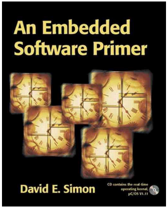
RTOS services —Part I

Kizito NKURIKIYEYEZU, Ph.D.

Readings

- Read Chap 6 of Simon, D. E. (1999). An Embedded Software Primer
- Topics
  - RTOS fundamentals
  - Tasks
  - Semaphores
  - Priority inversion



[1] Readings are based on Simon, D. E. (1999). An Embedded Software Primer.

Tasks and Task States

- Task—a subroutine in RTOS
- Embedded software application makes calls to the RTOS functions to start tasks, passing to the OS, start address, stack pointers, etc. of the tasks
- Task states
  1. Running—A task is running when it is actively being executed by a processor, and hence, makes progress. The number of tasks in the running state cannot exceed the total number of processors available in the system.
  2. Ready—A task is in the ready state when it is eligible for execution but no processors are currently available to execute it, because all of them are busy with other activities. A task does not make any progress when it is ready
  3. 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. Blocked

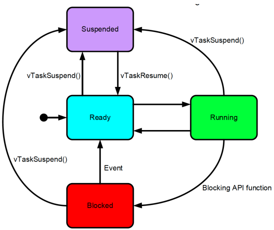


FIG 1. Task states transition in FreeRTOS[1]

[1] https://www.freertos.org/RTOS-task-states.html

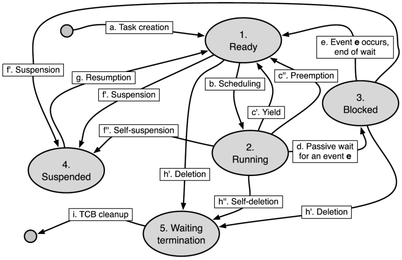


FIG 2. Task state diagram in the FreeRTOS operating system.

Task-based scheduling

- The scheduler keeps track of the states of each task
- It also decides which task should run
- Based on priorities
  - priorities set by user
  - non-blocked task with highest priority runs
- How does a scheduler know when a task has become blocked or unblocked?—The RTOS provides API for events to wait for or signal events that occurred
- What happen if all tasks are blocked —the scheduler will wait for something to happen. If nothing happen, it usually the programmer's fault (or the software is supposed to wait that long?!)
- What if two tasks of the same priories are ready?—depends on the RTOS and how it implements this behavior
- FreeRTOS store a full copy of the processor state in a data structure, known as Task Control Block also known as the

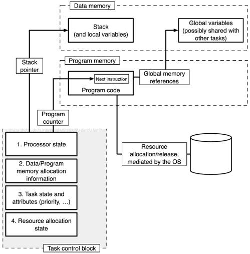


FIG 3. Main task control block components in FreeRTOS


```c
typedef struct tskTaskControlBlock{
  /*The location of the last item placed on the
     stack */
  volatile StackType_t * pxTopOfStack;
  /*The list that the state (Ready, Blocked,
     Suspended ). */
  ListItem_t xStateListItem;
  /*Used to reference a task from an event list. */
  ListItem_t xEventListItem;
  /*The priority of the task.  0 is the lowest
     priority. */
  UBaseType_t uxPriority;
  /*Points to the start of the stack. */
  StackType_t * pxStack;
  /*Descriptive name given to the task to
     facilitates debugging*/
  char pcTaskName[ configMAX_TASK_NAME_LEN ];
} tskTCB;
```

Task-based scheduling

- The TCB contains a full copy of the processor state[2] to allow the OS to switch from one task to another
  - Context switch—the OS saves the processor state of the previous task into its TCB and then restoring the processor state of the next task
  - Program counter —points to the next instruction that the processor will execute, within the task's program code
  - Stack pointer—locates the boundary between full and empty elements in the task stack
- The data and program memory allocation information keep a record of the memory areas currently assigned to the task.
- The task state and attributes are used by the operating system to schedule tasks in an orderly way and support inter-task synchronization and communication.
- Resource allocation state hold which resources (e.g., hardware devices connected to the system) that may need to

Task-based scheduling

- Deleted tasks —immediately ceases execution but its TCB is not immediately removed from the system. Instead, the task goes into the waiting termination state until the OS completes the cleanup operation[3]
- Task scheduling —the OS decides which task to move in the running state whenever a processor is available for use.
- The transition from the running to the blocked state is always under the control of the affected task and when specific event eventually occurs, the waiting task is returned to the ready state
- When a task is resumed, it unconditionally goes from the suspended state into the ready state. This happens regardless of which state it was in before being suspended.

