Solutions for December 2000 Examinations

Q1a A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints are not met, system failure is said to have occurred. It is essential that the timing constraints of the system are guaranteed to be met i.e. the system is predictable.

A soft real-time system is one where failure to meet its deadlines does not result in system failure, in contrast with a hard real-time system failure to meet its deadlines results in catastrophic system failure.

Q1b In a real-time system each task or process can be periodic, run at intervals, or aperiodic, event driven. It is the job of the scheduler to decide which task to run at a particular time. The decision is usually arrived at based upon some scheduling algorithm. The scheduler is also responsible for controlling the transitions of the tasks between states of running, blocked or ready. It maintains lists of ready and blocked tasks and carries out the task switching function where the execution of the currently running task is stopped and next task on the ready list is started.

Q1c The round robin scheduling algorithm is a simple non real-time algorithm. In round robin, each process is assigned a time interval called its *quantum*, during which it is allowed to run. If the process is still running at the end of its quantum, it is preempted and another process allowed to run. If the process has blocked or finished before the quantum has elapsed, then, of course, another process is run. The scheduler maintains a list of runnable processes, when the process uses up its quantum, it is put at the end of the list.

The rate monotonic algorithm is a priority based real-time algorithm for periodic tasks. In this algorithm, each task is assigned a fixed priority: the shorter the task period, the higher the priority. The schedulability test for a rate monotonic system is

$$\sum_{i=1}^{m} \frac{C_i}{T_i} \leq N(2^{1/N} - 1)$$

where is $N$ is the number of periodic tasks being scheduled. This is the sufficient condition that all processes complete their work before the end of their periods. As the number of tasks increases the scheduling bound converges to $\ln 2$ or 69%.

Q2a As with processes in non-real-time systems, real-time tasks must often share resources, using the same mechanisms, such as shared memory, semaphores, messages, and locks. Typically a task locks a resource when it is using it and unlocks it when done. A case can arise where a high-priority process needs the resource immediately in order to not miss a deadline, but a lower-priority process has the resource locked. *Priority inversion* is defined as the blocking of a high priority task due to a lower priority task locking a shared resource. Figure 1 illustrates the priority inversion problem. The task set consists of three tasks, $H$, $M$ and $L$ with their priorities in the order $P_H > P_M > P_L$. When task $H$
Figure 1: Priority inversion

requires use of resource 1, it is put in a blocked state waiting upon its release. The next task on the ready list would then be L which runs but is preempted by task M. Now task L needs to finish in order to release the resource, but it cannot because of task M which does not require the use of the same resource. Therefore, task H remains blocked unable to run because of the lower priority task L.

Priority inversion can be eliminated by using Priority Inheritance Protocol, where lower priority tasks inherit a higher priority when holding a shared resource. In the example above, the task L would have a higher priority so would not be blocked by M and thus be able to complete.

Q2b One method of scheduling aperiodic tasks is by creating a periodic, fixed priority task that handles all aperiodic task requests. The virtue of this approach is that the periodic polling task can be analyzed as a periodic task. The execution time of the task is the time associated with processing a event and the period of the task is its polling period. There two problems with this model:

- If many events occur during a polling period, the amount of execution time associated with the periodic poller may vary widely and on occasion cause lower priority periodic tasks to miss deadlines
- If an event occurs immediately after the polling task checks for events, the associated processing must wait an entire polling period before it commences.

Another way is by using an aperiodic server. An aperiodic server is a conceptual task that is endowed with an execution budget and a replenishment period. An aperiodic server will handle randomly arriving requests at its assigned priority as long as the budget is available. When the server’s computation budget has been depleted, requests will be executed at a background priority (i.e., a priority below any other tasks with real-time response requirements) until the server’s budget has been replenished. The execution budget bounds the execution time, thus preventing the first problem with the polling server. The aperiodic server provides on-demand service as long as it has execution time left in its budget, thus preventing the second problem. A method that uses the aperiodic server is the Priority exchange algorithm where the responsiveness of aperiodic tasks is maximized by using a high priority periodic server that handles the aperiodic task requests. The server exchanges its priority with that of
the pending, highest priority, periodic task, if no aperiodic task requests occur at the beginning of the server period. This algorithm has a high runtime overhead.

**Q3a** The foreground/background architecture is widely used because of its simplicity. It consists of a high priority foreground tasks and background tasks that run in the absence of the foreground tasks. An example can be a video arcade game, where the foreground consists of preemptive priority tasks such as interrupt routines triggered by the buttons and the background is for game situation processing.

**Q3bi** The system structure will be a main task that runs in the background reading the resistor and setting the variable \( T \). The foreground tasks will be interrupt driven such as the Port B and the Timer 1 handlers. The state diagram is shown as follows, the read resistor state is expanded to the right.

![State Diagram](image)

**Figure 2: State diagram**

**Q3bii** The basic processing of the main background task is to take readings of the resistor at intervals and set the value of \( T \). The Sync interrupt handler will simply set the value of Timer 1 such that it will overflow at time \( T \). This is simply the two’s complement of \( T \). When timer 1 overflows its interrupt handler will be called and the control pulse can be sent.

```c
//main routine
main()
{
    while (true) {
        select_A/D_channel();
        start_conversion();
        while ( !done_conversion() );
        T = result_from_A/D_conversion;
    }
}
//RB interrupt handler
int_sync()
{
```
Timer 1 = 16-bit two’s-complement of \( T \);
}

// Timer 1 interrupt handler
int_timer1()
{
    raise_control_line();
    delay(100 usec);
    drop_control_line();
}