Semaphore/Conditional Critical Region/Monitor. Circle one or more of Sem, CCR, or Mon, as applies. (1.5 pts. each)
1. Sem / CCR / Mon is object-oriented.
2. Sem / CCR / Mon Uses a “condition” data type.
3. Sem / CCR / Mon Can be used to implement the “producer/consumer with bounded buffer” design pattern.

1 is Mon only; 2 is Mon only; 3 is all three

True/False. Circle one of T or F. (2 pts. each)

4. T / F Conditional critical regions have broadcast semantics when a thread exits a critical region. That is, all threads waiting on that region are resumed. Each resumed thread will then retest its await condition.
5. T / F Signaling a condition variable in a Hoare-style monitor has the same scheduling guarantee for a signaled thread as found in a Brinch-Hansen-style monitor (i.e., that it is the next thread to run inside the monitor).
6. T / F Brinch-Hansen-style monitors are coded to allow for spurious wakeups of waiting processes.
7. T / F Hoare-style monitors are coded to allow for spurious wakeups of waiting processes.
8. T / F Early Java versions contained Hoare-style monitors, which Brinch Hansen criticized.

4, 5 are True; 6, 7, 8 are False

9. How does a monitor signal differ from a semaphore signal? Answer in terms of the data structures used. (6 pts.)

A monitor’s condition variable only has a blocked list. A monitor signal moves one thread from the condition variable blocked list to the ready list if the blocked list is nonempty. Otherwise the signal does nothing.
A semaphore has both a value and a blocked list. Like a monitor signal, a semaphore signal moves one thread from the blocked list to the ready list if the blocked list is nonempty. The difference is that when the blocked list is empty, a semaphore signal increments the value.

10. Is the following Brinch-Hansen-style monitor an adequate simulation of a general semaphore, or is there a logical error? Answer in terms of the program logic only. Do not base your answer on possible syntax errors in the Pascal-like code; note that “:=” is assignment and that the “=” in the if-then is the equality operator. (6 pts.)

simulated_general_semaphore = monitor
var value: integer;
    blocked: condition;

procedure entry sem_wait;
begin
    if value = 0 then blocked.wait;
    value := value – 1
end;

procedure entry sem_signal;
begin
    value := value + 1;
    blocked.signal
end;

begin (* monitor initialization section *)
    value := k (* the constant k is the initial value *)
end (* end of monitor *)

This is an adequate simulation.

Without a way to check the empty/nonempty status of the blocked list of the condition variable, the simulation always decrements in sem_wait and always increments in sem_signal.

Since the decrement is done only after the guarding if-then statement in sem_wait, the value can never be negative.

Furthermore, if sem_signal increments value and then the signal resumes a waiting thread, the newly resumed immediately decrements value. Thus there is no net change in value.
11. Consider the bounded-buffer monitor “buffer” and the skeletons of the monitor procedures “put” and “get” (which are arranged on this page to be shown side-by-side). 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. (10 pts.)

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

(* condition variable declarations *)
not_full, not_empty: condition;

procedure entry 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 (* put *)

procedure entry 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 (* get *)

