CS 417: Operating Systems

Spring 2025: Practice Final Exam

This exam is closed book and closed notes, except for one double-sided page of notes. All cell phones and laptops must be turned off. No calculators may be used.

You have three hours to complete this exam.

Part 1 True/False

For each statement below, circle whether it is true or false. You may add a brief explanation to explain your reasoning (which may be used to award partial points).

Q1 Unlike processes, threads share an address space with other threads in the same process.

True False

Q2 Unlike processes, threads share register state and stack as other threads in the same process

True False

Q3 Spin locks are simple but can result in starvations—the do not guarantee that a thread will *eventually* aqcuire a lock.

True False

Q4 Semaphores require special kernel support to implement beyond mutexes and condition variables.

True False

Q5 Threads are always worse than events because threads have to be implemented in the kernel, while events can be implemented in userspace.

True False

Q6 An `inode` stores metadata for a file or directory, such is its size, access permissions, and where its data is stored on disk.

True False

Part 2 Code Analysis

Read and analyze the following code snippet and answer the questions below. For multiple-choice answers, you may show your work and provide a justification in the space provided. Justifications will be used to award partial credit.

Producer/Consumer

The following code attempts to implement a common producer/consumer pattern in which a bounded buffer is shared between producer threads and consumer threads. Unfortunately, it is incorrect.

```
1  queue_t items[MAX];
2  cond_t fill, empty;
3  // Managed by queue_push and queue_pop to track how many items are
4  // currently in the queue
5  int numfull = 0;
6
7  int consume() {
8    mutex_lock(m);
9    if (numfull == 0) {
10     cond_wait(&fill, m);
11   }
12   int r = queue_pop(items);
13   cond_signal(empty);
14   mutex_unlock(m);
15   return r;
16 }
17
18 void produce(int value) {
19   mutex_lock(m);
20   if (numfull == MAX) {
21     cond_wait(&empty, m);
22   }
23   queue_push(items, value);
24   cond_signal(fill);
25   mutex_unlock(m);
26 }
```

Q7 What is one incorrect behavior that may result from the buggy code?

Q8 How would you fix the bug?

Broken List

Below is a program that attempts to insert twice into a thread-safe list.

```
1  typedef struct __node_t {
2    int key;
3    struct __node_t *next;
4  } node_t;
5
6  typedef struct {
7    mutex_t m;
8    node_t *head;
9  } list_t;
10
11 int list_prepend(list_t *list, int key) {
12     mutex_lock(&m);
13     node_t *n = malloc(sizeof(node_t));
14     if (n == NULL) { return -1; } // failed to insert
15     n->key = key;
16     n->next = list->head;
17     list->head = n; // insert at head
18     mutex_unlock(&m);
19     return 0; // success!
20 }
21
22 int main() {
23   list_t list;
24   list_prepend(&list, 1);
25   list_prepend(&list, 2);
26 }
```

Q9 Assuming `malloc` succeeds in both calls to `list_prepend`, what is the state of the list at the end of the program?

- ◯ head $\rightarrow 1 \rightarrow 2$
- ◯ head $\rightarrow 2 \rightarrow 1$
- ◯ head $\rightarrow 1$
- ◯ head $\rightarrow 2$
- ◯ The program never terminates.

Q10 Assuming `malloc` succeeds the first call to `list_prepend` but fails in the second one, what is the state of the list at the end of the program?

- ◯ head $\rightarrow 1 \rightarrow 2$
- ◯ head $\rightarrow 2 \rightarrow 1$
- ◯ head $\rightarrow 1$
- ◯ head $\rightarrow 2$
- ◯ The program never terminates.

Q11 Assuming `malloc` fails the first call to `list_prepend` and succeeds in the second call, what is the state of the list at the end of the program?

- ◯ head $\rightarrow 1 \rightarrow 2$
- ◯ head $\rightarrow 2 \rightarrow 1$
- ◯ head $\rightarrow 1$
- ◯ head $\rightarrow 2$
- ◯ The program never terminates.

Part 3 Design Question

Q12 One common storage medium is called "flash" storage or solid-state storage. Solid state drives (SSDs) are so-called because they have no mechanically moving parts (unlike the spinning magnetic disks or seek arm of a hard disk). They have much faster access times than magnetic hard disks, with three major downsides: 1) SSDs are much more expensive to manufacture and, thus, buy; 2) each writeable unit (a page) must be erased first before it can be updated; 3) each writable unit can only sustain a fairly small number of erases before it becomes unusable (this is called "wear").

In practice, SSDs have fairly sophisticated computers on them that try to mask the page wear using various "wear-leveling" algorithms—basically lying about which pages are actually being written to. But, for the purpose of this question, we'll assume they don't—if the OS reads, erases or writes to page 15, it's always the same set of molecules.

Your job is to design a file system for SSDs that minimizes wear on pages. SSDs are typically exposed to the file systems the same as hard disks—as a generic block device. Let's assume, for simplicity, that an SSD page is the same size as a block, so a block in your file system implementation always maps to a page.

Your file system should have all the same characteristics as the file-systems discussed in class—files, directories, inodes, etc, so you can elide design details that are common to other file systems. Instead, decide what *kind* of file system implementation to use—a regular naive block file system, a journaling file system—and how. How does your file system design affect wear on individual pages? How does it perform?