Please print in pen:

Waterloo Student ID Number:

WatIAM/Quest Login UserID:

Final Exam - Winter 2026 - SE 350

1. Before you begin, make certain that you have one **2-sided booklet with 9 pages**. You have **90 minutes** to answer as many questions as possible. The number in parentheses at the beginning of each question indicates the number of points for that question.

2. Please read all questions before starting the exam as some of the questions are substantially more time consuming. Read each question carefully. Make your answers as concise as possible. **If there is something in a question that you believe is open to interpretation, then please write your interpretation and assumptions!**

3. All solutions must be placed in this booklet. If you need more space to complete an answer, you may be writing too much. However, if you need extra space, use the blank space on the last page of the exam clearly labeling the question and indicate that you have done so in the original question.

Good Luck!

Question	Points Assigned	Points Obtained
1	34	
2	16	
3	20	
4	15	
5	15	
Total	100	

1. (34 points) True-False with explanation.

For each question:

- Circle your answer and write your explanation below each question.
- Explanations should not exceed 3 sentences.
- One point for correct true-false.
- One point for correct explanation.
- No points for any explanation if true-false is incorrect.

1. The OS directly schedules processes, not threads.

True False

- False. Threads are scheduled by the OS.

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

True False

- False. Each thread has its own stack.

3. 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

- False. 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.

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

True False

- False. Overhead of multithreading (e.g. context switching, thread creation, synchronization) can cause a program to run slower than a single threaded program, especially on a single core machine. For example, a large critical section may result in threads running sequentially.

5. During a system call, the kernel validates the arguments passed by the user program on the user memory space. If the arguments are successfully validated, the operating system can safely use them to process the system call.

True False

- False. The kernel copies over the arguments onto the kernel space before validating them to prevent against time-of-check to time-of-use attacks

6. The Linux exec system call creates a new process

True False

- False. exec will override the current process image with a new image. To spawn a new process in Linux, fork should be used instead.

7. Threads within the same process can share data (since they share the same memory space), but threads in different processes cannot share data.

True False

- False. There are many inter-process communication (IPC) methods that can be used for threads in different processes to share data.

8. A zombie process is one that got hung up trying to access a disk which is experiencing permanent read errors.

True False

- False. A zombie process is one that is not running anymore (has exited) but has not yet relayed its exit code to its parent.

9. The pipe system call creates a single file descriptor to read and write to it. Child processes created with fork share the file descriptor with their parents, so both parent and child have access to the file through the pipe file descriptor.

True False

- False. Pipe are implemented as kernel-managed buffers and are accessed through two file descriptors. When a process forks, the file descriptors are copied into the child process's file descriptor table; they are not shared.

10. The test&set instruction consists of two independent operations: (1) read the value from memory and (2) replace it with the value "1". Thus, it must be used in a loop to protect against the case in which multiple threads execute test&set simultaneously and end up interleaving these two operations from different threads.

True False

- False. The test&set instruction is atomic (that is the whole point), so there is no way for different test&set operations from different threads to interleave.

11. TCP/IP makes it possible for a server with IP address X and well-known port Y to simultaneously communicate with multiple remote clients without intermixing response streams.

True False

- True. The socket layer can distinguish between connections if the clients have different client IP addresses, different client ports, or both.

12. Every thread has its own heap.

True False

- False, Threads in the same process share the same heap.

13. Context switches are always involuntary.

True False

- False. A thread can call yield to relinquish the CPU.

14. In Unix systems, it is possible to read from a random location of a file.

True False

- True. seek can be used to read from anywhere in a file.

15. The function pthread_yield will always ensure that another thread gets to run.

True False

- False. It puts the calling thread on the ready queue. The scheduler picks the next thread to run from the ready queue and hence could pick the same thread to run again (e.g., if there are no other ready threads).

16. Special atomic read–modify–write hardware instructions (e.g., atomic test-and-set) are required to implement mutual exclusion correctly in a multiprocessor system.

