CS 5372
Specification and Design of Real-Time Systems

Lecture 4: Real Time Semaphores

1

Outline

- Lab work (20 min)
  - Configuring VxWorks Kernel
  - VxWorks Hostshell
- 09/19/2015 Quiz Solution (10)
- Quiz (15)
- Lecture Synchronization and Semaphores (45)
- Lab exercise – Binary and mutex sepmaphores (60)
- Semester Project (Rest of class)

2

Lab Work

- Configuring VxWorks Kernel
- VxWorks Hostshell

3

Last Weeks Quiz

- Question 1:
  - sysClkRateSet(1)
  - tid1 = sp(taskDelay, 500)
  - ts(tid1)
  - Observe the counter
  - tr(tid)

- Question 2:

```
void recur1(int level)
   {
   if (level>0)  {
       count1++;
       taskDelay(0);
       recur1(level-1); }
    }
void recur2(int level)
   {
   if (level>0)  {
       count2++;
       taskDelay(0);
       recur2(level-1); }
    }
```

Quiz

5

Condition synchronization

- **Condition synchronization** is when one task operation can be performed only after another task reaches specific state
  - This requires the first task to hold until it is appropriate to proceed
- Separate tasks may have access to the same memory location (**shared variable**) for the purpose of passing information -
  - concept of **flag**
  - Process of updating the shared variable is NOT **atomic** (or **indivisible**)
    - even *a*++ is a sequence of three steps (load, increment, store)

6

Synchronization - using flags

- The conventional implementation of waiting is either:
  - **busy waiting** (**spin-lock**): continuously checking the flag in a loop
  - **suspend/resume** the task depending on the flag condition
- Discussion:
  - The flag-based approach is highly inefficient
  - it uses CPU time for useless computations when busy waiting
  - the flag cannot/shouldn't be tested concurrently by many tasks resulting in **race condition**

7

Synchronization - busy waiting

- Task T1 executes empty loop testing the flag before entering a critical section;
- Task T2 sets the flag

What's a critical section

```
T1 (waiting)
…
while FLAG = "down"
    loop
        null;
    end loop;

do something
…

FLAG = "down"
```

```
T2 (signaling)
…
FLAG = "up"
…
…
```

8

Synchronization - suspend/resume

- Task T1 tests flag continuing when the flag is set;
- Task T2 sets the flag and resumes T1

```
T1 (waiting)
…

if FLAG = “down”
      suspend
end if;

do something
…


FLAG = “down”
```

```
T2 (signaling)
…
…
FLAG = “up”
resume T1
…
…
```

9

Critical Sections

- **Critical Section** (CS) is a segment of a task code that must be executed without giving control away to another task (in a mutually exclusive manner)
- The CS may represent code that is: (a) manipulating sensitive data or (b) accessing system resources (only one task at a time)
- CS requires synchronization before entry to provide mutual exclusion
- CS can be used for :
  - shared update (*modify value*)
  - shared access (*retrieve value*)
- Access protocol providing protection can be implemented in a form of: **flags, semaphores, monitors**

10

Critical Sections - Using Flags

- unprotected CS in a concurrent program may lead to unpredictable results
- Protection of shared data can be performed using shared variables (e.g. BOOLEAN flags)
- Solutions are non-trivial and difficult to scale-up to more than two processes
- Using "three flags" (a turn flag plus one for each process) or the Dijkstra's algorithm may alleviate some problems
- The problems are related to violation of **mutual exclusion** access and **race condition**

11

Mutual Exclusion – Using Flags: busy waiting (1)

- Process announces intention to enter CS
  - if opposite flag is "down" it enters CS
  - it sets flag "down" upon exit from CS

Signals intention to get into CS

```
T1
FLAG1 = "up"
while FLAG2 = "up"
   loop
            null;
   end loop;

<critical section>

FLAG1 = "down"
```

```
T2
FLAG2 = "up"
while FLAG1 = "up"
   loop
            null;
   end loop;

<critical section>

FLAG2 = "down"
```

Are we happy with this?

Is it possible for both tasks to be in the critical section at same time?

Other problems?

Can we make it better?

12

Mutual Exclusion – Using Flags: busy waiting (2)

- if opposite flag is “down” task sets “own” flag “up” and enters CS
- It sets “own” flag “down” upon exit from CS

```
T1
while FLAG2 = “up”
   loop
        null;
   end loop;

FLAG1 = “up”

<critical section>

FLAG1 = “down”
```

```
T2
while FLAG1 = “up”
   loop
        null;
   end loop;

FLAG2 = “up”

<critical section>

FLAG2 = “down”
```

Are we happy with this?

Is it possible for both tasks to be in the critical section at same time?

Can we make it better?

