Real-Time Operating Systems (Part I)

Embedded Software Design

熊博安

國立中正大學資訊工程研究所

pahsiung@cs.ccu.edu.tw

Textbooks: (a) Embedded Software Primer, David E. Simon, Addison Wesley
(b) Programming Embedded Systems with C and GNU Development Tools, 2nd Edition, M. Barr and A. Massa, Oreilly Media Inc.

Contents

- Tasks and Task States
- Tasks and Data
- Semaphores and Shared Data

Desktop OS v/s RTOS (1)

- Desktop OS:
  - Boot: OS takes control, sets up environment
  - Applications: Run under OS, independently
- Real-Time Embedded OS:
  - Boot: Application takes control, starts RTOS
  - Application: linked with OS, tied together

Desktop OS v/s RTOS (2)

- Desktop OS:
  - multiuser → need security, protection, etc.
  - check validity of pointers into system function
- RTOS:
  - single user → no need of security
  - for performance, pointers are not checked

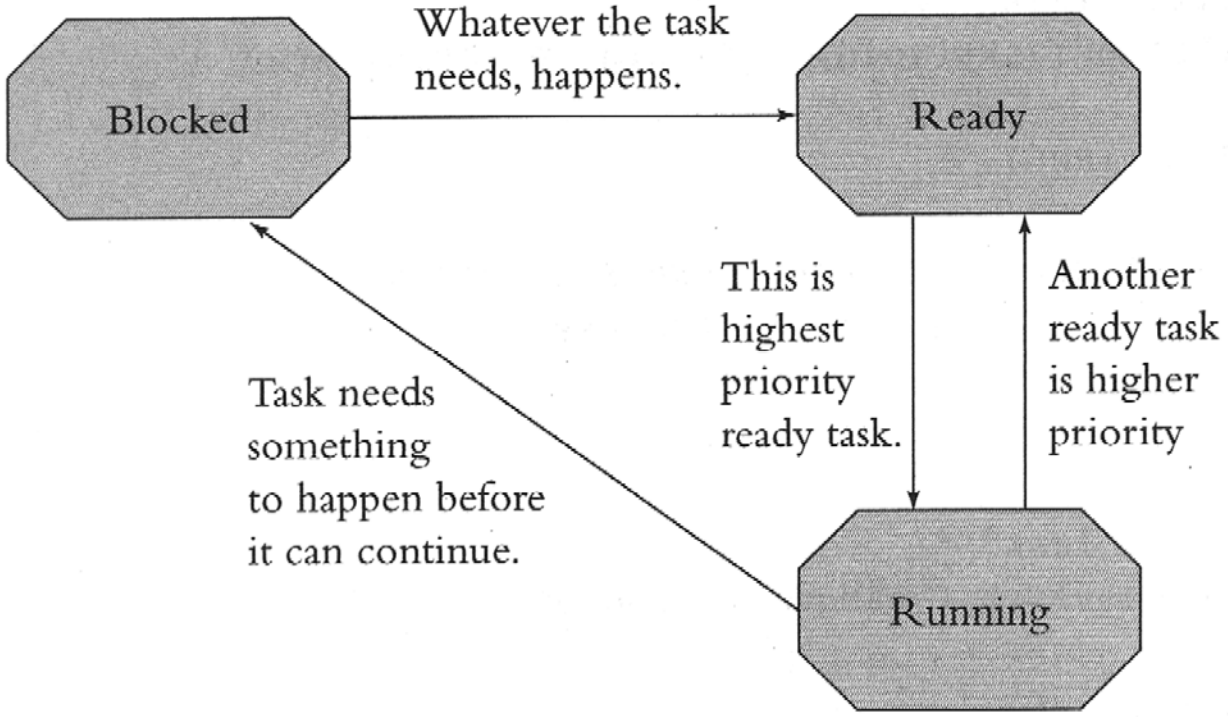
Desktop OS v/s RTOS (3)

- Desktop OS:
  - limited configuration
- RTOS:
  - extensive configuration: leave out all what you don’t need, e.g. file managers, I/O drivers, utilities, and even memory management

Tasks and Task States

- Task: a subroutine in RTOS
- Task States:
  - Running: using microprocessor to execute instructions
  - Ready: has instructions for microprocessor to execute, but is not yet executing
  - Blocked: has nothing for microprocessor, waiting for external event, e.g. network data handler with no data from network, button response task with button not yet pressed

Task States


Whatever the task
needs, happens.
Blocked
Ready
This is
highest
priority
ready task.
Another
ready task
is higher
priority
Task needs
something
to happen before
it can continue.
Running

Other Task States

- Finer distinctions of ready and blocked states:
  - suspended
  - pended
  - waiting
  - dormant
  - delayed

The Scheduler

- Keeps track of the states of each task
- Decides which task should run
- Based on priorities
  - priorities set by user
  - non-blocked task with highest priority runs

Consequences (1)

- Can a task go from ready to blocked state?
- Ans: NO!
- Reason:
  - A task goes to blocked state only when it decides for ITSELF if it needs to wait for something or has nothing to do.
  - To make this decision, it needs to execute some code, thus it is “running” before “blocked”!

Consequences (2)

- Can a blocked task wake up on its own (without any other task helping)?
- Ans: NO!
- Reason:
  - A blocked task will have something for microprocessor to do only if some OTHER task interrupts it and tells it that whatever it was waiting for has happened!
  - Otherwise, the task will be blocked forever.

Consequences (3)

- Can a task switch from ready to running or vice-versa on its own?
- Ans: NO!
- Reason:
  - Scheduler does all the switching between ready and running states.
  - A blocked task can move to ready, and immediately switch to running (if it has the highest priority).

Q/A about scheduler and task states (1)

- **Qs**: How does the scheduler know when a task has become blocked or unblocked?
- **Ans**: RTOS provides functions for tasks to tell scheduler:
  - what events the tasks want to wait for
  - to signal that events have happened

Q/A about scheduler and task states (2)

- **Qs**: What happens if all the tasks are blocked?
- **Ans**: Scheduler spins in some tight loop in the RTOS.
- If nothing ever happens, that’s your fault!
- Make sure something happens sooner or later by having an interrupt routine call some RTOS function to unblock a task.

Q/A about scheduler and task states (3)

- **Qs**: What if two tasks with the same priority are ready?
- **Ans**: Depends on RTOS.
  - Illegal to have two tasks with same priority
  - Time-slice between the two tasks
  - Run one until blocked, then run the other
  - Backup scheduling policy: round-robin, FIFO

Q/A about scheduler and task states (4)

- Qs. If one task is running and another, higher-priority task unblocks, does the task that is running get stopped and moved to the ready state right away?
- Ans.
  - Preemptive RTOS: Yes!
  - Nonpreemptive RTOS: No!

A Simple Example

- The classic situation

```
/* "Button Task" */
void vButtonTask (void)   /* High priority */
{
   while (TRUE)
   {
      !! Block until user pushes a button
      !! Quick: respond to the user
   }
}
```

This task will be unblocked as soon as the user pushes a button.

