Real-Time Operating System Basics

CSE4354/5354

Scope

- This is a very simplified survey of real-time operation system (RTOS) operation and computer science concepts, condensed to fit into two lecture periods
- This document is intended to show a minimum set of concepts required to complete the class projects
- It is recommended that you take an advanced computer sciences OS course if designing or using a RTOS

References

- **G. L. Peterson. *Myths about the mutual exclusion problem*. Information Processing Letters, 12(3):115--116, 1981**
- **Eisenberg, M.A., and McGuire, M.R. *Further comments on Dijkstra's concurrent programming control problem*. Comm. ACM 15, 11 (Nov. 1972), 999**
- **Knuth, D.E.: *The art of computer programming*. Fundamental algorithms. Addison-Wesley, 3rd edition (1997)**

References

- **Free Dictionary of Online Computing**
  - **http://wombat.doc.ic.ac.uk/foldoc/**
- **Scheduling (MUF and Quick Review of RM, EDF, and MLF)**
  - **http://www.ee.umd.edu/serts/bib/book article/rtp92.shtml**
- **Priority inversion on Mars Pathfinder**
  - **http://research.microsoft.com/~mbj/Mars_Pathfinder/**

RTOS Topics

- What is a RTOS?
- Tasks, Processes/Threads, and Kernel
- Multitasking
- Scheduling
- Critical Sections
- Inter-process Communication

What is a RTOS?

- A real-time operating system is one in which a set of computing tasks are considered correct only if the tasks are completed in a timely manner
- A real-time operating system executes these tasks correctly in a predictable (deterministic) manner

Common OS Terms

- **Processes – Complicated tasks require "heavy weight" code that requires a lot of state information and does not share a memory space with other processes**
- **Thread (of execution) – Simple tasks have "light weight" code that shares the memory space within the process**
- **Kernel – A part of the operating system responsible for managing system resources, scheduling and dispatching processes, and inter-process communications**

Task Properties

- Periodicity (Synchronicity)
  - Task can be timer-driven (periodic/synchronous)
  - Task can be event-driven (aperiodic/asynchronous)
  - Task can be a one-shot event (special aperiodic case)
- Temporal Attributes
  - Task duration (how long to complete a task)
  - Start deadline (must start by some time)
  - Stop deadline (must complete by some time)
- Deadline Type (hard or soft deadline)
- Relative Priority

Tasks

- A task has at least three states
  - Running, blocked, and ready
- Only one task can run at a time on the M4F controller (no parallel processing)
- Scheduler allows concurrent tasks using multi-tasking techniques (pseudo-parallelism)
- A task control block (TCB) keeps track of the state of a thread

Task Execution

- Task can be started by
  - Interrupt
  - Scheduler decisions
- Execution of interrupt task
  - Can run to completion
  - Can be preempted by a higher-priority interrupt
- Execution of dispatched tasks
  - Can run to completion (batch-like operation)
  - Can autonomously relinquish control (cooperative multi-tasking) before completion
  - Can be preempted by the scheduler (preemptive multi-tasking)

Kernel Scheduler

- Must decide which task to run
- Handling periodic tasks is straight-forward
- There is no apriori knowledge of when asynchronous events will occur, so the kernel must rely on statistics to determine scheduling (min time between tasks, average time between tasks, …)
- May also consider the timing and type of each task deadline to prioritize scheduling (hard deadlines have catastrophic consequences if missed)

Kernel Dispatcher

- Switches the context from a task to a new task selected by the scheduler
- Context switch steps
  - If preemptive multitasking, stops any currently running task if applicable and saves the context
  - Switches the context to the scheduler selected process (if restarting, then restores the context)
  - On the M4F, the context consists of the R0-12, SP, LR, PC, xPSR, S0-15, FPSCR, and other registers

Multi-tasking

- Cooperative
  - Kernel lets a task execute until completion (may temporarily interrupt it for interrupt processing)
  - If task is short, task runs to completion
  - If task is long, the task cooperates by relinquishing control back to the kernel so that other tasks can run
  - “Long” and “short” determine cooperation
  - Problem is a task “crashes” and does not relinquish control to other tasks
- Preemptive
  - Kernel assigns a quantum (time slice) to the task
  - If the task does not complete within the slice, the kernel preempts the task and lets another run

