Week 10. RTOS: Processes, Scheduling

18-342: Fundamentals of Embedded Systems

Rajeev Gandhi
INI & ECE
Carnegie Mellon University

Carnegie Mellon

Electrical & Computer ENGINEERING

Outline

- Multitasking systems
  - Processes, tasks and threads
- Process creation
- Process states
- Scheduling
  - FCFS scheduling
  - Shortest job first scheduling
  - Round robin scheduling
  - Multi-level round robin scheduling

- Reference – Chapter 6 of "Operating Systems Concepts" by Silberschatz, Galvin, Gagne (Sixth Edition) or any other similar chapter in another OS book

Running Multiple Tasks

- Problem: How do you run multiple tasks "simultaneously" on a single CPU?

- Multi-tasking ⇨ technique for sharing a single processor between multiple independent tasks
  - Cooperative multi-tasking ⇨ running task decides when to yield the CPU
  - Preemptive multi-tasking ⇨ another entity (the scheduler) decides when to make a running task yield the CPU

Multi-Tasking Requirements

- Need to perform at least three functions to allow multiple tasks to run on a single CPU
  - Scheduling: determine which task/process will be run next
  - Dispatching: saving the state of current task and restoring the state of the new task
  - Inter Task Communication: sharing data amongst multiple tasks
- Missing functions?
  - Memory protection/management

Process

- Informally, a **program in execution**
- Process is more than just the code
- It includes the current activity of the program, as represented by the PC and the contents of the processor's registers.
- Each process has its own text, data, heap and stack section
- **Program** is a **passive** entity
- **Process** is an **active** entity
  - Multiple processes can run the same program
  - Each process has its own state which may be different from the states of other processes (even if they are running the same program)

Process *vs.* Task *vs.* Thread

- The terms, **process** and **task**, are often used interchangeably
- Many operating systems also allow multithreading where a single process (task) can have multiple threads
  - These multiple threads share the text and data segments (physical memory), file descriptors and process priority
  - Each thread has its own private register set (including PC) and stack
- **Threads**, sometimes called **lightweight processes**, have their own copy of
  - Program counter
  - Register set
  - Stack space

Tasks in Embedded Systems

- Tasks in most Embedded Systems are periodic
- Periodic processes/tasks
  - Time-driven
  - Activated on a regular basis between fixed time intervals
  - Specified by $\{C_i, T_i\}$
    - C_i = worst-case compute time (execution time) for task τ_i
    - T_i = period of task τ_i
  - Example: periodic monitoring or sampling of sensors

Tasks

- Tasks in embedded systems are typically periodic and tend to be an infinite loop function

```
void MyTask(void *pdata)
{
      while(1){
      /*   some application code */
      /*  call an OS service: */

                mutex_lock(…);

       /*   some application code */
      /*  call more OS service: */

           mutex_unlock(…);

      /* more application code after which the task waits until the start of
the next time period*/
      sleep(until_start_of_next_time_period);
      }
}
```

Process Creation in Linux

- The only way (after system initialization) in which a new task is created for an existing process to execute `fork()`
  - The process that executes `fork()` is called *parent process*
  - The new process is called the *child process*
- The child process
  - Shares the text segment (the actual machine instructions)
  - Gets a copy of the data
  - Files that were opened by the parent process
- `fork()` is called *once* but returns *twice*
- `PID = fork()` returns a process identifier (PID)
  - In the child task, the value of the PID is zero
  - In the parent task, PID contains the child's PID
- This PID is useful for the parent task to keep track of all of its children
  - Example: Which child is alive, which is terminated

What happens on a `fork()` call?

```
PID = fork();

if (PID == 0) {
   /* This is the child task; add your code here for a small fee */

}
else {
   children[current_child] = PID;
   current_child++;
   /* This is the rest of the parent task; continue your code here.
    You can also wait here for your child task to return */

}
```