[3] In FreeRTOS, this is done by the idle task—which is executed when the system is otherwise idle.

Example—Underground tank monitoring

- The underground tank monitoring system monitors up to eight underground tanks by reading thermometers and the levels of floats installed in those tanks.
- To read the floats level in one of the tanks, the microprocessors must send a command to the hardware to tell it which tank to read from.
- When the hardware has obtained a new float reading a few milliseconds letter, it interrupt; the microprocessor can read the the level from the hardware at any time later.
- In the code Listing 3 below:
  - vLevelTask compute gasoline in the tank. It is time consuming but has low priority
  - vButtonTask is short and has higher priority
  - if a user pressess a button, the RTOS block the vLevelTask task and run the high priority vButtonTask task.

Scheduler

- Can a task go from ready to blocked state? —No
  - 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"!
- Can a blocked task wake up on its own ?—No
  - 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.
- Can a task switch from ready to running or vice-versa on its own? —No
  - 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).

Example—Underground tank monitoring

- Two tasks can be written independently of one another.
- The programmers does not need to work much how fast the task will respond.
- Code in Listing 2 ensures that the RTOS knows which tasks are available and how they should be prioritized.

```

void main(void){
  // Initialize (but do not start) the RTOS
  RTOS_Init();
  // Tell the RTOS about the tasks
  StartTask(vButtonTask, PRIORITY_HIGH);
  StartTask(vLevelsTask, PRIORITY_LOW);
  //Start the RTOS (This function never returns)
  RTOS_Run();
}
```


Example—Underground tank monitoring

```
// High priority task
void vButtonTask(void){
  while(true){
    //Block until the user presses a button
    // Quick: Respond to the user pressing the
       button
  }
}
// Low priority task
void vLevelTask(void){
  while(true){
    // Read the level of floats in tank
    // Calculate average float level
    // Do some interminable calculations
    // Figure out which tank to do next
```


Example—Underground tank monitoring

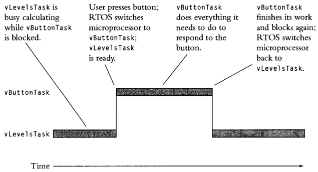


FIG 4. Tasks for an underground tank monitoring

Tasks and Data

- 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 variables
  - uninitialized and initialized variables
  - extern data types
- Shared data caused the shared-data problem[4] —use of “Reentrancy” characterization of functions to solve this

[4] The shared data problem occurs when several functions (or ISRs or tasks) share a variable. Shared data problem can arise in a system when another higher priority task finishes an operation and modifies the data or a variable before the completion of previous task operations.. See details at https://automaticaddison.com/what-is-the-shared-data-problem/

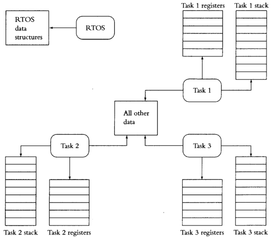


FIG 5. Data in an RTOS-based real-time system

Shared-Data Problems

- the shared object where the conflict may happen is a "resource"
- the parts of the code where the problem may happen are a critical sections

Critical section

critical section is a sequence of operations that cannot be interleaved with other operations on the same resource

- Two critical sections on the same resource must execute in mutual exclusion
- there are three ways to obtain mutual exclusion
  - implementing the critical section as an atomic operation
  - system-wide disabling the preemption
  - selectively disabling the preemption (e.g., using semaphores and mutex)

Shared-Data Problems

- The shared data problem occurs when several functions (or ISRs or tasks) share a variable.
- This problem can arise in a system when another higher priority task finishes an operation and modifies the data or a variable before the completion of previous task operations
- For example, in the code in Listing 4:
  - What would happen if the RTOS stops vCalculateTankLvelsTask(void) task and run vButtonTask(void) when the vCalculateTankLvelsTask(void) task was still in the middle of computing tankData[i].timeUpdated = getCurrentTime()?
  - In this case, the value displayed on the LCD will be wrong because the tankData[i].timeUpdated will contain the previous value,or worse, it might be even corrupted data[5]

