Operating Systems - Assignment 2

Aryan Nath

April 2025

Part 2: Questions

1. Construct your own example of a shared memory race condition (different from your textbook / slides / internet resources).

The following example shows a race condition over the *rear* attribute in a queue.

```
#include <stdio.h>
#include <pthread.h>

#define SIZE 5

int queue[SIZE];
int front = 0;
int rear = 0;

void* enqueue(void* arg) {
    int item = *(int*)arg;

    if ((rear + 1) % SIZE == front) {
        printf("Queue is full, cannot enqueue %d\n", item);
        return NULL;
    }

    // race condition here: two threads can both read the same rear
    queue[rear] = item;
    printf("Enqueued %d at position %d\n", item, rear);

    rear = (rear + 1) % SIZE; // updating shared variable without lock
    return NULL;
}

int main() {
    pthread_t t1, t2;
```

```
    int item1 = 10, item2 = 20;

    pthread_create(&t1, NULL, enqueue, &item1);
    pthread_create(&t2, NULL, enqueue, &item2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Front: %d, Rear: %d\n", front, rear);
    return 0;
}
```

In this implementation, a race condition occurs on the shared variable *rear* during concurrent enqueue operations. This happens as both threads simultaneously the same initial value of rear, and as a result they both write to the same position in the queue resulting in one thread overwriting the data written by the other thread. Hence, the *rear* pointer advances only once instead of twice and only a single element actually gets added to the queue.

2. Consider the following code snippet running on a modern Linux operating system. Assume that there are no other interfering processes in the system. Note that the executable "good_long_executable" runs for 100 seconds, prints the line "Hello from good executable" to screen, and terminates. On the other hand, the file "bad executable" does not exist and will cause the exec system call to fail.

```
int ret1 = fork();
if ret1 == 0 then
   printf("Child 1 started\n");
   exec("good_long_executable");
   printf("Chīld 1 finished\n");
else
   int ret2 = fork();
   if ret2 == 0 then
      sleep(10);
      printf("Child 2 started\n");
      exec("bad_executable");
      printf("Child 2 finished\n");
   else
      wait();
      printf("Child reaped\n");
      wait();
      printf("Parent finished\n");
   end if
end if
```

Write down the output of the above program with justification.

The output of the program is as follows:

```
Child 1 started
Child 2 started
fail system call(exec)
Child 2 finished
```

```
Child reaped
Hello from good executable
Parent finished
```

Explanation:

Parent process calls *fork()*, the child process enters the if block as its *ret1* value is 1 and the parent process enters the else block. The child process (let's call it child 1) prints "Child 1 started\n" and exec("good_long_executable") which makes it copy its program code from the default program of the parent process to that of the "good_long_executable", so it does not print "Child 1 finished".

After the second *fork()* in the else block (by the parent process) the second child process (child 2) is created. Child 2 enters the if block and prints "Child 2 started\n". It tries to run the "bad executable" but it fails, so it does not copy program code from "bad executable" and just prints "Child 2 finished\n".

The parent child then enters the else block and calls *wait()*. Since the "good_long_executable" takes a 100 seconds to finish I expect this to happen before the child 1 process finishes. So the parent process prints "Child reaped" as soon as the child 2 process finishes. It calls wait again, then after the child 1 process has printed "Hello from good executable" and finished execution, the parent process prints "Parent finished" and terminates.

3. The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1, share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process P_i ($i == 0$ or 1) is shown in the figure below. The other process is P_j ($j == 1$ or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem. (Mutual exclusion, progress, and bounded waiting)

```
while (true) {
    flag[i] = true;
    while (flag[j]) {
        if (turn == j) {
            flag[i] = false;
            while (turn == j)
                ; /* do nothing */
            flag[i] = true;
        }
    }
    /* critical section */
    turn = j;
    flag[i] = false;
    /* remainder section */
}
```

1. **Mutual Exclusion:**

We need to show that P_0 and P_1 cannot be in their critical sections simultaneously.

Suppose both processes are in their critical sections, then:

- Both processes passed their outer while loops.
- For both processes, either $flag[j]$ became false or they got through the if condition.

For both processes to be in their critical sections, both $flag[0]$ and $flag[1]$ must be true (as neither process has executed the code after critical section yet).

But if both flags are true, then each proces would enter the while loop checking the other's flag. The only way to exit is through the if statement, which requires $turn == j$.

However, turn cannot equal both 0 and 1 simultaneously. So if P_0 finds $turn == 1$, it will wait. If P_1 finds $turn == 0$, it will wait. If $turn == 0$, then P_0 can proceed but P_1 cannot, and vice versa.

Hence, mutual exclusion has been ensured.

2. **Progress:**
   We want to check that if no process is in its critical section and some process wants to enter, then only those processes competing to enter can participate in the decision, and the decision cannot be postponed indefinitely.

   If the process P_i wants to enter its critical section and P_j is not interested (that is, $flag[j] == false$), then P_i can enter without waiting.

   If both processes want to enter the *turn* variable ensures one will proceed. The proceed who doesn't have its turn currently will turn its flag to false, wait for its turn, then set its flag back to tru and proceed.

   Since *turn* is always assigned to the other process after exiting the critical section, each process will eventually get its turn of both are competing.

3. **Bounded Waiting:**
   We need to show that there exists a bound on the number of times other processes enter their critical sections after a process has made a request to enter its critical section and before that request has been granted.

When a process P_i wants to enter its critical section but is blocked because P_j is in its critical section, P_j will eventually exit and set $turn = i$.

And the next time both processes compete the following will happen:

- Process P_i will find $turn == i$ and proceed.
- Process P_j will find $turn == i$, set $flag[j] = false$, wait until $turn == j$.

This guarantees that P_i can enter its critical section next, so P_j can only enter its critical section at most once between successive entries by P_i. This satisfies the bounded waiting requirement.

Hence. we have proved that this solution by Dekker satisfies Mutual exclusion, progress, and bounded waiting.