1. **Short Answer**

a. The Process Control Block (PCB) is the most important data structure managed by the OS. From the following list, circle those items that very likely would be stored in the PCB.

<table>
<thead>
<tr>
<th>PID (process ID)</th>
<th>PC (program counter)</th>
<th>CPU registers</th>
<th>process code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU status info</td>
<td>link to next PCB</td>
<td></td>
<td>process state (ready, blocked, running)</td>
</tr>
<tr>
<td>test-and-set variable</td>
<td>semaphore used by this process</td>
<td>scheduling priority</td>
<td></td>
</tr>
<tr>
<td>previous predicted CPU burst</td>
<td></td>
<td>password for the user that created this process</td>
<td></td>
</tr>
</tbody>
</table>

b. The text and lectures discuss three standard ways of evaluating the performance of operating systems (scheduling algorithms in particular). The three methods are queueing theory analysis, simulation, and empirical experiments. For each of the dimensions of performance evaluation listed below, circle the method which is best for that dimension.

(i) **accuracy** - best in the ability to accurately predict the future performance of the system when deployed

- queueing theory
- simulation  **empirical experiments**

(ii) **cost/implementation overhead** - best (least) amount of design, programming, equipment, and effort to do the performance analysis

- queueing theory
- simulation
- empirical experiments

(iii) **flexibility** - maximum ability to model a wide variety of algorithms and workloads (not taking cost into account)

- queueing theory
- simulation  **empirical experiments**
c. Leslie Lamport developed the Bakery Algorithm. What problem does it solve?

*n-process software solution to the mutual exclusion/critical section problem*

d. Name two advantages of semaphores.

*\underline{Simplicity through abstraction}*

*Less error prone than software solutions*

*Efficient – implemented with little busywaiting; or no busywaiting (blocking semaphores)*

*Powerful – can solve critical section problem as well as a variety of n process synchronization problems*

e. How is atomicity of the wait(0 and signal()) kernel code guaranteed?

*By enclosing the code for wait and signal with the test-and-set code (assuming atomic hardware instruction test-and-set)*

```c
While(Test-and-set(&lock) ) /* busywait */
    Code for wait() or code for signal();
lock = FALSE;
```

f. How is the atomicity of the test-and-set() hardware instruction guaranteed?

*The hardware guarantees test-and-set to be indivisible/atomic relative to other hardware instructions. When the critical section problem first appeared in the 60s, the designers of computer hardware realized they had to provide such an instruction.*

g. What is the difference between the Unix fork and exec kernel calls?

*fork() creates a child process that co-exists with the parent (creator). The child runs a copy of the parent’s code.*

*exec() replaces the code of the process that calls it with the code passed to exec() in its parameter list. exec() does not create a new process.*

h. Write the formula for normalized turnaround time. What is the theoretical minimum value of normalized turnaround time?

\[
NTT = \frac{\text{turnaround}}{\text{runtime}} = \frac{\text{(waittime + runtime)}}{\text{runtime}}
\]

*Theoretical minimum = 1 (when there is zero waittime)*
2. Process State Transitions
In all these problems assume a multiprogrammed uniprocessor system with several processes running along with the one that the problem focuses on.

a. Indicate what state(s) a process occupies if it is starved by the SJF algorithm by drawing arrows to show the state transitions. If only one state, just circle that state.

```
ready
running
blocked
```

b. Indicate what state(s) a process occupies if it is livelocked by a faulty busywait solution to the critical section problem, by drawing arrows to show the state transitions. If only one state, just circle that state.

```
ready
running
blocked
```

c. Indicate what state(s) a process occupies if it is deadlocked by a faulty solution to the critical section problem that uses blocking semaphores. Draw arrows to show the transitions or if only one state, just circle that state.

```
ready
running
blocked
```

d. Indicate what state(s) a process occupies immediately before and after it executes wait(sema) when the semaphore value is 0. Draw states arrows to show the state transitions or if only one state, just circle that state.

```
ready
running
blocked
```
e. Indicate what state(s) a process could occupy while in critical section by listing those states.

- ready
- running
- blocked

3. Exponential Averaging

The Sun Computer Company OS uses exponential averaging with alpha = 0.1 to predict the next CPU burst for its SJF scheduling algorithm. The Moon Computer OS uses a technique in which the next predicted CPU burst is simply the average of all previous actual bursts. More precisely,

Sun Company’s formula: \[ \text{Tau}(n+1) = \alpha \times t(n) + (1-\alpha) \text{Tau}(n) \]  
Moon Company’s formula: \[ \text{Tau}(n+1) = \frac{1}{n} \times \sum_{i=1}^{n} t(i) \]

a. In the computation of the fifth predicted burst Tau(5), how much weight is given to \( t(2) \) under each system? Circle and label final answers. Show work for partial credit.

SUN: \( (0.9)^2(0.1) = 0.081 \)
MOON: \( \frac{1}{4} \) (the average of four previous bursts)

Show work: \[ TAU(5) = (0.1)\times t(4) + (0.9)\times TAU(4) \]
\[ = (0.1)\times t(4) + (0.9) \left[ (0.1)\times t(3) + (0.9)\times TAU(3) \right] \]
\[ = (0.1)\times t(4) + (0.9) \left[ (0.1)\times t(3) + (0.9) \left[ (0.1)\times t(2) + (0.9)\times TAU(2) \right] \right] \]

b. How many bursts must be completed before Sun’s exponential average system gives more weight to \( t(n) \) in Tau(n+1) than Moon’s straight average system? In other words, what is the smallest value of \( n \) for which this occurs? Your answer can be a formula. Circle and label final answers. Show work for partial credit.

