Semaphores and shared data

Reference: Simon Chapter 6

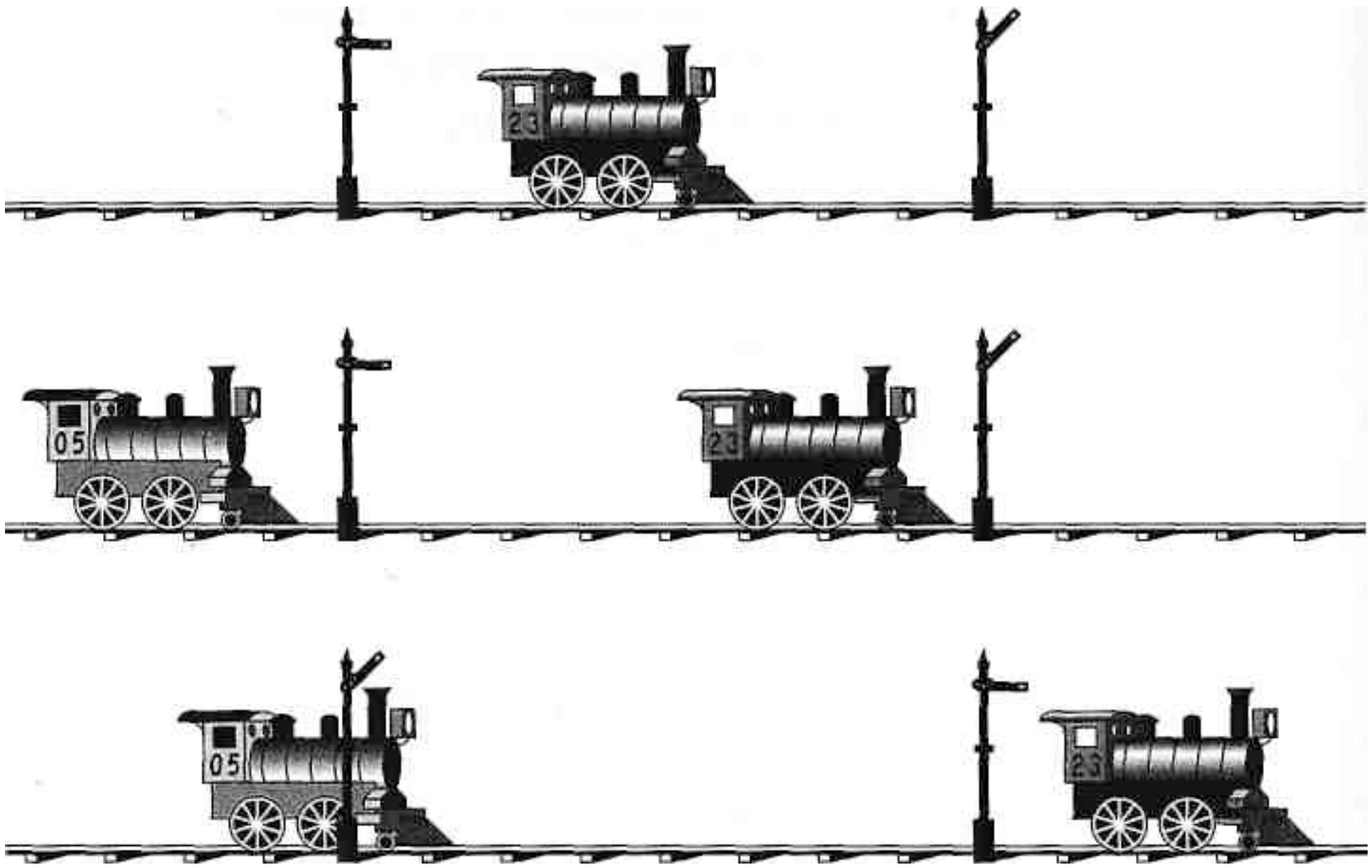
Nuno Alves (nalves@wne.edu), College of Engineering

Shared data problems

- RTOS can cause a new class of shared-data problems by:
  - Switching the microprocessor from task to task
  - Changing the flow of execution (similar to interrupts)

- Something called **semaphores** helps preventing shared data problems.

