A. (20) **Short Answers (be short)**

1. **What happens in a context switch?**

   In a context switch, the state of the current process (including all the registers that the process may be using, such as Program Counter, Stack pointer, etc.) is saved and the saved values of the next process is loaded from its PCB.

2. **What is the relationship between the thread concept and the Java thread controller object?**

   The Java thread controller object is NOT a thread itself; it can be used to control and schedule a thread.

3. **Explain how a CPU instruction such as the test-and-set is the basis for more complex synchronization abstractions?**

   The test-and-set instruction provides mutual exclusive access to a critical section. It is an atomic operation so only one process can be using the test-and-set instruction at a time.

4. **Where are the PCBs for processes stored? i) each process has it in its own memory or ii) the kernel has it in its memory or iii) managed by some user level OS service**

   ii) The kernel has it in its memory.
B. (20 pts) *Processes.*

Develop a simple simulator program for a multiprocessor system with N processors. Follow the following steps: (i) Create a process for every processor; (ii) in each process print your process ID and start executing a program you choose; Show the memory image of one child after the fork() system call.

System calls (slightly simplified from the real ones) that you need to use (arguments also shown for the calls): fork(); execlp(char[] prgname); getpid();

(i)

// Assume N is a defined constant;
// Assume the desired program to be executed is given from the command line (argv);
int main(int args, char* argv[])
{
    for(int i=0; i<N; i++)
        if(fork() == 0)
            {
                printf("Process ID: %d\n", getpid());
                execlp(argv[1]);
            }
}

(ii) Draw a figure as shown in the class slides, show stack pointers, program counter, etc.
C. (20) **Threads.**
(i) Complete the following Java program with a Consumer class that runs in its own thread and consumes values by using the `get()` method of the cubbyhole class. (ii) Modify the program such that you don’t need to extend the `Thread` class in the Producer and Consumer classes to support multiple threads.

```java
public class ProducerConsumerTest {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);

        Thread pThread = new Thread(p1);
        Thread cThread = new Thread(c1);

        p1.start();
        c1.start();

        pThread.start();
        cThread.start();
    }
}

public class Producer extends Thread implements Runnable {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #" + this.number + " put: " + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}

public class CubbyHole {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        available = false;
        return contents;
    }
}
```
public synchronized void put(int value) {
    while (available == true) {
        try {
            wait();
        } catch (InterruptedException e) { }
    }
    contents = value;
    available = true;
    notifyAll();
}

public class Consumer extends Thread implements Runnable {// add your code here
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i=0; i<10; i++) {
            System.out.println("consumer # " + this.number + " get: " + cubbyhole.get());
            try {
                sleep((int)(Math.random()*100));
            } catch (InterruptedException e) {} // for loop
        }
    }
}
D. (20) Solaris Threads.
Explain how Solaris is managing multiple threads. Describe what happens in the user space, what happens in the kernel, what is visible in the kernel, and what the benefits of having lightweight processes are.

In the user space, user-level threads are mapped to lightweight processes in many-to-many style. Each lightweight process maps exactly to one kernel thread; while kernel threads are mapped to the CPUs in many-to-many relations.

In this model, you can map a user-level thread to more than one lightweight process, which allows the thread to continue even if a blocking call was made to one of the kernel threads (mapped to the user-level thread).
E. (20) Synchronization
In the implementation of the monitor abstraction with semaphores there are three queues. One of the queues is not visible to applications. Let us call this queue the invisible queue. In the following implementation identify the way the invisible queue is implemented and explain the reason it is necessary.

<table>
<thead>
<tr>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>semaphore mutex; // (initially = 1)</td>
</tr>
<tr>
<td>semaphore next; // (initially = 0)</td>
</tr>
<tr>
<td>int next-count = 0;</td>
</tr>
</tbody>
</table>

Each external procedure $F$ will be replaced by

\[
\begin{align*}
\text{wait}(mutex); \\
\quad \ldots \\
\text{body of } F; \quad // \text{inside we can have processes calling x.wait and x.signal} \\
\quad \ldots \\
\text{if }(next-count > 0) \\
\quad \quad \text{signal(next);} \\
\text{else} \\
\quad \quad \text{signal(mutex);} \\
\end{align*}
\]

For each condition variable $x$, we have:

<table>
<thead>
<tr>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>semaphore x-sem; // (initially = 0)</td>
</tr>
<tr>
<td>int x-count = 0; // used to count blocked threads</td>
</tr>
</tbody>
</table>

The monitor operation $x.wait$ can be implemented as:

\[
\begin{align*}
x-count++;
\quad \text{if }(next-count > 0) \\
\quad \quad \text{signal(next);} \\
\text{else} \\
\quad \quad \text{signal(mutex);} \\
\quad \quad \text{wait(x-sem);} \\
\quad \quad x-count--; \\
\end{align*}
\]

The monitor operation $x.signal$ can be implemented as:

\[
\begin{align*}
\text{if }(x-count > 0) \{ \\
\quad \text{next-count++;} \\
\quad \quad \text{signal(x-sem);} \\
\quad \quad \text{wait(next);} \\
\quad \quad \text{next-count--;} \\
\}
\end{align*}
\]

The invisible queue is “next”.

The “next” queue is where processes are held when they yield to another process from the conditional queue (x-sem). When a process issues a signal to allow a process from the conditional queue, that process then needs to be suspended: this suspension happens in the “next” queue. The suspended process then gets resumed when either the allowed process performs a “wait” call or when that process finishes.