1) In a system with threads, is there one stack per thread or one stack per process ? Explain.

One stack per thread since each thread calls procedures on its own, so it must have its own stack for the local variables, return addresses, etc.

2) Consider a computer that does not have a TEST AND SET LOCK instruction but does have an instruction to swap the contents of a register and a memory word in a single indivisible action. Can that be used to write a routine **enter_region** such as the one found on page 63 of the text ? If so, explain how it works.

Yes it can. The memory word is used as a flag, with 0 meaning that no one is using the critical section and 1 meaning that someone is using them. Put a 1 in the register, and swap the memory word and the register. If the register contains a 0 after the swap, access has been granted. If it contains a 1, access has been denied. When a process is done, it stores a 0 in the flag in memory.

3) Can the Wait and Signal definitions defined on Slide 35 of Processes be used without disabling and enabling interrupts ? If not, what advantage(s) do Wait and Signal provide over the alternative; allowing the user to disable and enable interrupts directly around critical sections ? Assume the Wait and Signal routines are implemented as system calls, as they really are in protected mode OSs such as UNIX. Are mutual exclusion, progress and bounded waiting satisfied using semaphores with FIFO wait queues ?

Not as written. Some means of mutual exclusion must be provided around the code in Wait and Signal. Question 5 below asks you to do this using Peterson's solution but the solution requires that you pass a parameter to the scheduler in the Wait call. A hardware mechanism such as disabling interrupts is easy, less error prone and less restrictive than software methods.

Let us assume that disabling/enabling interrupts is the only acceptable method. In a protected mode system, user access to cli (disable interrupts) and sti (enable interrupts) is dangerous since it allows them to disable or monopolize the CPU. Therefore, by making Wait and Signal system calls, we can restrict user access to cli and sti while maintaining user access to a mutual exclusion/synchronization mechanism. Moreover, by restricting the use of cli and sti around the code in the Wait and Signal procedures, we have increased the degree of concurrency possible.

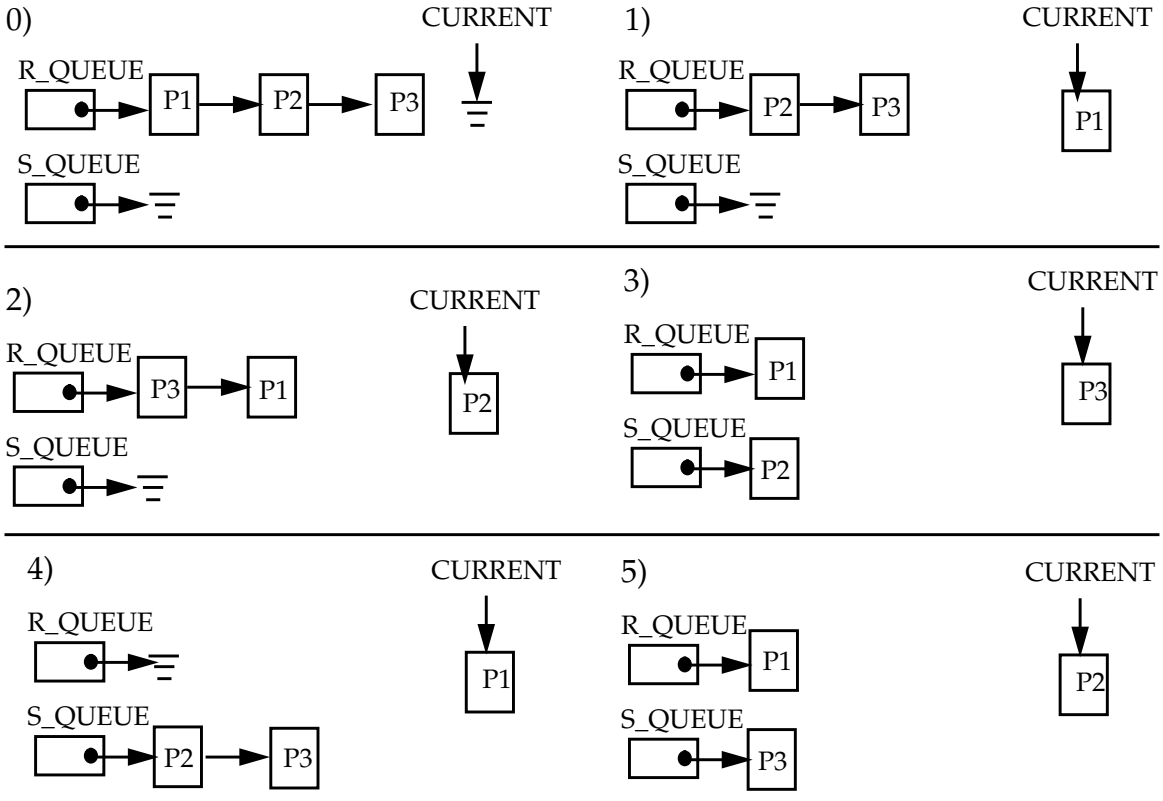
Mutual exclusion is provided since only one process can execute within Wait (assuming a cli instruction is placed before the update to S->val) and therefore, with S->val initially set to 1, only a single process can successfully execute Wait without blocking. Other processes decrement S->val to a negative value and block. The Signal operation increments S->val by 1 and allows only one process to complete the Wait call. Progress is guaranteed as well since a process executing within the critical section will eventually call Signal which causes another waiting process to unblock and enter

