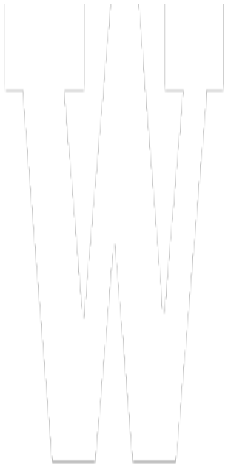
Lecture 8: Task, Threads, and Scheduling

Vikram Iyer

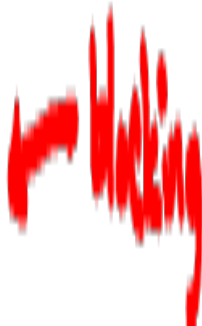
Adapted from material by Blake Hannaford



UNIVERSITY of WASHINGTON

Administrative

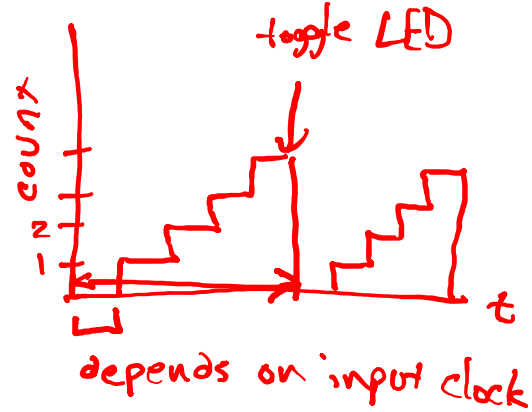
- Speaker trade-in
  - We have speakers with connectors on them
- Programming assignment 2 due Monday
  - Bit manipulation, more pointer practice, intro to structs
  - Meant to short so you can focus on Lab 2
- Lab 2 is posted, due 5/5
  - Longer than Lab 1, get started early
- Midterm on 5/12
  - Goal: make sure you know concepts, labs are worth more



Last time

- Walk through ATMega datasheet
  - Point out where to look for documentation in lab 2
- Details of 16 bit timer counter

UNIVERSITY of WASHINGTON



Plan for today

- How do we run a programs that have to do multiple things at once?

loop()

```
while(1) {
   task1_fcn();
   delay(50);
   task2_fcn();
   ...
}
```

blocking

- Intro to:
  - Tasks
  - Threads
  - Scheduling

Tasks, Processes and Threads

Processor State / Context

The CPU of a computer has several state variables. Once they are specified we know the exact state of the CPU.

These include:

- Program Counter (location in memory of next instruction)
- Value of each CPU register
- Processor Status Flags
- Stack Pointer

Extreme case: Intermittent Computing

Battery-free gameboy

Josiah Hester,
Georgia Tech

Tasks or Processes

A Task or Process is a unit of code and data which is described by one processor state when it is running.

Example Tasks:

- Flash an LED for 100ms, repeat every 1.5 seconds.
- Send a byte of data over a communication link whenever a hardware bit indicates the link is ready.
- Update display
- Every 60 seconds, read an analog voltage from a Thermistor, convert it to a resistance, calibrate to temperature.

A task is usually implemented in C by writing a function.

Context Switching

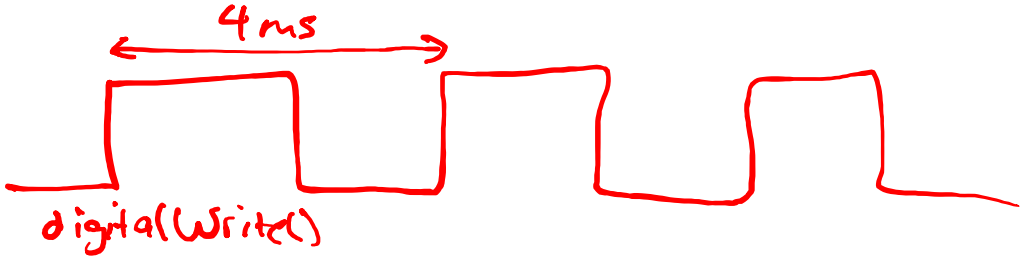
A computer must often switch from one task to another (don't we all?) and then back.

To restart a suspended task, we have to be able to restore its state. To restore its state, we have to save it.

Thus a context switch means

1. Save context of the processor for the current task.
2. Restore previously stored context of new task.
3. Start the new task.

Types of Tasks



Periodic tasks

- Found in Hard-Real time applications. Examples:
  1) Control: Every 10 ms., Read sensors → compute control → output command
  2) Audio: Every 22.727μsec, Get music sample → DSP filter → output to DAC.