Scheduling 101

- Schedule should be feasible
- Need to avoid task starvation
- Need to watch for deadlock (or prevent it)
- Sometimes, a watchdog timer is used to detect “lockups” in the system
- Latency of dispatcher and scheduler to should be minimized
- Should optimize context switching time
  - Too fast causes thrashing
    (too much managing and no working)
  - Too slow causes sluggishness

Non-Priority Scheduling

- **FIFO**
  - **First task into a queue executes until completion**
- **Round robin**
  - **Preemption method where N tasks are each dispatched in order with equal quantum**
- **Rate Monotonic (RM)**
  - **Shortest pending task is executed first**

Priority Scheduling

- Earliest Deadline First (EDF)
  - The most critical deadline (function of whether the deadline is hard or soft and the time to expiration) determines who goes first
- Minimum Laxity First (MLF)
  - Like EDF, but prioritization based on laxity
  - Laxity = time to deadline - execution time
- Maximum Urgency First (MUF)
  - An RM schedule with a constraint that some tasks are temporarily starved if CPU cycles are not available

Critical Sections

- When multiple tasks access the same hardware resource or shared memory conflicts can occur
- When a task needs exclusive access to a resource, it enters a critical section of code, where if another task interferes (collides), the result of the operation could be corrupted
- Mutexes, semaphores, and inter-process signaling can be used to prevent collisions by blocking entry into a critical section in use

Critical Sections

- Sub-optimal solution could be to disable interrupts for a few clocks
- Could only be used on short critical sections
- Can have dire impacts to system performance – bad practice
- Code snippet:

```
disable_interrupts(GLOBAL);
  critical section
enable_interrupts(GLOBAL);
```

- We will use alternative techiques

Inter-process (thread) Signaling

- Can be used to coordinate access to shared resources
- Can synchronize multiple tasks so that a task only runs when other data from other tasks are ready
  - Called the producer-consumer scenario
  - Can coordinate reading and writing data from keyboard, uart, … through a ring buffer in shared memory

Mutual Exclusion

- Mutual exclusion object (mutex) coordinates access to a shared resource as follows:
- As an atomic (indivisible) operation, the requesting task
  - Checks the status of a mutex
  - If unlocked, it
    - Locks the mutex
    - Enters the critical section
    - Unlocks the mutex
- If the mutex was locked, access is blocked and the task waits
- It could use busy waiting (spinlock), waiting in a queue, or a combination

Priority Inversion

- A case where a lower priority task runs instead of a higher priority task
- Example with mutex operation:
  - Low, medium and high priority tasks (L, M, and H)
  - L is running and locks a mutex that H needs
  - H starts running and is blocked
  - L would normally run and unlock the mutex, but M preempts L, leaving H blocked from running
- This example could be solved through priority inheritance (temporary elevation of L process priority to H process priority when H process is blocked waiting on the locked mutex)

Errant Mutual Exclusion Solution

- **Not atomic (another task/ISR could execute code between the while (lock) {} and lock = TRUE code)**
- **Conceptual code snippets:**

```
// init
bool lock = FALSE; // global
…
// task code
while (lock)
  {"busy wait (spinlock)"}
lock = TRUE;
  critical section
lock = FALSE;
```

Errant Interrupt Mutual Exclusion Solution

- **This solution is errant, since if locked, the system will lockup**
- **Conceptual code snippets:**

```
// init
bool lock = FALSE; // global
…
// process code
disable_interrupts(GLOBAL);
while (lock) {"wait"}
lock = TRUE;
enable_interrupts(GLOBAL);
  critical section
lock = FALSE;
```