A Simple Example

- The computational task

```
/* "Levels Task" */
void vLevelsTask (void)   /* Low priority */
{
   while (TRUE)
   {
      !! Read levels of floats in tank
      !! Calculate average float level

      !! Do some interminable calculation
      !! Do more interminable calculation
      !! Do yet more interminable calculation

      !! Figure out which tank to do next
   }
}
```

Blinking LED Example

- Two Operating Systems
  - eCos: an open-source real-time OS for embedded systems
    - Task = thread
  - Linux: embedded version of an open-source desktop OS
    - Task = pthread

Blinking LED in eCos

```
#define TICKS_PER_SECOND (100)
#define LED_TASK_STACK_SIZE (4096)
#define LED_TASK_PRIORITY (12)
/* Declare the task variables. */
unsigned char
  ledTaskStack[LED_TASK_STACK_SIZE];
cyg_thread ledTaskObj;
cyg_handle_t ledTaskHdl;
```

Task Information

Task Handle

Blinking LED in eCos

```
#include <cyg/kernel/kapi.h>
#include "led.h"
void blinkLedTask(cyg_addrword_t data)
{
  while (1)
  {
      /* Delay for 500 milliseconds. */
      cyg_thread_delay(TICKS_PER_SECOND / 2);
      ledToggle( );
} }
```

Blinking LED in eCos

```
void cyg_user_start(void)
{
    /* Configure the green LED control pin. */
    ledInit( );

    /* Create the LED task. */
    cyg_thread_create(LED_TASK_PRIORITY, blinkLedTask,
          (cyg_addrword_t)0, "LED Task", (void *)ledTaskStack,
          LED_TASK_STACK_SIZE, &ledTaskHdl, &ledTaskObj);

    /* Notify the scheduler to start running the task. */
    cyg_thread_resume(ledTaskHdl);
}
```

Scheduler runs only after cyg_user_start() exits!

Blinking LED in Linux

```
#include <unistd.h>
#include "led.h”
void blinkLedTask(void *param)
{
  while (1)
  {
     /* Delay for 500 milliseconds. */
     usleep(50000);
     ledToggle( );
  }
}
```

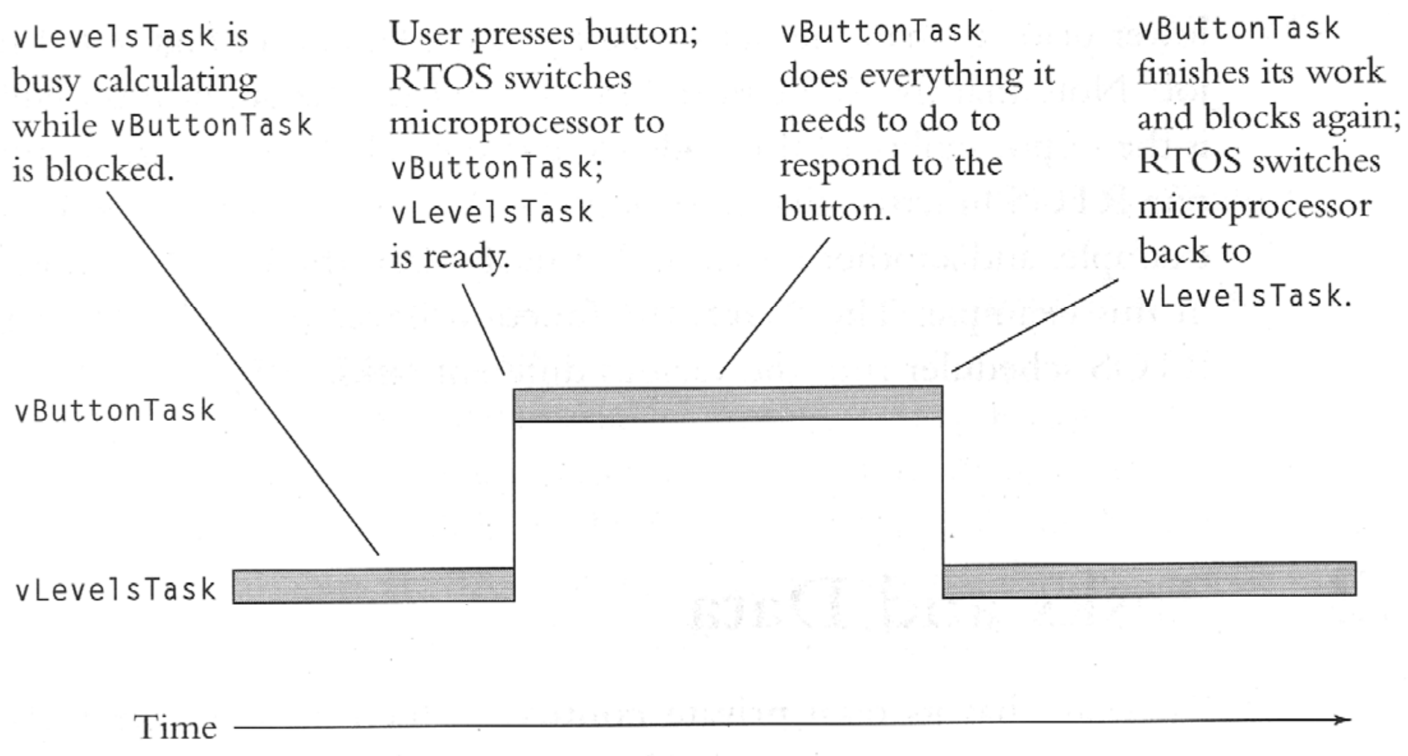
Time unit:
microseconds

Blinking LED in Linux

```
#include <pthread.h>
pthread_t ledTaskObj;
int main(void)
{
  /* Configure the green LED control pin. */
  ledInit( );
  /* Create the LED task using the default task
  attributes. No parameters */
  pthread_create(&ledTaskObj, NULL,
      (void *)blinkLedTask, NULL);
  /* Allow the LED task to run. */
  pthread_join(ledTaskObj, NULL);
  return 0;
}
```

Suspend main() until the pthread terminates!

A Simple Example (RTOS tasks)



The microprocessor's attention switches from task to task in response to the buttons.