True False

- False. Peterson's algorithm provides mutual exclusion using only regular reads and writes (assuming a sequentially consistent memory model and atomic reads/writes).

17. Context switching between two threads belonging to the same process is less expensive than context switching between two threads belonging to two different processes.

True False

- True. In the latter case, the kernel also needs to context switch the process address space state (i.e., translation), and the context switch will also result in more cache misses.

2 (16 points total) Networking is an Important Skill! The code below implements a server that handles multiple connections. (We ignore disconnections and other socket errors, as well as fork failure. You could also ignore those failures when answering the following questions.)

```
 1 bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr))
 2 listen(sockfd, 5);
 3 int count = 0;
 4 int status;
 5 while (1) {
 6   newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
 7   count++;
 8   if (fork() == 0) {
 9     // do some communication;
10     exit(0);
11   } else {
12     if (count > 1000) {
13       break;
14     }
15   }
16 }
```

1. **(4 points) [Close Sockets]** Insert code to close sockets whenever appropriate (see the example below). You might need to insert multiple close statements. You should close a socket as soon as it is not needed. You will not receive point for a delayed close.

   Example: After line 1: close(sockfd) # note this example might be incorrect

   After line 8: close(sockfd)

   After line 9: close(newsockfd)

   After line 11: close(newsockfd)

   After line 12 or 16: close(sockfd)

2. **(12 points) [Send]** Assume the server permits at most 3 concurrent client connections at any time. One approach would be to stop accepting new connections once three are active; however, this leaves additional clients without feedback. Instead, when the third connection is accepted, the server must send the message: “System is overloaded; reconnect later.” and then immediately close that connection. (As a result, only two connections remain active. This is acceptable.) Fill in the blanks on the following page to complete the code. Do not close sockets for this problem. You may need some of the following functions:

```
pid_t wait( int * status);
pid_t waitpid( pid_t pid, int * status, int options); /* 0 for options fine */
void exit( int status);
int send( int sockfd, const void* msg, int length, int flags );
int recv( int sockfd, void * buffer, int length, int flags );
```

```
bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr))
listen(sockfd, 5);
int count = 0;
int active = 0;
int status;
while (1) {
  while (active > 2) {
    if (wait(&status) > 0) {
      active--;
    }
  }
  newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
  count++;
  active++;
  if (fork() == 0) {
    if (active == 3) {
      send(newsockfd , “system is overloaded; reconnect later.” , 38 , 0);
    } else {
      // do some communication;
    }
    exit(0);
  } else {
    if (count > 1000) {
      break;
    }
  }
}
```

3. (20 points total) What The Fork. Eleanor, an SE student, wants to create a multithreaded program that she can use to tell everyone how much she loves her favorite operating systems class. She wrote the program below, but she finds that different runs produce different outputs.

```
void* func1(void* args) {
  printf("is\n");
  return NULL;
}

void* func2(void* args) {
  printf("SE350\n");
  return NULL;
}

int main(void) {
  pid_t pid;
  pthread_t pthread;
  int *ret = (int*) malloc(sizeof(int));
  int status;
  pid = fork();
  if (!pid) {
    pthread_create(&pthread, NULL, func2, (void*) ret);
    pthread_join(pthread, NULL);
  } else {
    printf("the\n");
  }
printf("best!\n");
return 0;
}
```

1. **(12 points)** List all of the possible outputs that her program could display when run. Assume that calls to fork and pthread_create always succeed. Add more columns if needed.

SE350 best! the best!	SE350 the best! best!	the best! SE350 best!	the SE350 best! best!		