begin (* monitor initialization section *)
in := 0; out := 0; count := 0
end (* end of monitor *)
```

12. Are nested monitor procedure calls susceptible to deadlock? Explain your answer. (4 pts.)

Yes. If a procedure in monitor M1 calls a procedure in monitor M2 and has to wait, it will release mutual exclusion in M2 but not M1. If the corresponding procedure in M2 that signals the waiting thread must be called from a procedure in M1, deadlock can result.

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

13. S / M / L Runs after every interrupt.
14. S / M / L Makes swapping decisions.
15. S / M / L Sets the CPU timer for a time slice.
16. S / M / L Can remove an entry from the ready list.
17. S / M / L Manages the submit queue(s) for longer-running batch (i.e., background) jobs.

13 is S only; 14 is M only; 15 is S only; 16 is S and M; 17 is L only

**FCFS/RR/MLFQ/SRTN.** Circle one or more of F, R, M, S, as applies. (2 pts. each)

18. F / R / M / S Is preemptive.
19. F / R / M / S Does not allow starvation.
20. F / R / M / S Has minimum average response time.

18 is R, M, and S; 19 is F and R; 20 is S only
21. Briefly explain the difference between policy and mechanism. (4 pts.)

Policy is a decision-making rule.
Mechanism is a hardware or software component that is used to implement a policy.

22. Briefly explain the difference between a CPU burst and a time slice. (4 pts.)

A CPU burst is the amount of time a thread will run until completion or a blocking event, if it is not preempted.
A time slice is the amount of time the dispatcher loads into the CPU timer when dispatching a thread. If the thread does not complete or block before the end of the time slice, a timer interrupt will give control of the CPU back to the OS.

23. Briefly explain the difference between a preemptive scheduling policy and a non-preemptive policy. (4 pts.)

A preemptive policy will seize control of the CPU from an executing thread.
A non-preemptive policy allows a thread to run to completion or a blocking event (e.g., I/O burst).

24. Identify at least two ways in which the highest-priority-first (HPF) policy with static priorities differs from the highest-response-ratio-next (HRRN) policy beyond what already appears in the names (i.e., one uses priority and one uses response ratio). Furthermore, simple vs. complex, fast vs. slow, etc., are too vague to identify the differences. Answer in such a way that you demonstrate that you know how each policy works. (4 pts.)

HRRN requires future knowledge, favors short jobs, and prevents starvation (the response ratio increases with waiting time)

25. Give the individual departure times (dep) and turnaround times (trn) for the following three processes using
a) First Come, First Served (FCFS);
b) Shortest Job First (SJF) where any ties (i.e., equal service times) are broken in FCFS order; and,
c) Shortest Remaining-Time Next (SRTN) where any ties (i.e., equal remaining 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. (18 pts.)

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival</th>
<th>Service</th>
<th>FCFS dep</th>
<th>FCFS trn</th>
<th>SJF dep</th>
<th>SJF trn</th>
<th>SRTN dep</th>
<th>SRTN trn</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td>8</td>
<td>10</td>
<td>9</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>1</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

**FCFS**

0 + + + + + + + + + + 6 7 8 9 10
A6 A5 A4 A3 A2 A1 B3 B2 B1 C1
A6 ^A B3 ^C C1

**SJF**

0 + + + + + + + + + + 6 7 8 9 10
A6 A5 A4 A3 A2 A1 C1 B3 B2 B1
C1 ^C B1 ^B

**SRTN**

0 + + + + + + + + + + 6 7 8 9 10
A6 B3 C1 B2 B1 A5 A4 A3 A2 A1
C1 ^A B1 ^B
26. Using the following values, demonstrate that the system is in a safe state by giving a feasible order in which the processes can complete (4 pts.)

<table>
<thead>
<tr>
<th>process</th>
<th>max_demand</th>
<th>allocated</th>
<th>remaining_claim</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>6</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>P4</td>
<td>8</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

Feasible order is P2, P3, P1, and then P4.

Although further detail is not needed, here is a sequence showing how the count of unused units increases as processes can run to completion and then can release their current allocation:

<table>
<thead>
<tr>
<th>unused_units</th>
<th>initially</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2 &gt;= 2</td>
<td>remaining claim of P2. P2 completes and releases 1 unit.</td>
</tr>
<tr>
<td>3</td>
<td>3 &gt;= 3</td>
<td>remaining claim of P3. P3 completes and releases 2 units.</td>
</tr>
<tr>
<td>5</td>
<td>5 &gt;= 4</td>
<td>remaining claim of P1. P1 completes and releases 2 units.</td>
</tr>
<tr>
<td>7</td>
<td>7 &gt;= 6</td>
<td>remaining claim of P4. P4 completes and releases 2 units.</td>
</tr>
</tbody>
</table>

which equals total_units 27

27. State the four necessary conditions for deadlock. (8 pts.)

1. Resources are exclusively held.
2. Resources cannot be preempted.
3. Processes hold on to resources and ask for more.

XC-1. Give the kernel-level implementation of semaphore wait logic, that is, the if-then-else logic covered in class and on quizzes. Assume the code is already inside an interrupt-disabled and spin-lock-protected region. (up to 3 pts.)

XC-2. Give the kernel-level implementation of semaphore signal logic, that is, the if-then-else logic covered in class and on quizzes. Assume the code is already inside an interrupt-disabled and spin-lock-protected region. (up to 3 pts.)

See the notes and/or exam 1 for the answers to XC-1 and XC-2.

XC-3. Below is an attempt at a simple implementation of a Brinch-Hansen-style monitor using semaphores. Do you see any possible problems? (up to 6 pts.)

The semaphores and counts used for the implementation are:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex(1);</td>
<td>//initialized to 1</td>
</tr>
<tr>
<td>condition_sem[N];</td>
<td>// array of N semaphores to match N conditions, all initially 0</td>
</tr>
</tbody>
</table>

The body of each procedure within the monitor becomes:

P(mutex);
*procedure_body*
V(mutex);

Each condition[i].wait becomes:

V(mutex);
P(condition_sem[i]);
P(mutex);

Each condition[i].signal becomes:

V(condition_sem[i]);

For a monitor signal at the end of a procedure, both the condition_sem and the mutex semaphores are signaled. There is no scheduling guarantee that a signaled thread from the condition_sem semaphore will necessarily run before a signaled thread from the mutex semaphore.

If there are no threads on the blocked list of the condition_sem semaphore, the value will be incremented. A subsequent thread that does a monitor wait on that condition will therefore not be blocked and instead allowed to enter the monitor without the desired state being true (e.g., a producer allowed to enter even through the bounded buffer has no empty slots).
XC-4. What is wrong with the following pthread code for a resource allocation function? Look for major pthread condition variable API issues, not minor issues in C programming such as whether a semi-colon is missing. (up to 3 pts.)

#define RESOURCES 10

// resource table
// row is indexed by resource id
// column 0 is busy flag (0=free, 1=in-use), column 1 contains the owning thread id when that resource is busy
int resourceTable[RESOURCES][2] = { {0,0} };

// global count of currently free resources and condition variable for thread waiting when there are no free resources
int availableResourceCount = RESOURCES;
pthread_cond_t resourcesAvailable;
...
pthread_cond_init( &resourcesAvailable, NULL );
...

// allocateResource function
// input argument is thread id of calling thread
// return value is a resource identifier
// this function will block the calling thread if there are no free resources (releaseResource() will signal)
int allocateResource( int tid ){
    int rid;

    *** should have explicit pthread_mutex_lock() here

    *** should be while, not if
    if( availableResourceCount == 0 ) pthread_cond_wait( &resourcesAvailable );

    *** the mutex lock used for mutual exclusion should be passed to pthread_cond_wait() as the second argument
    
    rid = 0;
    while( resourceTable[rid][0] != 0 ) rid++;
    if( rid >= RESOURCES ) { printf("**** error in allocation!\n"); exit(-1); } // error check – condition should never occur!
    resourceTable[rid][0] = 1;
    resourceTable[rid][1] = tid;

    availableResourceCount--;

    *** should have explicit pthread_mutex_unlock() here

    return rid;
}