the critical section. Since processes must call Signal when they leave the critical section, they can not prevent another process from entering while executing outside their critical region. The queue guarantees every waiting process will get in. A FIFO queue guarantees that a waiting process will have to wait at most n-1 times (bounded waiting)(where n is the number of processes contending for the resource) to enter the critical section.

4) Draw diagrams showing how the ready queue (R_QUEUE) and semaphore queue (S_QUEUE) change over time given three processes which share a critical section using Wait and Signal calls on a semaphore. The processes actions are described below. Assume all three processes are on the ready queue initially. Also assume that the process scheduled to run is removed from all queues and is pointed to be a global variable called CURRENT.

1) Process P1 runs and calls Wait. Since the semaphore value was initialized to 1, it immediately enters its critical section.
2) While in the critical section, the timer fires and P1 is preempted.
3) P2 runs and calls Wait.
4) P3 runs and calls Wait.
5) P1 runs and calls Signal.

Assume that a process waiting on a semaphore has priority over a process that is leaving the critical section (calling Signal).



5) Suppose you were programming a multi-threaded process in a system that does not provide Wait and Signal system calls. Suppose you wanted to run two threads which updated a shared buffer, therefore you need to provide mutual exclusion. Additionally, suppose it is important that each of these processes have a fair chance to update the buffer since each of them do something different to the buffer. Can you use Peterson's solution and Wait and Signal routines to provide this functionality ? If so, show how you would do this. Are mutual exclusion, progress and bounded waiting satisfied if you use FIFO wait queues in your implementation of Wait and Signal ?

If you answer no, then show deadlock that can occur:

```
P0:                                   P1:

flag0 = TRUE;                         flag1 = TRUE;
turn = 1;                             turn = 0;
while (flag1 && turn == 1);           while (flag0 && turn == 0);
Wait(&S);                             Wait(&S);
flag0 = FALSE;                        flag1 = FALSE;

critical section                      critical section

flag0 = TRUE;                         flag1 = TRUE;
turn = 1;                             turn = 0;
while (flag1 && turn == 1);           while (flag0 && turn == 0);
Signal(&S);                           Signal(&S);
flag0 = FALSE;                        flag1 = FALSE;
```

P0 executes Wait successfully and enters critical section. Context switch allows P1 to execute and calls Wait. Since Wait does not return, flag1 is not set back to FALSE. P0 runs again, sets flag0 to TRUE, turn to 1 and loops forever at while statement before Signal. Since it can not call Signal, it can not allow P1 to unblock and set flag1 back to FALSE, which is necessary in order for P0 to call Signal. Using a different set of flag and turn variables around the Signal does not work since mutual exclusion is then violated.

If you answer yes, then you must write Wait call as Wait(&S, &flagx) and indicate that the call to the scheduler within Wait (e.g., ChangeState(CURRENT, waiting, &flag)) must set flag to FALSE as the last statement before resuming execution of a process on the ready queue. It might be tempting to put the *flag = FALSE call in Wait after the Enqueue(CURRENT, S->L) but then it becomes possible that the current process ends up on both the ready queue and blocked queue. The adjustments to the block and ready queue MUST finish before another process is allowed to execute Wait or Signal !