A Simple Example (main())

- Assigning the priorities

```
void main (void)
{
    /* Initialize (but do not start) the RTOS */
    InitRTOS ();

    /* Tell the RTOS about our tasks */
    StartTask (vRespondToButton, HIGH_PRIORITY);
    StartTask (vCalculateTankLevels, LOW_PRIORITY);

    /* Start the RTOS.  (This function never returns.) */
    StartRTOS ();
}
```

Features of Using an RTOS

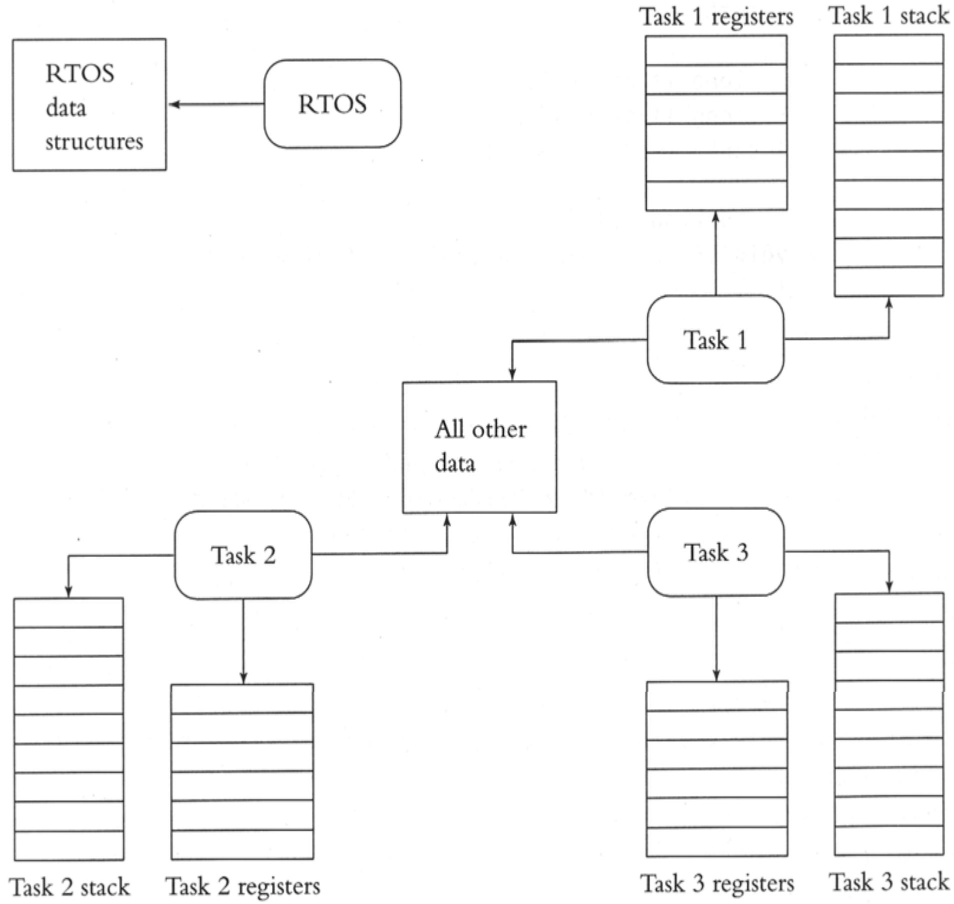
- Two tasks can be written independently of one another, and the system will still respond well.
- The RTOS will make the response good whenever the user presses a button by turning the microprocessor over to the task that responds to the buttons immediately.

Tasks and Data: Context

- Each task has its own private context.
  - the register values,
  - a program counter,
  - a stack.
- All other data is shared among all of the tasks in the system.
  - Global
  - static
  - initialized
  - ...

An Example

- A common data area



Sharing Data

- Two main functions

vRespondToButton

```
struct
{
   long lTankLevel;
   long lTimeUpdated;
} tankdata[MAX_TANKS];

/* "Button Task" */
void vRespondToButton (void)  /* High priority */
{
   int i;
   while (TRUE)
   {
      !! Block until user pushes a button
      i = !! ID of button pressed;
      printf ("\nTIME: %08ld   LEVEL: %08ld",
         tankdata[i].lTimeUpdated,
         tankdata[i].lTankLevel);
   }
}
```

vCalculateTankLevels

```
/* "Levels Task" */
void vCalculateTankLevels (void)   /* Low priority */
{
   int i = 0;
   while (TRUE)
   {
      !! Read levels of floats in tank i
      !! Do more interminable calculation
      !! Do yet more interminable calculation

      /* Store the result */
      tankdata[i].lTimeUpdated = !! Current time
         /* Between these two instructions is a
            bad place for a task switch */
      tankdata[i].lTankLevel = !! Result of calculation

      !! Figure out which tank to do next
      i = !! something new
   }
}
```

Shared-Data Problems

- Bug in previous slide (example task code)
- vCalculateTankLevels() and vRespondToButton() share the same data structure: tankdata[MAX_TANKS]
- The shared data structure could get corrupted or inconsistent (refer to Chapter 4)

Shared-Data Problems

- Another example
  - Task2 interrupts Task1

```
void Task1 (void)
{
    .
    .
    .
    vCountErrors (9);
    .
    .
    .
}

void Task2 (void)
{
    .
    .
    .
    vCountErrors (11);
    .
    .
    .
}

static int cErrors;

void vCountErrors (int cNewErrors)
{
    cErrors += cNewErrors;
}
```

