Lecture10: Scheduling III, scheduling worksheet examples

Vikram Iyer

Adapted from material by Blake Hannaford

Reminders and announcements

- Videos and midterm resources posted
- Lab 2 is posted, due 5/5
  - Tip: Start by just trying to get the timer to output to a pin, and then adjust the timing. Might be easier to debug once you have an output
  - Tip: for scheduling, start with simple tasks
- Midterm on 5/12
  - During class here 90 min + separate room for DRS
  - Goal: make sure you know concepts, labs are worth more

Plan for today

- Arduino Demos
- More scheduling examples and worksheets

Scheduler types

Periodic scheduler- Infinite Loop

- Most primitive of all
- Also known as Non-preemptive Round Robin.

```
while(1) {
   task1_fcn();
   task2_fcn();
   task3_fcn();
   ...
}
```

Scheduler types

- Each task **must** voluntarily return to scheduler quickly
- $C << P$, $P = T$
- Sample task:

```
task1_fcn(){
  compute a little bit
  return;
}
```

Scheduler types

Periodic scheduler- Synchronized Infinite Loop

- the top of the loop waits on a hardware clock.

```
while(1) {
   wait(CLOCK_PULSE);
   task1_fcn();
   task2_fcn();
   task3_fcn();
   ...
}
```

- Each task **must** voluntarily return to scheduler quickly, as above.
- $C << P$, $T = P$.

Scheduler types

Pre-emption

Each function in these schedulers must “voluntarily” return back to the main loop, and must make sure that it doesn’t compute for too much time. If a task gets stuck, it breaks the whole system.

The fix for this is **preemption** - the ability to break into a running task and stop it. In other words to force a context switch.

Scheduler Exercises

1. Our basic `while(1)` loop: Non-preemptive Round Robin (RR).
2. The same loop with synchronization timing loop or timer function: Synchronized Non-premptive RR
3. Special hacks to the synchronized non-preemptive RR to enable different task periods.
4. Adding a `sleep(int d)` delay option for tasks: Non-premptive synchronized RR with delays.
5. Making the scheduler dynamic: Non-premptive synchronized RR with `halt me()` function.

Diagrams

Each Line = 333 µsec.

Assume a system interrupt every 1,000 µsec.

Mark interrupts at left edge of every third line. Each time scheduler runs it requires 100µsec (approximately 1/3 of a line).

Scheduler		Task A	Task B	Task C
Description:		Periodic C = 200 µs P = D = 1 ms	Periodic C = 200 µs P = D = 2 ms	Periodic C = 200 µs P = D = 2 ms


```
while(1) {
  wait(CLOCK_PULSE);
  taskA();
  taskB();
  taskC();
  wait(CLOCK_PULSE);
  taskA();
}
```

C = time for computation

P = period

D = deadline for tasks B + C

Delay functions and scheduling

Sometimes tasks will need to wait. Examples:

- To create time-based events (such as Flash an LED for 0.5 sec)
- To wait for something external to happen such as for a device
- To free up CPU time for other processes.
- To determine for themselves how often they run:

```
while(1) {
  compute();
  sleep(20ms);      // Run every ~20ms
}
```

This task sets its own period, P , to 20ms. How accurate?

Delay functions and scheduling

A bad delay function:

```
#define DELAY      22000
for (i=0; i< DELAY ; i++) ; // just loop to use up time
```

The scheduler can help implement delays `sleep(int delay)`:

- returns to the scheduler
- sets a software counter to `delay`.
- schedules the next task.
- the task which called `OS TimeDelay()` will be set to Waiting (not started by the scheduler)
- Each tick, decrement `delay` value for each sleeping task. when the `delay` is over, set task to `Ready`

Delay functions and scheduling

Assume a time delay function: `sleep(int d)`

```
while(1) {
  compute();
  sleep(20ms);
}
```

Task List and Characteristics:

- The three tasks are labeled A, B, C
- The nature of the three tasks is as follows:

A P=2.0ms, C= 233μsec, D=1.0ms

B P=4.0ms, C= 233μsec, D=1.0ms, task pseudocode:

```
taskB(void) {
```

Scheduler	Task A	Task B	Task C
Description:	C = 200 µs P = 2 ms, D = 1 ms	C = 233 µs P = ~~4 ms~~, D = 2 ms	C = 100 µs P = D = 1 ms

```
taskB(void) {
  compute for 233 us;
  sleep(3);
  return;
}
```

taskA
taskB
taskC