Trains do two things with semaphores

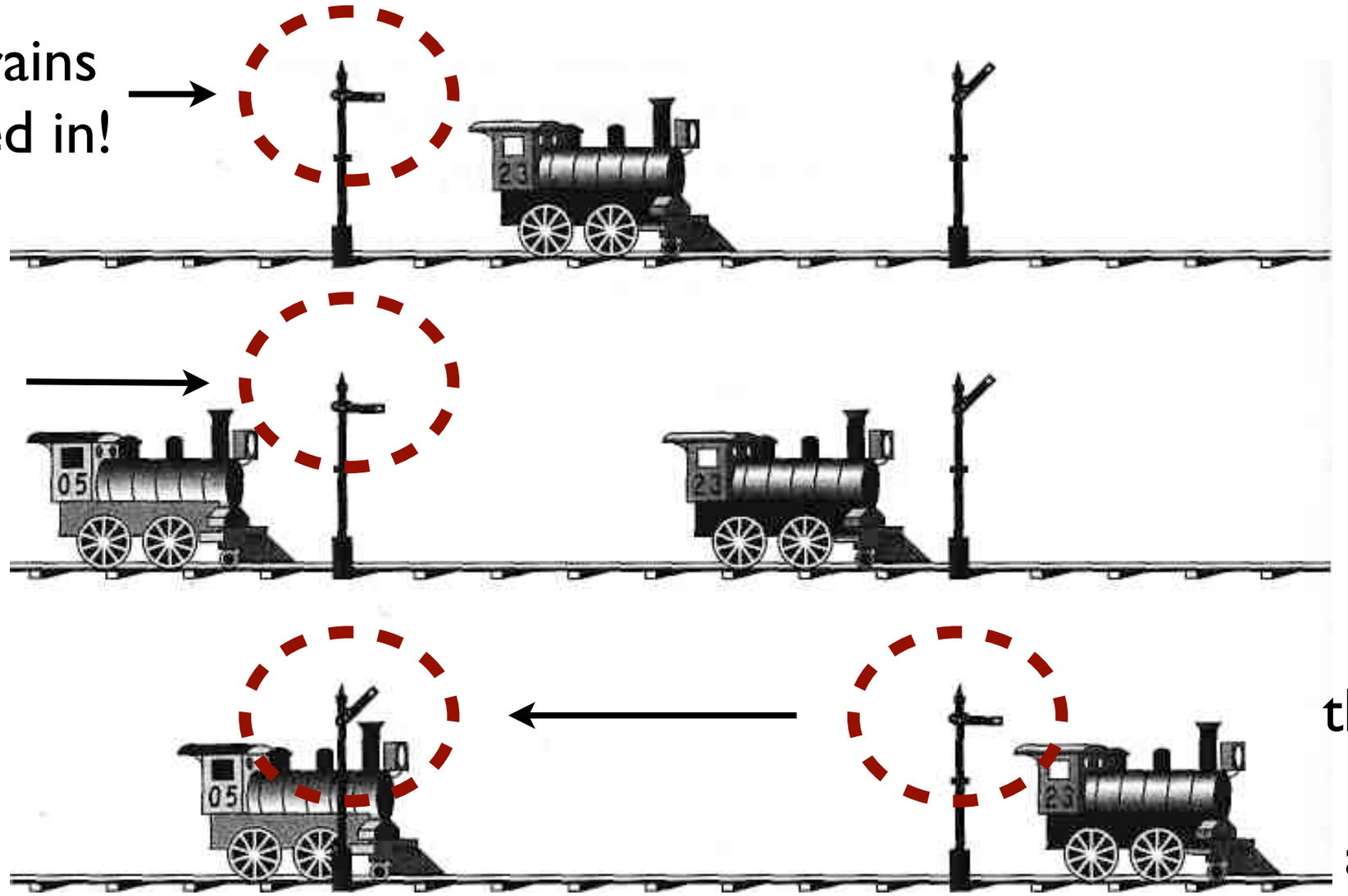


1. When a train leaves the protected section of track, it raises the semaphore.

2. When a train comes to a semaphore, it waits for the semaphore to rise, if necessary, passes through the (now raised) semaphore, and lowers the semaphore.

The typical semaphore in an RTOS works much the same way

No trains allowed in!



Train waits until this flag is raised...

Train leaves the protected section, allowing the other train in...

RTOS semaphores

- Most RTOS normally use the paired terms **take** and **release**. Other RTOS paired terms may be get-give, pend-post, wait-signal.
- Tasks can call two RTOS functions, **TakeSemaphore** and **ReleaseSemaphore**.
- If one task has called Take-Semaphore to take the semaphore and has not called ReleaseSemaphore to release it, then any other task that calls TakeSemaphore will be blocked until the first task calls ReleaseSemaphore.
- Only **one** task can have the semaphore at a time.

Tank monitoring system

- Imagine I have 300 watering tanks in my greenhouse

- I have one microprocessor that constantly monitors the water level at every single tank.

- It takes a while to determine the water level for each tank.

- Whenever I press a button I want to immediately know the water level for a particular tank.

Tank monitoring system implemented with semaphores

```
struct
{
  long ITankLevel;
  long ITimeUpdated;
} tankdata[MAX_TANKS];

/* "Button Task" - High priority */
void vRespondToButton (void)
{
  int i;

  while (TRUE)
  {
    !! Block until  user pushes a button
    i = !! Get ID of button pressed

    TakeSemaphore ();
    printf ("\nTIME: %081d  LEVEL: %081d",
          tankdata[i].ITimeUpdated,
          tankdata[i].ITankLevel);
     ReleaseSemaphore ();

  }
}
```

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

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

 StartTask  (vRespondToButton,  HIGH_ PRIOTY);
 StartTask  (vCalculateTankLevels, L0W_PRIOTY);

