp_naughty);

  return 0;
}
```

With the above example, the files would look like

```
nice.txt
```

```
child0.txt
child2.txt
```

```
naughty.txt
```

```
child1.txt
```

(a) **(2.0 pt)** [A]

```
thread_data_t* tdata = (thread_data_t*) threadarg
```

(b) **(2.0 pt)** [B]

```
FILE *fp = fopen(tdata->filename, "r")
```

(c) **(2.0 pt)** [C]

```
pthread_create(&thread_data_array[i].thread, NULL, count_good_bad, (void*)
&thread_data_array[i])
```

(d) **(2.0 pt)** [D]

```
int i = 0; i < num_children; i++
```

(e) **(2.0 pt)** [E]

```
thread_data_array[i].karma >= 0
```

(f) **(2.0 pt)** [F]

```
fprintf(fp_nice, "%s\n", thread_data_array[i].filename)
```

(g) **(2.0 pt)** [G]

```
fprintf(fp_naughty, "%s\n", thread_data_array[i].filename)
```

(h) **(4.0 pt)** What would happen if we did not call `pthread_join`?

The program would write the results of tabulation before all threads complete. However, the process would not wait for all threads to complete before exiting as we do not call `pthread_exit` from the main thread.

(i) **(4.0 pt)** How would you implement this using processes rather than threads?

Instead of calling `pthread_create`, you would call `fork` for each filename. `exec` is not acceptable since the original process would get replaced and never writes the results.
You would then use pipes to communicate results back to the main process which would write the results to `nice.txt` and `naughty.txt`. Other less desirable alternatives include sockets, files, or shared memory. For the latter two options, a `wait` would need to be used.

6. Improving pwords

Recall `pwords` from Homework 1 which used multiple threads to count words. Specifically, each thread handles one file and writes to a shared `word_count_list_t` struct that is protected by a lock. As you noticed, its performance does not improve much over the single-threaded implementation. In this problem, we will attempt to remedy this situation.

In the following code, we are still using `N` (assume at least 2) "worker threads", one to process each input file, but each thread will write to its own (non-shared) `word_count_list_t` struct first. Then, we are going to use one separate "merger thread" to merge any 2 ready `word_count_list_t` structs. All these `word_count_list_t` structs will be communicated through a currently **non-thread safe** `queue_t` struct.

The skeleton of the non-thread safe program is shown below. Your task is to make it thread-safe by **using the two provided semaphores only**. Your code may only include semaphore operations `sem_wait` and `sem_post`. Your solution must maximize the amount of work that can execute in parallel.

You may assume all calls to `fopen`, `fclose`, `pthread_create`, `pthread_join`, `malloc`, and other provided functions succeed.

```
/* FIFO queue. */
typedef struct queue queue_t;
/* Initializes empty queue into QUEUE. */
void queue_init(queue_t* queue);
/* Pops the front from QUEUE returns it. Undefined behavior when queue is empty. */
wordcount_list_t* queue_pop(queue_t* queue);
/* Pushes WCLIST to the end of QUEUE. */
void queue_push(queue_t* queue, word_count_list_t* wclist);

/* Initializes an empty word count list into WCLIST. */
void words_init(word_count_list_t* wclist);
/* Counts words into WCLIST from INFILE. */
void words_count(word_count_list_t* wclist, FILE* infile);
/* Sorts contents of WCLIST descending order in-place. */
void words_sort(word_count_list_t* wclist);
/* Prints contents of WCLIST. */
void words_print(word_count_list_t* wclist);
/* Returns a new merged word count list that combines the word counts of WCLIST1 and WCLIST2. */
word_count_list_t* words_merge(word_count_list_t* wclist1, word_count_list_t* wclist2);

int N;
sem_t sem_mutex;
sem_t sem_items;
queue_t queue;

typedef struct worker_args {
  int tid;
  char* name;
    pthread_t thread;
  word_count_list_t* wclist;
} worker_args_t;

void* worker_thread_func(void* arg) {
  worker_args_t* worker_args = (worker_args_t*)arg;

  FILE* infile = fopen(worker_args->name, "r");
  words_count(worker_args->wclist, infile);
  fclose(infile);
```

```
    __________[A]__________;
    queue_push(&queue, wclist);
    __________[B]__________;
    __________[C]__________;
  }

  void* merger_thread_func(void* arg) {
    for (int i = 0; i < N - 1; i++) {
      __________[D]__________;
      __________[E]__________;
      __________[F]__________;
      word_count_list_t* wclist1 = queue_pop(&queue);
      word_count_list_t* wclist2 = queue_pop(&queue);
      __________[G]__________;

      word_count_list_t* merged = words_merge(wclist1, wclist2);

      __________[H]__________;
      queue_push(&queue, merged);
      __________[I]__________;
      __________[J]__________;
    }
  }

  int main(int argc, char* argv[]) {
    N = argc - 1;
    sem_init(&sem_mutex, 0, 1);
    sem_init(&sem_items, 0, 0);
    queue_init(&queue);

    worker_args workers[N];
    for (int i = 0; i < N; i++) {
      workers[i].name = argv[i + 1];
      workers[i].tid = i;
      workers[i].wclist = malloc(sizeof(word_count_t));
      words_init(&workers[i].wclist);
      pthread_create(&workers[i].thread, NULL, worker_thread_func, &workers[i]);
    }

    pthread_t merger_thread;
    pthread_create(&merger_thread, NULL, merger_thread_func, NULL);

    for (int i = 0; i < N; i++) {
      pthread_join(workers[i].thread, NULL);
    }

    pthread_join(merger_thread, NULL);

    word_count_list_t* final_result = queue_pop(&queue);
    words_sort(final_result);
    words_print(final_result);

    return 0;
  }
```

(a) (3.0 pt) [A]

