Real Time Operating System III

Semaphore

Multiple Semaphore:

- Most RTOS allow you to have multiple semaphores.
- Semaphores are all independent of one another. i.e, if one task takes Semaphore A, another task can take semaphore B without blocking.
- Similarly, if one task is waiting for semaphore C that task will still be blocked even if some other task releases Semaphore D.

Semaphore Problems:

Forgetting to release Semaphore.

Forgetting to take Semaphore.

Taking the wrong Semaphore.

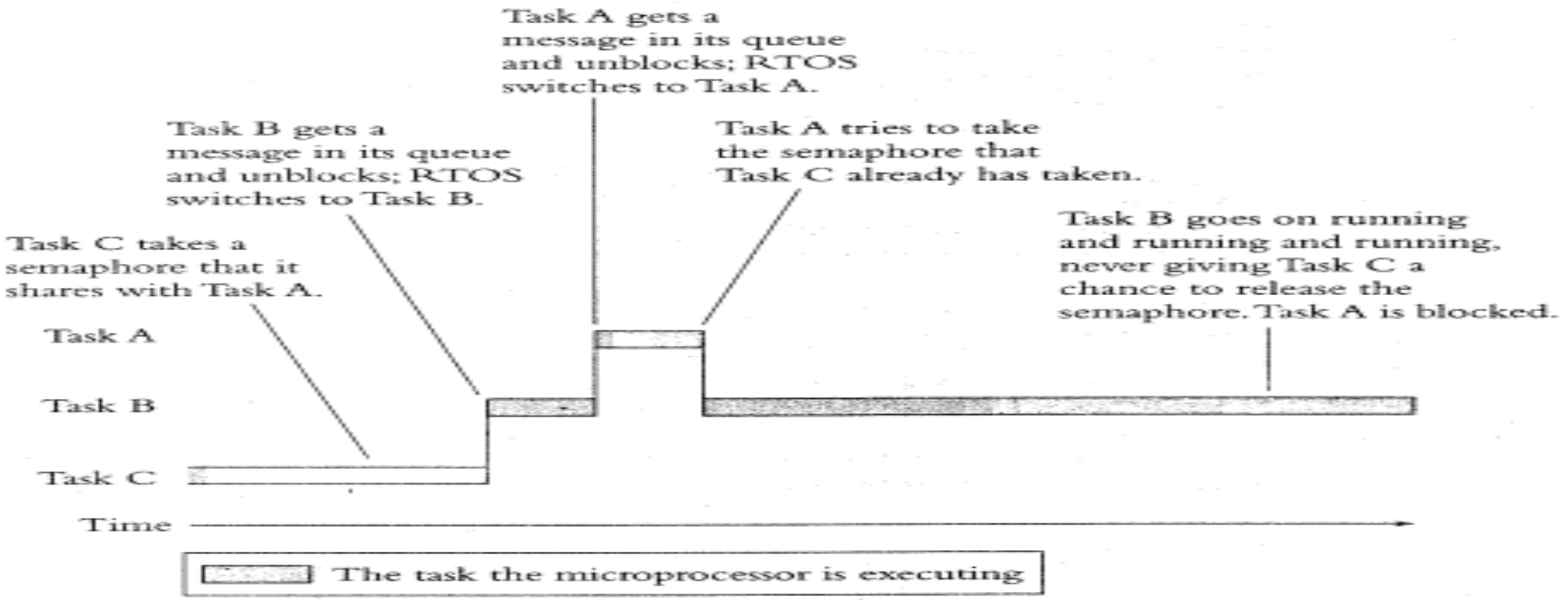
Holding a Semaphore for too long.

Semaphore

Different Types of Semaphore:

- Binary Semaphore
- Counting Semaphore
- Mutex Semaphore

Priority Inversion



Semaphore

Priority Inversion Problem can be reduced by using Priority Inheritance(which temporarily boost the priority of Low Priority Task i.e, Task C).

3 ways to protect shared data:

- Disabling Interrupts
- Disabling task switches
- Using Semaphores

Message Queue

Task must be able to communicate with one another to coordinate their activities or to share data.