 //Start the RTOS.
 StartRTOS  ();
}
```

Sequence of events for the “tank monitoring system” (part 1/2)

If the user presses a button while the levels task is still modifying the data and still has the semaphore, then the following sequence of events occurs:

1. The RTOS will switch to the "button task," just as before, moving the levels task to the ready state.

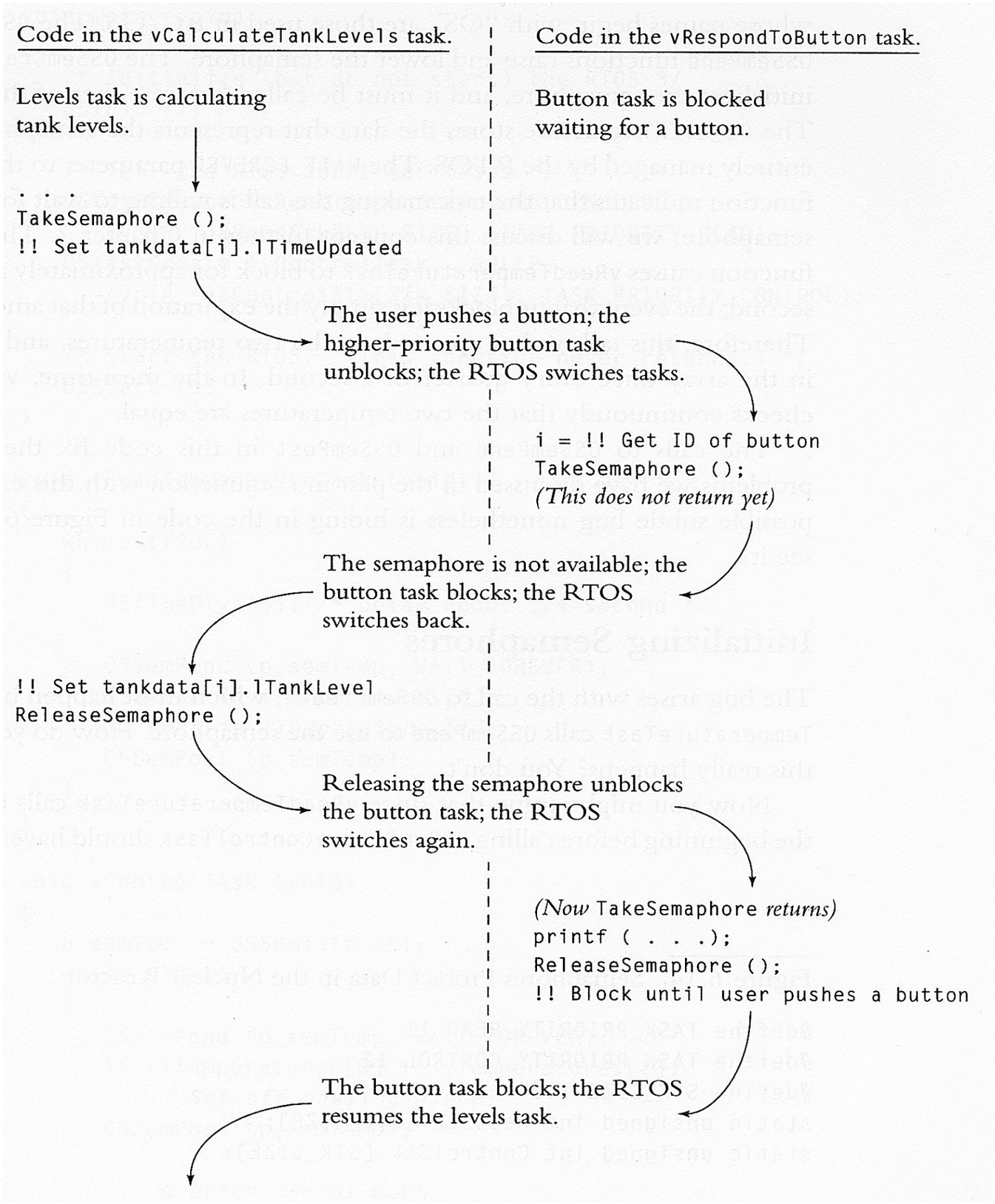
2. When the button task tries to get the semaphore by calling TakeSemaphore, it will block because the levels task already has the semaphore.

Sequence of events for the "tank monitoring system" (part 2/2)

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.

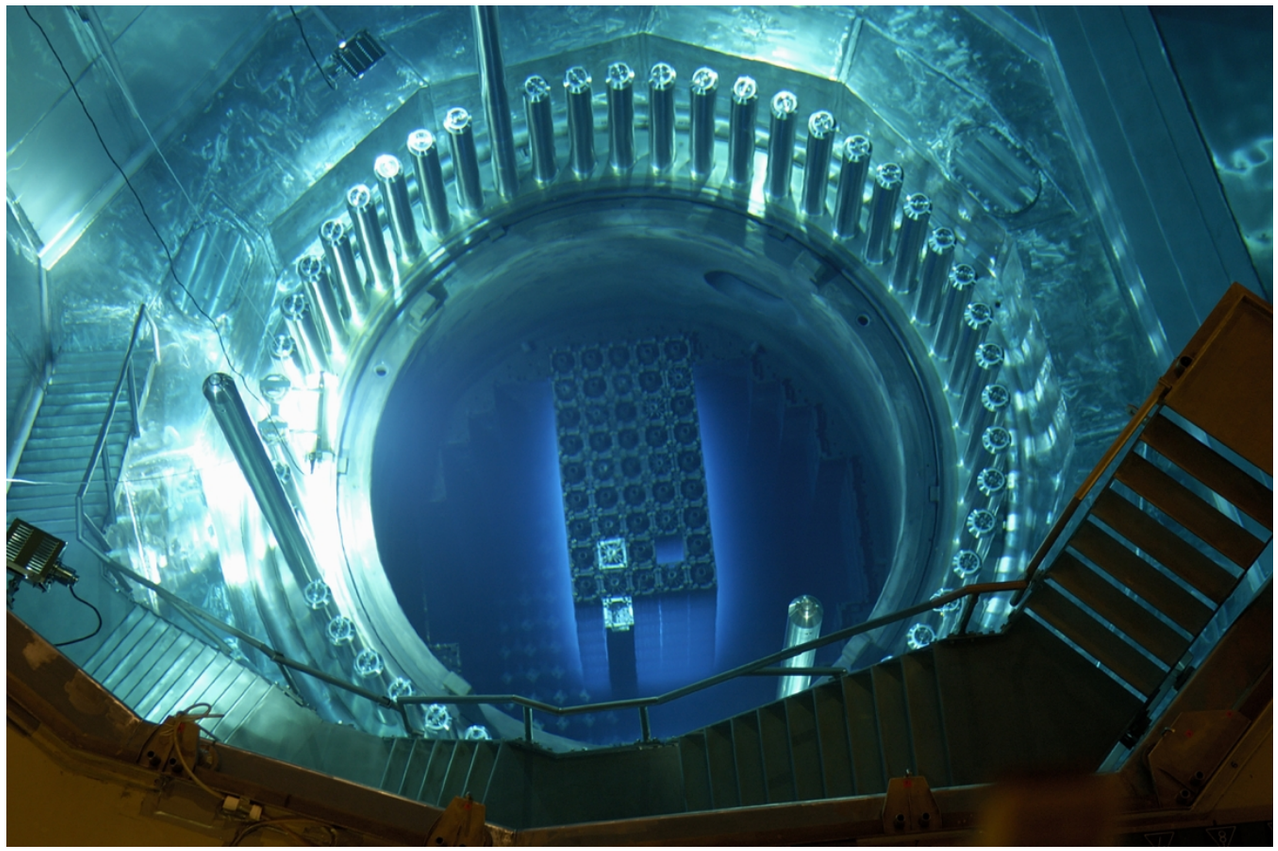
4. When the levels task releases the semaphore by calling ReleaseSemaphore, the button task will no longer be blocked, and the RTOS will switch back to it.

Entire sequence of events for the "tank monitoring system"



Another example: nuclear reactor temperature checking

Premise for the shared-data problem



- This is a nuclear reactor.
- Periodic temperature measurements occur at two separate locations.
- If the temperature at these two locations are different then sound the alarm!
- Different temperatures means we may have a nuclear fallout!


MASS
EFFECT
3

CPE 355 - Real Time Embedded Kernels - Spring '12

Nuno Alves (nalves@wne.edu), College of Engineering

WESTERN NEW ENGLAND UNIVERSITY | WNE

