ECE 254 S18 Midterm Solutions

J. Zarnett

May 22, 2018

(1.1)

1. Memory: Program A could modify memory in use by program B, which would

2. Printer: A program could cancel the print jobs ahead of its own in the queue, to ensure its print jobs get to go faster.

3. Files: anyone could read or write files belonging to any user.

4. CPU: A process could hog the CPU and not let other processes have a turn.

(1.2) Pipe initializes the file descriptors passed as the argument `int fd[]`. Fork makes the child an exact copy of the parent, including the file descriptors. To establish communication between the two processes, they have to be the same file descriptors; if fork is called first then both parent and child call pipe, but they get different file descriptors and cannot communicate because they have different pipes.

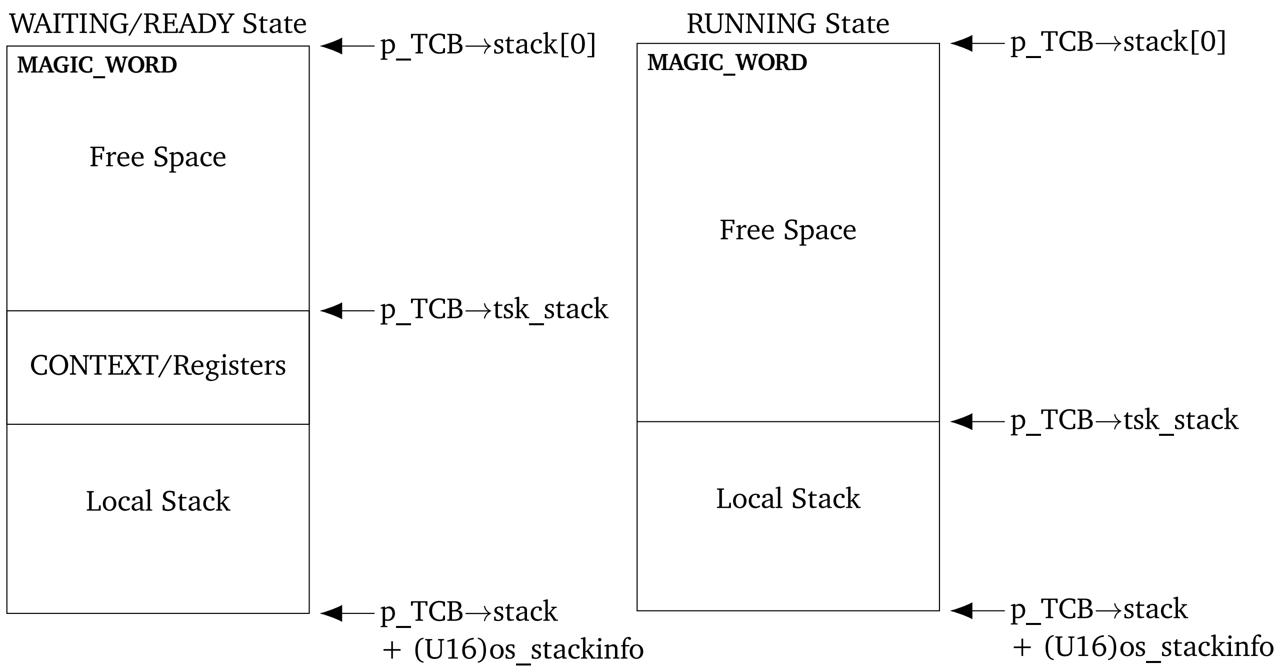
(1.3)

(3 Marks) WAITING/READY state:

- (1 mark) The MAGIC_WORD is at the start of the stack (in the free space section)
- The orientation of the stack doesn't matter, but the growth should be the same
- (1 mark) CONTEXT or REGISTERS is acceptable
- (1 mark) Correct end of stack address:
  - p_TCB→stack + (U16)os_stackinfo
  - p_TCB→stack[(U16)os_stackinfo >> 2]
- (-0.5 mark) If `os_stackinfo` is not properly cast to U16 (be lenient if they accidentally cast to something else).

(2 Marks) RUNNING state:

(2 marks) Remove the CONTEXT/REGISTERS section and point the `tsk_stack` to top of the local stack.



(1 Mark) For Effort If the student made a *reasonable* effort, then give them this mark.

(1.4)

If the signal is not handled, the program just terminates immediately.

If we handle the signal, it is possible to take some steps to clean up and close any open resources like files or network connections.

