Question 1 (5 points): A computer system has 1GB of memory. The kernel takes 200MB of the memory and the remaining memory is allocated to jobs (processes). Assume that each job requires 200MB of memory. This system can have 4 jobs in memory, i.e. the degree of multiprogramming it can support is 4. If each job (process) spends 80% of time waiting for I/O, what is the CPU utilization? Now suppose that we add an additional 1GB of memory, thus enabling a degree of multiprogramming of 9. What will be the CPU utilization for this new system configuration?

Question 2 (4 points): For each of the following three cases, identify the conditions under which the scheduler will change the status of a process:
   a. Running to Ready
   b. Swapped to Running
   c. Running to Waiting (Blocked)
   d. Ready or Waiting to Swapped

Question 3 (3 points): Why is a separate stack in the kernel memory space used for handling system call functions and interrupt handlers for a process, instead of using the process stack?

Question 4 (2 points): Select the correct answer and briefly justify your answer.
When a process is executing in the user-mode, what is the state of its kernel stack?
(a) The kernel stack for the process is always empty.
(b) The kernel stack for the process is always non-empty.
(c) It is not possible to assert either (a) or (b) will always hold.

Question 5 (6 points):
When a UNIX process executes fork(), does the child process inherit
   a. any pending signals of the parent?
   b. the signal handlers of the parent process?
   c. the signal mask of the parent?
Explain why/why not in each case.

Question 6 (5 points): Read Implementation of Processes & Threads in Linux from the textbook Modern Operating Systems (Chapter 10, Section 10.3.3).
Consider a multithreaded process in which one of the threads executes a call to fork(). Discuss what happens after this call to fork(). Your answer must include points which cover file descriptors and mutex locks.

Question 7 (20 points): Consider a system that has five jobs, with respective processing time requirements of 5, 2, 4, 1, and 3 units. Now consider a system that processes these jobs using the processor sharing discipline (e.g., the RR time-quantum is extremely small).
   (i) What is the average turnaround time for the jobs in this system?
   (ii) What is the average waiting time for the jobs?
   (iii) What is the throughput of this system?
   (iv) What is the minimum value for the turnaround time for a job in this system?
   (v) What is the maximum value for the turnaround time for a job in this system?
Now consider the same system but with the scheduling discipline changed to FCFS. Answer the above six questions for this system for the worst case and the best case condition in terms of the
average turnaround time for the jobs. (For this you will need to identify the order of these five jobs
in the FCFS queue.)

Question 8 (20 points): Measurements of a certain system have shown that the average process runs for a time T before blocking for I/O. A process switch requires a time of S, which is effectively wasted (overhead). For round-robin scheduling with quantum Q, give a formula for CPU efficiency for each of the following cases:
   a) Q = ∞
   b) Q > T
   c) S < Q < T
   d) Q = S
   e) Q nearly 0

Question 9 (10 points): Consider a real-time system with three periodic tasks. Task A has period of 75 units and requires 15 units of processing time. Task B has period 50 units and requires 20 units of processing time. Task C has period of 100 units and requires 20 units of processing time. Suppose that these tasks are all arrive at time 0, and they are scheduled with static priority using the Rate Monotonic Scheduling (RMS) model.
   a. Is the RMS condition for guaranteeing the existence of a feasible schedule satisfied by this system?
   b. Does a feasible schedule exist when using RM based scheduling of the tasks in this system? If yes, then show a feasible schedule, otherwise show a case where a task will miss its deadline.

Question 10 (15 points): Consider the following unreliable implementation of sleep() given in the book by Stevens and Rago (available online), Chapter 10, Figure 10.7

```
#include <signal.h>
#include <unistd.h>

static void
sig_alarm(int signo)
{
    /* nothing to do, just return to wake up the pause */
}

unsigned int
sleep1(unsigned int seconds)
{
    if (signal(SIGALRM, sig_alarm) == SIG_ERR)
        return(seconds);
    alarm(seconds);    /* start the timer */
    pause();           /* next caught signal wakes us up */
    return(alarm(0));  /* turn off timer, return unslept time */
}
```

The authors outline three problems with this code. A more reliable implementation of sleep() is given in Chapter 10, Figure 10.29. Which of the problems mentioned earlier are now corrected in this code?

With this more reliable implementation of sleep(), explain what happens in the following case: We call alarm(10) and 3 wall-clock seconds later, do a sleep(5). What happens now? The sleep() call will return in 5 seconds (assuming that some other signal is not caught during that time), but will another SIGALRM be generated 2 seconds later? If yes, explain how the code will ensure delivery of SIGALRM for the original alarm(10) call.
If no, then modify this code to achieve the same. Also, the originally intended disposition for SIGALRM should be preserved.

**Question 11 (10 points):** The goal of this question is to help you understand the implementation of coroutines in C using `setjmp()` and `longjmp()`. To solve this question, you **MUST** read section 7.10 in chapter 7 of the Stevens and Rago book (available online).

Consider the attached C program, `bampow.c`. When this program is compiled and executed, the following output can be observed:

```
$ gcc -o bampow bampow.c
$ ./bampow
[1]: bam! pow!
[2]: bam! pow!
[3]: bam! pow!
[4]: bam! pow!
[5]: bam! pow!
$
```

The provided program implements a simple coroutine, where control jumps back and forth between two functions. Each function prints a single word and then jumps to the other function. This continues back and forth until each function has executed 5 times.

Your task is to extend this program so that control jumps back and forth amongst THREE functions, instead of TWO. You have to implement the `my_wow()` function. You are free to make changes to the other parts of the code. However, you will not get any points if your code does not use `setjmp()` and `longjmp()` as intended.

Your output should look like this:
```
$ ./bampow
[1]: bam! pow! wow!
[2]: bam! pow! wow!
[3]: bam! pow! wow!
[4]: bam! pow! wow!
[5]: bam! pow! wow!
$
```

**Submission instructions:**
Submit an archive file (.tar, .gzip, .zip or .rar) containing
1. A PDF document for your solutions to questions 1-10.
2. Your modified version of `bampow.c`