Suppose we have 2 tasks which has to report on a network. A separate task called "ErrorTask" is created that is responsible for reporting the error conditions on the network. When T1,T2 discover error, it reports that error to ErrorTask and then does it job.

Use of Queue:

```
Void AddToQueue(int iData);
Void ReadFromQueue(int *p_iData);

Void Task1(void)
{
If(!!problem arises)
        vLogError(ERROR_TYPE_X);
}
```

Message Queue

```
Void Task2(void)
{
If(!!problem arises)
        vLogError(ERROR_TYPE_Y);
}

Void vlogError(int iErrorType)
{
AddToQueue(iErrorType);
}

Static int cErrors;
Void ErrorTask(void)
{
        int iErrorType;
        while(forever)
                {
                        ReadFromQueue(&iErrorType);
                        ++cErrors;
                }
}
```

Mailbox

Mailbox are much like Queues.

RTOS has functions to create, to write to and to read from Mailboxes.

Functions to check whether the mailbox contains any messages and to destroy it if it is no longer needed.

Variations:

- Some RTOS allow a certain number of messages in mailbox and some allow only one.
- Some RTOS allow unlimited number of messages in each mailbox.
- Some RTOS prioritize mailbox messages.

Mailbox

Example of Multi Task System:

int sndmsg(unsigned int uMbId, Void *p_vMsg, unsigned int uPriority);

void *rcvmsg(unsigned int uMbId, Unsigned int uTimeout);

void *chkmsg(unsigned int uMbId);

uMbId – identifies the mailbox on which to operate.

sndmsg function adds p_vMsg into the queue of messages held by the uMbId mailbox with the priority indicated by uPriority. Returns error if uMbId is invalid or if too many messages are already pending in mailboxes.

Rcvmsg function returns the highest priority message from the specified mailbox.

uTimeout Parameter limits how long it will wait if there are no messages.

chkmsg function returns the first message in the mailbox; it returns a null immediately if the mailbox is empty.

Pipes

Pipes are also much like Queues.

RTOS can create them, write to them and read from them.

- Some RTOS allow you to write message for varying lengths into pipes.
- In Some RTOS, pipes are byte oriented.

  Suppose we have three task Task A, Task B and Task C. Task A writes 11 bytes to the pipe, Task B writes 19 bytes to the Pipe. If Task C has to read 14 bytes from the Pipe then it will read 11 bytes from Task A and 3 bytes from Task B. 16 bytes will remain in the pipe.
- Some RTOS use standard C library functions like fread and fwrite to read from and write to pipes.

Events

An Event is a Boolean flag that can set or reset.

- More than one task can block waiting for same event and the RTOS will unblock all of them and run them in priority basis.
- RTOS form group of events and tasks can wait for any subset of events within the group.
- Different RTOS deal in different ways with the issue of resetting an event after it has occurred and tasks that were waiting for it have been unblocked. Some RTOS reset events automatically and other require task software to do this.

Comparison of Intertask Communication

- Semaphores are usually fastest and simplest method but only 1-bit message can pass through semaphore saying that it has been released.

- Events are a little more complicated than semaphore and take a little more microprocessor time than semaphores. The advantages of events over semaphores is that task can wait for any one of several events at the same time, whereas it can only wait for one semaphore.

- Queues allow you to send a lot of information from one task to another. The advantage of Queue over Events is that task can wait on only one queue or mailbox or pipe at a time, the fact that you can send data through a queue makes it even more flexible than events.

Kernel Services

Set of System Calls that can be used to perform operations on kernel objects or can be used in general to facilitate:

- Memory Management
- Timer Functions
- Interrupt Processing

Memory Management

- C library functions use malloc and free to allocate and free the memory.
- RTOS engineers usually avoid these functions since they are slow and unpredictable.
- They use functions that allocate and free fixed-size buffers since this is fast.
- You can set up pools, all the buffers are the same size.
- RTOS uses reqbuf and getbuf to allocate a memory buffer from a pool.

  void *getbuf(unsigned int uPoolId, unsigned int uTimeout);

  void *reqbuf(unsigned int uPoolId);

- Each returns a pointer to the allocated buffer; the only difference between them is that if no memory buffers are available, getbuf will block the task that calls it, whereas reqbuf will return a null pointer.

uPoolId – the pool from which the memory buffer is to be allocated.

uTimeout – length of time that the task is willing to wait for a buffer if none are free.

