Part 1: (60 points - 4 points for each problem)

1. Which one is the advantage of multiprocessor systems?
   (b) Increased reliability
   (a) Increased modularity  (c) Increased security  (d) None of above

2. System calls to allocate and free memory belongs to which of the operating system service category below:
   (d) Process control
   (a) Device manipulation  (b) File manipulation  (c) Information maintenance

3. When the currently running process issues a system call for I/O:
   (c) It is marked blocked and its PCB added to the tail of the device queue associated with the I/O request.
   (d) It is added to an I/O queue also referred to as blocked list
   (a) It is added to the tail of the ready queue to do its I/O.
   (b) Its PCB is entered into the ready queue for later execution.

4. The scheduler that brings processes into memory and swaps them out on disk as needed is referred to as:
   (c) Medium-term scheduler  (a) Short-term scheduler  (b) Long-term scheduler  (d) None of the above

5. When we have a message-passing scheme between processes P and Q as: send (P, message) and receive (Q, message), this has the following property:
   (d) The processes do not need to know each other’s identities.
   (a) The link may be unidirectional or bidirectional.
   (b) The link is associated with only one process.
   (c) There can be more than one link between a pair of processes.

6. Which is not the property of a signal in UNIX systems?
   (d) None of the above.
   (a) A signal is generated by the occurrence of a particular event.
   (b) A generated signal is delivered to a process.
   (c) Once delivered, the signal must be handled.

7. In an operating system a utility which lets the users issue and execute commands from the keyboard is called:
   (b) Command Interpreter  (a) Terminal Handler  (c) Kernel  (d) None of the above

8. Which is not a primary reason for providing an environment that allows process cooperation:
   (c) Concurrency  (b) Computational speedup  (a) Modularity  (d) Information sharing
9. The primary difference between user-level threads and kernel threads is:
(a) User level threads do not use OS services via system calls, where kernel threads
require system calls.
(b) User level threads are independent of each other, whereas kernel threads can
write into each other’s memory space.
(c) User level threads require memory management where kernel threads do not.
(d) None of above.

10. CPU burst distribution is generally characterized as
(a) Constant  (b) Linear
(c) Polynomial  (d) Exponential or hyper-exponential

11. Which is not the function of a dispatcher?
(a) Switching context  (b) Switching to use mode
(c) Jumping to the proper location in the user program to restart that program
(d) None of above

12. Which is not a CPU scheduling criterion?
(a) CPU utilization  (b) Burst time  (c) Throughput  (d) Response time

13. Which is a preemptive scheduling?
(a) RR  (b) FCFS  (c) SJF  (d) None of the above

14. If the time quantum is very large, a RR (Round-Robin) scheduling is the same as:
(a) FCFS  (b) SJF  (c) SRTF  (d) multilevel queue

15. Which is not a solution the critical-section problem?
(a) Conditional critical region  (b) Shared memory  (c) Monitor  (d) Semaphore

Part 2: (90 points)

1. Describe the process status. (10 points)
   - new: The process is being created.
   - running: Instructions are being executed.
   - waiting: The process is waiting for some event to occur.
   - ready: The process is waiting to be assigned to a process.
   - terminated: The process has finished execution.

2. What is a thread? What are the benefits of using threads? (10 points)
   1. A thread, a lightweight process (LWP), is a basic unit of CPU
      utilization. It shares with other threads within the same process its
      code, data, open files and signals.
   2. Benefits:
      - Responsiveness: Multithreading an interactive application may allow
        a program to continue running even if part of it is blocked or is
        performing a lengthy operation, thereby increasing responsiveness to
the user.

- Resource Sharing: Threads share the memory and the resources of the process to which they belong.
- Economy: Allocating memory and resources for process creation is costly.
- Utilization of MP (multiprocessor) Architectures: Each thread may be running in parallel on a different processor.

3. What is a race condition? How can we prevent a race condition? (10 points)

1. The race condition is the situation where several processes access and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
2. To prevent race conditions, concurrent processes must be synchronized.