13

Mutual Exclusion – Using Flags: “turn” flag

- if “turn” flag is not pointing to other process, the process enters CS
- it sets “turn” flag to other process upon exit from CS
- Initially “turn” is set to an arbitrary value of 1 or 2.

```
T1
   while TURN = 2
     loop
        null;
     end loop;

   <critical section>

   TURN = 2
```

```
T2
   while TURN = 1
     loop
        null;
     end loop;

   <critical section>

   TURN = 1
```

14

Semaphore Definition and Properties

- **Semaphore** is a software construct which allows the value stored in a memory location to be tested and updated depending on the result of testing in a single atomic operation: **TEST_AND_SET**
- **Semaphore** is an **integer variable** taking the values from 0 to a predefined max (*binary=1, counting>1*) combined with two indivisible operations that can be requested by active programs
- Each semaphore has a queue for the suspended processes and **indivisible operations** (WAIT/SIGNAL, TAKE/GIVE, PEND/RELEASE, P/V - Dijkstra)
- RTOS defines semaphores with their functions (*create, delete, wait, signal*)

15

Semaphore Operations

- WAIT/**TAKE** - the entry protocol checking the semaphore's value
  - if the value is greater than zero it decrements the semaphore,
  - if the value is zero it will queue the process on the queue associated with the semaphore
- SIGNAL/**GIVE** - the exit protocol checking the semaphore's queue;
  - if there is a process waiting it is activated
  - if no process is waiting, the semaphore is incremented

16

Types of VxWorks Semaphores

- They are highly optimized providing the fastest intertask communication mechanism
- Used primarily for addressing: (???)
  - **synchronization** - coordinating external events
  - **mutual exclusion** - access to shared resources
- Types of VxWorks semaphores: (???)
  - **Binary**: fastest and general purpose low overhead
    - *SemBCreate*()
  - **Counting**: keeps track of the number of times a semaphore is given, optimized for guarding multiple instances of a resource
    - *SemCCreate*()
  - **Mutual Exclusion**: special (**mutex**) semaphore optimized for mutual exclusion (priority inheritance, deletion safety, recursion)
    - *SemMCreate*()

17

VxWorks Semaphore operations

- Semaphore control routines (*semLib.h*)
  - *sem[BMC]Create*( ): allocate and initialize a [binary, mutual exclusion, counting] semaphore
  - *semDelete*( ): terminate and free a semaphore
  - *semTake*( ): take a semaphore
  - *semGive*( ): give a semaphore
  - *semFlush*( ): unblock all tasks that are waiting for a semaphore
  - When a semaphore is created, the queue type is specified. Tasks can be queued in
    - priority order (SEM_Q_PRIORITY) or
    - FIFO (SEM_Q_FIFO)

18

VxWorks Binary Semaphore

- It can be viewed as a flag that is available(full) or unavailable(empty)
- When a task is waiting for an event (e.g. a change in a data structure, completion of an atomic transaction, a change in the status of external HW, the expiration of a timer) binary semaphore can be used for notification that the event has occurred
- The binary semaphore can be taken or given by any task
  - Bad design can lead to data or behavior corruption

- Group Discussion:
  - Binary sems are used for either *mutual exclusion or synchronization*
  - Depending on the use of the sem, you, as a designer, will either initiate the sem as either *empty or full*.
  - When used for ***mutual exclusion***, the sem is **initially set to?**
    - **Full**
    - a task first takes and then gives back the semaphore
  - When used for ***synchronization*** the sem is **initially set to?**
    - **Empty**
    - one task waits to take the semaphore, which is given by another task

19

VxWorks Counting Semaphore

- Count is incremented when semaphore is **given** and is decremented when semaphore is **taken**
- When count reaches zero, a task that tries to **take** the semaphore is blocked
- When a semaphore is given and there are no other tasks blocked, then the count is incremented (which implies that a semaphore that is given twice can be taken twice without blocking)
- Example: (do this on your own time)
  - use of three tape drives might be coordinated using a counting semaphore with an initial count of 3
  - initial count specified as an argument to *semCCreate*( )
  - use *semTake* four times – what is the count?
  - use *semGive* two times – what is the status of tasks?