Example Problem

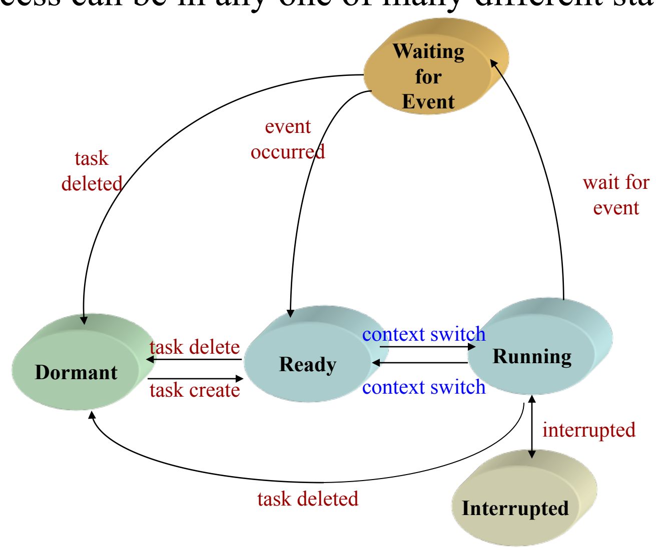
A task running on a system executes the following

```
int main()
{
        int pid, pid0;
        pid = fork();
        if (pid == 0) {
                pid0 = fork();
                printf("Child task\n");
                return 1;
        }
        else
                printf("Parent task\n");
        return;
}
```

How many times is "Child task" printed on the console?
How many times is "Parent task" printed on the console?

Process State

- A process can be in any one of many different states



Process Control Block

- **Process Control Block (PCB)**
  - Also called a **Task Control Block (TCB)**
  - OS structure which holds the pieces of information associated with a process
- Process state: new, ready, running, waited, halted, etc.
- Program counter: contents of the PC
- CPU registers: contents of the CPU registers
- CPU scheduling information: information on priority and scheduling parameters
- Memory-management information: Pointers to page or segment tables
- Accounting information: CPU and real time used, time limits, etc.
- I/O status information: which I/O devices (if any) this process has allocated to it, list of open files, etc.

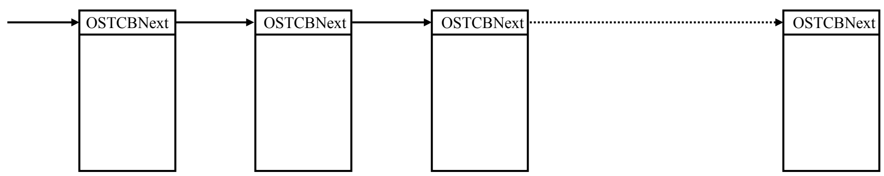
Example PCB

- Here is an example of a Process Control Block (for Lab 4)

```
typedef struct os_tcb {
  OS_STK         *OSTCBStkPtr;      /* Pointer to the loc where context stored */
  struct os_tcb  *OSTCBNext;        /* Pointer to next TCB in the TCB list  */
  struct os_tcb  *OSTCBPrev;         /* Pointer to previous TCB in the TCB list  */
  INT16U          OSTCBDly;          /* Nbr ticks to delay task or,
                                     /*timeout waiting for event        */
  INT8U           OSTCBStat;         /* Task status    */
  INT8U           OSTCBPrio;         /* Task priority (0 == highest,63 == lowest)  */
  INT8U           OSTCBX;            /* Bit position in group  corresponding to task
                                        priority (0..7) */
  INT8U           OSTCBY;            /* Index into ready table corresponding to task
                                        priority        */
  INT8U           OSTCBBitX;         /* Bit mask to access bit position in ready
                                        table */
  INT8U           OSTCBBitY;         /* Bit mask to access bit position in ready
                                        group */
} OS_TCB;
```

Ready queue

- Kernels often maintain the list of currently runnable tasks on ready queues
- Typically a linked list
  - On a context switch, the scheduler can go through this run queue and look for the highest priority task (in a priority based system) that is ready to run
- Maintain run queue as a linked list of TCBs of all tasks in your system



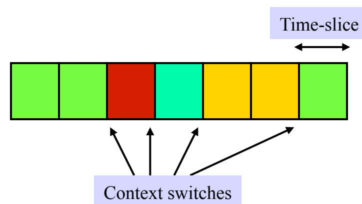
- How do you find the highest priority task in this list?
- Disadvantages?

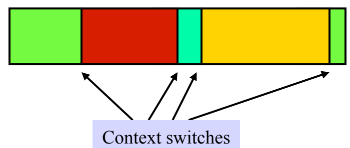
Context Switch

- The CPU's replacement of the currently running task with a new one is called a "context switch"
- Simply saves the old context and "restores" the new one
  1. Current task is interrupted
  2. Processor's registers for that particular task are saved in the TCB
  3. Task is placed on the "ready" list to await the next time-slice
  4. Memory usage, priority level, etc. is updated in the TCB (if needed)
  5. New task's registers and status are loaded into the processor
     - This generally includes changing the stack pointer, the PC and the PSR (program status register)
  6. New task starts to run

When *Can* A Context-Switch Occur?

