A context switch occurs when the operating system changes the currently running process. This involves saving all of the state (or context) of the current process into its Process Control Block (PCB) and then restoring the state of the new process. This state includes the control and status registers, such as the program counter, stack pointer, and other general purpose registers.

Two examples of events that can trigger a context switch are a timeout (the current process has used up its allowed time on the CPU) or an I/O event (the current process must be blocked).

A context switch is considered expensive because it is “wasted time” on the CPU, in the sense that no useful work is accomplished. The work spent doing a context switch is the overhead that must be paid for have a multiprogrammed operating system.

Yes. Two kernel-level threads share memory and other process resources, but have their own stack and control block. This means that each thread can operate concurrently on its own processor, using its own program counter, stack and other registers, while still sharing memory and other process resources controlled by the operating system.

A semaphore can suspend processes waiting on a condition, whereas with a hardware test-and-set each process must busy wait. This is generally implemented by the operating system with a FIFO queue, so that the first process suspended is the first process re-activated when the signal occurs.

A monitor is a programming language construct that supports synchronization through a programmer-defined set of methods. The procedures and data of the monitor may only be accessed through its methods, similar to a class in an object-oriented programming language. Mutual exclusion is enforced by allowing only one process at a time to access a method and thus “enter” the monitor.

The monitor still includes wait and signal operations to support synchronized communication between processes. For example, a producer-consumer monitor may want a producer to signal any waiting consumers that a shared buffer is not empty any more.
Section B: Problems (10 points each)

1. Consider the traffic deadlock shown at right. Give four alternatives for breaking or preventing the deadlock, with each alternative violating one of the four necessary conditions for deadlock. Use the spaces below. You have at your disposal a large crane that can safely move individual cars out of the way and a construction crew.

a) Mutual exclusion
   Build an overpass or underpass at the intersection, allowing multiple cars at a time to occupy the area.

b) Hold and wait
   Do not allow a car to enter the intersection unless it can completely clear it.

c) No preemption
   Use the crane to lift a car out of one of the intersections when deadlock occurs.

d) Circular wait
   Do not allow more than three intersections to be blocked at a time.

2. The following is the current resource allocation state for an operating system.

<table>
<thead>
<tr>
<th>Available</th>
<th>Claim</th>
<th>Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1  R2  R3  R4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2   1   0   0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

   a) Use the Banker's Algorithm to show that the system is in a safe state.
      • The only process that can run immediately is P2, since it needs [0 1 0 0].
      • After P2 is done, Available becomes [3 3 2 0].
      • Now only P3 can run, since it needs [2 0 1 0].
      • After P3 is done, Available becomes [3 4 2 1].
      • Now only P1 can run, since it needs [0 0 2 1].
      • After P1 is done, Available becomes [5 5 2 2].
      • Now P4 can run, since it needs [2 1 2 2].
      • After P4 is done, all processes are complete and Available becomes [5 6 2 2].

   b) Give an example of a request that leads to an unsafe state (and potential deadlock). Explain why this request will lead to an unsafe state.
      • In the example shown above, only P2 can run initially, and it needs 1 of resource R2. If P4 requests 1 of R2, then P2 will be unable to run. Hence, P4's request for 1 of R2 will lead to an unsafe state.
Section C: Problems (20 points each)

1. A father and his three daughters work in a restaurant that makes cheese pizzas. Making a pizza requires three ingredients: dough, sauce, and cheese. One daughter has an infinite supply of dough, another has an infinite supply of sauce, and a third has an infinite supply of cheese.

Pizzas are made as follows: The father first places a pizza pan on the counter. The daughter with the dough forms a pizza on the pan, then the second daughter spreads sauce on the pizza and finally the third daughter adds cheese. When the third daughter is done, the father puts the pizza in the oven and then starts the next pizza.

Write code for the father and a general procedure that can be used for any daughter, making sure that the father and daughters are synchronized properly with semaphores.

```java
String ingredient[3];
Semaphore mutex[4];

Procedure father begin
    while (true) begin
        place_pan_on_counter;
        V(mutex[0]);
        P(mutex[3]);
        put_pizza_in_oven;
    end;
end;

Procedure daughter(i) begin
    while (true) begin
        P(mutex[i]);
        do_pizza(ingredient[i]);
        V(mutex[i+1];
    end;
end;

procedure main begin
    ingredient[0] = dough;
    ingredient[1] = sauce;
    ingredient[2] = cheese;
    mutex[0] = 0;
    mutex[1] = 0;
    mutex[2] = 0;
    mutex[3] = 0;
    father.start();
    daughter.start(0);
    daughter.start(1);
    daughter.start(2);
end;
```