CS466 Chapter 6

Introduction to Real-Time Operating Systems

Introduction to Real-Time Operating Systems

- RTOS is an acronym pronounced “are toss”
- Others use the term real-time kernel RTK.
- Usually the kernel means some subset of the most basic services offered by a larger RTOS.
- In this case, things like network support, debugging tools, memory management, etc are considered to be part of the RTOS but not part of the kernel.
- In this text, we ignore such distinctions and use the term RTOS indiscriminately.

Introduction to Real-Time Operating Systems

- Most real-time systems are very different from Windows or Unix.
- In a desktop system, the O/S takes control of the machine as soon as it is turned on and then it lets you start your applications. Your applications are compiled and linked separately from the operating system.
- In an embedded system, your application and the RTOS are linked together. At boot-up time, your application normally gets control first, and then it starts the RTOS.
- Many RTOS products do not protect themselves as carefully as do desktop operating systems.
- Most RTOS products save memory by including only those services that you need.
- One can write there own RTOS, but you also can and probably should buy one!

Tasks and Task States

- The basic building block of software written under an RTOS is the task
- Under most RTOS schemes, the task is simply a subroutine
- At some point in your program, you make one or more calls to a function in the RTOS that starts tasks, telling it what subroutine is the starting point for the task, the priority of the task, and some general parameters, where the stack for the task should be located, etc.

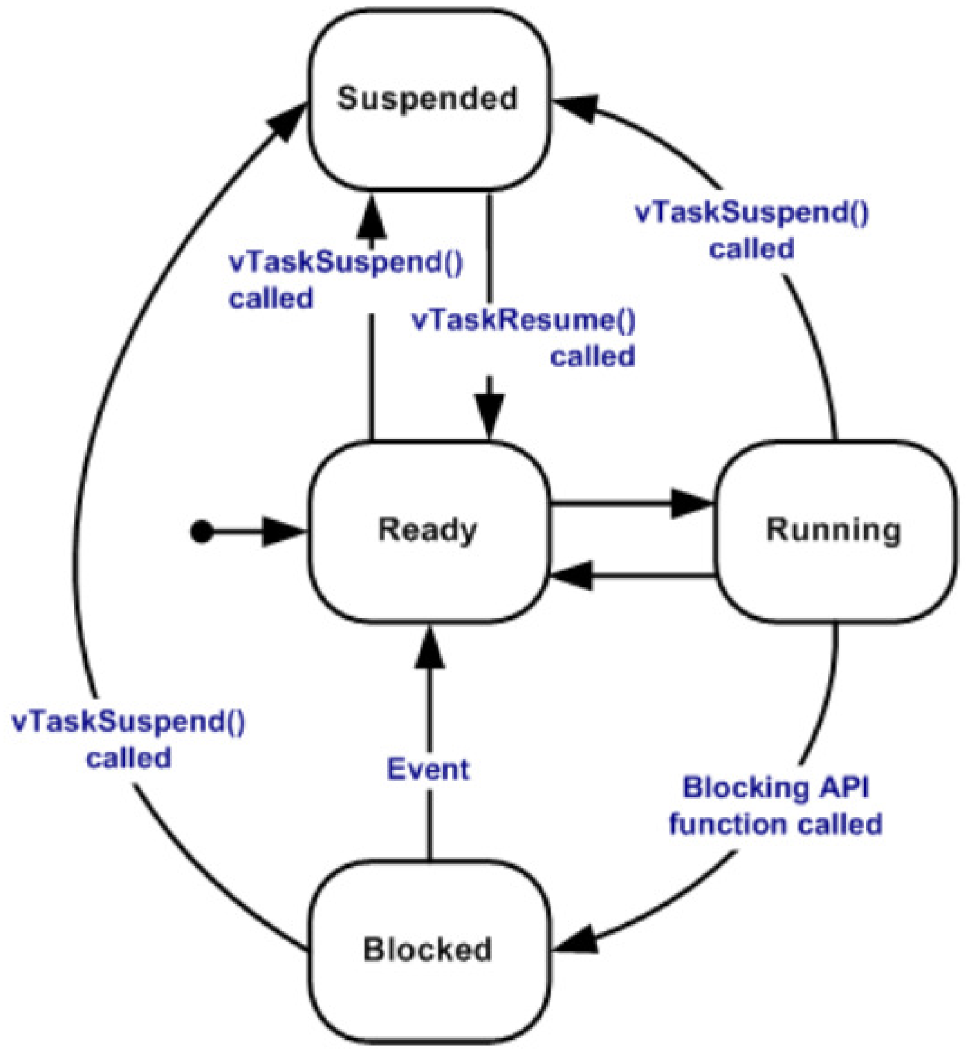
Tasks and Task States

- Running – the microprocessor is executing the instructions that make up this task. Only one task can be in this state.
- Ready – some other task is in the running state but this task has things to do if the microprocessor becomes available. Any number of tasks can be in this state.
- Blocked – this task hasn't anything to do right now, even if the microprocessor becomes available. Tasks get into this state because they are waiting for some external event. Any number of tasks can be in this state.

Tasks and Task States

- Most RTOS's seem to offer several other task states like suspended, pended, waiting, dormant, delayed, etc.
- These generally amount to fine distinctions among various categories of the blocked and ready states discussed earlier.
- For example, MicroC/OS-II has five states, namely DORMANT, READY, RUNNING, WAITING (for an event), or ISR (interrupted).