Memory Management

❖The relbuf function frees a memory buffer.

```
void relbuf(unsigned int uPoolId, void *p_vBuffer);
```

This function does not check the p_vBuffer really points to a buffer in the pool indicated by uPoolId. If you code passes invalid value for p_vBuffer, the results are catastrophic.

❖In most Embedded System, the application software gets the control first then the Operation System.

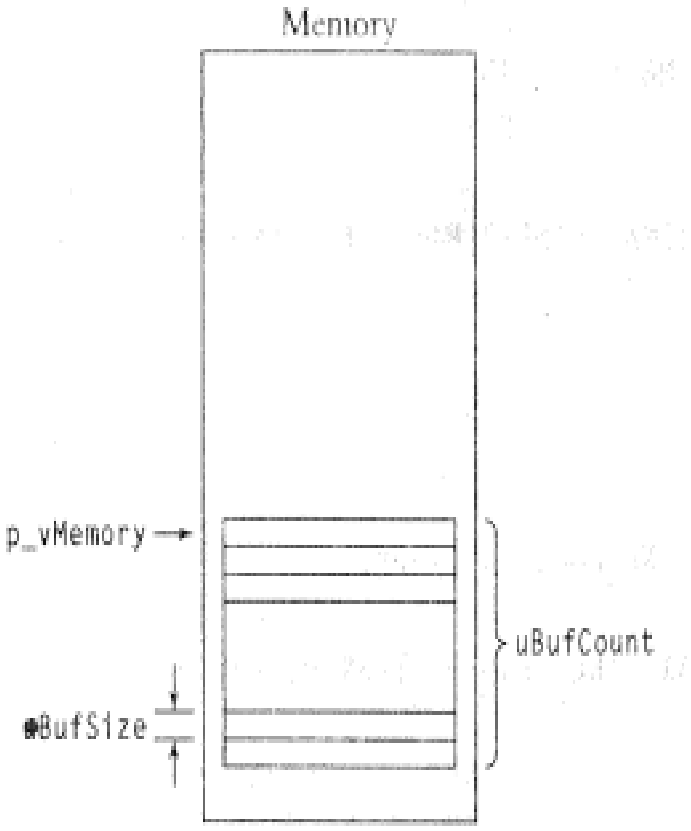
❖When it starts, RTOS has no way of knowing what memory is free and what memory your application is already using.

❖init_mem_pool will manage a pool of memory buffers.

Memory Management Contd…

init_mem_pool:

```
int init_mem_pool(
        unsigned  int uPoolId,
        void *p_vmemory,
        unsigned      int uBufSize,
        unsigned      int uBufCount,
        unsigned      int uPoolType);
```



Memory Fragmentation

❖Memory fragmentation is the state of the disk/drive of your computer where the files are not in continuity and there is some gap between files.

❖For Example: there are 25 files named a-y on a particular drive of your computer in continuity. You chose to delete 3 files namely d, m and r. The continuity is broken and this position is known as fragmented memory.

❖Disk Defragmentation is the process with which you can rectify this fault and increase the data access speed of your computer.

❖It rearranges the files on a disk drive in such a way that there are minimum gaps or none at all, with maximum continuity which increases the data access speed.