20

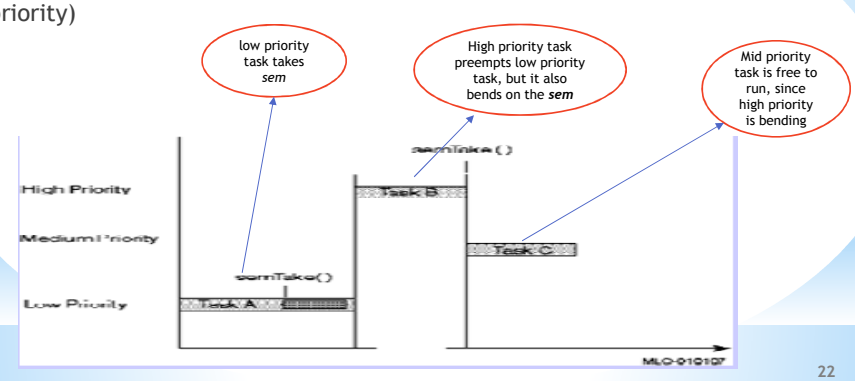
VxWorks Mutex Semaphore

- Mutex used primarily for the mutual exclusion
- Can be taken recursively i.e. taken more than once by the task that owns it before being finally released
- Before being released it must be ***given*** the same number of times it is ***taken***;
  - this is tracked by a count that increments with each *semTake()* and decrements with each *semGive()*
- Can be given only by the task that took it (it has an "owner")
- Cannot be given from an Interrupt Service Routine
- *semFlush( )* operation is illegal
- Mutex supports **Priority Inversion** and **Deletion Safety**

21

Priority Inversion

- HIGH task can not execute waiting for semaphore taken by LOW tasks, thus MEDIUM task runs
- Mutual Exclusion semaphore can prevent **priority inversion (PI)**
- **Priority-inheritance** algorithm assures that a task that owns a resource executes at the priority of the highest-priority task blocked on that resource (when the execution is complete, the task gives up the resource and returns to its original priority)



22

Priority Inversion - Protocols

- Priority Inheritance Protocol
  - Priority level of lower (current owner of mutex) is raised to that of the higher protocol waiting on the mutex
  - Priority level is dropped back to original level after releasing the mutex
- Ceiling Priority Protocol
  - Priority level of a task holding the mutex is raised to the highest priority of all tasks that might request the mutex

How do these protocols solve the problem in the previous slide?

23

Ceiling Priority Protocol

- Priority of every task is known a priori
- For a given resource (e.g. mutex), the priority ceiling is the hieghest priority of any task that might require the resource
- Example: R1 is required by T1 (priority of 100), T2 (priority of 120), and T3 (priority of 150)
  - Priority ceiling of R1 = 100 (remember low number → high priority)
- Rules: When a task T requests R
  1. If R is in use, T is blocked
  2. If R is free, R is allocated to T. T's execution priority is raised to the priority ceiling of R if it's higher. At any given time, T's execution priority equals the highest ceiling of all resources held by T
  3. When T releases the resource with the highest ceiling, its priority is dropped to the next-highest ceiling of its held resources
  4. T goes back to its original priority once it releases all resources.

24

Priority Ceiling - Protocol

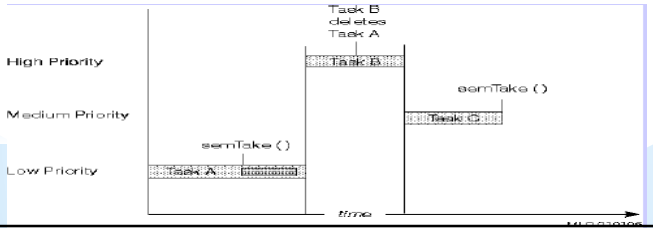
- The *current* priority ceiling of for a running system at anytime, is the highest priority ceiling of all resources *used at that time*
- Example: R1 (priority of 90), R2 (priority of 120), R3 (priority of 130), and R4 (priority of 150)
  - Current priority ceiling of the system = 90
- Rules: When a task T requests R
  1. If R is in use, T is blocked
  2. If R is free, and if the priority of T is higher than the current priority ceiling, R is allocated to T
  3. If the current priority ceiling belongs to one of the resources T currently holds, R is allocated to T, otherwise T blocks
  4. The task that blocks T inherits T's priority if it's higher and executes at this priority until it releases every resource whose priority ceiling is higher than or equal to that of T. The task then returns to it's previous priority

Examples

25

VxWorks Deletion Safety

- **How would deleting a task holding a semaphore cause a deadlock?**
- **Deletion Safety:** HIGH task deleting LOW task prevents it (LOW) from eventually giving the semaphore up, and hence, blocking indefinitely MEDIUM task
- **Deletion Safety** protects a task in a critical region *(when guarded by the semaphore)* from an unexpected deletion
- Deletion may leave the resource in a corrupted state and the semaphore guarding the resource unavailable - thus preventing other tasks the access to the resource



26

VxWorks Implementation

- **Priority Inversion** is disabled by using the SEM_INVERSION_SAFE option with the SEM_Q_PRIORITY option
  - *semID = semMCreate (SEM_Q_PRIORITY | SEM_INVERSION_SAFE)*
- **Deletion Safety** is assured by using the SEM_DELETE_SAFE option with the SEM_Q_FIFO option
  - *semID = semMCreate (SEM_Q_FIFO | SEM_DELETE_SAFE)*