Interrupt Masking Mutex Solution

- **Interrupts make atomic on single core processors, but could degrade performance or cause interrupt loss – bad practice**
- **Conceptual code snippets:**

```
// init
bool lock = FALSE; // global
…
// task code
bool ok = FALSE;
while (!ok)
{
  disable_interrupts(GLOBAL or TASK_SWITCH_ISR);
  if (!lock) {ok = lock = TRUE;}
  enable_interrupts(GLOBAL or TASK_SWITCH_ISR);
}
  critical section
lock = FALSE;
```

Software Mutex Solutions

- **Dekker**
  - **First software-only solution with 2 tasks**
- **Peterson**
  - **Simpler than Dekker's solution with 2 tasks**
- **Eisenberg and McGuire**
  - **Simple N task solution with rotating priorities**

Peterson's Mutex Solution

- **Two task solution, simpler than Dekker's algorithm**
- **Code snippets:**

```
bool busy[2] = {FALSE, FALSE}; // global
int turn = 0; // global
…
// task i code; i = 0..1
busy[i] = TRUE;
turn = 1-i;
while (busy[1-i] && turn != i) { };
  critical section
busy[i] = FALSE;
```

Peterson's Mutex Solution

- Suppose task 0 is waiting to enter the critical section (executing the while loop)
- 2 conditions allow entry into the critical section:
  - Task 1 not busy (trying to use the resource or using the resource)
  - OR it is task 0's turn
- The critical section in task 0 can run if the last code task 1 executed was
  - Before busy[i] = TRUE or after busy[i] = FALSE (other task is not busy)
  - After turn = 1 - i (turn given away by process 1) (task 0's turn)

Hardware Mutex Solution

- On some processors, there is hardware support for atomic operations on memory
- For instance, on the single core M4F, there are variants of the load and store commands, LDREX and STREX, that allow locking and exclusive access to memory

Semaphores

- A synchronization method that shares a collection of resources across tasks
- Binary semaphore are 0 or 1 valued
- Counting semaphores have values ≥ 0
- Functions (three different nomenclatures)

  wait( ), down( ), or P( )

  - Puts task to sleep and adds task to a queue if count = 0

  - Decrements count and allow the task to proceed if count > 0

  post(), signal( ), up( ), or V( )

  - Increments count

  - If count = 1 after incrementing, then a task may be waiting to run in the queue

  - If task waiting in queue, wake up the task
- Generally, a task either waits or posts, not both

Semaphore Implementation

- **Conceptual view of a semaphore structure**
- **Code snippet**

```
typedef struct _semaphore
{
  int count;
  int queue_count;
  int process_queue[MAX_SIZE];
} semaphore;
```

Semaphore Functions

- **Conceptual Atomic Functions**
  - **wait(semaphore& s)**

```
while (s.count == 0)
{ s.task_queue[queue_count++]=
      current_task;
  dispatch other task(s)
  task(s) post the semaphore and queues up waiting
  task(s), prog flow eventually returns }
s.count--;
```

  - **post(semaphore& s)**

```
s.count++;
if (s.count == 1)
{ if (queue_count > 0)
      select task to run, dec s.count,
      set state to READY state,
      return to caller }
```

Producer-Consumer Example

- Consider a solution with three synchronization primitives
  - One mutex (key) used to control access to inventory
  - One counting semaphore (needed) used to control producing
  - One counting semaphore (available) used to control consuming
- Maximum number of inventory items is N

Semaphore Example

- **Code snippet**

```
mutex key = false;
semaphore needed, available;
needed.count = N;
available.count = 0;
```

Semaphore Example

- **Code snippet**

```
// producer task
wait(needed);
lock(key);
  add widget
unlock(key);
post(available);
…
// consumer task
wait(available);
lock(key);
  remove widget
unlock(key);
post(needed);
```

Common Bottlenecks for RTOS

- **Serving**
  - **Polling overhead (use interrupts or semaphores when possible to mitigate)**
  - **Latency of interrupts may affect performance**
- **Multi-tasking**
  - **Latency of context switching (save as little as absolutely possible)**
  - **Impact of blocking (use sleep and wake instead of busy waiting when possible)**
- **Computational**
  - **Complex mathematical functions (use lookup tables when possible)**