[5] You should have learned this in your previous classes. For a refresher, please read about non-atomicity due to multiple CPU instructions and why this might corrupt data


```c
struct{
  long tankLevel, timeUpdated;
}tankData[MAX_TANKS];
void vButtonTask(void){
  int i;
  while(true){
    i= getPressedButtonId();
    updateLCD(tankData[i].tankLevel, tankData[i].
       timeUpdated);
  }
}
void vCalculateTankLvelsTask(void){
  int i;
  while(true){
    tankData[i].tankLevel = getCurrentTankLevel;
    tankData[i].timeUpdated = getCurrentTime();
    i = getNextTankId();
  }
}
```

Reentrancy

Reentrant function

A function that works correctly regardless of the number of tasks that call it between interrupts

- A Reentrant function
  - can be called by more than one task and will always work correctly,
  - even if the RTOS switches from one task to another in the middle of executing the function.
- Characteristics of reentrant functions
  - Only access shared variable in an atomic-way, or when variable is on callee's stack
  - A reentrant function calls only reentrant functions
  - A reentrant function uses system hardware (shared resource) atomically

How to check reentrancy?

Apply the following three 3 rules to check if a function is reentrant[6],[7]

1. Does not use variables in a nonatomic wayunless
   - they are stored on stack of the calling task, or
   - they are private variables of the task
   - does not use global and static data[8]
2. Does not call any non-reentrant functions
3. Does not use hardware in a nonatomic way[9]

[6] https://www.geeksforgeeks.org/reentrant-function/
[7] IBM has a nice tutorial on how to write reentrant and threadsafe code
[8] Though there are no restrictions, but it is generally not advised. because the interrupt may change certain global values and resuming the course of action of the reentrant function with the new data may give undesired results.
[9] for more information, see Jack Ganssle's introduction to reentrancy

Example —non-reentrant function

In Listing 5 Both fun1() and fun2() are not reentrant

- fun1() is NOT reentrant because it uses global variable i
- fun2() is NOT reentrant because it calls a non-reentrant function

```
int i;
int fun1(){
  return i * 5;
}
int fun2(){
  return fun1() * 5;
}
```

LISTING 5: Example non-reentrant functions

Example —reentrant function

In Listing 6, both fun1() and fun2() are reentrant

```
int fun1(int i){
  return i * 5;
}
int fun2(int i){
  return fun1(i) * 5;
}
```

LISTING 6: Example of reentrant functions

Example —non-reentrant function

Is the code in Listing 7 reentrant?

```c
bool error_flag = false;
void update_display(int j){
  if (!error_flag ){
    printf("\n Value: %d", j);
    j=0
    error_flag = true;
  }
  else{
    printf("\n Could not update the display");
    error_flag = false;
  }
}
```

LISTING 7: Example of non-reentrant functions

Example —non-reentrant function

The code in Listing 7 is not reentrant:

- non-atomic use of fError
- the printf() function may benon-reentrant[10]

[10] The C standard explicitly states that the functions in the standard library are not guaranteed to be reentrant and may modify objects with static storage duration. Thus, a signal handler cannot, in general, call standard library functions.

Reentrancy—some considerations

Is the code in Listing 8 reentrant?

- The function modifies a nonstack variable —thus, it should be non-reentrant.
- However, this may or may not be the case
- Maybe! Depends on microprocessor and compiler

```c
static int errors;
void update_errors(void){
  ++errors;
}
```

LISTING 8: Is this code reentrant?

Reentrancy—some considerations

- For AVR microcontrollers, the code would not be reentrant
- The compiler implemented the increment using three (load, increment, and store) machine instructions. —Thus, this operation is not atomic.

```
update_errors:
push r28
push r29
in r28,__SP_L__
in r29,__SP_H__
lds r24,errors
lds r25,errors+1
adiw r24,1
sts errors+1,r25
sts errors,r24
ret
```

LISTING 9: Assembly using AVR GCC

Reentrancy—some considerations

- For an Intel 8086 architecture, the code would be reentrant [11]
- The inc instruction increases by 1 the value of a variable. It is atomic in this case[12].

```
_errors:
.proc   _update_errors: near
inc     _errors
ret
```

LISTING 10: Assembly for 80x86 CPU

[11] The Intel 8086 is a 16-bit microprocessor chip designed by Intel in the late 1970s https://en.wikipedia.org/wiki/Intel_8086

[12] In other CPU architecture, increment is usually three operations: Load, Increment, then Store.