```
#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;

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 ();
}
```

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

    OSSemPost (p_semTemp);
    !! Do other useful work
  }
}


void vReadTemperatureTask (void)
{
  while (TRUE)
  {
    //Delay of 0.25 seconds
    OSTimeDly (5);
    OSSemPend (p_semTemp, WAIT_FOREVER);
    !! read in iTemperatures[0];
    !! read in iTemperatures[1];
    OSSemPost (p_semTemp);
  }
}
```

```
#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;

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 ();
}
```

OS_EVENT structure (defined in the RTOS) stores the data that represents the semaphore.

```
void vControlTask (void)
{
                                                  REVER);
                                                  tures[1])
    OSSemPost (p_semTemp);
    !! Do other useful work
  }
}


void vReadTemperatureTask (void)
{
  while (TRUE)
  {
    //Delay of 0.25 seconds
    OSTimeDly (5);
    OSSemPend (p_semTemp, WAIT_FOREVER);
    !! read in iTemperatures[0];
    !! read in iTemperatures[1];
    OSSemPost (p_semTemp);
  }
}
```

A task rather than an interrupt routine reading the temperatures.

OSSemPost and OSSemPend functions raise and lower the semaphore.

WAIT_FOREVER parameter to the OSSemPend function indicates that the task making the call is willing to wait forever for the semaphore.

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

    OSSemPost (p_semTemp);
    !! Do other useful work
  }
}
```

```
void vReadTemperatureTask (void)
{
  while (TRUE)
  {
    //Delay of 0.25 seconds
    OSTimeDly (5);
    OSSemPend (p_semTemp, WAIT_FOREVER);
    !! read in iTemperatures[0];
    !! read in iTemperatures[1];
    OSSemPost (p_semTemp);
  }
}
```

vControlTask checks continuously that the two temperatures are equal.

The calls to OSSemPend and OSSemPost in this code fix the shared-data problems.

OSTimeDly function causes current task to block for a certain time; the event that unblocks it is simply the expiration of that amount of time.

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

    OSSemPost (p_semTemp);
    !! Do other useful work
  }
}
```

```
void vReadTemperatureTask (void)
{
  while (TRUE)
  {
    //Delay of 0.25 seconds
    OSTimeDly (5);
    OSSemPend (p_semTemp, WAIT_FOREVER);
    !! read in iTemperatures[0];
    !! read in iTemperatures[1];
    OSSemPost (p_semTemp);
  }
}
```

The function OSSemInit initializes a semaphore.

```
#define TASK_PRIORITY_READ  11
#define TASK_PRIORITY_CONTROL  12
#de
sta                                    ZE];
sta                                    _SIZE];
static int iTemperatures[2];
OS_EVENT *p_semTemp;

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 (
```

...What is the problem with this code???

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

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

    OSSemPost (p_semTemp);
    !! Do other useful work
  }
}
```

```
void vReadTemperatureTask (void)
{
  while (TRUE)
  {
    //Delay of 0.25 seconds
    OSTimeDly (5);
    OSSemPend (p_semTemp, WAIT_FOREVER);
    !! read in iTemperatures[0];
    !! read in iTemperatures[1];
    OSSemPost (p_semTemp);
  }
}
```

OSSemInit must happen before vRead-TemperatureTask calls OSSemPend to use the semaphore.

How do you know that this really happens? You don't.

... vReadTemperatureTask calls OSTimeDly at the beginning before calling OSSemPend, vControlTask **should** (but not necessarily) have enough time to call OSSemInit.

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

    OSSemPost (p_semTemp);
    !! Do other useful work
  }
}
```

```
void vReadTemperatureTask (void)
{
  while (TRUE)
  {
    //Delay of 0.25 seconds
    OSTimeDly (5);
    OSSemPend (p_semTemp, WAIT_FOREVER);
    !! read in iTemperatures[0];
    !! read in iTemperatures[1];
    OSSemPost (p_semTemp);
  }
}
```

How do you know that there isn't some higher-priority task that takes up all of the delay time in vReadTemperatureTask?

OSTimeDly is an attempt at ensuring the system will work as desired.

Best solution is to put OSSemInit in some start-up code that's guaranteed to run first... such as the main function.

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

    OSSemPost (p_semTemp);
    !! Do other useful work
  }
}
```

```
void vReadTemperatureTask (void)
{
  while (TRUE)
  {
    //Delay of 0.25 seconds
    OSTimeDly (5);
    OSSemPend (p_semTemp, WAIT_FOREVER);
    !! read in iTemperatures[0];
    !! read in iTemperatures[1];
    OSSemPost (p_semTemp);
  }
}
```

Reentrancy and semaphores

Semaphores make a function reentrant

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

static int cErrors;
static SEMAPHORE semErrors;

void vCountErrors (int cNewErrors)
{
  OSSemPend (&semErrors,SUSPEND);
  cErrors += cNewErrors;
  OSSemPost (&semErrors);
}
```

- The code that modifies **cErrors** is surrounded by calls to semaphore routines.
- In the language of data sharing, we have protected **cErrors** with a semaphore.
- Whichever task calls vCountErrors second will be blocked when it tries to take the semaphore.
- We have made the use of **cErrors atomic** and therefore have made the function vCountErrors reentrant.

