**Please print in pen:**

Waterloo Student ID Number:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

WatIAM/Quest Login UserID:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

UNIVERSITY OF WATERLOO

**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

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

True False

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

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

True False

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

6. The Linux exec system call creates a new process

True False

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

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

True False

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

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

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

12. Every thread has its own heap.

True False

13. Context switches are always involuntary.

True False

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

True False

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

True False

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

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

**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

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 );
```

```
1 bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr))
2 listen(sockfd, 5);
3 int count = 0;
4 int active = 0;
5 int status;
6 while (1) {
7   while (active > 2) {

8     _____________________________


9     _____________________________


10    _____________________________


11  }
12  newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
13  count++;
14  active++;
15  if (fork() == 0) {

16    _____________________________


17    _____________________________


18    } else {
19      // do some communication;
20    }
21    exit(0);
22  } else {
23    if (count > 1000) {
24      break;
25    }
26  }
27 }
```

**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.

```
 1. void* func1(void* args) {
 2.   printf("is\n");
 3.   return NULL;
 4. }

 5. void* func2(void* args) {
 6.   printf("SE350\n");
 7.   return NULL;
 8. }

 9. int main(void) {
10.   pid_t pid;
11.   pthread_t pthread;
12.   int *ret = (int*) malloc(sizeof(int));
13.   int status;
14.   pid = fork();
15.   if (!pid) {
16.     pthread_create(&pthread, NULL, func2, (void*) ret);
17.     pthread_join(pthread, NULL);
18.   } else {
19.     printf("the\n");
20.   }
21. printf("best!\n");
22. return 0;
23.}
```

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.

| | | | | | |
|---|---|---|---|---|---|
| | | | | | |

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

```
1. void* func1(void* args) {
2.   printf("is\n");
3.   return NULL;
4. }

5. void* func2(void* args) {
6.   printf("SE350\n");
7.   return NULL;
8. }

9. int main(void) {
10.   pid_t pid;
11.   pthread_t pthread;
12.   int *ret = (int*) malloc(sizeof(int));
13.   int status;

14.   ___________________________________

15.   pid = fork();
16.   if (!pid) {

17.     ___________________________________

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

20.     ___________________________________

21.     ___________________________________

22.     ___________________________________


23.   } else {

24.     ___________________________________

25.     printf("the\n");

26.     ___________________________________

27.   }

28.     ___________________________________


29. printf("best!\n");
30. return 0;
31.}
```

**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 $0 \leq F \leq 1$).

**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) {









}

void spinlock_unlock(spinlock_t *lock) {









}
```