- Characterized by three attributes:

1. **P , Period**: the regular time interval between runs of this task.
2. **C, Computing Resources**: How much CPU time does it require each time. Obviously C <= P . (C may not be the same each time)
3. **D, Deadline**: How quickly must the task ~~me~~ completed after it is started each time tick. C < D < P

Types of Tasks

Periodic task pseudocode

```
task() {
  compute for C seconds;
  return();
}
```

Types of Tasks

Intermittent tasks

- Found in all types of applications. Examples:
  1) send an email every night
  2) Save all data when power is going down
  3) Send a message to plant operator when tank runs low.
  4) Calibrate a sensor on startup

- Characterized by two attributes:

1. C, Computing Resources: How much CPU time does it require each time?
2. D, Deadline: How quickly must the task be completed after it is started. (whenever that happens to be).

Types of Tasks

Intermittent task pseudocode

```
task() {
  compute for C seconds;
  if(! done){
    return(); ← return to scheduler
  }
  else {
    halt_me(); ← end this task
  }
}
```

Types of Tasks

Background tasks

- A soft real time or non real time task.
- Lower Priority
- Will be accomplished only as CPU time is available when no hard real time tasks are ready.
- Characterized by: C, Computing Resources: How much CPU time does it require each time between scheduler accesses.
- Typical Code Structure:

```
task() {
  compute for C seconds;
  if(! done){return();}
  else { halt_me()}
}
```

Types of Tasks

Complex tasks

- Found in all types of applications
- Examples: 1) Microsoft Word 2) Apache Web Server.
- Characteristics:
  1. Continuous need for CPU time.
  2. Frequent requests for I/O which free up the CPU.
  3. Waits for user input which free up CPU.
- Typical Code Structure:

```
function() {
  while(1) {
    compute for C seconds;
    request_IO();// also starts scheduler
  }
}
```

Task States

Intermittent task pseudocode

Tasks are in one of four states:

1. Running
2. Ready to Run (but not running)
3. Waiting (for something other than the CPU.)
4. Inactive

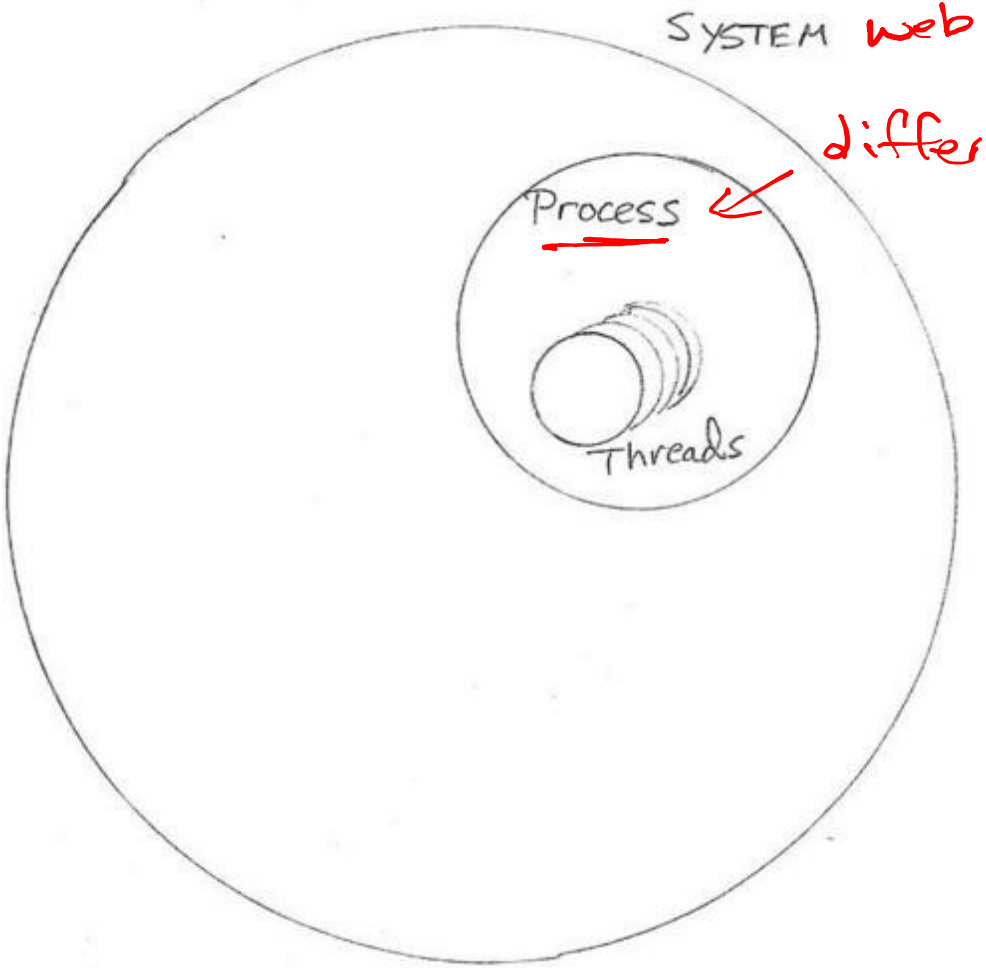
Threads: Smaller units of computation

With any complex computing system we naturally break it up into units. For example our CSE/ECE474 lab documents have: “tasks”

Each unit has

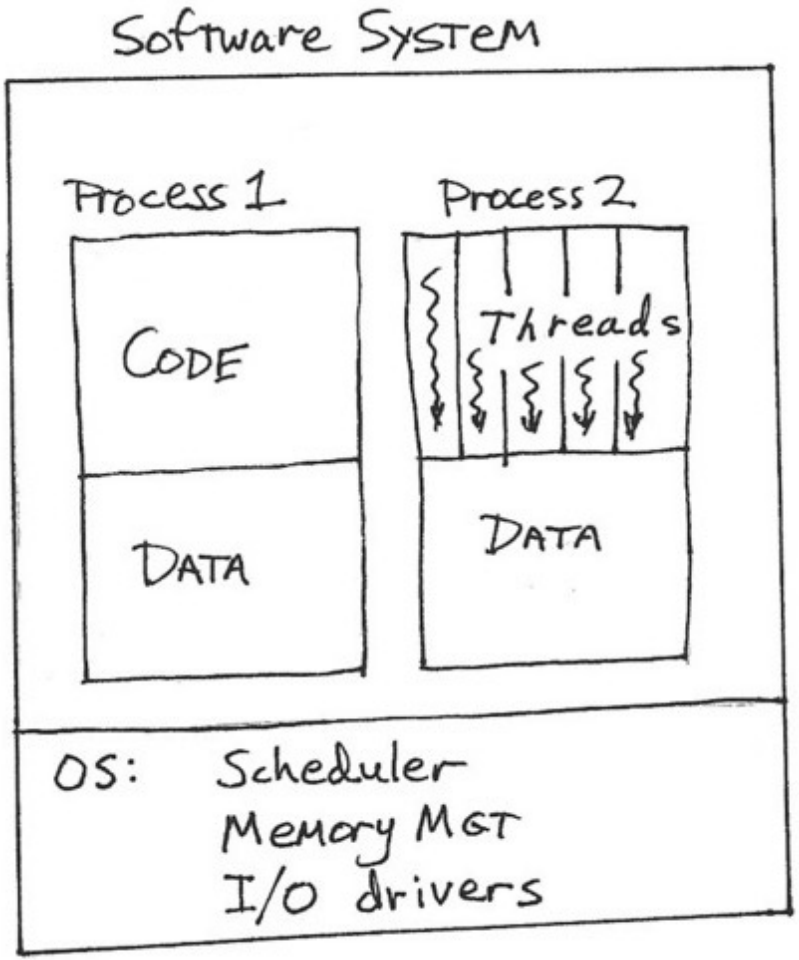
- Code/instructions
- Data
- Context/State
- Resources (memory, I/O devices, semaphores)

Task States



Threads: Smaller units of computation

- A Thread is a unit of computation with code and context, but limited private data. Threads may even share code with each other.
- Threads are owned by a program/process.
- Threads are like pieces within a program which can be scheduled by the OS independently.



A complete software system with two processes.

Process 1: normal

Process 2: threaded

Thread context switching

- Since threads have a smaller context than programs, context switching is faster.
- Only save/restore CPU state
- No need to change memory setup
- Example use case: web server

```
while(1) {
    OS_read_network();    // get incoming request
    compute;              // figure out what html page to get
    OS_Disk_I/O();        // read disk file for www page
    OS_write_network();   // send page to user
```

Thread context switching

Alternative 1: Run above as a single process.

Problem: When process is waiting for network, disk is idle and vice versa.

Alternative 2: Set up several threads of same code:

Advantage: Scheduler can start a thread which is done with disk I/O while another thread is waiting on network and vice versa.