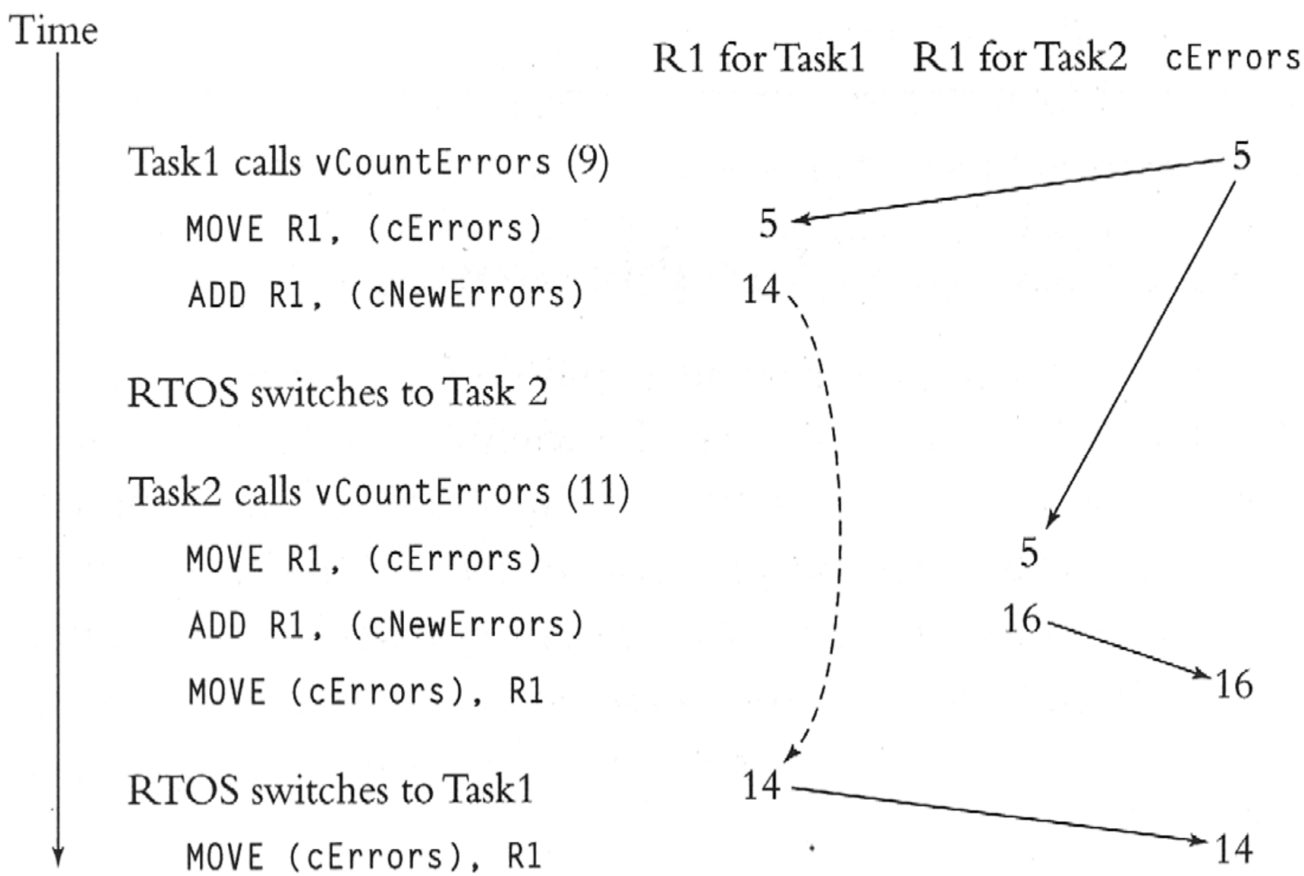
A clearer examination

- The assembly code

```
; Assembly code for vCountErrors
;
; void vCountErrors (int cNewErrors)
;{
;       cErrors += cNewErrors;
        MOVE R1, (cErrors)
        ADD R1, (cNewErrors)
        Move (cErrors), R1
        RETURN
;}
```

A clearer examination

- The flow



Reentrancy

- Reentrant function
  - can be called by more than one task and
  - will always work correctly,
  - even if RTOS switches from one task to another in the middle of executing the function.
- vCountErrors() is not a reentrant function.

How to check reentrancy?

- Apply 3 rules to check if a function is reentrant

1. Does not use variables in a nonatomic way unless
   - they are stored on stack of the calling task, or
   - they are private variables of the task
2. Does not call any non-reentrant functions
3. Does not use hardware in a nonatomic way

Review of C Variable Storage

```
static int static_int;
int public_int;
int initialized = 4;
char *string = "Where does this string go?";
void *vPointer;

void function (int parm, int *parm_ptr)
{
   static int static_local;
   int local;
   .
   .
   .
}
```

fixed locations in memory

on stack

fixed location in memory

on stack

Applying Reentrancy Rules

Qs: Is this reentrant?

Ans: NO!

Violates rules:

(1) non-atomic use of fError

(2) printf() may be non-reentrant

```
BOOL fError;   /* Someone else sets this */

void display (int j)
{
   if (!fError)
   {
      printf ("\nValue: %d", j);
      j = 0;
      fError = TRUE;
   }
   else
   {
      printf ("\nCould not display value");
      fError = FALSE;
   }
}
```

Gray Areas of Reentrancy

- Is the following code reentrant?

```
static int cErrors;
void vCountErrors(void) {
    ++cErrors;
}
```

Is incrementing `cErrors` atomic?

- Maybe! Depends on microprocessor and compiler
  - 8051: 9 assembly instructions (non-reentrant!)
  - 80x86: 2 assembly instructions (reentrant!)

Non-reentrant function in 8051

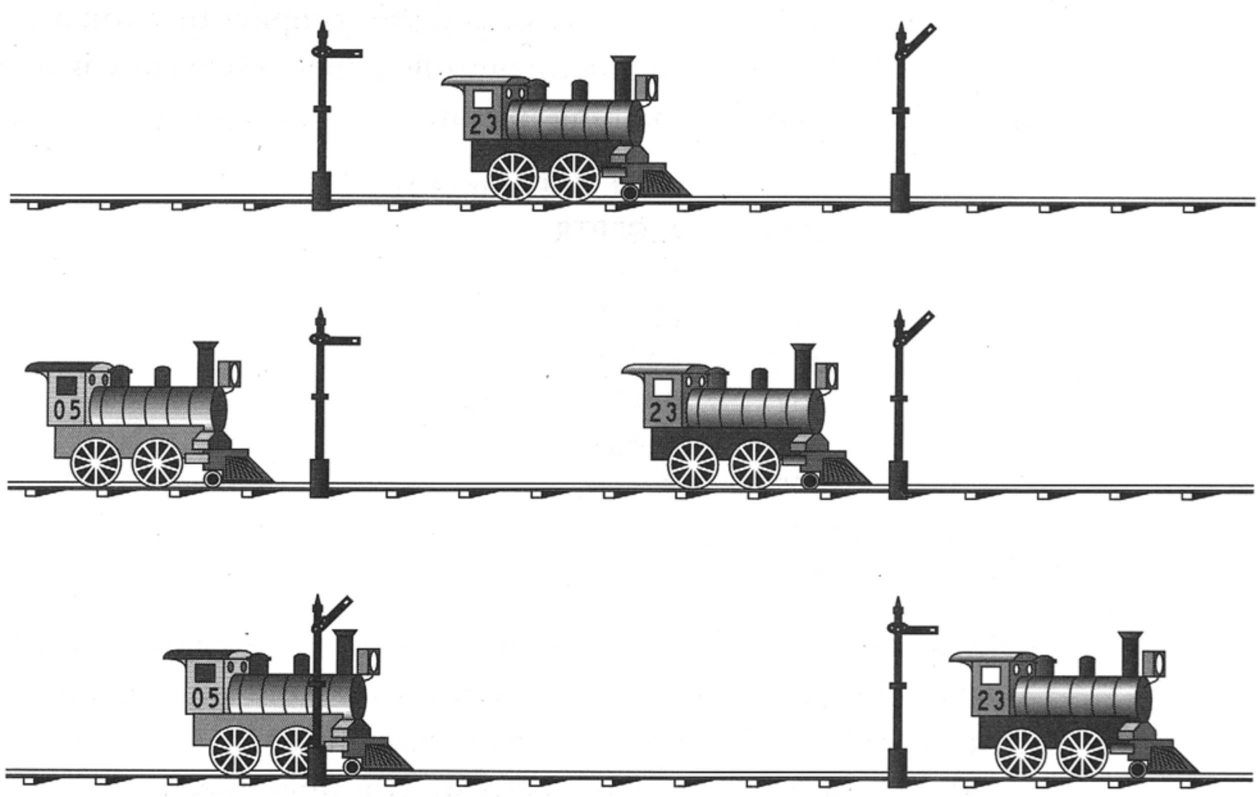
```
  MOVDPTR,#cErrors+01H
  MOVX    A,@DPTR
  INC     A
  MOVX    @DPTR,A
  JNZ     noCarry
  MOVDPTR,# cErrors
  MOVX    A,@DPTR
  MOVX    @DPTR,A
noCarry:
  RET
```