Reentrancy and semaphores

```
void Task1  (void)
{
  (...)
  OSSemPend (&semErrors,SUSPEND);
  vCountErrors (9);
  OSSemPost (&semErrors);
  (...)
}
void Task2 (void)
{
 (...)
  OSSemPend (&semErrors,SUSPEND);
  vCountErrors (11);
  OSSemPost (&semErrors);
  (...)
}

static int cErrors;
static SEMAPHORE semErrors;

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

- Does this code still work if the calls to OSSemPend and OSSemPost are now around the calls to **vCountErrors** instead of being within the function itself?

Reentrancy and semaphores

```
void Task1  (void)
{
  (...)
  OSSemPend (&semErrors,SUSPEND);
  vCountErrors (9);
  OSSemPost (&semErrors);
  (...)
}
void Task2 (void)
{
 (...)
  OSSemPend (&semErrors,SUSPEND);
  vCountErrors (11);
  OSSemPost (&semErrors);
  (...)
}

static int cErrors;
static SEMAPHORE semErrors;

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

- Does this code still work if the calls to OSSemPend and OSSemPost are now around the calls to **vCountErrors** instead of being within the function itself?

- Yes!

Multiple semaphores

Multiple semaphores

- The semaphore functions all take a parameter that identifies the semaphore that is being initialized, lowered, or raised.

- The semaphores are all independent of one another: 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.

What's the advantage of having multiple semaphores?

- Whenever a task takes a semaphore, it is potentially slowing the response of any other task that needs the same semaphore.

- In a system with only one semaphore, if the lowest-priority task takes the semaphore to change data in a shared array of temperatures, the highest-priority task might block waiting for that semaphore.

- By having one semaphore protect the temperatures and a different semaphore protect the error count, you can build your system so the highest-priority task can modify the error count even if the lowest-priority task has taken the semaphore protecting the temperatures. Different semaphores can correspond to different shared resources.

How does the RTOS know which semaphore protects which data?

- **It doesn't.** If you are using multiple semaphores, it is up to you to remember which semaphore corresponds to which data.
- A task that is modifying the error count must take the corresponding semaphore.
- 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).

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

void taskA(void)
{
   (...)
   //Initialize the semaphore
   semPrinter = OSSemInit(0);
   //take the semaphore
   OSSemPend (&semaphoreA,SUSPEND);
   (...)
}