2. **(8 Points)** Modify Eleanor's program such that the output will always be:

```
SE350
is
the
best!
```

Fill in the blanks to show your changes to the original program.
• You must use pthread_create at least once.
• You may not call printf or the helper functions (func1/func2) directly.
• Write at most one statement per line. You may not need all line

```
void* func1(void* args) {
  printf("is\n");
  return NULL;
}

void* func2(void* args) {
  printf("SE350\n");
  return NULL;
}

int main(void) {
  pid_t pid;
  pthread_t pthread;
  int *ret = (int*) malloc(sizeof(int));
  int status;

  ________________________________

  pid = fork();
  if (!pid) {

    ________________________________

    pthread_create(&pthread, NULL, func2, (void*) ret);
    pthread_join(pthread, NULL);

    pthread_create(&pthread, NULL, func1, (void*) ret);

    pthread_join(pthread, NULL);

    exit(0);


  } else {

    wait(&status);

    printf("the\n");

    ________________________________

  }

    ________________________________


printf("best!\n");
return 0;
}
```

4. (15 points) After graduation, Eleanor is hired by a large company called *Yooyle*. On her first day, her supervisor sends her a piece of code and asks her to determine what fraction of the code is parallelizable (denoted by **F**). Eleanor measures the program's execution time using 1, 2, 4, and 8 cores and computes the following speedups (relative to 1 core):

Number of cores	2	4	8
Measured speedup	1.7	2.5	3.3

Using these measurements, provide a tight bound on the parallelizable fraction F. Clearly justify your bound (for example, you may not simply state $0 \le F \le 1$).

Amdahl's law provides the ideal speedup on N cores for a program with parallelizable fraction F:

$$S_{\text{Amdahl}}(N) = \frac{1}{(1-F) + \frac{F}{N}}$$

In practice, measured speedups are less than or equal to what Amdahl's law predicts for a given F is: $S_{\text{measured}}(N) \le S_{\text{Amdahl}}(N, F)$. This inequality reflects real-world overheads (synchronization, communication, load imbalance) that reduce actual speedup below the ideal. To find a bound on F, we rearrange the inequality. Taking reciprocals (and flipping the inequality direction since both sides are positive):

$$\frac{1}{S_{\text{measured}}} \ge (1-F) + \frac{F}{N} \Rightarrow F \ge \frac{1 - \frac{1}{S}}{1 - \frac{1}{N}} = \frac{N - \frac{N}{S}}{N-1}$$

Computing the Bound

For each measurement, we calculate the lower bound on F:

At $N = 2$ **cores,** $S = 1.7$**:** $F \ge \frac{2-\frac{2}{1.7}}{2-1} = \frac{2-1.176}{1} = 0.824$

At $N = 4$ **cores,** $S = 2.5$**:** $F \ge \frac{4-\frac{4}{2.5}}{4-1} = \frac{4-1.6}{3} = \frac{2.4}{3} = 0.800$

At $N = 8$ **cores,** $S = 3.3$**:** $F \ge \frac{8-\frac{8}{3.3}}{8-1} = \frac{8-2.424}{7} = \frac{5.576}{7} = 0.797$

Tight Bound

All three measurements must be satisfied simultaneously. The tightest bound is determined by the **maximum** of the per-measurement lower bounds (since F must satisfy all three inequalities at once):

$$F \ge 0.824$$

5. (15 points) Spinlock Using `swap`. We want to implement a lock using the atomic read–modify–write instruction `swap`. The semantics of `swap` are given below.

```
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

Implement the functions `spinlock_lock`, and `spinlock_unlock`. You may use busy waiting. **You must use only the `swap` instruction for atomic read–modify–write operations. The use of any other atomic primitive (e.g., test-and-set, compare-and-swap, fetch-and-add, etc.) is not allowed.**

```
typedef struct {
  int flag; // 0 = unlocked, 1 = locked
} spinlock_t;

void spinlock_initialize(spinlock_t *lock) {
  lock->flag = 0; // initially unlocked
}

void spinlock_lock(spinlock_t *lock) {

  int key = 1;
  while (key == 1) {
    swap(&lock->flag, &key);
    // If lock->flag was 0, swap sets it to 1
    // and key becomes 0 -> locked.
    // If lock->flag was 1, key remains 1 -> keep spinning.
  }

}

void spinlock_unlock(spinlock_t *lock) {

  lock->flag = 0; // release the lock

}
```