When handling the signal we would likely (1) use `pthread_cancel` to inform the various threads they are cancelled, (2) close any open files and network connections, and (3) deallocate resources. Any two of these would be acceptable.

The system does not allow handling `SIGKILL` because that would provide a system administrator no way to terminate a misbehaving process.

If you ignore `SIGSEGV` and the instruction is restarted, it will cause a segmentation fault again (because the pointer will still be invalid) and the program can fall into an infinite loop of repeating this segmentation fault over and over again.

(2.1)

Signalling: Your friend texts you when they are leaving so you know when to meet them at the library.

Rendezvous: Submitting the code for marking after all group members have committed their parts to the version control repository.

(Note that just saying something like "meeting up" is not sufficient here; it has to be an example where we are waiting for each side to signal it is okay to proceed)

Mutual Exclusion: a coffee shop where there is a key to access a washroom; a person must have the token (key) to access the resource.

(2.2)

The exact pseudocode that you write varies, but here are the things needed to get full marks for this question:

`sem_init` should set value to the initial value, initialize the mutex, and set the `queue` pointer to `0` or `NULL` (1 mark)

`sem_destroy` needs to deallocate whatever you allocated in the initialization function (here, just destroy the mutex) (0.5 marks)

`sem_wait` should lock the mutex, decrement the current value of the semaphore and check if it is less than 0. If that is the case, then allocate a new `threadinfo`, initialize it with this thread's ID, and ask the OS to block this thread. Unlocking the mutex must be before requesting the OS to block the OS. (4 marks)

`sem_signal` should lock the mutex, increment the current value of the semaphore, check and see if there is anything in the queue. If there's something in the queue, ask the OS to unblock that thread, remove the node from the linked list, and deallocate that node. Then unlock the mutex. (3 marks)

(2.3)

The consumer cannot starve. Even if the producer has priority, eventually the producer will fill the buffer to its capacity C and be blocked. At this point the consumer will definitely get a chance to run.

The producer also cannot starve. At the start of time the buffer is empty so the consumer can't run so the producer must get a chance. And this argument is really just the mirror image of the one for how the consumer cannot starve: eventually the consumer will empty the buffer and at that point the producer will definitely get a chance to run.

No, they cannot become deadlocked. A formal argument is possible but not necessary here. To become deadlocked each process would have to be blocked waiting for the other one. As it is now the only way for the producer to be blocked is if the buffer is full; the only way for the consumer to be blocked is if the buffer is empty. Thus for both to be blocked at the same time the buffer would have to be simultaneously full and empty, which obviously cannot happen.

(2.4)

Any three of the following:

Line 9: Semaphore `empty_list` is initialized wrong. It will cause the program to never make any progress. Solution is to change the initial value parameter to 1.

Line 7: `tasks` is allocated, but not initialized. This means the number of tasks contains a garbage value and tests like if it equals zero will not work as intended. Solution is to add a line setting the value to 0 anywhere in the `init` function.

Line 37: Decrement of `*tasks` is outside of the critical section and therefore a race condition can occur. This could lead to an incorrect value of the number of tasks remaining. Correct this by moving the decrement before the unlock of the mutex.

Line 24: Dobby does `sem_wait` on `full_list`. This will cause a deadlock if a thread is waiting on this semaphore and Dobby is supposed to put more tasks in the list (should be easily found by mismatch of wait/signal calls). The call on line 24 should actually be `sem_post` instead.

(3)

```
void find_and_resolve_deadlock() {
  struct proc* head = find_deadlock();
  while( head != NULL ) {

    /* Find the max process ID */
    pid_t max = 0;
    struct proc* i = head;
    while ( i != NULL ) {
      if ( i->pid > max ) {
        max = i->pid;
      }
      i = i->next;
    }

    /* Use UNIX kill() function with signal 9 on our victim process */
    kill( max, 9 );

    /* Deallocate memory allocated in find_deadlock */
    struct proc* j = head;
    while( j != NULL ) {
      struct proc* temp = j;
      j = j->next;
      free( temp );
    }

    /* If we are done then head will be set to NULL, otherwise continue */
    head = find_deadlock();
  }
}
```

- 3 marks for finding the max process ID
- 1 mark for kill()-ing the chosen process
- 1 mark for correct use of signal 9
- 2 marks for deallocation of the memory
- 2 marks for correct looping / termination of the algorithm