Tasks and Task States – FreeRTOS 8.1+



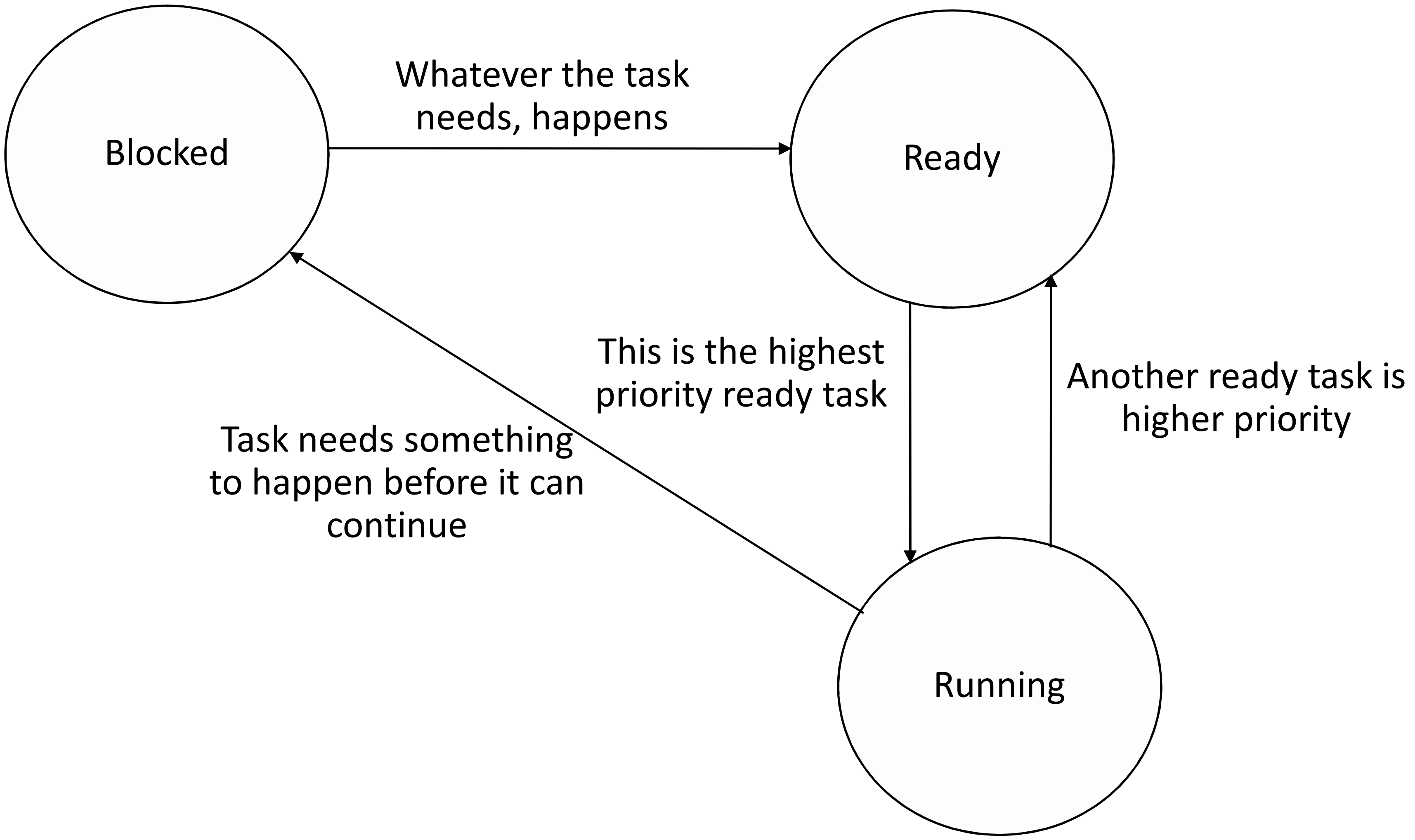
The Scheduler

- The scheduler is a part of the RTOS that keeps track of the state of each task and decides which one task should go into the running state. It is sometimes also known as the dispatcher.
- Unlike the schedulers in Unix or Windows, the scheduler is entirely simpleminded about which task gets the processor: it looks at priorities which you assign to the tasks and among tasks that are not in the blocked state, the one with the highest priority runs.
- If a higher priority task hogs the processor for a long time while lower-priority tasks are waiting in the ready state, that's just too bad!!!

The Scheduler (cont)

- A task will only block because it decides for itself that it has run out of things to do. Other tasks in the system cannot decide for a task that needs to wait for something. A consequence that a task must be running just before it is blocked so that it can execute the instructions that determine that it has nothing else to do.
- While a task is blocked, it never gets the microprocessor. Therefore, and interrupt routine or some *other* task in the system must be able to signal that whatever the task was waiting for has happened. Otherwise, the task will be blocked forever.
- The shuffling of tasks between the ready and the running states is entirely the work of the scheduler.

Task States – figure 6.1


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

Common Questions…

- How does the scheduler know when a task has become blocked or unblocked?
- What happens if all of the tasks are blocked?
- What if two tasks with the same priority are ready?