void interruptA (void)
{
    //Release the semaphore.
    OSSemPost (&semaphoreA);
}
```

The semaphoreA will only be released after the interruptA occurs.

Semaphore problems

Semaphore problem cause #1: Forgetting to take the semaphore

- Semaphores do NOT solve all our shared-data problems.
- In fact, your systems will probably work better, the fewer times you have to use semaphores.
- Semaphores only work if you use them perfectly.
- Forgetting to take the semaphore: Semaphores only work if every task that accesses the shared data, for read or for write, uses the semaphore.

Semaphore problem cause #2: Forgetting to release the semaphore

- Forgetting to release the semaphore: If any task fails to release the semaphore, then every other task that ever uses the semaphore will sooner or later be blocked as they wait to take that semaphore.

Semaphore problem cause #3: Taking the wrong semaphore

- Taking the wrong semaphore: If you are using multiple semaphores, then taking the wrong one is as bad as forgetting to take one.

Semaphore problem cause #4: Holding a semaphore for too long

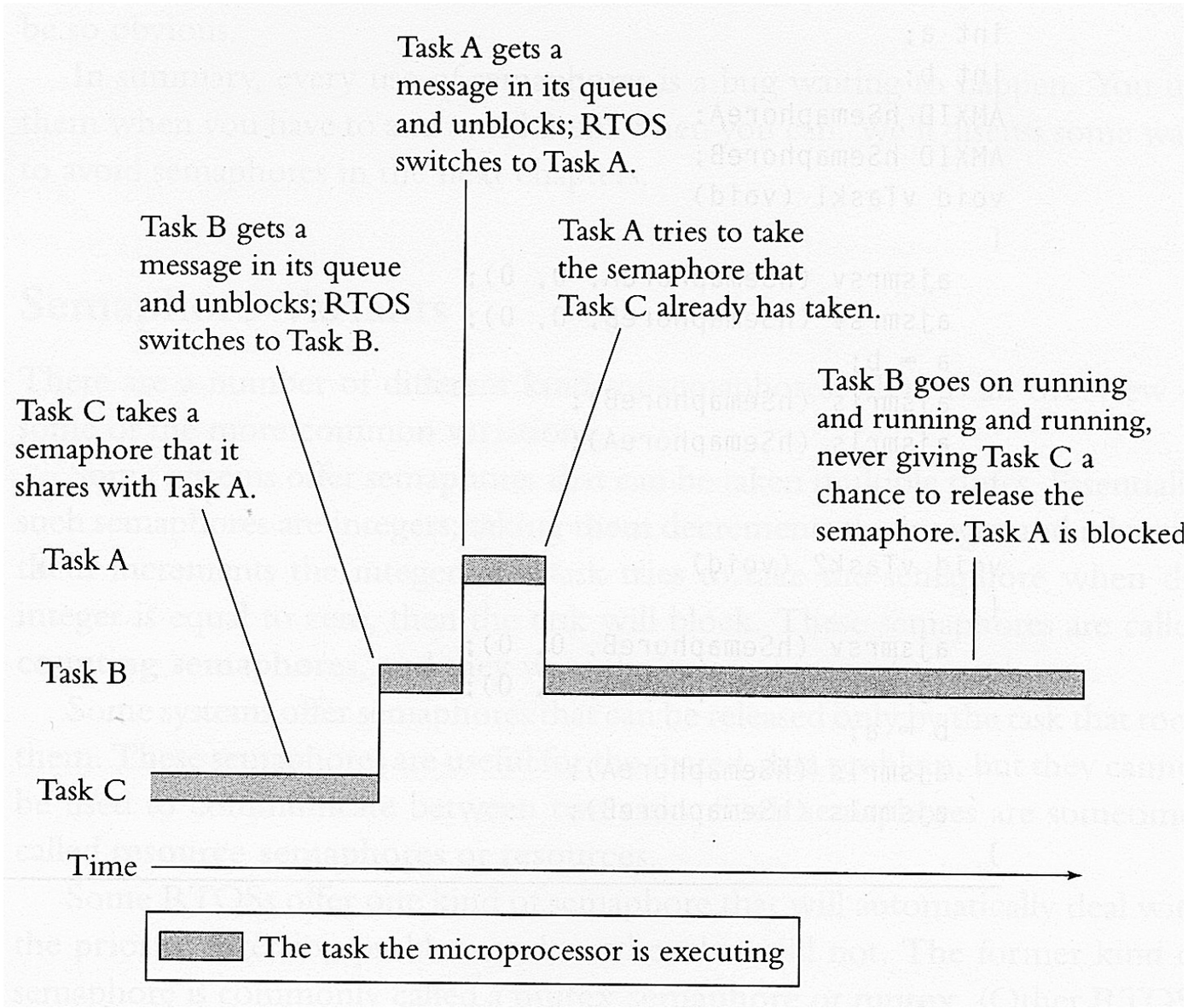
- Holding a semaphore for too long: Whenever one task takes a semaphore, every other task that subsequently wants that semaphore has to wait until the semaphore is released.