Rentrant Function in 80x86

```
INC     (cErrors)
RET
```

Semaphores and Shared Data

- The scenario



RTOS Semaphores

- Functions:
  - raise & lower
  - get & give
  - take & release
  - pend & post
  - p & v
  - wait and signal
  - take (for lower) & release (for raise)

RTOS functions for binary semaphore

- A binary semaphore
  - Only one task can have the semaphore at a time.
- TakeSemaphore
  - block until the semaphore is released
  - take the semaphore
- ReleaseSemaphore
  - release a taken semaphore

Semaphores Protect Data

- The tank application

```
struct
{
   long lTankLevel;
   long lTimeUpdated;
} tankdata[MAX_TANKS];

/* "Button Task" */
void vRespondToButton (void)  /* High priority */
{
   int i;
   while (TRUE)
   {
      !! Block until user pushes a button
      i = !! Get ID of button pressed
      TakeSemaphore ();
      printf ("\nTIME: %08ld   LEVEL: %08ld",
         tankdata[i].lTimeUpdated,
         tankdata[i].lTankLevel);
      ReleaseSemaphore ();
   }
}
```

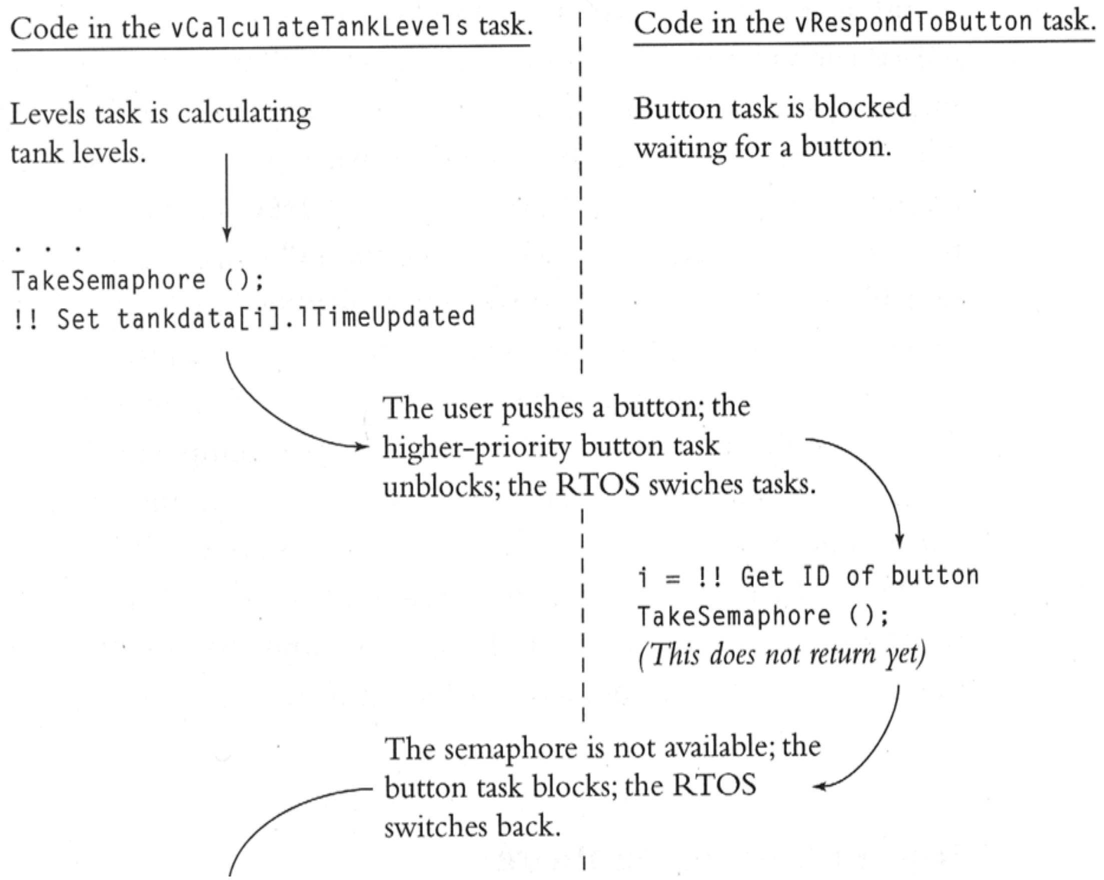
```
/* "Levels Task" */
void vCalculateTankLevels (void)   /* Low priority */
{
   int i = 0;
   while (TRUE)
   {
      .
      .
      .
      TakeSemaphore ();
      !! Set tankdata[i].lTimeUpdated
      !! Set tankdata[i].lTankLevel
      ReleaseSemaphore ();
      .
      .
      .
   }
}
```

The Sequence of Events

- If a user presses a button while the levels task is still modifying the data and still has the semaphore,
  - The RTOS will switch to the "button task," just as before, moving the levels task to the ready state.
  - When the button task tries to get the semaphore by calling TakeSemaphore it will block because the levels task already has the semaphore.
  - The RTOS will then look around for another task to run.
  - When the levels task releases the semaphore by calling ReleaseSemaphore, the button task will no longer be blocked.