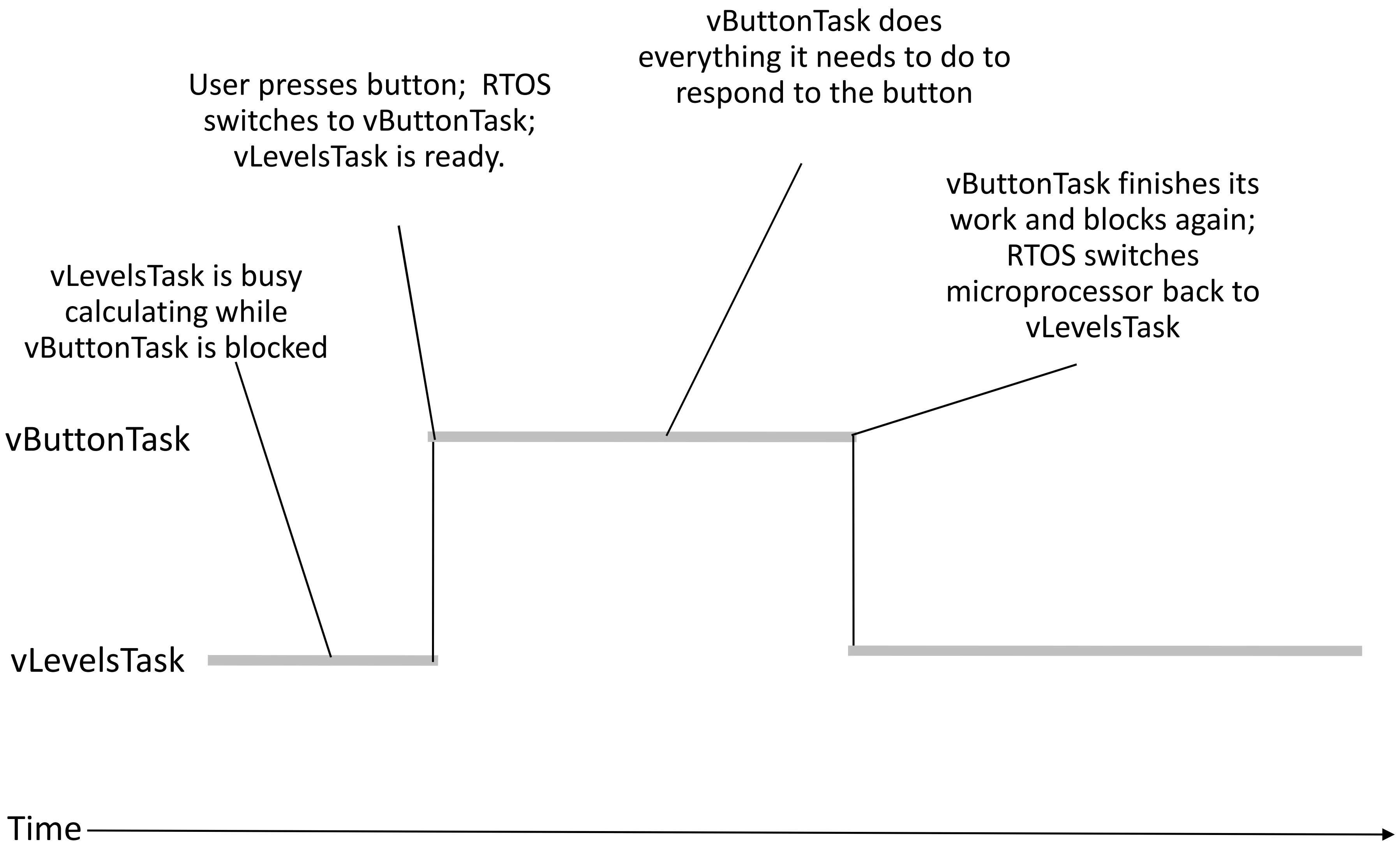
A simple example

- The next slide, figure 6.2 shows a classic example where an RTOS can make a difficult system easy to build
- In this case, the vLevelsTask uses up a lot of computer time figuring out how much gasoline is in each tank, and will use up as much computing time as it can get.
- However, when the user pushes a button, it is required that the system responds quickly. When the button is pressed, the vButtonTask unblocks.
- The RTOS will stop the low-priority vLevelsTask in its tracks, move it to the READY state, and run the high-priority vButtonTask to let it respond quickly to the user.

Figure 6.2 Uses for Tasks

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

/* “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
       !! Figure out which tank to do next
   }
}
```

Figure 6.3 Button Response


User presses button; RTOS switches to vButtonTask; vLevelsTask is ready.
vButtonTask does everything it needs to do to respond to the button
vButtonTask finishes its work and blocks again; RTOS switches microprocessor back to vLevelsTask
vLevelsTask is busy calculating while vButtonTask is blocked
vButtonTask
vLevelsTask
Time

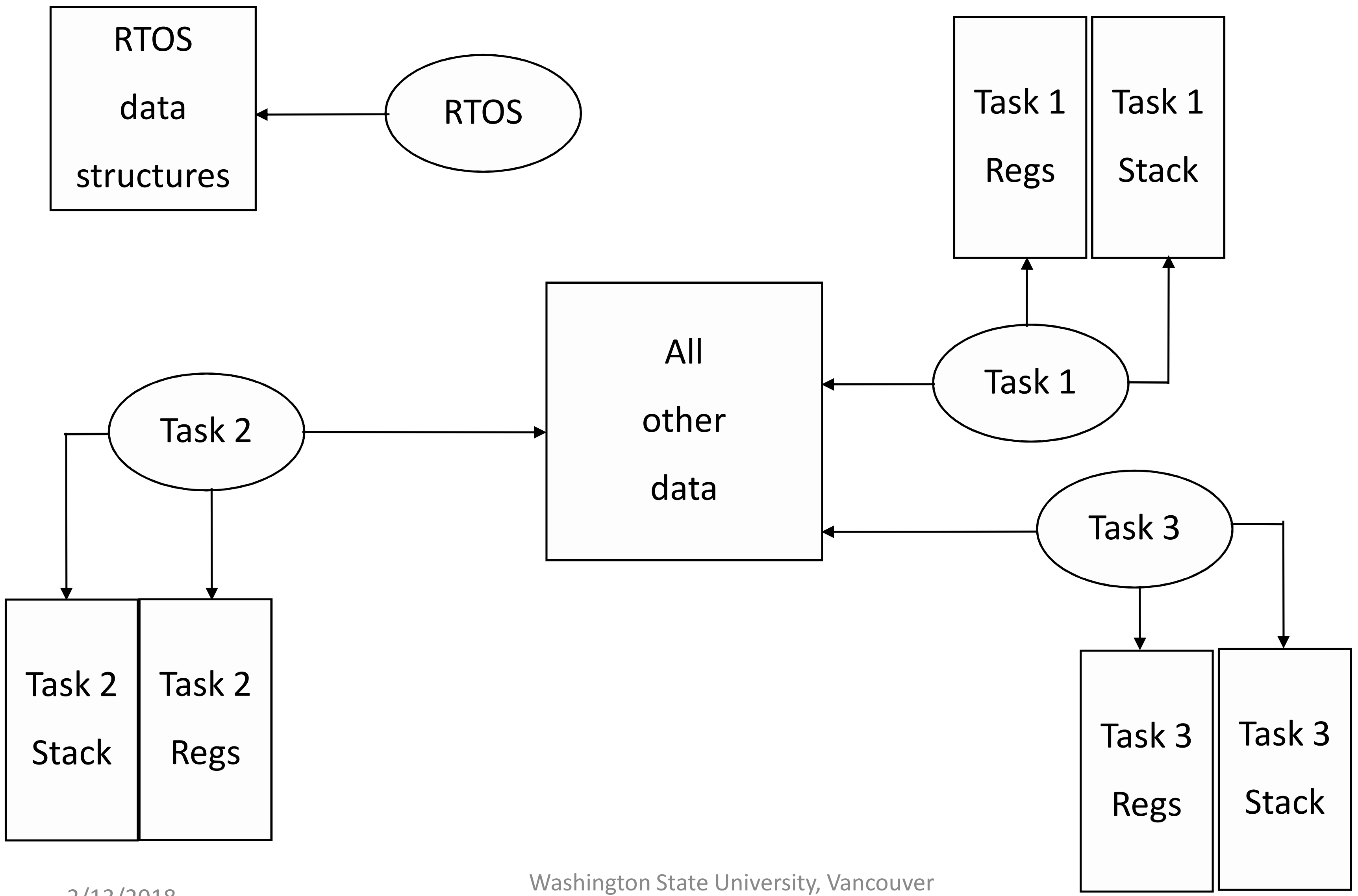
RTOS initialization

- It is common to have some code which runs first that tells the RTOS which of the functions are tasks and that (for example) the calculation task has a lower priority than the button task.
- A HUGE feature of using a RTOS is that one can write a task without being terribly concerned about someone else writing another task and generally the system will respond well.

Tasks and Data

- Each task has its own private context, which includes…
  - register values
  - program counter
  - stack
- However, all other data---global, static initialized, un-initialized, and everything else is shared. Thus, tasks in an RTOS are really more like threads than processes if you are familiar with Unix or Windows.

Figure 6.5 Task Data in an RTOS



Shared-Data Problems

- Sharing data between tasks has the same sort of sharing data problems as the examples in chapter 4 when bugs cropped up because of sharing data between task and isr code.
- The example in the text of figure 6.6 is an example…since setting vCalculateTankLevels might be setting in the array and it is not atomic, it could be preempted out.

Figure 6.7 Tasks sharing code

