INSTRUCTIONS:

1. There are a total of 4 problems on 10 pages worth a total of 100 points. Please take a moment to make sure that your test is complete and readable.

2. SHOW YOUR WORK. Partial credit is more easily awarded in these cases.

3. No questions will be answered during the examination. If you have any questions, please make reasonable assumptions, state your assumptions and proceed with the solution.

4. Please sign the statement below, before you begin.

HONOR CODE:

By signing my name below I hereby certify that I can neither give, nor receive assistance in completing this examination.
**Problem P1 [20 points]: Real-Time Scheduling**

A preemptive foreground/background system has three interrupt-driven cycles, shown below (ignore context switch time):

<table>
<thead>
<tr>
<th>Task</th>
<th>Cycle</th>
<th>Actual Execution Time</th>
<th>Priority ((1 is lowest)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10 ms</td>
<td>4 ms</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>20 ms</td>
<td>5 ms</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>40 ms</td>
<td>10 ms</td>
<td>2</td>
</tr>
<tr>
<td>Background</td>
<td>5 ms</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

(a) Draw an execution timeline for this system.
(b) What is the system time-loading (CPU utilization factor)?
(c) If we now consider context switch time of 1 ms, redraw the execution timeline.
(d) What is the system time-loading factor with the context switch time? *(Hint: utilization depends upon what happens to the task that misses its deadline. Assume that the task missing its deadline is aborted.)*

*The problem asks you to draw the schedule timeline. The 40 ms interval task misses its deadline. Utilization factor depends upon what happens to the task that misses its deadline. If the task missing the deadline is aborted, utilization without background is 33/40, else it is 0.95.*
Problem P2 [25 points]: Max data rate on interrupt-driven IO

In Homework 5, you attempted the problem of determining maximum data rate in case of polled waiting loops. The answer was a max data rate of 2.8 MB/second. Under the same assumptions related to the choice of the process, memory speed etc, let us now consider the problem of determining maximum data transfer rates in interrupt driven IO operations.

In case of interrupt-driven IO, the timing of interrupt driven transfers depends upon the time to execute the longest instruction, hardware response time, time to execute the ISR.

Assuming an ISR shown below, compute number of memory cycles, IO operations to determine the ISR time. Using this compute the maximum data transfer rate and how does it compare to the answer in Problem P4 in HW5.

Assume longest instruction latency to be 540 ns and hardware response time of 450 ns. To the first order you can assume that the ‘clean up’ code is the same size as the code needed to get to the first data.

;; Sample ISR code: a programmable interrupt controller that implements interrupt priority, actually receives the interrupt and forwards to the CPU if the interrupts are enabled (STI) along with interrupt type code on the data bus for the CPU to reader.

_Serial_Input_ISR:
STI ; enable higher priority interrupts
PUSH EAX ; preserve current contents
PUSH EDX ; of EAX and EDX
MOV DX, 02FDh ; Retrieve data and clear the request
IN AL, DX

; Latency ends here. IO port has been read
; operations after this are clean up operations after the IO has been done.
MOV [_serial_data], AL ; save data
MOV AL, 00100000b ; send EOI command to interrupt controller
OUT 20h, AL
POP EDX
POP EAX
IRET

[Hint: determine number memory operations from the ISR including stack operations, number of IO operations, then determine the latency from interrupt request to execution of ISR.]
From the ISR, 4 memory cycles and 1 IO read occurs before the data is retrieved from the serial device.

Thus, the total latency from interrupt request to service is $0.54 + 0.45 + 4 \times 0.06 + 0.03 = 1.26$ micro seconds.

The maximum transfer rate is determined by the time required to handle each interrupt. The complete ISR requires 15 memory cycles, 2 IO transfers or about 0.96 microseconds. For a total transfer time of 1.95 microseconds, the maximum data transfer rate is 513 KB/seconds. This is several times slower than the polled waiting loop, but has the advantage of a predictable latency.
**Problem P3 [10, 15, 10 points]: Priority Inversion**

Consider a multithreaded implementation in which each task is implemented as a thread. Tasks can have static or dynamic priorities depending upon scheduling chosen. High priority threads have precedence over low-priority threads.

a. Explain what is *priority inversion*? Under what conditions is a priority inversion *bounded*, that is, blocking time of the high priority thread is bounded (by the critical section duration of the lower priority thread that owns the resource)? Under what conditions is a priority inversion *unbounded*?

*Priority inversion occurs when a higher priority task (thread) has to wait for the completion of a lower priority task that shares its resource. This priority inversion is bounded because its duration lasts only as long as the critical section of the lower priority thread/task that owns the needed resource.*

*Unbounded priority inversion occurs when the lower priority task blocking a resource is preempted by intermediate priority tasks thus delaying the original high priority task. Since there can be many such intermediate priority tasks, the higher priority task can be delayed without a priori bound.*
Priority Inheritance Protocol (PIP) and Priority Ceiling Protocols (PCP) are two ways to address priority inversion. When a high-priority thread attempts to lock a mutex already locked by a lower-priority thread, the PIP temporarily raises the priority of the low-priority thread to match that of the blocked thread until the low-priority thread unlock the mutex. PCP immediately raises the priority of the low priority thread when it locks the mutex, rather than waiting for a subsequent lock attempt by a higher-priority thread.

a. Comment on the differences in implementation of PIP versus PCP. For instance, how are the two different in terms of who does the priority related work: application or the RTOS kernel? In other words, which of these can be implemented without application or without kernel participation. Entirely in application code or entirely in the kernel code?