Consequence #1: Priority inversion

Lets say consider a system with 3 different tasks:

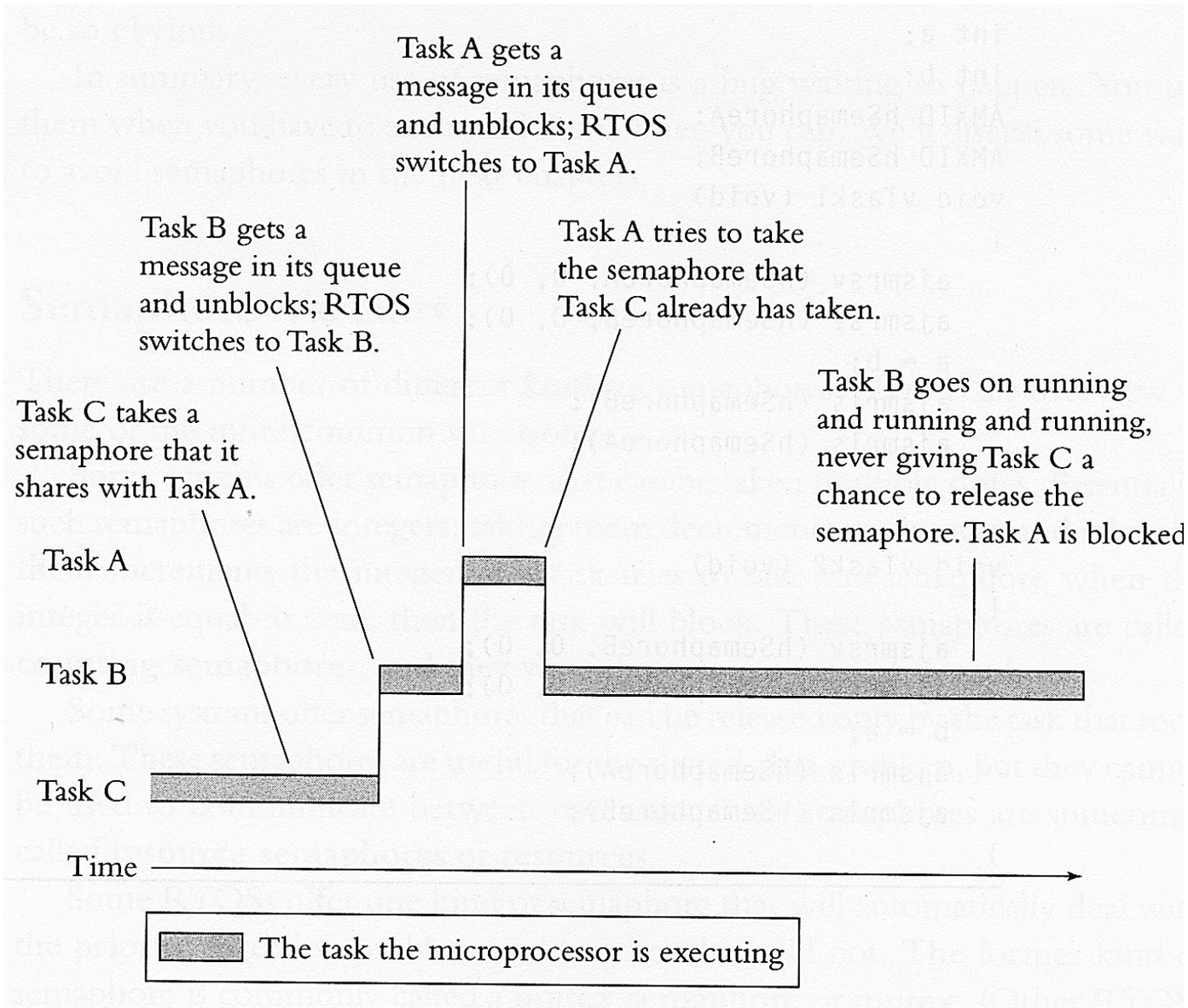
- TaskC (low priority)
- TaskB (medium priority)
- TaskA (high priority)

According to our RTOS paradigm, when the system demands a high priority task (TaskA) to be executes, it must be done immediately!

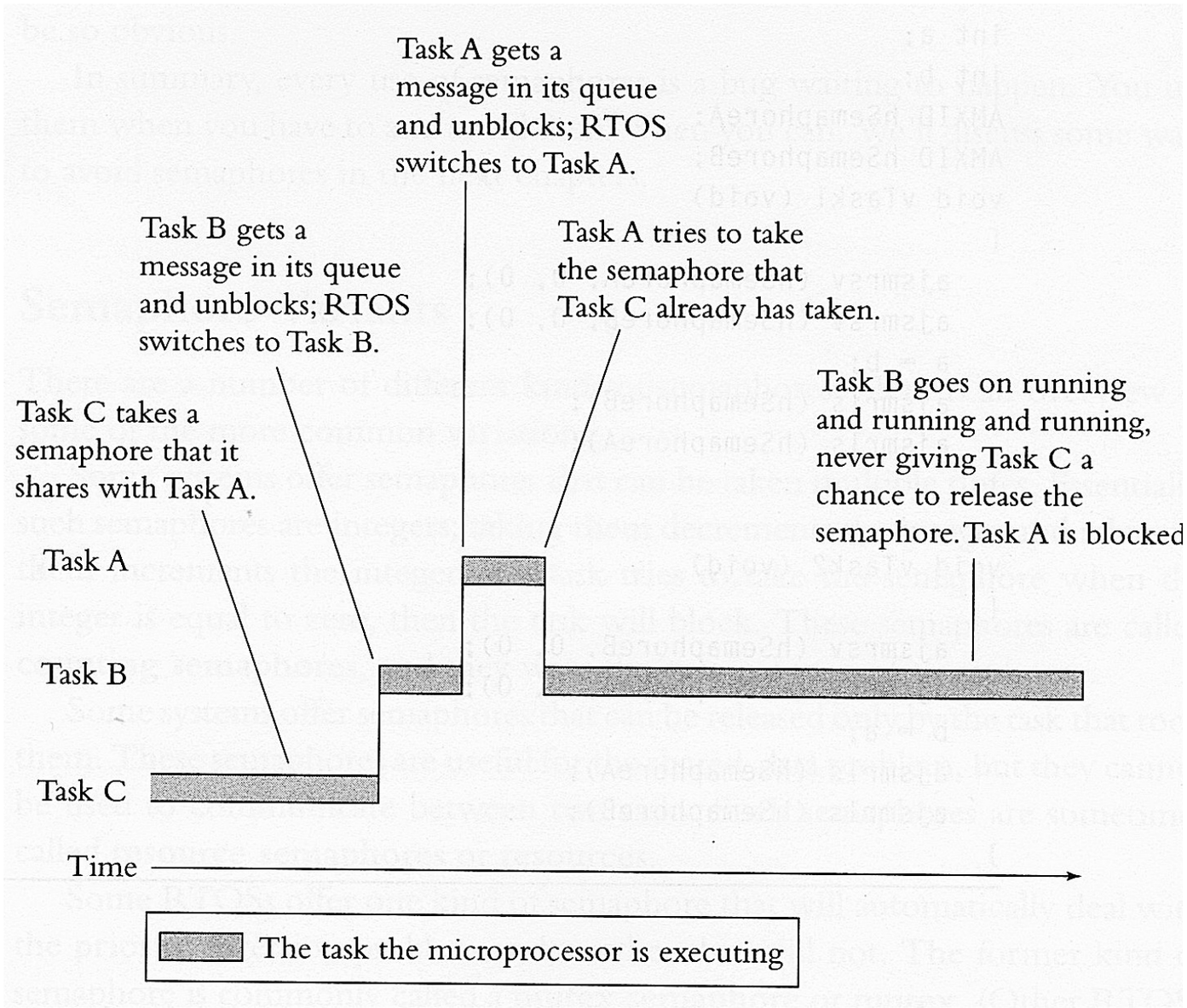


Consequence #1: Priority inversion

A nasty semaphore problem can arise if the RTOS switches from a low-priority task (Task C) to a medium-priority task (Task B) after Task C has taken a semaphore.



Consequence #1: Priority inversion

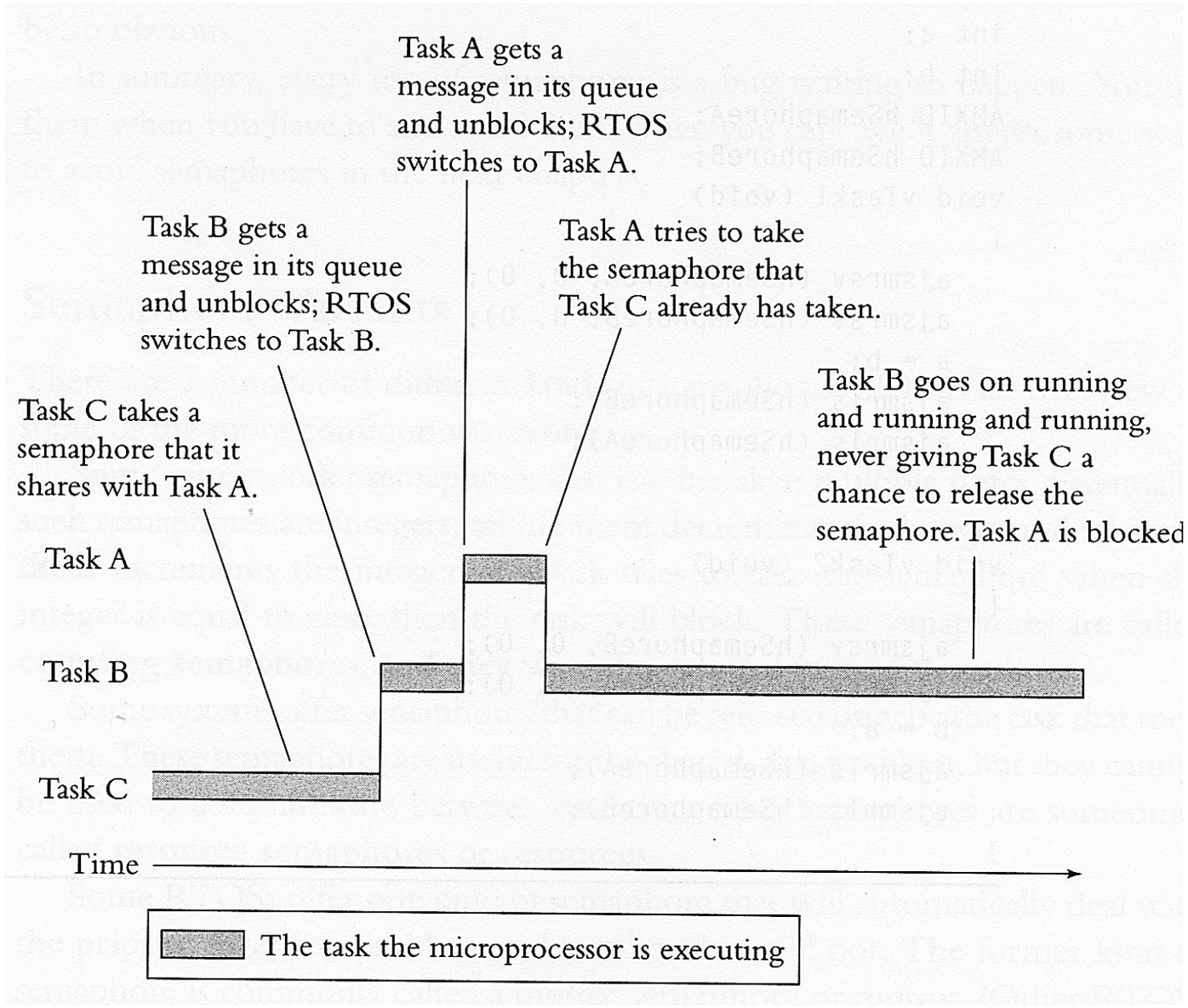


A high-priority task (Task A) that wants the semaphore then has to wait until Task B gives up the microprocessor: Task C can't release the semaphore until it gets the microprocessor back.

Consequence #1: Priority inversion

No matter how carefully you code Task C, Task B can prevent Task C from releasing the semaphore and can thereby hold up Task A indefinitely.

This is problematic! TaskA, which is a high priority task should have been executed immediately!



Consequence #2: Deadly embrace