Stack Memory and Heap Memory

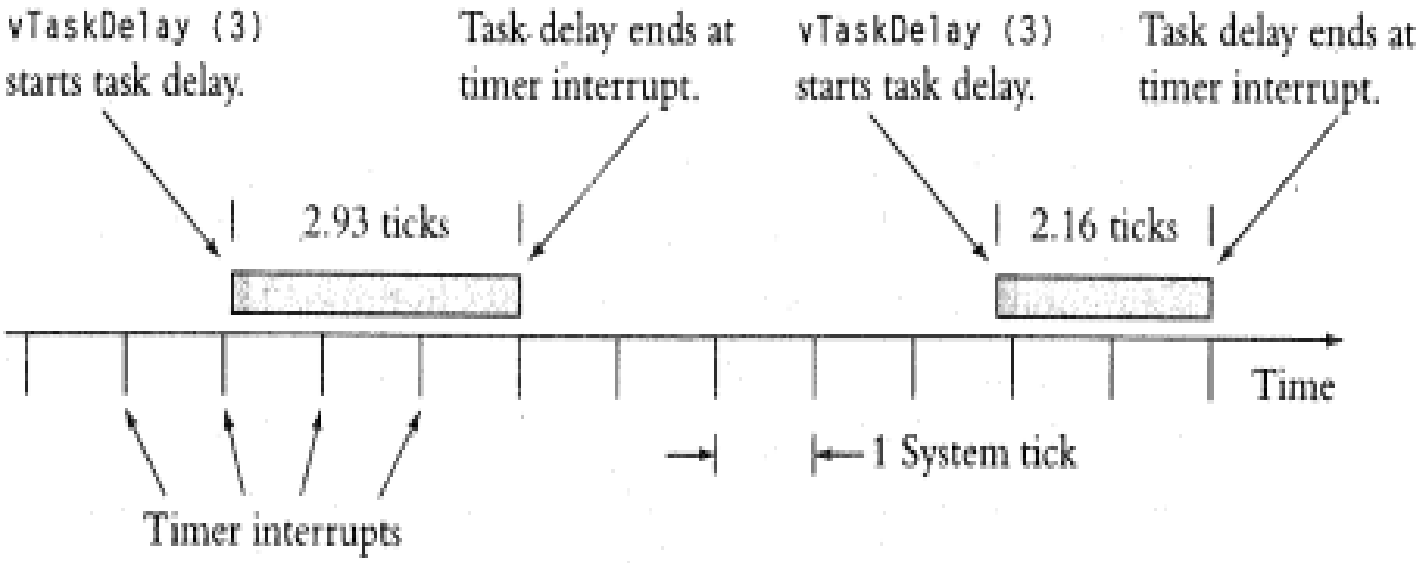
Stack Memory	Heap Memory
It is an area of RAM which stores temporary data .	It is an area of RAM that represents the dynamic memory of the system.
It is statically allocated.	It is dynamically allocated.
Operates in Last in first out basis.	Operates based on the address.
The types of data stored in stack include local variables, return address, functions arguments etc.	The types of data stored in heap include Transient data objects etc.
If the memory area allocated for the stack isn't large enough, the executing code writes to the area allocated below the stack and an overflow situation occurs.	Inefficient use of the allocated data space will corrupt the entire heap memory area and most likely result in an application crash with few traces of how the crash happened.
Overestimating stack usage wastes memory resources, which increases cost.	Less costly.

Timer Functions

- Most Embedded Systems keep track of passage of time.
- One Service that RTOS offer is a function that delays a task for a period of time(blocks until the period of time expires).
- **taskDelay** function takes the number of system ticks as its parameter.
- Delay produced by **taskDelay** are accurate to the nearest system tick.
- RTOS works by setting up a single hardware timer to interrupt periodically every millisecond and bases all timings on that interrupt. This timer is often called heartbeat timer.

Timer Functions

- If one of your tasks passes 3 to taskDelay that task will block until the hearbeat timer interrupts three times.
- If the system needs accurate timing:
  - One is to make the system tick short enough that RTOS timings fit your definition of extremely accurate.
  - Second is to use a separate hardware timer for those timings that must be extremely accurate.



Interrupt Processing

Rule 1: An interrupt routine must not call any RTOS function that might block the caller.

Example:

```
Static int iTemperatures[2];
Void interrupt vReadTemperatures(void)
{
GetSemaphore(SEMAPHORE_TEMPERATURE);/*Not Allowed */
iTemperatures[0]=!! Read in value from hardware;
iTemperatures[1]=!! Read in value from hardware;
GiveSemaphore(SEMAPHORE_TEMPERATURE);
}
Void interrupt vTaskTestTemperatures(void)
{
int iTemp0, iTemp1;
While(true){
GetSemaphore(SEMAPHORE_TEMPERATURE);
iTemp0=iTemperatures[0];
iTemp1=iTemperatures[1];
 GiveSemaphore(SEMAPHORE_TEMPERATURE);
If(iTemp0!=iTemp1)
            !!set off howling alarm.
}
```