**PIP can be transparent to the application, that is, it can be implemented entirely into the kernel. To acquire a resource, the thread must make a kernel call to a mutex associated with that resource. So, when a high-priority thread attempts to lock the mutex already locked by a lower-priority thread, PIP can temporarily raise the priority – entirely in the kernel as a part of kernel handling of the mutex – of the low-priority thread to match that of the blocked thread until the low-priority thread unlocks the mutex (again by a kernel call to release the mutex).**

Keep in mind that priority inversion also happen when critical sections are protected by a semaphore (instead of a mutex). In this case, the semaphores are a bit more complex since they must maintain record of who has acquired or tried to acquire the semaphore. Most kernels do not support PIP with semaphores.

**As mentioned in the problem statement, in case of PCP, the priority of the low-priority thread is raised immediately when it locks the mutex, rather than waiting for a subsequent lock attempt by a higher-priority thread. This can be easily implemented within the application, simply by adding a call to the kernel to raise the priority before locking the mutex (and another to correspondingly lower the priority upon releasing the mutex). Clearly, in case of PCP, the priority ceiling value must be known ahead of time within the mutex, and this value must be highest among all the threads that attempt to lock the same mutex.**

**PCP can also be implemented transparently to the application. In this case, the priority ceiling value is provided as a calling parameter when the associated mutex is created.**
b. The Mars pathfinder software included three threads: a high-priority bus
management thread designed to run quickly and frequently, a medium priority
communications thread that ran for a much longer time, and a low-priority
meteorological thread that ran infrequently. They all shared the shared onboard bus.
Explain how this could lead to a priority inversion? Explain using a timing diagram
if that is helpful.

*To publish its data, the low-priority meteorological thread had to acquire the (shared) bus. In the meantime, the communications thread woke and preempted the meteorological thread. Finally, the management thread woke up, but it was blocked since it could not acquire the bus.*

*When the high priority management thread couldn’t meet its deadline, an alarm went off that reinitialized the computer via a hardware reset.*
Problem P4 [20 points]: Watchdog Timers and Priority Inversion

We have earlier considered the problem of how an RTOS kernel can monitor deadline misses. Let us consider one possible alternative using a watchdog timer in case of a multithreaded application.

A Watchdog timer is simply a hardware counter that counts down towards zero at a fixed rate. The kernel software is expected to periodically restart the counter so that it never really reaches zero. If the counter does reach zero, a failure is assumed to have occurred. The timer can then send a reset signal to the CPU and/or put the CPU in a safe state prior to doing so.

Clearly, in real-time multithreaded applications, it is important that the watchdog timer verify more than just the fact that the CPU is executing instructions. One can restart the watchdog timer by adding a few lines of code to the system timer tick’s ISR. When triggered by some event, the time a task requires to compute the corresponding output response must be less than any specified deadline. Thus, the kernel task to restart the Watchdog time must do so only after verifying that all such deadlines have been met.

One way to do this is by associating an integer ‘deadline’ and two Boolean flags (‘busy’ and ‘failed’) with each TASK as shown below. All flags are cleared during initialization. Whenever an event occurs that triggers the execution of a task, the event computes the future deadline for the task, records it and sets ‘busy’ to TRUE, and then makes the kernel call that releases the pending task (see Task Trigger below).

### Task Trigger (ISR or another thread)

```c
....
deadline[TASK] = current_time() + LIMIT;
busy[TASK] = TRUE;
SemaphorePost(...);
...
```

Once the task has completed computing the output response, it sets its failed flag to TRUE if it has missed the deadline, and it unconditionally clears its busy flag to FALSE. See below:

### Task to compute output response

```c
while (TRUE) {
    SemaphorePending(...);

    /* Compute output response */

    if (current_time() > deadline[TASK])
        failed[TASK] = TRUE;

    busy[TASK] = FALSE;
}
```
Write the code for the kernel software that periodically checks each task. A task has missed a deadline if either its failed flag is TRUE or its busy flag is TRUE and the current time is greater than its posted deadline. If none of the tasks has failed, the watchdog software restarts the hardware counter. Comment on the priority at which you would run this Watchdog checking software. Assume that the total number of tasks is statically known and available as a variable “NoOfTasks”.

```c
while (TRUE) {
    int now, task;
    sleep (... /* for less than counter period */);
    now = current_time();
    for (task=0; task < NoOfTasks; task++) {
        If (failed[task]) break;
        If (busy[task] && now > deadline[task]) break;
    }
    if (task == NoOfTasks) restart_counter();
}
```