```
static int cErrors;

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

void Task1(void)
{
   …
   vCountErrors(9);
   …
}
void Task2(void)
{
   …
   vCountErrors(11);
   …
}
```

Reentrancy

- vCountErrors() is not reentrant
- Reentrant functions are functions that can be called by more than one task and will always work correctly.
- Also … reentrant functions will work correctly if called by an isr

Reentrancy rules!

- A reentrant function may *not* use variables in a non-atomic way unless they are stored on the stack of the task that called the function or are otherwise the private variables of that task.
- A reentrant function may *not* call any other functions that are not themselves reentrant.
- A reentrant function may *not* use the hardware in a non-atomic way.

C Variable Storage review

```
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;
  …
}
```

Question: Which of these variables are stored on the stack and which in a fixed location in memory?

C Variable Storage review - ans

```
static int static_int;       /* fixed location – shared by any task */
int public_int;          /* ditto except fcns in other C files can access */
int initialized = 4;         /* ditto – initial value makes no difference */
char *string = “Where does this string go?”;  /* ditto */
void *vPointer;          /* pointer itself in fixed location and shared */

void function(int parm, int *parm_ptr)    /* parm is on the stack */
{                    /* parm_ptr is also on the stack */
   static int static_local;      /* fixed location in memory smallest scope
   */
   int local;            /* is on the stack */
   …
}
```

Figure 6.10 – Reentrancy Ex.

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

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

QUESTION: IS this re-entrant?

Reentrancy cont

- The answer is NO for two reasons:
  - The boolean variable fError is in a fixed location in memory and is shared by any task that might call display(). The use of fError is not atomic, because the RTOS might switch between the time that is is tested and the time that it is set. (Rule 1 violation)
  - This function may violate Rule 2 as well…printf() must be reentrant…don’t count on printf() being reentrant unless so stated…

Gray Areas of reentrancy

```
static int cErrors;

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

The question is…is incrementing cErrors atomic(it is a non-stack variable). The answer is maybe or “it depends”. On an 8-bit micro-processor it will not be while for a 32-bit microprocessor it might be. Good advice is not depend on this and use another technique(semaphores for example).

6.3 Semaphores and shared data

- The RTOS offers a new service, called a semaphore to deal with the shared data problem.
- Duh…railroad barons discovered that it was a bad thing for trains to run into each other
- Discuss the semaphore train picture on page 154
- The word semaphore is one of the most slippery in the embedded systems world.
- Many RTOS's have multiple types of semaphores. These might be binary semaphores, counting semaphores, mutual exclusion semaphores, synchronization semaphores, etc…

Semaphores and shared data

- Also, RTOS vendors are not consistent about the terms. They may use raise and lower, get and give, take and release, pend and post, p and v, wait and signal, or any combination…
- The author uses take for lower and release for raise. The MicroC author uses pend and post. FreeRTOS uses Take and Give
- A typical RTOS semaphore works like this: tasks can call two RTOS functions say TakeSemaphore and GiveSemaphore. If one task has called TakeSemaphore and has not called GiveSemaphore any other task that calls TakeSemaphore will block until the first task calls GiveSemaphore.
- Only one task can have the semaphore at a time.

Semaphores Etymology

- The canonical names *V* and *P* come from the initials of Dutch words. *V* stands for *verhogen* ("increase"). Several explanations have been offered for *P*, including *proberen* for "to test" or "to try,"[3] *passeren* for "pass," and *pakken* for "grab." Dijkstra's earliest paper on the subject[1] gives "passeren" (pass) as the meaning, and mentions that the terminology is taken from that used in railroads. Dijkstra subsequently wrote that he intended *P* to stand for the portmanteau *prolaag*,[4] short for *probeer te verlagen*, literally "try to reduce," or to parallel the terms used in the other case, "try to decrease."[5][6][7] This confusion stems from the fact that the words for *increase* and *decrease* both begin with the letter *V* in Dutch, and the words spelled out in full would be impossibly confusing for those not familiar with the Dutch language.
- In ALGOL 68, the Linux kernel,[8] and in some English textbooks, the *V* and *P* operations are called, respectively, *up* and *down*. In software engineering practice, they are often called *signal* and *wait*, *release* and *acquire* (which the standard Java library[9] uses), or *post* and *pend*. Some texts call them *vacate* and *procure* to match the original Dutch initials.

Semaphores

- Semaphores can be used to protect data by guarding access to the data. This effectively protects the shared data problem in which a task can have half-modified data which of course is corrupted data.

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

void vRespondToButton(void)    /* high priority  Button Task */
{
    int i;
    while(TRUE)
    {
        !! Block until user pushes a button
        i = !! Get ID of button pressed
        TakeSemaphore();
        printf("\nTIME:  %08li      LEVEL:  %08li", tankdata[i].lTimeUpdated, tankdata[i].lTankLevel);
        ReleaseSemaphore();
    }
}
void vCalculateTankLevels(void)    /* low priority Levels Task */
{
    int i = 0;
    while(TRUE)
    {
        …
        TakeSemaphore();
        !! Set tankdata[i].lTimeUpdated
        !! Set tankdata[i].lTankLevel
        ReleaseSemaphore();
    }
}
```

Semaphore example 6.12

Before the levels task (vCalculateTankLevels) updates the data in the structure, it calls TakeSemaphore() to take (lower) the semaphore. If the user presses a button while the levels task is still modifying the data and still has the semaphore, the following sequences occurs:

1. The RTOS will switch to the “button task” just as before, moving the levels task to the ready state.
2. When the button task tries to get the semaphore by calling Take(), it will block because the levels task already has the semaphore.
3. The RTOS will then look around for another task to run and will notice that the levels task is still ready. With the button task blocked, the levels task will get to run until it releases the semaphore, even though it is a lower priority than the button task.
4. When the levels task releases the semaphore by calling Give(), the button task will no longer be blocked, and the RTOS will switch back to it.

Semaphores

- Review the execution flow on page 157 to make sure you understand the previous 4 steps…
- Note that the FreeRTOS calls to take and release a semaphore are called xSemaphoreTake () and xSemaphoreGive(), respectively

Nuclear reactor example

- The next several slides are equivalent to Figure 6.14 on page 158. The call to xSemaphoreCreate() MUST!! be made before vReadTemperatureTask() calls to use the semaphore.
- The problem is that there is no guarantee that this happens…
- Do not fool around…put the calls to the semaphore initialization in some start-up code that is guaranteed to run first…
  - Initialize
  - Start
  - Activate

Figure 6.14

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

static int iTemperatures[2];
SemaphoreHandle_t semTemp;
```