6) Consider the procedure put_forks on page 78 of the dining philosophers solution. Suppose that the variable **state[i]** was set to THINKING *after* the two calls to **test**, rather than *before*. How would this change affect the solution for the case of three phi-

losophers ? For 100 philosophers ?

The change would mean that after a philosopher stopped eating, neither of his neighbors could be chosen next. With only two other philosophers, both of them neighbors, the system would deadlock. With 100 philosophers, all that would happen would be a slight loss of parallelism.

7) Measurements of a certain system have shown that the average process runs for a time T before blocking on I/O. A process switch requires a time S, which is effectively wasted (overhead). For round-robin scheduling with quantum Q, give a formula for the CPU efficiency for each of the following.
a) $Q = \infty$
b) $Q > T$
c) $S < Q < T$
d) $Q = S$
e) Q nearly 0

Efficiency = Useful CPU time/total CPU time.
When $Q >= T$, a process runs for T to completion and a process switch occurs for S. Therefore a) and b) are $T/(T+S)$. When the quantum is shorter than T, each run of T will require T/Q process switches, wasting a time ST/Q. The efficiency here is then $T/(T+ST/Q)$ which reduces to $Q/(Q+S)$. For d), we just substitute Q for S and find that the efficiency is 50%. For e), as Q approaches 0 so does the efficiency.

8) Five batch jobs A through E, arrive at a computer center at almost the same time. They have estimated running times of 10, 6, 2, 4, and 8 minutes. Their (externally determined) priorities are 3, 5, 2, 1, and 4, respectively, with 5 being the highest priority. For each of the following scheduling algorithms, determine the mean process turnaround time. Ignore process switching overhead.
a) Round robin
b) Priority scheduling.
c) First-come, first-served (run in order 10, 6, 2, 4, 8)
d) Shortest job first.

For (a), assume that the system is multiprogrammed, and that each job gets its fair share of the CPU. For (b) through (d) assume that only one job at a time runs, until it finishes. All jobs are completely CPU bound.

For round robin, during the first 10 minutes each job gets 1/5 of the CPU. At the end of 10 minutes, C finishes. During the next 8 minutes, each job gets 1/4 of the CPU, etc. The finishing times for the five jobs are 10, 18, 24, 28 and 30 for an average of 22 minutes. For priority scheduling, finish times are 6, 14, 24, 26 and 30 for an average of 20 minutes. Average is 19.2 minutes for FCFS and 14 minutes for SJF.

9&10) Consider the following set of processes, with the length of the CPU-burst time given in milliseconds

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5, all at time 0.

a) Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a nonpreemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1) scheduling. (5pts)

b) What is the turnaround time of each process for each of the scheduling algorithms in part a ? (5pts)

c) What is the waiting time of each process for each of the scheduling algorithms in part a ? (5pts)

d) Which of the schedules in part a results in the minimal average waiting time (over all processes)? (5pts)

FCFS

P1	P1	P1	P1	P1	P1	P1	P1	P1	P1	P2	P3	P3	P4	P5	P5	P5	P5	P5
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

SJF

P2	P4	P3	P3	P5	P5	P5	P5	P5	P1	P1	P1	P1	P1	P1	P1	P1	P1	P1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

NPP

P2	P5	P5	P5	P5	P5	P1	P1	P1	P1	P1	P1	P1	P1	P1	P1	P3	P3	P4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

RR

P1	P2	P3	P4	P5	P1	P3	P5	P1	P5	P1	P5	P1	P5	P1	P1	P1	P1	P1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Turnaround time | waiting time:

FCFS: 10, 11, 13, 14 and 19 | 0, 10, 11, 13, 14 (9.6)

SJF: 19, 1, 4, 2, 9 | 9, 0, 2, 1, 4 (3.2)

NPP: 16, 1, 18, 19, 6 | 6, 0, 16, 18, 1 (8.2)

RR: 19, 2, 7, 4, 14 | (5-1 + 8-6 + 10-9 + 12-11 + 14-13), 1, (2-0 + 6-3), 3, (4-0 + 7-5 + 9-8 + 11-10 + 13-12) = 9, 1, 5, 3, 9 (5.4)

Minimal average waiting time SJF = 3.2