- Time-slicing
  - Time-slice: period of time a task can run before a context-switch can replace it
  - Driven by periodic hardware interrupts from the system timer
  - During a clock interrupt, the kernel's scheduler can determine if another process should run and perform a context-switch
  - Of course, this doesn't mean that there is a context-switch at every time-slice!
- Preemption
  - Currently running task can be halted and switched out by a higher-priority active task
  - No need to wait until the end of the time-slice





Context Switch Overhead

- *How often do context switches occur **in practice**?*
  - It depends – on what?
- *System context-switch time vs. processor context-switch time*
  - **Processor** context-switch time = amount of time for the CPU to save the current task's context and restore the next task's context
  - **System** context-switch = amount of time from the point the highest priority task was ready for context-switching to when it was actually swapped in
- *How long does a **system** context-switch take?*
  - System context-switch time is a measure of responsiveness
  - Time-slicing ⇨ time-slice period + processor context-switch time
  - Preemptive ⇨ processor context-switch time
  - Preemption is mostly preferred because it is more responsive (system context-switch = processor context-switch)

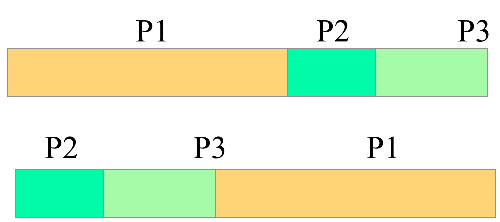
Process Scheduling

- What is the scheduler?
  - Part of the operating system that decides which process/task to run next
  - Uses a scheduling algorithm that enforces some kind of policy that is designed to meet some criteria
- Criteria may vary
  - *CPU utilization* - keep the CPU as busy as possible
  - *Throughput* - maximize the number of processes completed per time unit
  - *Turnaround time* - minimize a process' latency (run time), i.e., time between task submission and termination
  - *Response time* - minimize the wait time for interactive processes
  - *Real-time* - must meet specific deadlines to prevent "bad things" from happening

FCFS Scheduling

- **First-come, first-served** (FCFS)
  - The first task that arrives at the request queue is executed first, the second task is executed second and so on
  - Just like standing in line for a roller-coaster ride
- FCFS can make the wait time for a process very long

Process	Total Run Time
P1	12 seconds
P2	3 seconds
P3	3 seconds



If arrival order is P1, P2, P3

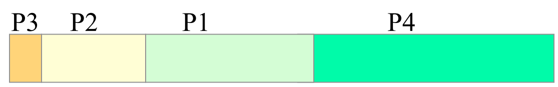
If arrival order is P2, P3, P1

- FCFS is a non-preemptive scheduling policy

Shortest-Job-First Scheduling

- Schedule processes according to their run-times

Process	Total Run Time
P1	5 seconds
P2	3 seconds
P3	1 second
P4	8 seconds

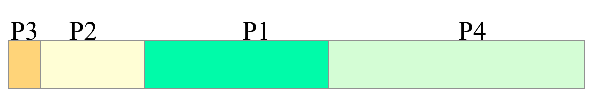


- May be run-time or CPU burst-time of a process
  - **CPU burst time** is the time a process spends executing in-between I/O activities
  - Generally difficult to know the run-time of a process
- Can be either preemptive or non-preemptive
  - Preemptive shortest-job-first is often called shortest-remaining-time-first scheduling

Priority Scheduling

- Shortest-Job-First is a special case of priority scheduling
- **Priority scheduling** assigns a priority to each process. Those with higher priorities are run first.
  - Priorities are generally represented by numbers, e.g., 0..7, 0..4095
  - No general rule about whether zero represents high or low priority
  - We'll assume that higher numbers represent higher priorities

Process	BurstTime	Priority
P1	5 seconds	6
P2	3 seconds	7
P3	1 second	8
P4	8 seconds	5



Priority Scheduling (con't)

- Who picks the priority of a process?
- What happens to low-priority jobs if there are lots of high-priority jobs in the queue?