Figure 6.14

```
void main(void)
{
    /* tell the RTOS about our tasks */
    xTaskCreate(vReadTemperatureTask,
                TASK_PRIORITY_READ);
    xTaskCreate(vControlTask,
                TASK_PRIORITY_CONTROL);


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

Figure 6.14

```
void vReadTemperatureTask(void)
{
    while(TRUE)
    {
        vTaskDelay(100); /* delay about 1/10 second */
        // this is a futile attempt to wait for
        // sem to be created (Dangerous and Poor Form)

        xSemaphoreTake( semTemp, portMAX_DELAY);

        !!  Read in iTemperatures[0];
        !!  Read in iTemperatures[1];

        xSemaphoreGive( semTemp, NULL);
    }
}
```

Figure 6.14

```
void vControlTask(void)
{
    /* create the semaphore */
    p_semTemp = xSemaphoreCreateBinary();

    while(TRUE)
    {
        xSemaphoreTake( semTemp, portMAX_DELAY);

        if(iTemperatures[0] != iTemperatures[1])
            !!  Set off the howling alarm;

        xSemaphoreGive( semTemp, NULL);

        !! Do any other useful work
    }
}
```

- Semaphores can make a function reentrant.
- However, put the calls to take and release the semaphore INSIDE the function so that a user cannot forget!
- See figure 6.15
- Consider using multiple semaphores to protect different critical data.
- How does the RTOS know which semaphore protects which data???
- It doesn’t, that is the engineer’s job to be consistent in the definition and use of a semaphore

Figure 6.15 make a function reentrant

```
static int cErrors;
// assume semaphore is created during setup
static SemaphoreHandle_t _semErrors;

void Task1(void)
{
    vCountErrors(9);
}
void Task2(void)
{
    vCountErrors(11);
}

void vCountErrors(int cNewErrors)
{
    xSemaphoreTake(_semErrors, portMAX_DELAY);
    cErrors += cNewErrors;
    xSemaphoreGive(_semErrors);
}
```

Semaphores as signaling devices

- A typical use of a semaphore is to use it as a signaling device from one task to another or from an ISR to a task. (Remember, ISR's can take a semaphore but can *not* block if the semaphore is not available).
- Typically for this case, some initialization code will create the semaphore as already taken. The ISR will give the semaphore, allowing the blocked task to perform its work synchronized with the ISR .

Semaphores as resource protection

- Another use of a semaphore is to block concurrent task access to a critical piece of code.
- While semaphores will work, We typically refer to them as a 'Mutex' if guaranteeing 'Mutual Exclusion' around a non-reentrant critical exception.

Figure 6.15b Semaphore Types

```
static int cErrors;
// All FreeRTOS semaphores and Mutex's are managed with a
// handle of a coinsistant type.
static SemaphoreHandle_t _semExample[4];

void resourceInit(void){
   // Normal binary semaphore created empty
   _semHandle[0] = xSemaphoreCreateBinary();

   // Counting semaphore, max and initial counts provided at creation
   _semHandle[1] = xSemaphoreCreateCounting(max, initial);

   // Creates a mutex semaphore, I believe created full (available)
   _semHandle[2] = xSemaphoreCreateMutex();

