1. Give the kernel-level implementation of semaphore wait, that is, the if-then-else logic covered in class and on quizzes. (10 pts.)

<answer>

\[
\text{if ( S.value == 0 )} \\
\; \; \; \; \; \; \text{block calling thread and place on S.blocked_list;} \\
\text{else} \\
\; \; \; \; \; \; \text{S.value = S.value - 1;}
\]

2. Give the kernel-level implementation of semaphore signal, that is, the if-then-else logic covered in class and on quizzes. (10 pts.)

<answer>

\[
\text{If ( empty( S.blocked_list ) )} \\
\; \; \; \; \; \; \text{S.value = S.value + 1;} \\
\text{else} \\
\; \; \; \; \; \; \text{select thread from S.blocked_list and move to ready_list;}
\]

Semaphore/Conditional Critical Region/Monitor. Circle one or more of Sem, CCR, or Mon, as applies. (1.5 pts. each)

4. Sem / CCR / Mon Programmer must code any mutual exclusion on his/her own.
5. Sem / CCR / Mon Provides a signal operation that has the effect of a no-op when the wait queue is empty.

<answers>

3 - Mon only; 4 - Sem and arguably CCR (thus a poor question); 5 - Mon only

Short-Term/Medium-Term/Long-Term Scheduling. Circle one or more of S, M, L, as applies. (1.5 pts. each)

6. S / M / L Is called a swapper.
7. S / M / L Makes CPU allocation decisions.
8. S / M / L Manages the submit queue(s) for longer-running batch (i.e., background) jobs.

<answers>

6 - M only; 7 - S only; 8 - L only
FCFS/RR/MLFQ/SRTN. Circle one or more of F, R, M, S, as applies. (2 pts. each)
10. F / R / M / S Uses time slices.
11. F / R / M / S Requires future knowledge.

<answers>
9 - R,M,S; 10 - R,M; 11 - S only

T/F. Circle one of True or False. (1.5 pts. each)
12. T / F Conditional critical regions can solve any synchronization problem that semaphores can.
13. T / F Monitors can solve any synchronization problem that semaphores can.
14. T / F Signaling a condition variable in a Brinch-Hansen-style monitor has the same scheduling guarantee for a signaled process as found in a Hoare-style monitor (i.e., that it runs next).
15. T / F Signaling a condition variables in pthreads has the same scheduling guarantee for a signaled thread as found in a Hoare-style monitor (i.e., that it runs next).
16. T / F All solutions we studied to the readers/writers synchronization problem allowed writers to starve.
17. T / F RCU (read-copy-update) is a popular alternative to a readers-writers lock and is used in the Linux kernel.

<answers>
all are true except 15 and 16

Use this table for questions 18-21, where you are asked about resource allocation using the Banker’s algorithm.

<table>
<thead>
<tr>
<th>process</th>
<th>max_demand</th>
<th>allocated</th>
<th>remaining_claim</th>
<th>unused_units = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

18. T / F Can safely grant request of P2 for 1 unit.
19. T / F Can safely grant request of P2 for 3 units.
20. T / F Can safely grant request of P3 for 1 unit.
21. T / F Can safely grant request of P3 for 3 units.

<answers>
all are true except 21
22. Identify and correct the errors in this Producer/Consumer with Bounded Buffer. (6 pts.)

```plaintext
var mutex: semaphore(1); (* mutual exclusion of buffer access, initially 1 *)
empty: semaphore(1); (* number of empty slots in buffer, initially 1 *)
full: semaphore(1); (* number of items in buffer, initially 1 *)
buffer[1..10] of item_type; (* buffer with space for ten items *)

procedure producer(i:integer);
begin
  var p_item: item_type;
  p_item = produce();
  wait(mutex);
  wait(mutex);
  wait(empty);
  wait(full);
append(p_item); (* to shared buffer *)
signal(mutex);
signal(empty);
end;
end;

procedure consumer(i:integer);
begin
  var c_item: item_type;
  c_item = take(); (* from shared buffer *)
signal(mutex);
signal(full);
consume(c_item);
end;

<answers>

var mutex: semaphore(1); (* mutual exclusion of buffer access, initially 1 *)
empty: semaphore(1); (* number of empty slots in buffer, initially 1 *)
full: semaphore(1); (* number of items in buffer, initially 1 *)
buffer[1..10] of item_type; (* buffer with space for ten items *)

procedure producer(i:integer);
begin
  var p_item: item_type;
  p_item = produce();
  wait(mutex);
  wait(mutex);
  wait(empty);
  wait(full);
append(p_item); (* to shared buffer *)
signal(mutex);
signal(empty);
end;
end;

procedure consumer(i:integer);
begin
  var c_item: item_type;
  c_item = take(); (* from shared buffer *)
signal(mutex);
signal(full);
consume(c_item);
end;
```