27

Semaphores: Avoiding Mistakes

- To prevent misuse mutex semaphores to provide access to a resource:
  - Write a library of routines to access the resource
  - Initialization routine creates the semaphore
  - Routines in this library obtain exclusive access by calling *semGive()* and *semTake()*
  - All uses of the resource are through this library
  - Mutual exclusion semaphores can not be used at interrupt time
  - Keep the critical section of the code short,
    - i.e.code between *semTake()* and *semGive()*

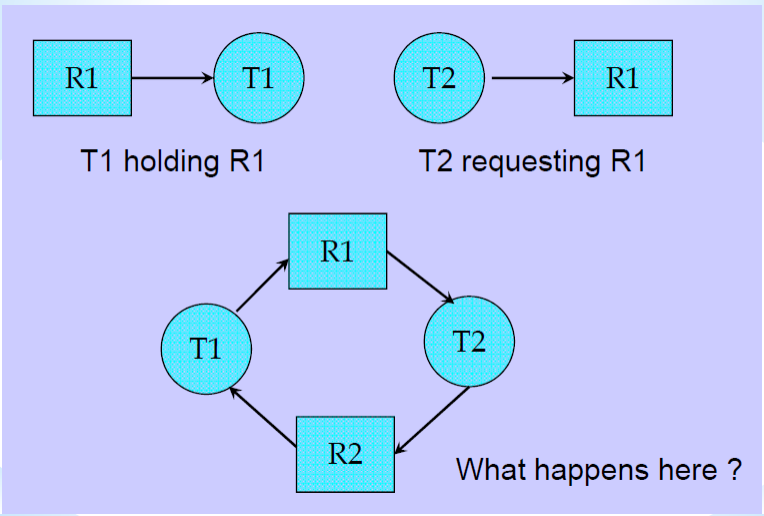
28

Deadlocks

- Deadlock is synchronization problem involving conflicting needs for resources in two or more concurrent processes
- Deadlock requires four conditions to occur:
  - Mutual Exclusion – two tasks use resource at a time
  - Resource Holding – task holds resource and requests another
  - No Preemption – resource allocated for as long as needed
  - Circular Wait – task waits for another (linear ordering of resources)

29

Deadlock Model



30

Example: Deadlock on Message

- Deadlock occurs when the receiver blocks:

```
//task T1
…
receive T2
…
send(T2, messageA)
```

```
//task T2
…
receive T1
…
send(T1, messageB)
```

31

Summary

- Binary Semaphores
  - Create a binary semaphore for the given event
  - Tasks waiting for the event blocks on a *semTake()*
  - Task or ISR detecting the event calls *semGive()* or *semFlush()*
- Mutual Exclusion Semaphores (have owners)
  - Create a mutual exclusion semaphore to guard the resource.
  - Before accessing the resource, call *semTake()*
  - To release the resource, call *semGive()*
- Caveats:
  - Keep critical sections short
  - Better make all accesses to the resource through a library of routines
  - Mutex semaphores can not be used at interrupt time
  - If not properly used may cause deadlocks
  - *taskLock/Unlock* and *intLock/Unlock* may be also an option

32

Next Week

- No Class
- Teamwork on semester project

- Monday 10/05/2015
  - Team project proposal presentation
  - Reading: Chapter 7
  - Lab1 Due at beginning of class.

33

Lab Session on Semaphores

34

Semester Project Teams

- Team1
  - Herandy Vasquez
  - Edgar Sanchez
  - Walter Copenhaver
  - Juan Angulo
- Team2
  - John Vasquez
  - Esthela Gallardo
  - Eduardo Dragone
  - Viannete Felix
- Team3
  - Christian Hughes
  - Juan Soto
  - Vince Fonseca
  - Jorge Venegas
- Team4
  - Yanet Garay
  - Florencia Larsen
  - Elvijs Ostrovskis
  - Roberto Fierro
- Team5
  - Ernesto Vasquez
  - Julio Corral
  - Christopher Mckye
  - 'Upama Rahman
- Team6
  - Robert Mccain
  - David Reyes
  - Mark Eby

Semester Project

- DigitalHome (DH) Project:
  - www.softwarecasestudy.org
  - SRS 1.3
- I'm open for something totally different
- Team Assignment Monday 10/05/2015
  - Team presentation of project proposal (ppt due at class time)
  - Project non-functional requirements: The project must
    - A1) use at least two VxWorks concurrently running tasks.
    - A2) use at least one VxWorks semaphore for synchronization.
    - A3) use at least one VxWorks message queue for communication.
    - A4) use at least one VxWorks watchdog for timeout and/or periodic operation.
    - A5) use at least one VxWorks interrupt handler for external interrupt.
    - A6) use time-stamping for identifying time of events (use both ticks and sec/nsec)
    - A7) capture/record/display the system events and actions
    - A8) be in a form of a downloadable program

36