   // A recursive mutex may be taken multiple times by a one thread.
   // uses special take and give API's
   _semHandle[3] = xSemaphoreCreateRecursiveMutex();
}
```

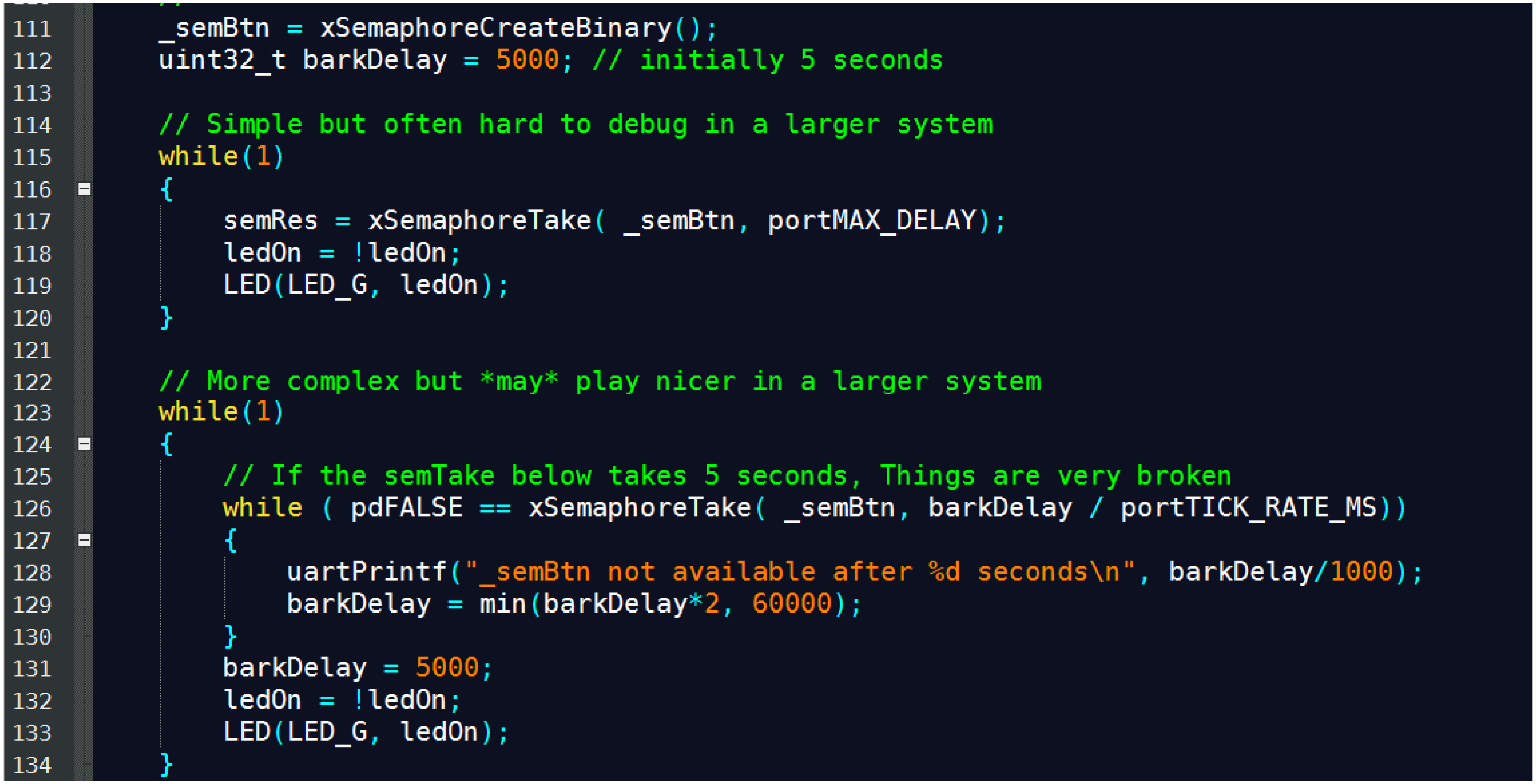
Semaphore problems…

- Tried and True ways to mess up with semaphores…
  - Forgetting to take the semaphore
  - Forgetting to release the semaphore
  - Taking the wrong semaphore
  - Holding a semaphore for too long
  - Perverse Priority Inversion shown on next slide…

Holding a semaphore too long..

- In larger complex systems there are lots of tasks and a hung task can be very difficult to detect.
- A printer called Condor at HP had over 120 concurrent tasks to manage data and a mechanism that could keep 17 pages in flight at any time.
- We learned to never block forever, If it was possible to set error timeouts you could produce reasonable logging to help engineers triage the issue but keep your task up and running.
- The following code shows a simple forever-block or a more complex barking algorithym that produces logging, Could be a LED, GPIO whatever.

Holding a semaphore too long..



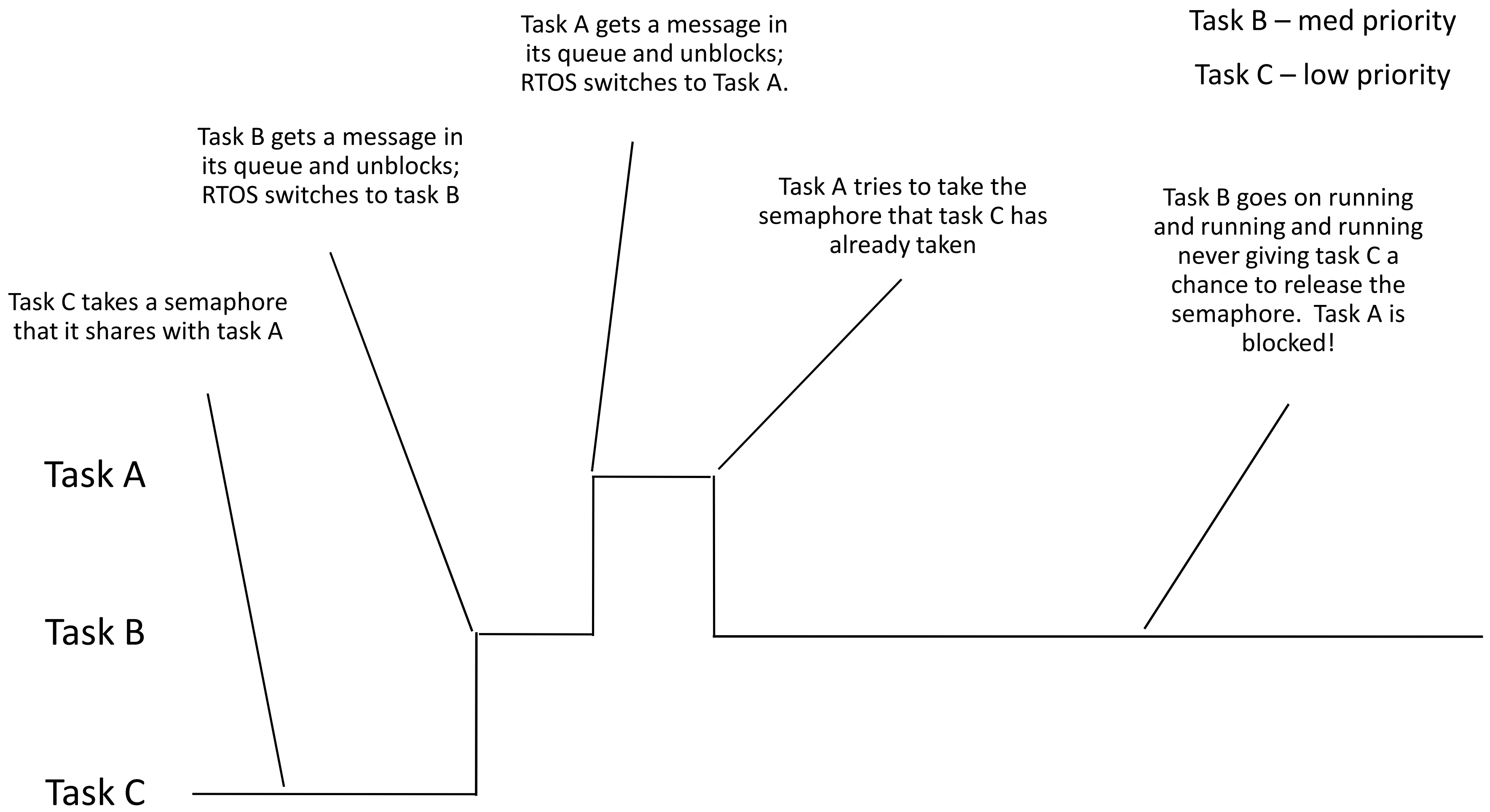
```
    _semBtn = xSemaphoreCreateBinary();
    uint32_t barkDelay = 5000; // initially 5 seconds

    // Simple but often hard to debug in a larger system
    while(1)
    {
        semRes = xSemaphoreTake( _semBtn, portMAX_DELAY);
        ledOn = !ledOn;
        LED(LED_G, ledOn);
    }