- The `mutex` semaphore variable should be a 1 here instead of its current 1 value.
- The `empty` semaphore variable should be 10 instead of its current 1 value.
- The `full` semaphore variable should be 0 instead of its current 1 value.
- The `buffer[1..10]` line should indicate that the buffer has space for ten items instead of for one item.
- In the `producer` procedure, the order of the semaphore waits on `mutex` and `empty` should be reversed.
- In the `producer` procedure, the `signal(mutex)` statement should be after the `append(p_item)` statement.
- In the `consumer` procedure, the `signal(full)` statement should be after the `take()` statement.
23. Consider the bounded-buffer data structure “buffer” and the skeletons of the procedures “put” and “get”. Add the necessary conditional critical region statements to produce a correctly-synchronized solution of the bounded-buffer problem using conditional critical regions. (6 pts.)

```plaintext
var buffer:
    shared record
        buf: array[0..N-1] of item_type;
        in: integer := 0;
        out: integer := 0;
        count: integer := 0;
    end;

procedure put( input item: item_type )
    begin
        await( count < N ) region buffer
        begin
            buf[in] := item;
            in := (in + 1) mod N;
            count := count + 1;
        end
    end

procedure get( output item: item_type )
    begin
        await( count > 0 ) region buffer
        begin
            item := buf[out];
            out := (out + 1) mod N;
            count := count - 1;
        end
    end

<answer>
var buffer:
    shared record
        buf: array[0..N-1] of item_type;
        in: integer := 0;
        out: integer := 0;
        count: integer := 0;
    end;

procedure put( input item: item_type )
    begin
        await( count < N ) region buffer
        begin
            buf[in] := item;
            in := (in + 1) mod N;
            count := count + 1;
        end
    end

procedure get( output item: item_type )
    begin
        await( count > 0 ) region buffer
        begin
            item := buf[out];
            out := (out + 1) mod N;
            count := count - 1;
        end
    end
```
24. Consider the bounded-buffer monitor “buffer” and the skeletons of the monitor procedures “put” and “get”. Add condition variable declarations and the necessary if-then statements and wait and signal operations on the condition variables to produce a correctly-synchronized solution of the bounded-buffer problem using a Brinch-Hansen-style monitor. (6 pts.)

```
buffer = monitor
    var buf: array[0..N-1] of item_type;
    in: integer;
    out: integer;
    count: integer;

    (* condition variables *)

procedure put( input item: item_type )
begin
    buf[in] := item;
    in := (in + 1) mod N;
    count := count + 1;
end

procedure get( output item: item_type )
begin
    item := buf[out];
    out := (out + 1) mod N;
    count = count - 1;
end
```
buffer = monitor

var
  buf: array[0..N-1] of item_type;
  in: integer;
  out: integer;
  count: integer;

(* condition variables *)

not_full: condition;
not_empty: condition;

procedure put( input item: item_type )
begin
  if( count = N ) then not_full.wait;
  buf[in] := item;
  in := (in + 1) mod N;
  count := count + 1;
  not_empty.signal;
end

procedure get( output item: item_type )
begin
  if( count = 0 ) then not_empty.wait;
  item := buf[out];
  out := (out + 1) mod N;
  count := count – 1;
  not_full.signal;
end

begin
  in := 0;
  out := 0;
  count := 0;
end
25. Give the individual departure times (dep) and turnaround times (trn) for the following three processes using:
   a) First Come, First Served (FCFS);
   b) Round Robin (RR), with a time slice of 1 where preempted jobs go to the back of the queue after any arrivals have been added;
   c) Shortest Remaining-Time Next (SRTN) where any ties (i.e., equal remaining times) are broken in FCFS order; and,
   d) Shortest Job First (SJF) where any ties (i.e., equal service times) are broken in FCFS order.
Assume that zero time is required for a process to arrive, to be added to the queue, or to be dispatched. (24 pts.)

<table>
<thead>
<tr>
<th>process name</th>
<th>arrival time</th>
<th>service time</th>
<th>FCFS dep</th>
<th>FCFS trn</th>
<th>RR dep</th>
<th>RR trn</th>
<th>SRTN dep</th>
<th>SRTN trn</th>
<th>SJF dep</th>
<th>SJF trn</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>5</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>3</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>2</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
</tr>
</tbody>
</table>

<answers>

<table>
<thead>
<tr>
<th>process name</th>
<th>arrival time</th>
<th>service time</th>
<th>FCFS dep</th>
<th>FCFS trn</th>
<th>RR dep</th>
<th>RR trn</th>
<th>SRTN dep</th>
<th>SRTN trn</th>
<th>SJF dep</th>
<th>SJF trn</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>5</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>3</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>2</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
<td>________</td>
</tr>
</tbody>
</table>
26. State the four necessary conditions for deadlock. (8 pts.)

<answer>
1) resources are exclusively held (i.e., not shared)
2) resources cannot be preempted
3) processes hold on to allocated resources while requesting more
4) circular waiting

Extra credit.

X1. Although Java threads, introduced in 1995, were influenced by the monitor concept, identify a (some would say “the”) major difference from monitors. (5 pts.)

<answer>
There are no condition variables and thus only one wait queue per object.

X2. Why do pthreads use a while statement to test a condition prior to calling pthread_cond_wait(), whereas in the original monitor concept an if statement would suffice before invoking a wait operation? (5 pts.)

<answer>
There is no guarantee that a signaled thread will run immediately after the signal. Thus the condition could change between the time the signal is executed and the signaled thread starts running.

(Also, pthread implementations allow for the possibility of “spurious wakeups”, that is, a waiting thread could start running even though it had not been signaled.)