Operating Systems Quiz 2 ECE344, Winter 2021

Duration: 1 hour

Examiner: D. Yuan

Instructions

Examination Aids: This is an open book exam.

All questions have been provided in this exam booklet. You need to provide your answers in Quercus.

If any of the questions appear unclear or ambiguous to you, then make any assumptions you need, state them and answer the question that way. If you believe there is an error, state what the error is, fix it, and respond as if fixed.

Please be brief and specific as possible. Clear, concise answers will be given higher marks than vague, wordy answers. Marks will be deducted for incorrect statements in an answer.

Work independently.

MARKING GUIDE

Q1: (2)
Q2: (4)
Q3: (5)
Q4: (4)
TOTAL: (15)

Question 1 (2 points). (A) Discuss the pros and cons of implementing different tabs of a web browser using different threads versus different processes.

Pros of threads: efficiency
Cons of threads: less secure and less reliable. A bug in one tab would bring down the entire browser.

The pros and cons of using processes are the opposite.

(B) If we implement browser tabs in threads, should we use user-level threads (all of them mapped to a single kernel-level thread) or kernel-level threads? Why?

We should use kernel-level threads. Because otherwise an I/O performed by one tab will block all other tabs.

Question 2 (4 points): `fork() and exec()`

Assume the following:

- The first newly created process has PID 100, the next one 101, and then 102, and so on.
- getpid() returns the PID of the calling process
- getppid() returns the PID of the parent process
- The PID of the parent of process 100 is 99

(A) (2 points) Consider the following code:

```
int main(int argc, char ** argv){
    int i;
    for(i = 0; i < 10; i += fork()){
        printf("%d: I'm %d, parent is %d\n", i, getpid(), getppid());
    }
}
```

Describe all possible output of this program.

0: I'm 100, parent is 99
0: I'm 101, parent is 100
0: I'm 102, parent is 101
...
(and keeps going on and on)

The order of the lines shouldn't change.

Note: it's OK to assume the first process has PID 99.

(B) (1 point) Consider the following code:

```
int main(int argc, char ** argv){
    int i;
    for(i = 0; i < 10; i += fork()){
        printf("%d: I'm %d, parent is %d\n", i, getpid(), getppid());
        exec(helloworld);
    }
}
```

Whereas "helloworld" is the executable of a program of the following:

```
int main(int argc, char ** argv){
    printf("Hello World! I'm %d\n", getpid());
}
```

Describe all possible output of this program.

0: I'm 100, parent is 99
Hello World! I'm 100

(C) (1 point) Consider the following code:

```
int main(int argc, char ** argv){
    int i;
    for(i = 0; i < 10; i += fork()){
        printf("%d: I'm %d, parent is %d\n", i, getpid(), getppid());
        exec(a.out);
    }
}
```

Whereas "a.out" is the name of the executable of this program. Describe the possible output of this program.

0: I'm 100, parent is 99
0: I'm 100, parent is 99
0: I'm 100, parent is 99
...
(And it repeats forever.)

Question 3 (5 points) (Robin, Adrian): Consider this C program:

```c
void work_thread(void* arg){
    int * value = arg;
    /* Mark A */
    *value ++;
    /* Mark B */
    printf("num = %d\n", *value);
    /* Mark C */
}

void calculate(int n, int *value){
    int i;
    tid_t * tids = malloc(n * sizeof(tid));
    for(i = 0; i < n; i++){
        tids[i] = thread_create(work_thread, value);
        /* Mark D */
    }
    /* Mark E */
}

int main(int argc, char ** argv){
    int* value = malloc(sizeof(int));
    *value = 0;
    calculate(344, value);
}
```

Answer each sub-question. **The sub-questions are independent of each other, instead of being cumulative.**

The semantic of thread_create() is the same as we discussed in the lecture. The second argument of thread_create() is the argument passed to the function specified in thread_create()'s first argument.

(A) Describe all possible outputs of this program.

If we assume the main thread doesn't exit before the child threads, then there will be 344 lines of outputs, each line has the following form:
num = XX
where XX can be any number ranging from [1, 344], and any combinations of these 344 lines can occur.

For example, this could be an output:
num = 1
num = 2
...
num = 344

And this could also be an output:
num = 1
num = 1
...
num = 1
(there are 344 of them, or less numbers if main thread exits before any child threads.)