CPU is used more efficiently and system delivers more pages per second with same hardware performance.

Potential Problem: more time spent context switching.

Thread context switching

Types of threads

- There are lots of possibilities for size of *units*.
- The more context a thread contains, the “heavier” it is.

  Heavier threads take more time for context switching.
- OS designer determines what kind of threads to support — application programmer gets only one option.

RTOS Scheduling Concepts

Real time operating System

Time

- Time is measured by a piece of hardware which records actual clock time: a *real time clock*:
- "Time Quanta" "Time Slice"
- "Ticks" Let's call the period of time ticks T
- Typically $T < P_{min}$. Sometimes

  $T << P_{min}$

RTOS Scheduling Concepts: Real time

Real Time: A software system with specific speed or response time requirements.

Soft Real Time: If the deadlines are not met, performance is considered low.

Hard Real Time: A computer system in which at least one task must meet deadlines in time. If the deadlines are not met, the system has failed.

Super Hard Real Time: Mostly Periodic Tasks: Task periods = the OS time tick, task compute times (C) and deadlines (D) are very short. $P = D = T, C << P$.

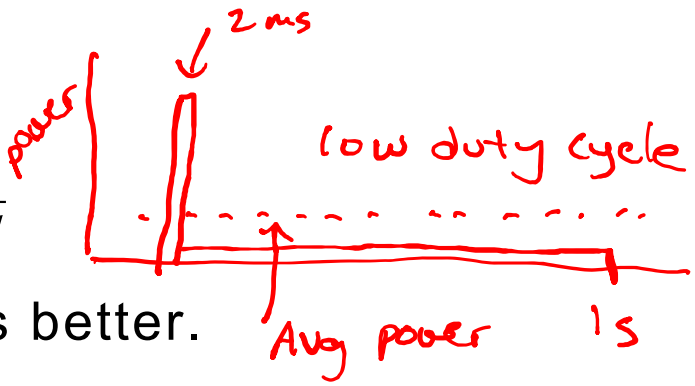
RTOS Scheduling Concepts

At each opportunity, OS asks:

If *N* tasks are **Ready**, which one should I run?

If you can show that scheduler always can achieve deadlines, the system is “deterministically schedulable”.

RTOS Scheduling Concepts



1. **CPU utilization** $U_{CPU} = 1 - \frac{idle}{period}$

   In traditional systems, closer to 100% is better.

   But U_{CPU} = 100% is not safe for real-time systems

   Goal: low load - 40%, high load - 90%.

2. **Throughput**: Work units completed per unit time.

3. **Turnaround Time**: Total time from task start to completion (waiting + execution + I/O)

RTOS Scheduling Concepts

4. **Waiting Time**: Choose scheduler which minimizes waiting time
5. **Response Time**: How quickly system responds to external asynchronus events.

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:

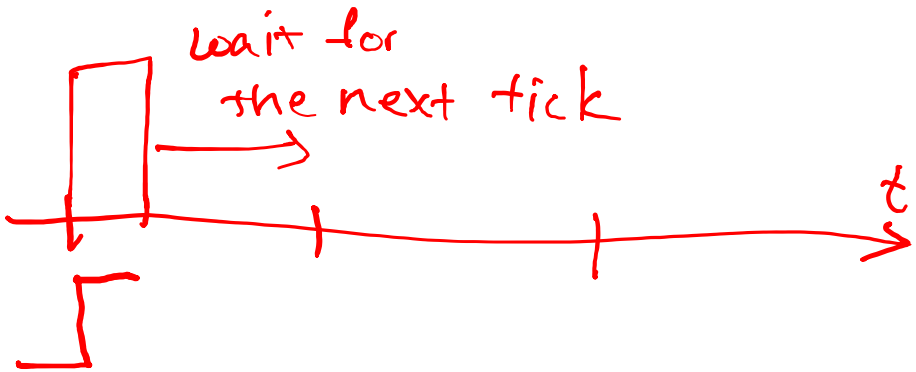
```
task1_fcn(){
  compute a little bit
  return; let scheduler
          run other things
}
```

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 types

1. Non-preemptive Round Robin (RR)- Basic `while(1)` loop
2. Synchronized non-preemptive RR- The same loop with a timing function for synchronization
3. Modifications to Synchronized RR to enable different task periods
4. Adding a `sleep(int d)` delay option for tasks
5. Making the scheduler dynamic with a `halt_me()` function

Scheduler		Task A	Task B	Task C

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

Task	P	C	D
A	1	0.2	P
B	2	0.2	P
C	2	0.2	P