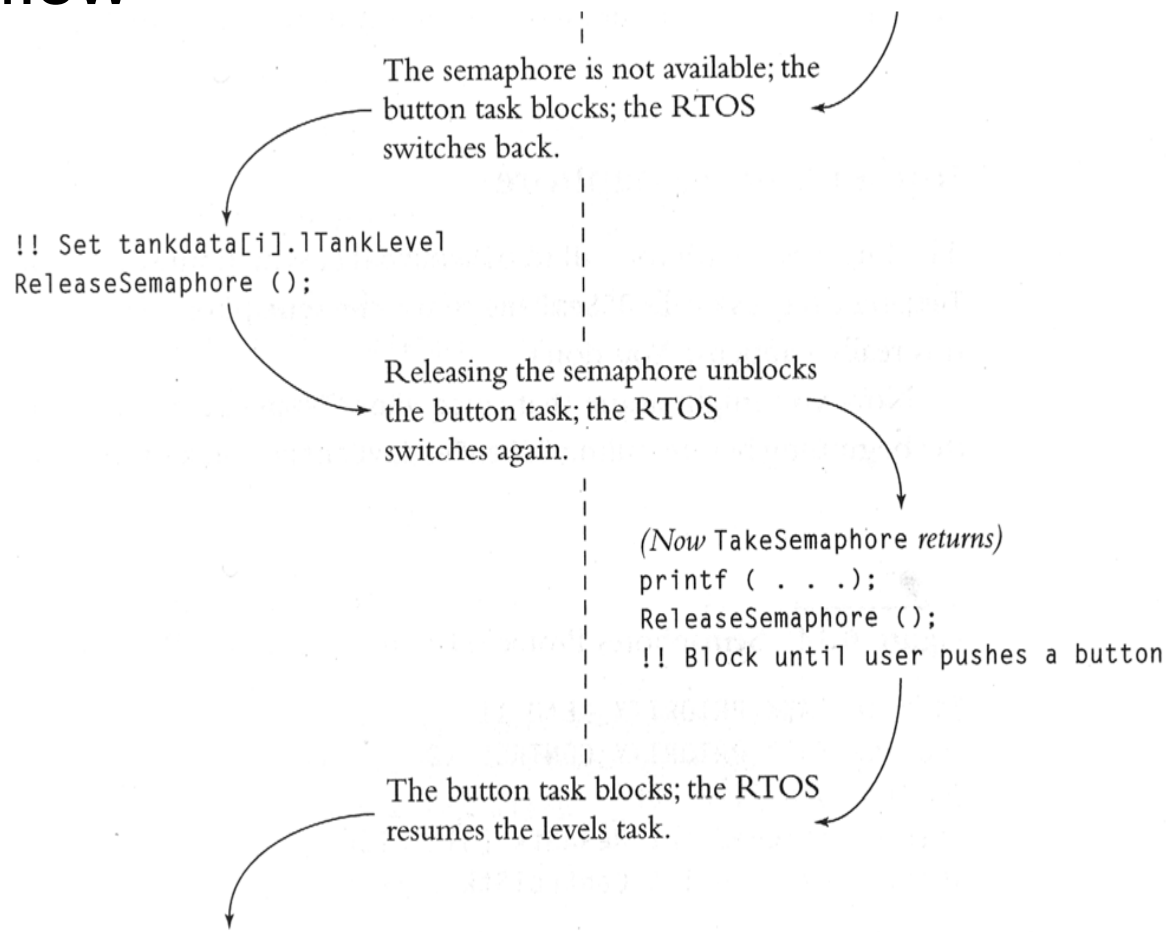
Execution Flow

- The flow



Execution Flow

- The flow



The Nuclear Reactor System

- MicroC/OS RTOS
- Semaphore-related functions
  - OSSemPost(): release the semaphore
  - OSSemPend(): take the semaphore
  - OSSemCreate(): initialize the semaphore
- Related data structures
  - OS_EVENT: the data representing the semaphore
  - WAIT_FOREVER: indicates that the task making the call is willing to wait forever
- Other functions
  - OSTimeDly(): block functions

The Code

- The data structures

```
#define TASK_PRIORITY_READ 11
#define TASK_PRIORITY_CONTROL 12
#define STK_SIZE 1024
static unsigned int ReadStk [STK_SIZE];
static unsigned int ControlStk [STK_SIZE];

static int iTemperatures[2];
OS_EVENT *p_semTemp;
```

The Code

- The main function

```
void main (void)
{
    /* Initialize (but do not start) the RTOS */
    OSInit ();

    /* Tell the RTOS about our tasks */
    OSTaskCreate (vReadTemperatureTask,  NULLP,
        (void *)&ReadStk[STK_SIZE], TASK_PRIORITY_READ);
    OSTaskCreate (vControlTask,  NULLP,
        (void *)&ControlStk[STK_SIZE], TASK_PRIORITY_CONTROL);

    /* Start the RTOS.  (This function never returns.) */
    OSStart ();
}
```

The Code

- Two tasks

```
void vReadTemperatureTask (void)
{
   while (TRUE)
   {
      OSTimeDly (5); /* Delay about 1/4 second */

      OSSemPend (p_semTemp, WAIT_FOREVER);
      !! read in iTemperatures[0];
      !! read in iTemperatures[1];
      OSSemPost (p_semTemp);
   }
}
```

```
void vControlTask (void)
{
   p_semTemp = OSSemInit (1);
   while (TRUE)
   {
      OSSemPend (p_semTemp, WAIT_FOREVER);
      if (iTemperatures[0] != iTemperatures[1])
         !! Set off howling alarm;
      OSSemPost (p_semTemp);

      !! Do other useful work
   }
}
```

A potential bug!

Initializing Semaphores in Nuclear Reactor

- How do you know that OSSemCreate happens before OSSemPend in vReadTemperatureTask?
  - Because of delay by calling OSTimeDly(5)?
    - Some higher priority task might take up all the delay introduced!
  - Change Task Priorities?
    - Someone later might change back the task priorities and not know of the time bomb!
- Correct solution
  - Place OSSemCreate BEFORE OSStart in main!

Reentrancy and Semaphores

- Now adding a semaphore to the previous code (using Nucleus RTOS system calls)

```
static int cErrors;

void vCountErrors (int cNewErrors)
{
   cErrors += cNewErrors;
}
```

```
static int cErrors;
static NU_SEMAPHORE semErrors;

void vCountErrors (int cNewErrors)
{
   NU_Obtain_Semaphore (&semErrors, NU_SUSPEND);
   cErrors += cNewErrors;
   NU_Release_Semaphore (&semErrors);
}
```

Atomic now!

Mutex Task Synchronization in eCos

- Two tasks share a variable **`gSharedVariable`**
  - **`incrementTask`**: increments the variable value
  - **`decrementTask`**: decrements the variable value
- A mutex is used to protect this shared variable

Mutex Task Synchronization in eCos (main program)

```
#include <cyg/kernel/kapi.h>
#include <cyg/infra/diag.h>
cyg_mutex_t sharedVariableMutex;
int32_t gSharedVariable = 0;

void cyg_user_start(void) {
    /* Create the mutex for accessing the shared variable. */
    cyg_mutex_init(&sharedVariableMutex);

    /* Create the increment and decrement tasks. */
    cyg_thread_create(INCREMENT_TASK_PRIORITY, incrementTask,
          (cyg_addrword_t)0, "Increment Task",
                    (void *)incrementTaskStack, INCREMENT_TASK_STACK_SIZE,
                    &incrementTaskHdl, &incrementTaskObj);
    cyg_thread_create(DECREMENT_TASK_PRIORITY, decrementTask,
                    (cyg_addrword_t)0, "Decrement Task",
                    (void *)decrementTaskStack, DECREMENT_TASK_STACK_SIZE,
                    &decrementTaskHdl, &decrementTaskObj);

    /* Notify the scheduler to start running the tasks. */
    cyg_thread_resume(incrementTaskHdl);
    cyg_thread_resume(decrementTaskHdl);
    diag_printf("eCos mutex example.\\n");
}
```

Mutex Task Synchronization in eCos (incrementTask)

```
void incrementTask(cyg_addrword_t data) {
  while (1) {
      /* Delay for 3 seconds. */
      cyg_thread_delay(TICKS_PER_SECOND * 3);

      /* Wait for the mutex to become available. */
      cyg_mutex_lock(&sharedVariableMutex);
      gSharedVariable++;
      diag_printf("Increment Task: shared variable
                   value is %d\\n", gSharedVariable);

      /* Release the mutex. */
      cyg_mutex_unlock(&sharedVariableMutex);
  }
}
```