    // More complex but *may* play nicer in a larger system
    while(1)
    {
        // If the semTake below takes 5 seconds, Things are very broken
        while ( pdFALSE == xSemaphoreTake( _semBtn, barkDelay / portTICK_RATE_MS))
        {
            uartPrintf("_semBtn not available after %d seconds\n", barkDelay/1000);
            barkDelay = min(barkDelay*2, 60000);
        }
        barkDelay = 5000;
        ledOn = !ledOn;
        LED(LED_G, ledOn);
    }
```

Priority Inversion – Figure 6.17

Task A – high priority

Task B – med priority

Task C – low priority

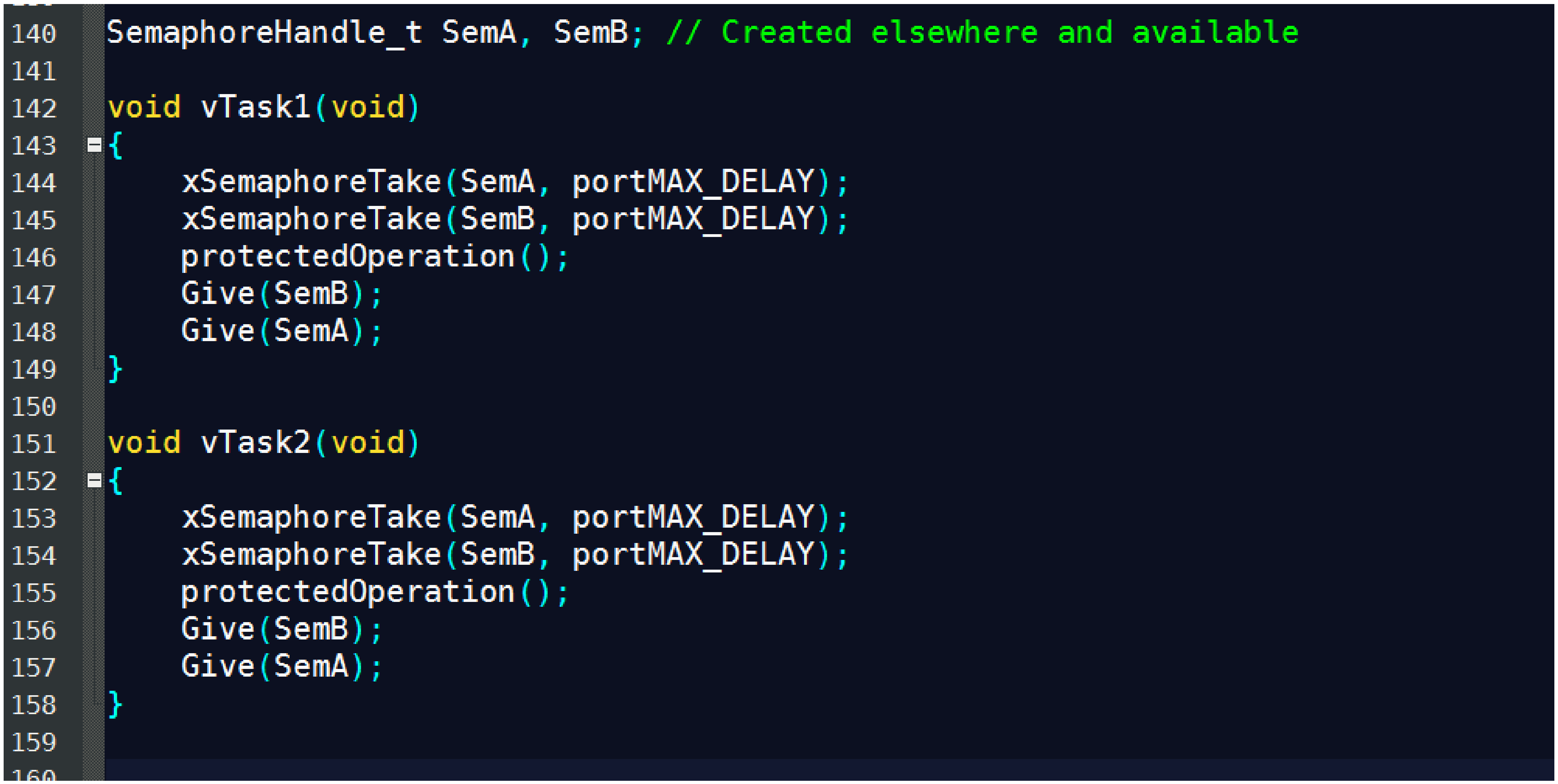

Task A gets a message in its queue and unblocks; RTOS switches to Task A.
Task B gets a message in its queue and unblocks; RTOS switches to task B
Task A tries to take the semaphore that task C has already taken
Task B goes on running and running and running never giving task C a chance to release the semaphore. Task A is blocked!
Task C takes a semaphore that it shares with task A
Task A
Task B
Task C

Priority Inversion

- No matter how carefully you code Task C, Task B can prevent Task C from releasing the semaphore and thereby can hold up Task A indefinitely.
- Some RTOS's solve this problem with priority inheritance…they temporarily boost the priority of the priority of Task C to that of Task A whenever Task C holds the semaphore that Task A is waiting for.
- FreeRTOS does provide priority inheritance but only for mutexes.
- There is no strict rule and each OS needs to be checked for features like this

Deadly Embrace – Figure 6.18

(Sounds overly dramatic, I prefer the term deadlock..)

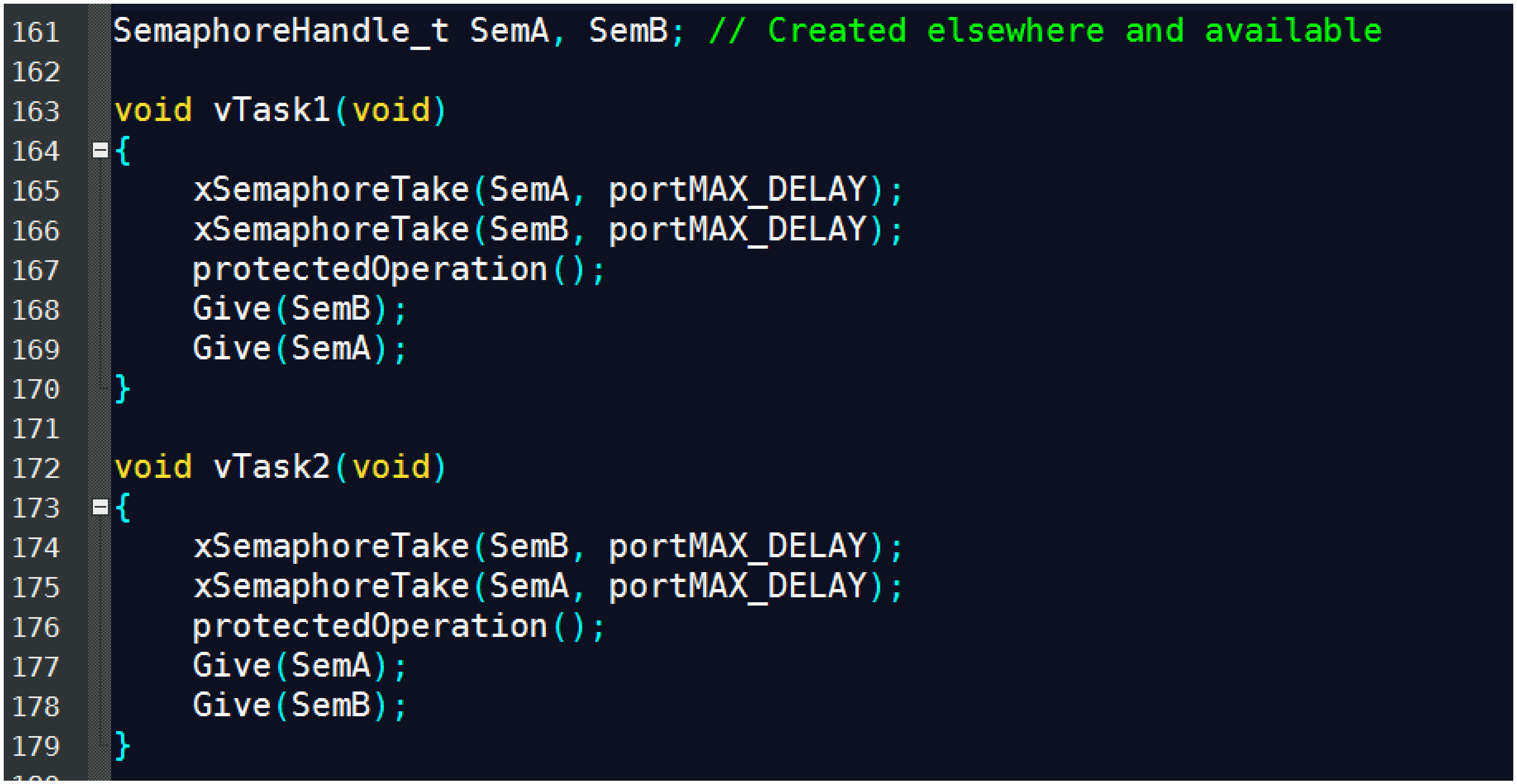