(B) If we put the following code at ***Mark D***, what are the possible outputs?

```
thread_join(tids[i]);
```

num = 1
num = 2
...
num = 344

(C) If we put the following code at ***Mark E***, what are the possible outputs?

```
for(i = 0; i < n; i++){
        thread_join(tids[i]);
    }
```

Same as (A).

(D) If we **only** want to ensure the final value of `*value` in the `main` function will always be 344 (and we don't care about anything else), what's the **minimal** patch that will work using lock_acquire() and lock_release()? By minimal it means both the minimal lines of

code and minimal critical region. Answer using the Mark A-E, for example, calling lock_acquire() and Mark A and lock_release() at Mark E.

Call lock_acquire() at Mark A and lock_release() at Mark B.

(E) Now consider the following code snippet:

```
void work_thread(int value){
    value ++;
    printf("num = %d\n", value);
}

void calculate(int n){
    int i;
    tid_t * tids = malloc(n * sizeof(tid));
    for(i = 0; i < n; i++){
        tids[i] = thread_create(work_thread, i);
    }
}

int main(int argc, char ** argv){
    calculate(344);
}
```

Describe and explain all the possible outputs of this program

num = 1
num = 2
...
num = 344
The order of the lines might change, but not the content.
Again, main thread could exit so there could be less # of output lines.

Question 4 (4 points) Concurrency bugs. Next we will show you two code snippets from *real-world software systems*. Unfortunately both have data races. For each sub-question, you are asked to show the interleaving that results in a failure in the following format:

```
Thread 1          Thread 2
```

```
Instruction A
                  Instruction B
Instruction C
```

It shows that Thread 1 executes Instruction A first, followed by Thread 2's Instruction B, then followed by Thread 1's Instruction C. The time goes from the top to the bottom.

Below are the definitions of locks, semaphores, and condition variables in OS161; you can assume they are properly implemented for you, and they have the same semantics as in OS161.

```
1  struct semaphore {
2          char *name;
3          volatile int count;
4  };
5
6  struct semaphore *sem_create(const char *name, int initial_count);
7  void              P(struct semaphore *);
8  void              V(struct semaphore *);
9
10 struct lock {
11         char *name;
12         ...
13 };
14
15 struct lock *lock_create(const char *name);
16 void         lock_acquire(struct lock *);
17 void         lock_release(struct lock *);
18
19 struct cv {
20         char *name;
21         ..
22 };
23
24 struct cv *cv_create(const char *name);
25 void       cv_wait(struct cv *cv, struct lock *lock);
26 void       cv_signal(struct cv *cv, struct lock *lock);
27 void       cv_broadcast(struct cv *cv, struct lock *lock);
28
```

You will need to fix each bug by using these interfaces.

For each question you should introduce the **minimal** changes -- **unnecessary code will be penalized.**

(A) (2 marks) Consider the following code:

```
/* Thread 1 */
```

```
if (t->ptr)
   puts(t->ptr);

/* Thread 2 */
t->ptr = NULL;
```

The variable `t` is shared between the two threads. Show the interleaving that would lead to a failure, and fix the bug using **Semaphores**. What's the value of the initial_count when you create the semaphore?

Buggy interleaving:

```
Thread 1:          Thread 2:
if (t->ptr)
                   t->ptr = NULL;
puts(t->ptr)
```

puts will crash the program.

Fix:

```
/* Thread 1 */
P(sem);
if (t->ptr)
   puts(t->ptr);
V(sem);

/* Thread 2 */
P(sem);
t->ptr = NULL;
V(sem);
```

The initial_count of the semaphore is 1.

(B) (2 marks) Consider the following code:

```
/* Thread 1 */
p = init_ptr(...);

/* Thread 2 */
```

```
state = p->state;
```

The initial value of p is NULL. Show the interleaving that would lead to a failure, and fix the bug using **Condition Variables**.

Buggy interleaving:

```
Thread 1:          Thread 2:
                   state = p->state;
p = init_ptr(...);
```

So as long as the statement in thread 2 happens before the one in thread 1, there will be a crash.

Fix:

```
/* Thread 1 */
p = init_ptr(...);
lock_acquire(lock);
initiated = 1;
cv_signal(cv, lock);
lock_release(lock);

/* Thread 2 */
lock_acquire(lock);
while (!initiated) // replacing while w/ if is fine
   cv_wait(cv, lock);
lock_release(lock);
state = p->state;
```

`initiated` is a global variable with initial value to be 0.