Mutex Task Synchronization in eCos (decrementTask)

```
void decrementTask(cyg_addrword_t data) {
   while (1) {
        /* Delay for 7 seconds. */
        cyg_thread_delay(TICKS_PER_SECOND * 7);

        /* Wait for the mutex to become available. */
        cyg_mutex_lock(&sharedVariableMutex);
        gSharedVariable--;
        diag_printf("Decrement Task: shared variable
                     value is %d\\n", gSharedVariable);

        /* Release the mutex. */
        cyg_mutex_unlock(&sharedVariableMutex);
   }
}
```

Mutex Task Synchronization in Linux (main program)

```
#include <pthread.h>
pthread_mutex_t sharedVariableMutex;
int32_t gSharedVariable = 0;

int main(void) {
   /* Create the mutex for accessing the shared variable using
    * the default attributes. */
   pthread_mutex_init(&sharedVariableMutex, NULL);

   /* Create the increment and decrement tasks */
   pthread_create(&incrementTaskObj, NULL,
                  (void *)incrementTask, NULL);
   pthread_create(&decrementTaskObj, NULL,
                  (void *)decrementTask, NULL);

   /* Allow the tasks to run. */ pthread_join(incrementTaskObj, NULL);
   pthread_join(decrementTaskObj, NULL);
   return 0;
}
```

Mutex Task Synchronization in Linux (incrementTask)

```
#include <stdio.h>
#include <unistd.h>

void incrementTask(void *param) {
   while (1) {
        /* Delay for 3 seconds. */
        sleep(3);

        /* Wait for the mutex before accessing the GPIO
               registers. */
        pthread_mutex_lock(&sharedVariableMutex);
        gSharedVariable++;
        printf("Increment Task: shared variable value is %d\\n",
               gSharedVariable);
        /* Release the mutex for other task to use. */
        pthread_mutex_unlock(&sharedVariableMutex);
   }
}
```

Mutex Task Sychronization in Linux (decrementTask)

```
void decrementTask(void *param) {
  while (1) {
      /* Delay for 7 seconds. */
      sleep(7);

      /* Wait for the mutex to become available. */
      pthread_mutex_lock(&sharedVariableMutex);
      gSharedVariable--;
      printf("Decrement Task: shared variable value is
            %d\\n", gSharedVariable);

      /* Release the mutex. */
      pthread_mutex_unlock(&sharedVariableMutex);
  }
}
```

Multiple Semaphores

- Some RTOS allows you to have as many semaphores as you like.
- Advantage
  - In a system with only one semaphore, if the lower-priority task takes the semaphore to change data, the higher-priority task might block waiting for the semaphore.
- How does the RTOS know which semaphore protects which data?
  - It doesn’t.
  - You must decide what shared data each of your semaphores protects!

Semaphores as a Signaling Device

- Another common use of semaphores is as a simple way to communicate
  - from one task to another or
  - from an interrupt routine to a task.
- For example,
  - printing task
  - formatting task

Semaphores as a Signaling Device

- Data structures

```
/* Place to construct report. */
static char a_chPrint[10][21];

/* Count of lines in report. */
static int iLinesTotal;

/* Count of lines printed so far. */
static int iLinesPrinted;

/* Semaphore to wait for report to finish. */
static OS_EVENT *semPrinter;
```

Semaphores as a Signaling Device

- Functions

```
void vPrinterTask(void)
{
   BYTE byError;   /* Place for an error return. */
   Int wMsg;

   /* Initialize the semaphore as already taken. */
   semPrinter = OSSemInit(0);

   while (TRUE)
   {
      /* Wait for a message telling what report to format. */
      wMsg = (int) OSQPend (QPrinterTask, WAIT_FOREVER, &byError);

      !! Format the report into a_chPrint
      iLinesTotal = !! count of lines in the report

      /* Print the first line of the report */
      iLinesPrinted = 0;
      vHardwarePrinterOutputLine (a_chPrint[iLinesPrinted++]);

      /* Wait for print job to finish. */
      OSSemPend (semPrinter, WAIT_FOREVER, &byError);
   }
}
```

Semaphores as a Signaling Device

- Functions

```
void  vPrinterInterrupt (void)
{
   if (iLinesPrinted == iLinesTotal)
      /* The report is done.  Release the semaphore. */
      OSSemPost (semPrinter);

   else
      /* Print the next line. */
      vHardwarePrinterOutputLine (a_chPrint[iLinesPrinted++]);
}
```

Semaphore Task Synchronization in eCos (main program)

```
#include <cyg/kernel/kapi.h>
#include <cyg/infra/diag.h>
#include "led.h"
cyg_sem_t semButton;

void cyg_user_start(void) {
    /* Configure the green LED control pin. */
    ledInit( );
    /* Create the semaphore with an initial value of zero. */
    cyg_semaphore_init(&semButton, 0);
    /* Create the producer and consumer tasks. */
    cyg_thread_create(PRODUCER_TASK_PRIORITY, producerTask,
          (cyg_addrword_t)0, "Producer Task", (void *)producerTaskStack,
          PRODUCER_TASK_STACK_SIZE, &producerTaskHdl, &producerTaskObj);
    cyg_thread_create(CONSUMER_TASK_PRIORITY, consumerTask,
          (cyg_addrword_t)0, "Consumer Task", (void *)consumerTaskStack,
          CONSUMER_TASK_STACK_SIZE, &consumerTaskHdl, &consumerTaskObj);
    /* Notify the scheduler to start running the tasks. */
    cyg_thread_resume(producerTaskHdl);
    cyg_thread_resume(consumerTaskHdl);
    diag_printf("eCos semaphore example - press button SW0.\\n");
}
```

Semaphore Task Synchronization in eCos (producerTask)

```
#include "button.h"

void producerTask(cyg_addrword_t data) {
   int buttonOn;

   while (1) {
        /* Delay for 10 milliseconds. */
        cyg_thread_delay(TICKS_PER_SECOND / 100);

        /* Check whether the SW0 button has been pressed. */
        buttonOn = buttonDebounce( );

        /* If button SW0 was pressed, signal consumer task. */
        if (buttonOn) cyg_semaphore_post(&semButton);
   }
}
```

Semaphore Task Synchronization in eCos (debounce)

```
int buttonDebounce(void) {
   static uint16_t buttonState = 0;
   uint8_t pinState; pinState = buttonRead( );

   /* Store the current debounce status. */
   buttonState = ((buttonState << 1) | pinState | 0xE000);

   if (buttonState == 0xF000)
       return TRUE;

   return FALSE;
}
```

Semaphore Task Synchronization in eCos (consumerTask)

```
void consumerTask(cyg_addrword_t data) {
  while (1) {
     /* Wait for the signal. */
     cyg_semaphore_wait(&semButton);

     diag_printf("Button SW0 was pressed.
                              \\n");

     ledToggle( );
  }
}
```