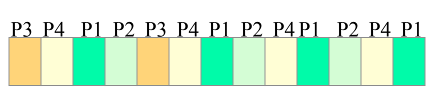
Round-Robin Scheduling

- Each process is executed for a small amount of time called a **time-slice** (or **time quantum**)
- When the time slice expires, the next process is executed in a **round-robin** order
- Each time slice is often several timer ticks

Process	BurstTime
P1	4 seconds
P2	3 seconds
P3	2 seconds
P4	4 seconds

Quantum is 1 “unit” of time (10ms, 20ms, …)

The following picture assumes quantum= 1seconds



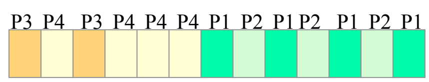
Multi-level Round-Robin Scheduling

- Each process *at a given priority* is executed for a **time-slice**
- When the time slice expires, the next process in **round-robin** order at the same priority is executed -- unless there is now a higher priority process ready to execute
- Each time slice is often several timer ticks

Process	BurstTime	Priority
P1	4 seconds	6
P2	3 seconds	6
P3	2 seconds	7
P4	4 seconds	7

Quantum is 1 “unit” of time (10ms, 20ms, …)

The following picture assumes quantum= 1seconds



Interactions Between Processes

- Multiple processes/tasks running concurrently on the same system might interact
  - Need to make sure that processes do not get in each other’s way
  - Need to ensure proper sequencing when dependencies exist
  - *Rest of the lecture*: how do we deal with **shared state** between processes/tasks running on the same processor?

Race Condition

- **Race condition** - outcome depends on the particular order in which the operations takes place
- Example: two processes in a system share and modify a global variable `x`

process one

```
while (1){
  x = x + 1;
}
```

process two

```
while (1){
  x = x + 1;
}
```

```
LDR    r1, [r0,#0]
ADD    r1, r1,#1
STR    r1, [r0,#0]
```

If "x=x+1" can execute as a single indivisible (atomic) action, then there is no race condition

Race Condition

- Consider two different schedules, assume initial value of `x` initialized to 0

process one

```
while (1){
  LDR    r1, [r0,#0]
  ADD    r1, r1,#1
  STR    r1, [r0,#0]
}
```

process two

```
while (1){
  LDR    r1, [r0,#0]
  ADD    r1, r1,#1
  STR    r1, [r0,#0]
}
```

- *Schedule 1*: Process 1 executes one while loop ➔ `x=1,` then process two executes one while loop ➔ `x=2`
- *Schedule 2*: Process 1 executes while loop but gets preempted before executing `ADD r1, r1, #1`
  - Process 2 executes one while loop `x`➔`1`
  - After executing one while loop, Process 2 gets preempted and Process 1 is executed again
  - Process 1 executes from the `ADD` instruction and completes the while loop `x`➔`1`
- Two different schedules generate two different results ➔ race condition

What is the Critical-Section Problem?

- A number of processes share a common segment of code (or data), called the critical section
- When one process is executing its critical section, no other process must be allowed to execute its critical section
- Problem: Design a protocol for the processes to cooperate
- Correct solution must satisfy:
  - Mutual exclusion: No two processes may be simultaneously in their critical sections
  - Progress: No process running outside the critical section may block another process from entering the critical section
  - Bounded waiting: There is a bound on the number of times that other processes can get into the critical section after P makes a request to enter the critical section and before that request is granted

Solution 1 – Taking Turns

- Use a shared variable to keep track of whose turn it is
- If a process, Pi , is executing in its critical section, then no other process can be executing in its critical section
- Solution 1 (turn is initially set to 1)

process one
```
while(turn != 1)
    ; // loop till turn == 1
  x = x + 1;
  turn = 2;
```

process two
```
while (turn != 2)
     ; //loop till turn = =2
  x = x + 1;
  turn = 1;
```

- Hmmm…..what if Process 1 sets *turn = 2,* but Process 2 never enters its critical section?
- We have mutual exclusion, but does Process 1 make progress?