```c
SemaphoreHandle_t SemA, SemB; // Created elsewhere and available

void vTask1(void)
{
    xSemaphoreTake(SemA, portMAX_DELAY);
    xSemaphoreTake(SemB, portMAX_DELAY);
    protectedOperation();
    Give(SemB);
    Give(SemA);
}

void vTask2(void)
{
    xSemaphoreTake(SemA, portMAX_DELAY);
    xSemaphoreTake(SemB, portMAX_DELAY);
    protectedOperation();
    Give(SemB);
    Give(SemA);
}

```

Deadly Embrace (cont)

- Of course, these problems would be easy if they were as illustrated in this example
- However, they virtually never appear on one page of code and are typically distributed over many functions/files/pages/etc.
- Is there a generic way to manage the issue?

Deadly Embrace – Figure 6.18b



```
SemaphoreHandle_t SemA, SemB; // Created elsewhere and available

void vTask1(void)
{
    xSemaphoreTake(SemA, portMAX_DELAY);
    xSemaphoreTake(SemB, portMAX_DELAY);
    protectedOperation();
    Give(SemB);
    Give(SemA);
}

void vTask2(void)
{
    xSemaphoreTake(SemB, portMAX_DELAY);
    xSemaphoreTake(SemA, portMAX_DELAY);
    protectedOperation();
    Give(SemA);
    Give(SemB);
}
```

Deadly Embrace (cont.)

- In general, If you acquire binary locks in any order, make sure that you release them in reverse order. p(a), p(b), v(b), v(a)
- If all tasks agree on the order the problem is easy..

- What if I have 10 developers authoring 20 tasks, with 15 semaphores protecting 15 resources (100, 200, 150, 150)….
- Is there still a generic way to manage the issue?

- The Linux kernel has a workable solution

Deadly Embrace (cont.)

- We manage semaphores/mutexes with what's known as 'Handles' or 'Opaque Pointers'
- Each opaque pointer actually points to a region in memory called a C structure.
- No two unique handles can occupy the same memory.
- As a result we can sort any arbitrary list of semaphores by their handle values.
- Then if *all* the threads sort the handles the same way and always take/give in ascending handle value, we are clear.

Ways to protect shared data

- Disable interrupts
  - Drastic – affects the response time of all ISRs
  - Also disables task switches
  - It is the only method that works if you have shared data between tasks and ISRs… (excepting a queue with head and tail done correctly)
  - It is fast
- Taking a mutex or semaphore
  - The most targeted way to protect data
  - Response times of users are unchanged
  - Generally doesn’t work for ISR routines
- Disabling task switches
  - Somewhat in between the other two
  - No effect on interrupt routines
  - Stops response for all other tasks cold

Chapter 6 summary

- A typical **RTOS** is smaller and offers fewer services than a standard operating system and is more closely linked to the application.
- **RTOS**'s are widely available for sale, and it generally makes sense to buy one rather than write another one for yourself.
- The **task** is the main building block for software written in an RTOS environment.
- Each task is always in one of three general states: **running**, **ready**, **blocked**. The scheduler runs the highest priority ready task.
- Each task has its own stack; however, other data in the system is shared by all tasks. Therefore, shared data problems can occur.

Chapter 6 Summary(cont)

- A function that works properly even if it called by more than one task is called a **reentrant** function
- **Semaphores** can solve the shared-data problem. Since only one task can take a semaphore at a time, semaphores can prevent shared data from causing bugs. Semaphores have two associated operations – **take** and **give** (p,v) (acquire, release), (down, up).. The name depends on the OS
- Your tasks can use semaphores to signal one another
- Your tasks can use mutexes to create more friendly critical sections
- You can introduce any number of ornery bugs with semaphores. **Priority inversion** and **deadly embrace** are two of the more obscure. Forgetting to take or release or using the wrong ones are common ways to cause problems.
- What can happen if a function returns in several places?

Chapter 6 Summary(cont)

- The **mutex**, the **binary semaphore**, and the **counting semaphores** are all common.
- Three methods to protect data…disabling interrupts, taking semaphores, and disabling task switches.
- Semaphore/Mutex management is largely convention, Except for simple take/give checking I have not seen a good automated tool that can locate errors.