Semaphore Task Synchronization in Linux (main program)

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include "led.h"

sem_t semButton;

int main(void) {
    /* Configure the green LED control pin. */
    ledInit( );

    /* Create semaphore for this process only and with an initial value of 0. */
    sem_init(&semButton, 0, 0);

    /* Create the producer and consumer tasks */
    pthread_create(&producerTaskObj, NULL, (void *)producerTask, NULL);
    pthread_create(&consumerTaskObj, NULL, (void *)consumerTask, NULL);
    printf("Linux semaphore example - press button SW0.\\n");

    /* Allow the tasks to run. */
    pthread_join(producerTaskObj, NULL);
    pthread_join(consumerTaskObj, NULL);
    return 0;
}
```

Semaphore Task Synchronization in Linux (producerTask)

```
#include <unistd.h>
#include "button.h"

void producerTask(void *param) {
   int buttonOn;
   while (1) {
        /* Delay for 10 milliseconds. */
        usleep(10000);

        /* Check if the SW0 button has been pressed. */
        buttonOn = buttonDebounce( );

        /* If button SW0 was pressed, signal consumer */
        if (buttonOn) sem_post(&semButton);
   }
}
```

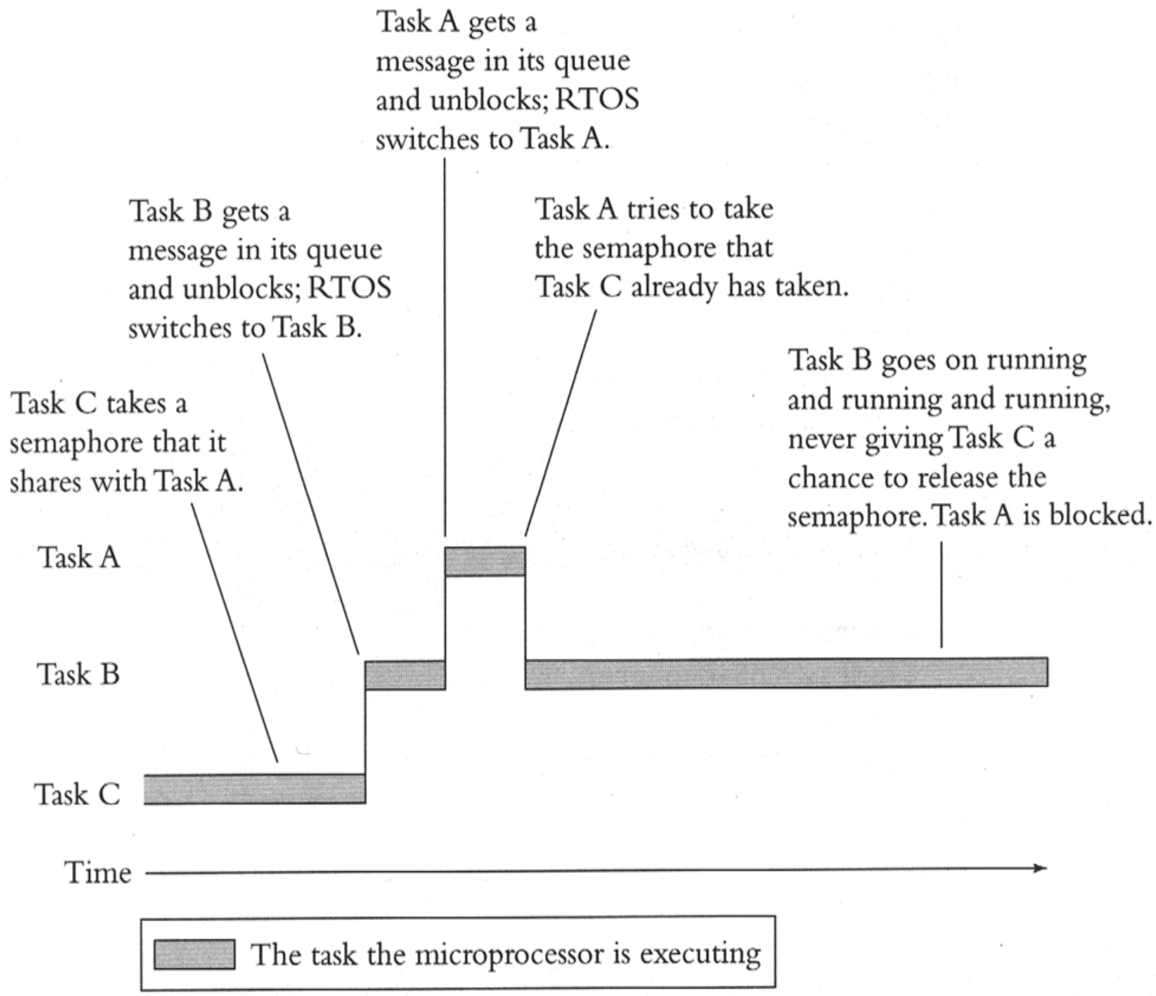
Semaphore Task Synchronization in Linux (consumerTask)

```
void consumerTask(void *param) {
  while (1) {
     /* Wait for the signal. */
     sem_wait(&semButton);

     printf("Button SW0 was pressed.\\n");

     ledToggle( );
  }
}
```

Semaphore Problems

- Forgetting to take the semaphore
- Forgetting to release the semaphore
- Taking the wrong semaphore
- Holding a semaphore for a long time
- Priority Inversion
- Causing a deadly embrace
- Use semaphores only when you have to!
- Avoid them when you can!

Priority Inversion



Deadly Embrace

- AMX RTOS code
- Both tasks may block

```
int a;
int b;
AMXID hSemaphoreA;
AMXID hSemaphoreB;
void vTask1 (void)
{
   ajsmrsv (hSemaphoreA, 0, 0);
   ajsmrsv (hSemaphoreB, 0, 0);
   a = b;
   ajsmrls (hSemaphoreB);
   ajsmrls (hSemaphoreA);
}

void vTask2 (void)
{
   ajsmrsv (hSemaphoreB, 0, 0);
   ajsmrsv (hSemaphoreA, 0, 0);
   b = a;
   ajsmrls (hSemaphoreA);
   ajsmrls (hSemaphoreB);
}
```

Semaphore Variants

- Counting semaphores
  - take = decrement integer
  - release = increment integer
  - block when integer = 0
- Resource semaphores
  - released only by task that took them
- Mutex semaphores
  - automatically handle priority inversion problem
  - (not all RTOS call such semaphores mutexes!)

Ways to Protect Shared Data

- **Disabling interrupts**
  - Most drastic, affects all other tasks
  - Only method if task & interrupts share data
  - Fast (single instruction)
- **Taking semaphores**
  - Most targeted
  - Response times of interrupts and non data-sharing tasks are unaffected
  - Not work for interrupts
- **Disabling task switches**
  - In-between the above two
  - No effect on interrupt routines
  - Affects all other tasks