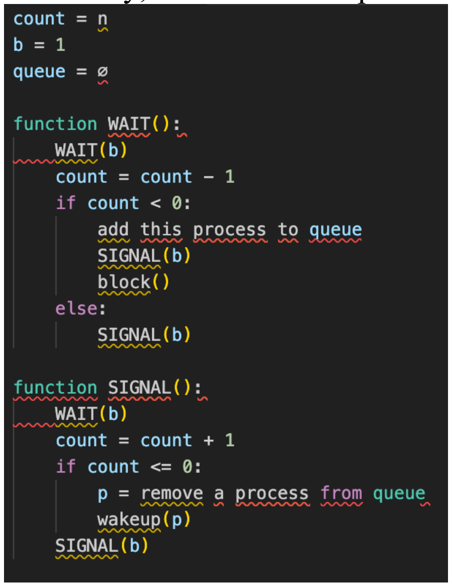
CSEN177, Winter 2025 Homework 3

1. The race condition occurs because if this code is run twice, the two threads from the two independent executions of the get_account function will run concurrently. They share the variable account_count without synchronization, causing them to read and update the same memory location. As a result, one thread can overwrite the other's changes. A simple fix is to use a mutex to lock the section of the code that reads account_count, writes to the array, and increments account_count. This ensures that only one thread can perform these operations at a time, preventing the race condition.
2. Peterson's solution works fine when scheduling is nonpreemptive. When a process starts its entry section, it isn't interrupted until it finishes, so the ordering is preserved. But with preemptive scheduling, things can go wrong. A process might get interrupted right after setting its flag but before it properly sets the turn, which means both processes could end up in their critical sections at the same time. That breaks mutual exclusion unless you add extra safeguards such as memory barriers) to enforce the correct ordering
3. No, the problem doesn't happen under round-robin scheduling. With round robin, every process, regardless of priority will eventually get a slice of CPU time. That means the low-priority process will still run at some point, allowing it to finish its work so that the high-priority process isn't stuck waiting forever. Under pure priority scheduling, however, a low-priority process might never run if there's always a higher-priority process ready to go, which can lead to the situation where the high-priority process is effectively blocked forever. Creating a starvation situation.
4. Yes. The standard semaphore-based solution in Figure 2-28 can work for any number of producers and consumers. Because it uses counting semaphores and a mutex for mutual exclusion, multiple threads can safely coordinate their accesses to the shared buffer. Each producer waits for an empty slot, locks the buffer, adds an item, unlocks the buffer, and signals a filled slot; similarly, each consumer waits for a filled slot, locks the buffer, removes an item, unlocks the buffer, and signals an empty slot. This mechanism works the same way regardless of how many producers and consumers are running.
5. This algorithm does achieve mutual exclusion, only one process can be in the critical section at a time. But it fails the progress and bounded-waiting requirements. Because each process always resets turn to its own ID (P0 sets it to 0, P1 sets it to 1), whichever process starts first keeps the turn set to its ID, causing the other process to wait forever. That means there's no guarantee the other process will ever get its turn, violating bounded waiting and progress.

6. WAIT(b) and SIGNAL(b) are standard binary semaphore (mutex) operations that can be implemented using simple machine instructions like test-and-set or compare-and-swap. The variable count tracks how many “permits” remain; if count is already 0 or negative, any additional thread calling WAIT() must block. Processes that need to wait are placed in a queue, and when a SIGNAL() occurs, if there are blocked processes, one is awakened and allowed to proceed.

   This builds a counting semaphore on top of a single binary semaphore plus a queue. The key is that the binary semaphore b ensures the updates to count and the queue are done atomically, while count keeps track of the semaphore’s integer value.



```
count = n
b = 1
queue = ∅

function WAIT():
    WAIT(b)
    count = count - 1
    if count < 0:
        add this process to queue
        SIGNAL(b)
        block()
    else:
        SIGNAL(b)

function SIGNAL():
    WAIT(b)
    count = count + 1
    if count <= 0:
        p = remove a process from queue
        wakeup(p)
    SIGNAL(b)
```