```
sem_wait(&sem_mutex)
```

(b) (3.0 pt) [B]

```
sem_post(&sem_items)
```

(c) (3.0 pt) [C]

```
sem_post(&sem_mutex)
```

(d) (3.0 pt) [D]

```
sem_wait(&sem_items)
```

(e) (3.0 pt) [E]

```
sem_wait(&sem_items)
```

(f) (3.0 pt) [F]

```
sem_wait(&sem_mutex)
```

(g) (3.0 pt) [G]

```
sem_post(&sem_mutex)
```

(h) (3.0 pt) [H]

```
sem_wait(&sem_mutex)
```

(i) (3.0 pt) [I]

```
sem_post(&sem_mutex)
```

(j) (3.0 pt) [J]

```
sem_post(&sem_items)
```

(k) **(3.0 pt)** Describe the purpose of the two semaphores `sem_mutex` and `sem_items`.

`sem_mutex` provides mutual exclusion to `queue` to prevent concurrent accesses. `sem_items` acts as a counter for how many items are left in `queue` to merge.

(l) **(4.0 pt)** Would launching 2 merger threads work? Why or why not?

It won't work as the merger threads may fail to proceed. Suppose there are a total of 2 files, and they have been processed and added to the queue. Then, suppose merger thread 1 calls `sem_wait(&sem_items)` first, then merger thread 2 calls `sem_wait(&sem_items)`. They will then get stuck in their own second down() call for forever.

(m) **(4.0 pt)** Can this approach be faster than original `pwords`? Why or why not?

Yes, it can be faster, since files are processed independently without synchronization, and the merging thread can run concurrently or in parallel with the word counting threads.

7. Multithreading Fun

Consider the following two threads. Assume each instruction is atomic.

```
# ================================================
# Syntax
# load  r, a    loads a into r
# store r, b    stores r into b
# add   r, c    adds c to r and stores into r
# sub   r, c    subtracts c from r and stores into r
# mul   r, c    multiples r by c and stores into r
# ================================================

x = 1
y = 3

# Thread A
load  r1, x
add   r1, 1
store r1, y

# Thread B
load  r1, y
sub   r1, 1
store r1, x
```

(a) **(4.0 pt)** What are all the possible values of x and y? List them as tuples separated by commas (e.g. (1, 2), (3, 4))

(0, 1), (1, 1), (1, 2), (2, 2), (2, 3), (3, 3), (3, 4)
When the threads don't interleave, the only possible values are (1, 2) and (2, 3). When the threads interleave, the possible values are (0, 1), (1, 1), (1, 2), (2, 2), (2, 3), (3, 3), and (3, 4).

(b) **(4.0 pt)** Now consider a third thread executing in parallel with threads A and B. Is there any sequence of execution that produces `x = 4` and `y = 4` after all threads complete? If yes, provide an example of such execution by writing the exact execution order where each line is the line of code and which thread it belongs to. If not, explain why not.

```
# Thread C
load  r1, x
mul   r1, 2
store r1, x
```

```
load  r1, x    # Thread C
load  r1, y    # Thread B
sub   r1, 1    # Thread B
load  r1, x    # Thread A
add   r1, 1    # Thread A
mul   r1, 2    # Thread C
store r1, x    # Thread C
store r1, x    # Thread B
store r1, y    # Thread A
```

(c) **(4.0 pt)** How would you use locks in thread A and thread B such that these threads can produce no more than three distinct values between both `x` and `y`?

Surrounding each thread's code with a lock, so each thread becomes a critical section. The only possible values are 1, 2, and 3 across both `x` and `y` resulting from the tuples (1, 2), (2, 2), (2, 3).

8. References

```
/* Semaphore */
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);

/* Lock */
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);

/* Condition Variable */
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *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);

/* Process */
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[]);

/* Thread */
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);
void pthread_exit(void *retval);

/* High-Level IO */
FILE *fopen(const char *pathname, const char *mode);
int fclose(FILE *stream);
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);
fprintf(FILE * restrict stream, const char * restrict format, ...);

/* Low-Level IO */
int open(const char *pathname, int flags);
int close(int fd);
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);

/* Socket */
int socket(int domain, int type, int protocol);
int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, structure sockaddr *addr, socklen_t *addrlen);
int connect(int sockfd, struct sockaddr *addr, socklen_t addrlen);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

9. Clarifications

The following are selected clarifications made during the exam. Some clarifications have not been included and instead corrected within the exam since they were typos and didn't significantly affect the exam.

True or False

- Regarding Pintos for (5)

Select All

- "Defined to" means "applicable to" for (1)
- Thread execution does not include creation or exit for (4)
- Switching refers to thread switching for (5)
- "Select all of the following that are true of interrupts." is the prompt for (6)

Short Answer

- Assume all syscalls don't error when called for (1).

Santa's List

- Nonnegative karma (karma greater than or equal to 0) should be written to `nice.txt`. everything else will go to `naughty.txt`.
- `ssize_t getline(char ** restrict linep, size_t * restrict linecapp, FILE * restrict stream)` is the method header for `getline`

Multithreading Fun

- `r1` is a global variable.
- Assume all threads run to completion.
- Thread C is not used for (c).
- Describe a situation with exactly three distinct values for (c).

No more questions.