```
int a;
int b;
static OS_EVENT *semaphoreA;
static OS_EVENT *semaphoreB;

void vTaskl (void)
{
  OSSemPend (&semaphoreA,SUSPEND);
  OSSemPend (&semaphoreB,SUSPEND);
  a = b;
  OSSemPost (&semaphoreB);
  OSSemPost (&semaphoreA);
}

void vTask2  (void)
{
  OSSemPend (&semaphoreB,SUSPEND);
  OSSemPend (&semaphoreA,SUSPEND);
  b = a;
  OSSemPost (&semaphoreA);
  OSSemPost (&semaphoreB);
}
```

- In this code Taskl and Task2 operate on variables a and b after getting permission to use them by getting semaphores **SemaphoreA** and **SemaphoreB**.

- Do you see the problem?

Consequence #2: Deadly embrace

```
int a;
int b;
static OS_EVENT *semaphoreA;
static OS_EVENT *semaphoreB;

void vTaskl (void)
{
  OSSemPend (&semaphoreA,SUSPEND);
  OSSemPend (&semaphoreB,SUSPEND);
  a = b;
  OSSemPost (&semaphoreB);
  OSSemPost (&semaphoreA);
}

void vTask2  (void)
{
  OSSemPend (&semaphoreB,SUSPEND);
  OSSemPend (&semaphoreA,SUSPEND);
  b = a;
  OSSemPost (&semaphoreA);
  OSSemPost (&semaphoreB);
}
```

- If vTaskl calls gets SemaphoreA, but before it can call OSSemPend to get SemaphoreB, the RTOS stops it and runs vTask2.

- The task vTask2 now calls OSSemPend and gets SemaphoreB.

- When vTask2 then calls OSSemPend to get SemaphoreA, it blocks, because another task (vTaskl) already has that semaphore.

Consequence #2: Deadly embrace

```
int a;
int b;
static OS_EVENT *semaphoreA;
static OS_EVENT *semaphoreB;

void vTaskl (void)
{
  OSSemPend (&semaphoreA,SUSPEND);
  OSSemPend (&semaphoreB,SUSPEND);
  a = b;
  OSSemPost (&semaphoreB);
  OSSemPost (&semaphoreA);
}

void vTask2  (void)
{
  OSSemPend (&semaphoreB,SUSPEND);
  OSSemPend (&semaphoreA,SUSPEND);
  b = a;
  OSSemPost (&semaphoreA);
  OSSemPost (&semaphoreB);
}
```

- The RTOS will now switch back to vTaskl, which now calls OSSemPend to get SemaphoreB.

- Since vTask2 has SemaphoreB, however, vTaskl now also blocks.

Consequence #2: Deadly embrace

```
int a;
int b;
static OS_EVENT *semaphoreA;
static OS_EVENT *semaphoreB;

void vTaskl (void)
{
  OSSemPend (&semaphoreA,SUSPEND);
  OSSemPend (&semaphoreB,SUSPEND);
  a = b;
  OSSemPost (&semaphoreB);
  OSSemPost (&semaphoreA);
}

void vTask2  (void)
{
  OSSemPend (&semaphoreB,SUSPEND);
  OSSemPend (&semaphoreA,SUSPEND);
  b = a;
  OSSemPost (&semaphoreA);
  OSSemPost (&semaphoreB);
}
```

- Deadly-embrace problems would be easy to find and fix if they always looked as "clean" as this code.
- However, deadly embrace is just as deadly if vTaskl takes the first semaphore and then calls a subroutine that later takes a second one while vTask2 takes the second semaphore and then calls a subroutine that takes the first.
- In this case the problem will not be so obvious.