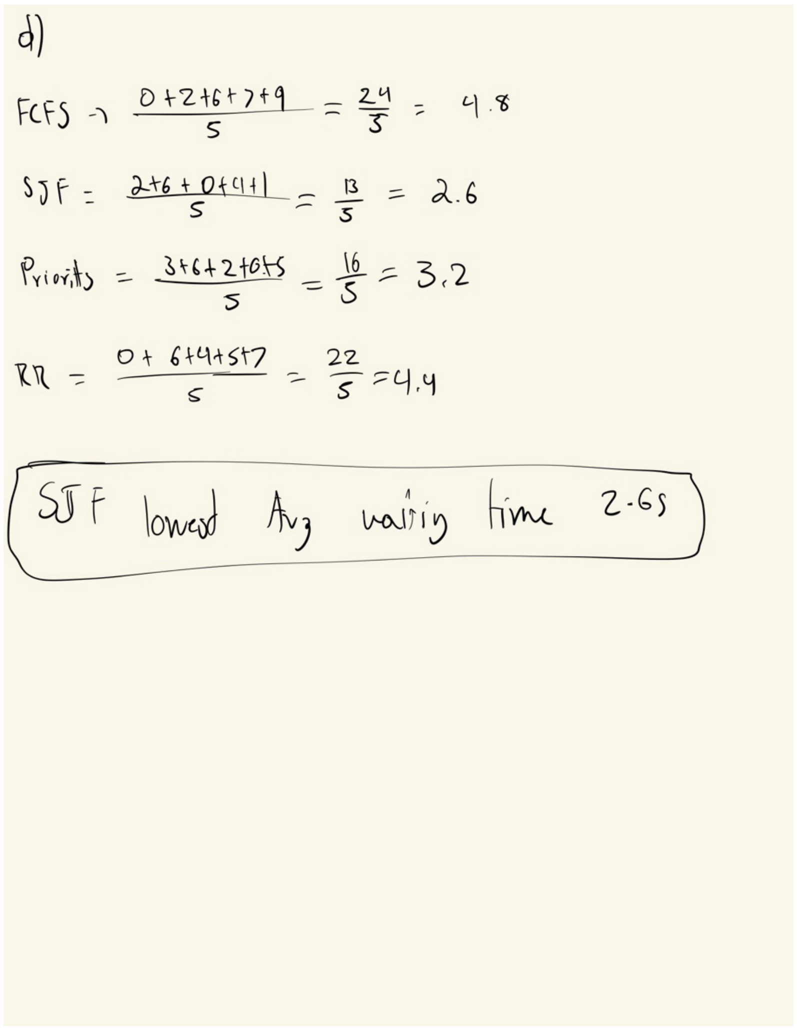
7.

b) TAT = At - Ct   All At = 0   So ct = TAT

	P_1	P_2	P_3	P_4	P_5
FCFS	2	6	7	9	10
SJF	4	10	1	6	2
Priority	5	10	3	2	6
RR	2	10	5	7	8

c) WT = TAT - Burst

	P_1	P_2	P_3	P_4	P_5
FCFS	0	2	6	7	9
SJF	2	6	0	4	1
Priority	3	6	2	0	5
RR	0	6	4	3	7



d)

FCFS $\rightarrow \frac{0+2+6+7+9}{5} = \frac{24}{5} = 4.8$

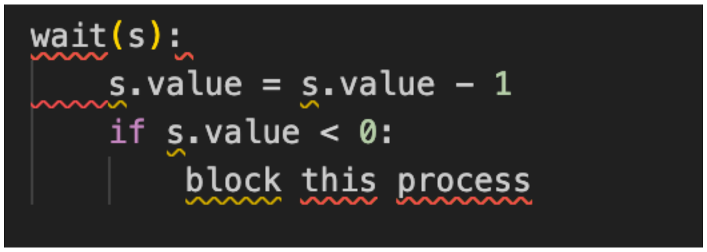
SJF $= \frac{2+6+0+4+1}{5} = \frac{13}{5} = 2.6$

Priority $= \frac{3+6+2+0+5}{5} = \frac{16}{5} = 3.2$

RR $= \frac{0+6+4+5+7}{5} = \frac{22}{5} = 4.4$

$\boxed{\text{SJF lowest Avg waiting time } 2.6s}$

8. Having different time slices per queue allows interactive or short-burst jobs in the higher-priority queues to get smaller, more frequent CPU quanta results in faster response times, while longer-running, CPU-bound jobs in lower-priority queues can have larger quanta to reduce the overhead of frequent context switches. This setup helps ensure that short, interactive tasks get snappy performance, and longer tasks don’t pay excessive context-switch overhead. In a very complex figure this can help optimize resources and time a system is processes for the fastest and most reliable experience.
9. Both processes believe they have successfully entered the critical section, violating mutual exclusion. Ensuring that **wait()** (and **signal()**) are atomic prevents any such interleaving from causing incorrect results.



```
wait(s):
    s.value = s.value - 1
    if s.value < 0:
        block this process
```

10. A race condition arises when both deposit(amount) and withdraw(amount) access and modify the shared balance at the same time. For example, if the balance is 100, and one thread reads it just before the other thread modifies it, both might base their calculations on the same outdated value, leading to a final balance that's incorrect (like 80 instead of 120). A straightforward way to prevent this is to enforce mutual exclusion around the balance updates. For instance, you can use a mutex lock (or any other synchronization mechanism) so that only one function at a time can read and update the shared balance, eliminating the possibility of interleaving operations that lead to the wrong final result.