Solution 2 – Status Flags

- Have each process check to make sure no other process is in the critical section
- Use 2 shared variables (`P1inCrit` and `P2inCrit`)

initially, P1inCrit = P2inCrit = 0;

process one

```
while (1){
  while(P2inCrit == 1);
    P1inCrit = 1;
    x = x + 1;
    P1inCrit = 0;
}
```

process two

```
while (1) {
while (P1inCrit == 1);
    P2inCrit = 1;
    x = x + 1;
    P2inCrit = 0;
  }
```

- Do you see any problems here?
  - The algorithm depends on the exact timing of the two processes

Solution 2 Does **not** Guarantee Mutual Exclusion

process one

```
while (1){
  while(P2inCrit == 1);
    P1inCrit = 1;
    x = x + 1;
    P1inCrit = 0;
}
```

P2 sneaks in

process two

```
while (1){
  while (P1inCrit == 1);
    P2inCrit = 1;
    x = x + 1;
    P2inCrit = 0;
}
```

	P1inCrit	P2inCrit	
Initially	0	0	
P1 checks P2inCrit	0	0	P1 jumps out of while loop
P2 checks P1inCrit	0	0	P2 jumps out of while loop
P1 sets P1inCrit	1	0	
P2 sets P2inCrit	1	1	
P1 enters crit. section	1	1	
P2 enters crit. Section	1	1	

Solution 3: Enter the Critical Section *First*

- Set your own flag before testing the other one

P2 sneaks in

process one	process two

`while (1){`
`  P1inCrit = 1;`
`  while (P2inCrit == 1);`
`  x = x + 1;`
`  P1inCrit = 0;`
`}`
	`while (1){`
	`  P2inCrit = 1;`
	`  while (P1inCrit == 1);`
	`  x = x + 1;`
	`  P2inCrit = 0;`
	`}`

	P1inCrit	P2inCrit
Initially	0	0
P1 sets P1inCrit	1	0
P2 sets P2inCrit	1	1
P1 checks P2inCr	1	1
P2 checks P1inCrit	1	1

- Each process waits indefinitely for the other

Deadlock - when the processes can do no more useful work

Peterson's Solution – Take Turns & Use Status Flags

process one

```
while (1){
 P1inCrit = 1;
 turn=2;
 while (P2inCrit == 1 &&
       turn==2){};

  x = x + 1;
  P1inCrit = 0;
}
```

process two

```
while (1){
  P2inCrit = 1;
  turn=1;
  while (P1inCrit == 1 &&
              turn==1){};

  x = x + 1;
 P2inCrit = 0;
}
```

- Initially, `turn` = 1 and `P1inCrit` = `P2inCrit` = 0;
- Ensures Progress, Mutual Exclusions, Bounded Waiting

Hardware Solutions to the Critical Section Problem

- Disabling (and enabling) interrupts
  - Cannot allow user programs to disable interrupts
- Special Instructions
  - TAS - Test and Set instruction
    - Both of the following steps are executed **atomically**
      - TEST the operand and set the CPU status flags so that they reflect whether it is zero or non-zero
      - Set the operand, so that it is non-zero
- Example
  - Initialize `lockbyte` to 0

```
Process 1                           Process 2
LOOP:  TAS lockbyte                 LOOP:  TAS lockbyte
       BNZ LOOP                            BNZ LOOP
         critical section                    critical section
       CLR lockbyte                        CLR lockbyte
```

Called a **busy-wait**
(or a **spin-lock**)

(TAS is not an ARM instruction)

Hardware Solutions (contd)

- Alternatively, hardware can provide a special instruction to swap the contents of two words **atomically**

```
void Swap(boolean *a, boolean *b) {
boolean temp = *a;
*a = *b;
*b = temp;
}
```

```
Process 1                           Process 2
while(1) {                          while(1) {
   key1 = true;                      key2 = true;
   while(key1 == true)               while(key2 == true)
       Swap(&lock, &key1);            Swap(&lock, &key2);
   x=x+1;                            x=x+1;
   lock = false;                     lock = false;
};                                  };
```

Initialize `lock=false`

Hardware Solutions (contd)

- Atomic SWP instruction on the ARM processor

- `SWP<cond> {B} Rd, Rm, [Rn]`

  Equivalent to

  ```
  temp = Mem[Rn]
  Mem[Rn] = Rm
  Rd = temp
  ```

- `SWP` combines a load and a store in a single, atomic operation

  ```
  SWPB   r1, r1,[r0]
  ```

  Atomically swaps the contents of memory location pointed by `r0` with contents (bits 0-7) of register `r1`

Critical Section For More Than Two Processes

- How do you extend previous algorithm for more than two processes?
- The **Bakery Algorithm**
  - On arrival at a bakery, the customer picks a token with a number and waits until called
  - The baker serves the customer waiting with the lowest number
    - Same applies today at AAA and DMV ;-)