SUN: weight on \( t(n) \) in TAU(n+1) is 0.1
MOON: weight on \( t(n) \) in TAU(n+1) is \( 1/n \)
Find smallest value of \( n \) for which \( 1/10 > 1/n \). Answer = 11 bursts.
Using Semaphores for Registration in CIS 415 (10 pts)

Define appropriate semaphores and write code using wait() and signal to coordinate students’ code so that at most 30 students take CIS 415 simultaneously. In addition, your code must ensure that the student has completed CIS 314 and CIS 313 before registering for CIS 415.

Semaphore cis415 = 30;
Array of semaphores cis314done[MAXSTUDENTS] = 0;
Array of semaphores cis313done[MAXSTUDENTS] = 0;

Code for student[i]

TAKE_CIS415() {
    P(cis313done[i]);
P(cis314done[i]);
P(CIS415);
    Listen to Ginnie Lo blah blah;
    Eat a snack;
    Learn a little;
    Take a test or two;
    V(CIS415);
}

TAKE CIS314() {
    /* assume no enrollment constraint in CIS 314 */
    Listen to Eric Wills blah blah;
    Learn a little;
    Take a test or two;
    V(cis314done);
}

TAKE CIS313() {
    /* assume no enrollment constraint in CIS 313 */
    Listen to Chris Wilson blah blah;
    Learn a little;
    Take a test or two;
    V(cis313done);
}
6. Synchronization and Scheduling (20 pts)

Write pseudo-code for a Scheduler that gathers statistics on how long a process is blocked waiting on semaphores. **For simplicity, assume a given process only uses one semaphore during its lifetime.** Use the data structures and pre-defined procedures given to you in Homework 1 (reproduced for you on the yellow sheet).

Your code should accomplish the following:

- Whenever a process blocks on a semaphore, compute how long it stays blocked on that semaphore, and write the information to the statistics file (semaphore name and time blocked before signaled)
- When the process exits (becomes a zombie) print out the total time blocked on the semaphore over the lifetime of the process.
- Assume that the scheduler is passed the following information when called from a wait() or signal() kernel call: the name of the semaphore and the process ID of the calling process.

The basic code has been provided for you. You are to add your new code in the spaces provided. You need not use all the lines provided (the purpose of the lines is to keep your answers neat so I can grade them more easily). Add comments next to your code to help explain your solution.

You are free to define any new fields in the PCB, any new global variables or any new procedures.. Define these here, give initial values, give short explanation:

---

- **PCB.startblock** – time a process starts blocking on semaphore
- **PCB.totalblocked** – total time a process is blocked on the semaphore (initially 0)
- **PCB.sema** – the ID of the semaphore
- **int blocked**; to compute recent blockage time
- **writestats(PID, SEMA, time)** – write recent increment of blocked time to file

---
Scheduler () {

/* The cases for I/O routine call, I/O interrupt handler have been omitted */

If (called by wait synchronization kernel call) {
    /* This call was made by process PID on semaphore SEMA to block the process */

    Running_Process.startblock = readtime();

    Running_Process.sema = SEMA;

    Append(Running_Process, Wait_Queue);
    Running_Process = Head(Ready_Queue); /* get next process
    start_time = read_time(); /* record its start time
    if (Running_Process != NIL) Context_Switch (Running_Process, Tau);
    else IDLE(); /* start next process or idle until next interrupt

} else if (called by signal(sema) kernel call) {
    /* This call was made by process PID on semaphore SEMA to unblock a process waiting on SEMA */

    PCB = findprocess(SEMA); /* find a process blocked on SEMA */

    blocked = readtime() – PCB.startblock();

    writestats(PCB.PID, PCB.sema, blocked); /* length of recent blockage on SEMA */

    PCB.totalblocked += blocked;

    Append(Running_Process, Ready_Queue); /* append the signaling process to RQ;

    *** Note: the next statement appends the unblocked process to the RQ*/

    Append(PCB,Ready_Queue); /* put it in Ready_Queue
    Running_Process = Head(Ready_Queue); /* get next process
    start_time = read_time(); /* record its start time
    if (Running_Process != NIL) Context_Switch (Running_Process, Tau);
    else IDLE(); /* start next process or idle until next interrupt

}
} else if (called by timer interrupt handler) {
    /* after time quantum expired
     * not relevant to process blocked on semaphore */
    
    Append(Running_Process, ReadyQueue);  /* time quantum is up
    Running_Process = Head(ReadyQueue);   /* get next process
    start_time = read_time();            /* record its start time
    if (Running_Process != NIL) Context_Switch(Running_Process, Tau);
    else IDLE(); /* start next process or idle until next interrupt

} else if (called by finished process) {
    /* process is done
     * write the total synch block time to the statistics file */
    writestats(Running_Process.PID, Running_Process.sema,
                Running_Process.totalblocked);

    Append(Running_Process, Zombie_Queue);  /* process is now dead
    Running_Process = Head(ReadyQueue);    /* get next process
    start_time = read_time();              /* record its start time
    if (Running_Process != NIL) Context_Switch(Running_Process, Tau);
    else IDLE(); /* start next process or idle until next interrupt

}