4. How many processes will be created when the following program is executed? Assume that all fork system calls are successful. What will be printed? (Hint: Be careful and draw a picture.) (12 points)

```c
main()
{
    int i=1;
    int ret_val = 1;

    while(i <= 2)
    {
        if (ret_val == 0) { /* Child’s code */
            printf("In child %d. \n", i);
            exit(0);
        } else { /* Parent’s code */
            fork();
            printf("In parent %d. \n", i);
            i = i + 1;
        }
    }
}
```

There are 4 processes (1 parent and 3 child processes) created when this program is executed. The following could be printed:

In parent 1.
In parent 1.
In parent 2.
In parent 2.
In parent 2.
5. Add the semaphores necessary to synchronize processes A, B, C, D, and E so that process A must finish executing before B starts, process B must finish before C or D starts, and process D must finish before process E starts. Show your solution. Remember to indicate the initial value of each semaphore. (12 points)

The relations among processes can be represented in the following diagrams.

```
S2 / C
  S1 /
A ---- B
  \    S4
  S3 \ D ---- E
```

The program can be as follows:

```c
semaphore S1, S2, S3, S4 = 0, 0, 0, 0;

Process A:
------------
- do work of A
signal(S1); /* Let B start */

Process B:
------------
wait(S1); /* Block until A is finished */
- do work of B
signal(S2); /* Let C start */
signal(S3); /* Let D start */

Process C:
------------
wait(S2); /* Block until B is finished */
- do work of C

Process D:
------------
wait(S3); /* Block until B is finished */
- do work of D
signal(S4); /* Let E start */

Process E:
------------
wait(S4); /* Block until D is finished */
- do work of E
```

6. Suppose that the following processes arrive for execution at time 0 in the order A, B, C: (12 points)
<table>
<thead>
<tr>
<th>Process</th>
<th>Run Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>1=high</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>3=low</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

(a) Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a nonpreemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1) scheduling.

(b) What is the waiting time of each process for each of the scheduling algorithms?

(c) What is the turnaround time of each process for each of the scheduling algorithms?

a. The four Gantt charts are

```
0 2 5 6 0 1 3 6
+-+---+-+ +-+--+---+
| A| B |C| FCFS |C| A| B | SJF
+-+---+-+ +-+--+---+

0 2 3 6 0 1 2 3 4 6
+-+---+-+ +-+-----------+
|A |C| B | Priority |A|B|C|A| B| RR
+-+---+-+ +-+-----------+
```

b. Waiting time:

<table>
<thead>
<tr>
<th></th>
<th>FCFS</th>
<th>SJF</th>
<th>Priority</th>
<th>RR</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

c. Turnaround time:

<table>
<thead>
<tr>
<th></th>
<th>FCFS</th>
<th>SJF</th>
<th>Priority</th>
<th>RR</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

7. Consider the following semaphore definition: A semaphore S is an integer variable that can only be accessed via two indivisible (atomic) operations: (12 points)

```c
wait(S): while (S <= 0) /* do nothing */ ;
        S = S - 1;

signal(S): S = S + 1;
```
A solution to the mutual exclusion problem, using these operations is shown below:

shared int S = 1; /* S is initialized to 1 */

Each process executes the following code:

```
while (TRUE) {
    wait(S);
    // in critical section
    signal(S);
}
```

(a) Does the above solution involve any busy-waiting? Explain.

(b) What is the "priority inversion" problem?

(c) If priority scheduling is used, can the above solution lead to the priority inversion problem? If so, give an example; otherwise, explain why not.

a. This semaphore definition involves the busy waiting because a waiting process is looping for the shared variable S.

b. A priority inversion problem is a situation in which a low-priority process is blocking a high-priority process.

c. Yes, the above solution can lead to the priority inversion problem if a high-priority process is waiting for a low-priority process to leave the critical section and signal the shared variable S.

8. Consider the following proposed solution to the dining philosophers problem:

semaphore fork[5] = 1,1,1,1,1; /* each semaphore is initialized to 1 */

philosopher (int i) /* philosopher i=0,1,2,3,4 executes this code */
{
    while(1) {
        think();

        DOWN(fork[i]);
        DOWN(fork[(i+1)%5]);

        eat();

        UP(fork[(i+1)%5]);
        UP(fork[i]);
    }
}
Briefly explain how the above solution may lead to deadlock. In the above solution, add a single semaphore to make the solution deadlock free. Be sure to indicate the initial value of the semaphore. (12 points)

a. Suppose all philosophers execute the first wait/DOWN operation, before any have a chance to execute the second wait/DOWN operation; that is, they all grab one fork (chopstick). Then, deadlock will occur and no philosophers will be able to proceed. This is called a circular wait.

b.

semaphore fork[5] = 1,1,1,1,1; /* each semaphore is initialized to 1 */
semaphore count = 4; /* Number of philosophers who can grab their forks */

philosopher (int i) /* philosopher i=0,1,2,3,4 executes this code */
{
    while(1) {

        think();

        DOWN(count);
        DOWN(fork[i]);
        DOWN(fork[(i+1)%5]);

        eat();

        UP(fork[(i+1)%5]);
        UP(fork[i]);
        UP(count);
    }
}