Race conditions

Race condition is an issue that hinders program correctness when two or more tasks are allowed uncontrolled access to some shared variables or, more generally, a shared resource

- Race condition zones appear only as a consequence of task splittingand, even in that case, their location in the schedule is well known in advance.
- In RTOS-based application, predicting race condition is hard to predict because the task switching points are nowchosen autonomously by the OS scheduler instead of being hard-coded in the code.

Semaphores and shared data

Semaphores

- Semaphore was proposed by Edsger W. Dijkstra[13] in 1965 which is a very significant technique to manage concurrent processes by using a simple integer value, which is known as a semaphore.
- Semaphore[14]—a flag that is used to control access to shared resource
- Semaphores are used to avoid shared-data problems in RTOS
- In theory, a semaphore is a shared counter that can be incremented and decremented atomically.
- According to its abstract definition, a semaphore is an object that contains two items of information
  - a value v—represented as a nonnegative integer
  - a queue of tasks q—which are waiting on the semaphore.

[13] https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

[14] https://www.guru99.com/semaphore-in-operating-system.html

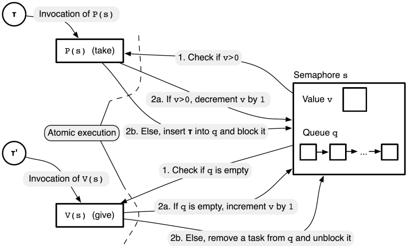


FIG 6. Abstract structure of a semaphore and behavior of its primitives


```
int s =0;
void semaphore_init (){
  s =0;
}
void P(){
  if(s == 0){
    //block any other tasks to access the semaphore
    while(s == 0){/*Do nothing. Just spin around*/}
  }
  else{
    s--;
  }
}
void V(){
  s++;
}
```

LISTING 11: Semaphore pseudocode

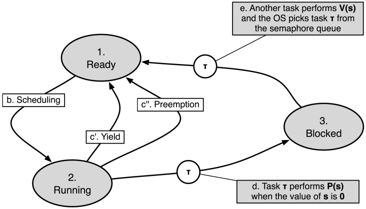


FIG 7. Task states and transitions involved in semaphore operations in FreeRTOS
Note that semaphore primitives are tied to the task state diagram because their execution may induce the transition of a task from one state to another.

- If task τ enter the critical section, the primitive P(s) will execute, and find the initial value of s, $s = 1$. it will decrement the value to $s = 0$, and will be allowed to proceed into its critical region immediately
- If another task τ'tries to enter the critical section while task τ is executing, task τ' will be blocked because the current value of semaphore $s = 0$

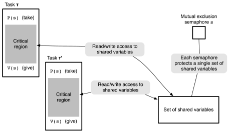


FIG 8. Usage of a semaphore and its primitives for mutual exclusion.

Mutual exclusion with semaphores

How to use a semaphore for critical sections

- before entering the critical section, perform a wait
- after leaving the critical section, perform a post

```
void CriticalTask(void){
  // other code
  ...
  semaphore_take();
  <critical section>
  semaphore_release()
  ...
  // other code
}
```


Example—Underground tank monitoring

```
struct{
  long tankLevel, timeUpdated;
}tankData[MAX_TANKS];

void vCalculateTankLvelsTask(void){
  int i;
  while(true){
    TakeSemaphore();
    tankData[i].tankLevel = getCurrentTankLevel;
    tankData[i].timeUpdated = getCurrentTime();
    ReleaseSemaphore();
    i = getNextTankId();
  }
}
```

Listing 12: Solving the underground tank monitoring problem with

Example—Underground tank monitoring

If the user presses a button while the vCalculateTankLvelsTask(void) task is still modifying the data, and still has the semaphore, then:

- The RTOS will switch to the vButtonTask(void) task and moved the vCalculateTankLvelsTask(void) task to the ready state.
- When the vButtonTask(void) task tries to get the semaphore by calling TakeSemaphore(), it will block because the semaphore is already taken by the vCalculateTankLvelsTask(void) task.
- The RTOS will then look for another task to run and will switch back to the vCalculateTankLvelsTask(void) task since it is in the ready state.
- The vCalculateTankLvelsTask(void) task will until completion, releases the semaphore

Semaphores in FreeRTOS[16]

FreeRTOS provides four different semaphore implementations:

1. counting semaphores
   - Equivalent to the canonical definition of a semaphore
   - The slowest implementation
   - The value of *s* can be declared when the semaphore is declared
2. Binary semaphores
   - Their value can only be either one or zero, but they can still be used for either mutual exclusion or task synchronization.
   - Faster than the one of counting semaphores.
3. Mutex semaphores
   - they must only be used as mutual exclusion semaphores, i.e., the P(s) and V(s) primitives on a mutex semaphore *s* must always appear in pairs and must be placed as brackets around critical regions.
   - cannot be used for task synchronization
4. Recursive mutex semaphores[15]

Binary semaphore

- A binary semaphore —only one task can have the semaphore at a time.
- Two functions to control the semaphore:
  1. TakeSemaphore()
     - block until the semaphore is released
     - take the semaphore
  2. ReleaseSemaphore()—release a taken semaphore

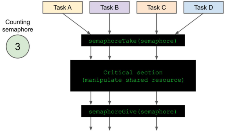


FIG 9. Concept of a semaphore

Working principle—principle: if one task has called the TakeSemaphore() function, and has not yet called ReleaseSemaphore()function to release it, then any other task

Semaphores in FreeRTOS

- The four kinds of semaphore are created using functions listed in Table 1[17].

TAB 1. Semaphore creation and deletion primitives of FreeRTOS

Function	Purpose	Optional
xSemaphoreCreateCounting	Create a counting semaphore	*
xSemaphoreCreateBinary	Create a binary semaphore	-
xSemaphoreCreateMutex	Create a mutex semaphore	*
xSemaphoreCreateRecursiveMutex	Create a recursive mutex	*
vSemaphoreDelete	Delete a semaphore (of any kind)	-

- If semaphore the creation fails (e.g., no heap memory available), the function returns a NULL pointer as shown in Listing 13.

[17] Detailed info on variation semaphores API is found at https://www.freertos.org/a00113.html


```c
SemaphoreHandle_t xSemaphore;
void vSemaphoreExampleTask( void * pvParameters ){
    /* Attempt to create a semaphore. */
    xSemaphore = xSemaphoreCreateBinary();
    if( xSemaphore == NULL )
    {
      /* There was insufficient FreeRTOS heap
         available for the semaphore to be created
          successfully. */
    }
    else
    {
    /*  The semaphore can now be used and its
       handle is stored in the xSemahore variable.
         Note that calling xSemaphoreTake() on the
        semaphore here will fail until the
       semaphore has first been given. */
  }
```


Semaphore manipulation in FreeRTOS

- Once created, a semaphore can be manipulated with function listed

TAB 2. Semaphore Manipulation Primitives of FreeRTOS

Function	Purpose	Optional
xSemaphoreTake	Perform a P() on a semaphore	-
xSemaphoreGive	Perform a V() on a semaphore	-
xSemaphoreTakeFromISR	P() from an interrupt handler	-
xSemaphoreGiveFromISR	V() from an interrupt handler	-
xSemaphoreTakeRecursive	P() on a recursive mutex	*
xSemaphoreGiveRecursive	V() on a recursive mutex	*

- Except the mutual exclusion semaphores, most semaphores are acted upon by means of the functions xSemaphoreTake() and xSemaphoreGive(), the FreeRTOS counterpart of P() and V(), respectively

```c
BaseType_t xSemaphoreTake(SemaphoreHandle_t
```

Remarks

The FreeRTOS implementation of semaphores is slightly different from the canonical algorithms:

- The canonical algorithm block the caller for an unlimited amount of time. This is not reasonable for a RT system. Thus, the function xSemaphoreTake() has a second argument, xBlockTime that specifies the maximum blocking time:
  - if xBlockTime==portMAX_DELAY, the function blocks the caller until the semaphore operation is complete, i.e., it behaves like the canonical algorithm.
  - If xBlockTime==0, the function returns an error indication to the caller when the operation cannot be performed immediately.
  - Any other value is interpreted as the maximum amount of time the function will possibly block the caller, expressed as an integral number of clock ticks.
- Canonical algorithm is assumed to never fail. However, in the real world, things go wrong. For this reason, the return value

Potential issues in using semaphores

- The initial values of semaphores – when not set properly or at the wrong place
- The symmetry of takes and releases must match
  - each take must have a corresponding release somewhere in the application
  - Avoid Taking the wrong semaphore unintentionally (issue with
- Holding a semaphore for too long can cause waiting tasks—deadline to be missed
- Priorities could be inverted and usually solved by priority inheritance/promotion
- Semaphore work only if you use them perfectly—and there is no guarantees that you will
- **SUMMARY**—Using semaphore is a bug waiting to happen. Use them sparingly.

Priority inversion

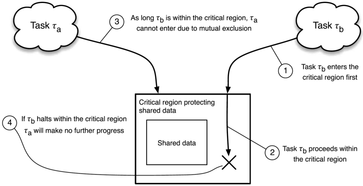


FIG 10. Shortcoming of lock-based synchronization when a task halts. If task τ_b is delayed while it is within its critical region, τ_a and any other tasks willing to enter a critical region associated with the same lock will be blocked and possibly be unable to make any further progress. Even though τ_b proceeds normally, if the priority of $\tau_a$is higher than the priority of τ_b, the way mutual exclusion is implemented goes against the concept of task priority, because a higher-priority task is forced to wait until a lower-priority task has completed part of its activities.

Priority inversion —Principle

Priority inversion is a bug that occurs when a high priority task is indirectly preempted by a low priority task.

- For example, the low priority task holds a mutex that the high priority task must wait for to continue executing[18].
- In this case, the high priority task (Task H) would be blocked as long as the low priority task (Task L) held the lock.
- This is known as bounded priority inversion as the length of time of the inversion is bounded by however long the low priority task is in the critical section (holding the lock)[19].
- Unbounded priority inversion occurs when a medium priority task (Task M) interrupts Task L while it holds the lock. It's called "unbounded" because Task M can now effectively block Task H for any amount of time, as Task M is preempting Task L —which still holds the lock

[18] We will talk about mutex later. An interested reader can read a few discussion Stackoverflow

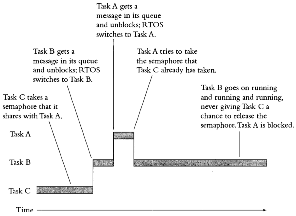


FIG 11. Priority Inversion —Task A has the highest priority, Task B a medium priority and Task C the lowest priority. Priority inversion happen when the RTOS switches from a low-priority task to a medium priority after the lowest priority task has taken a semaphore. If the high priority task wants the semaphore, it will have to wait until the medium task blocks. The lowest priority cannot release the semaphore since it is blocked; thus, holds up the highest priority indefinitely

Bounded priority inversion

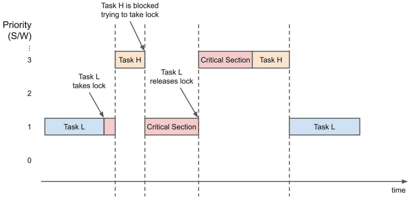


FIG 12. Bounded priority inversion
the high priority task is blocked as long as the low priority task holds the lock

Unbounded priority inversion

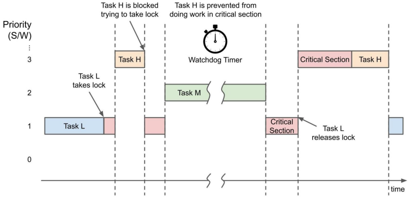


FIG 13. Unbounded priority inversion
Unbounded priority inversion occurs when a medium priority task interrupts a high priority task while it holds the lock

Priority inversion —trivia

- Priority inversion nearly ended the Mars Pathfinder mission in 1997
- After deploying the rover, the lander would randomly reset every few days due to an intermittent priority inversion bug that caused the watchdog timer to trigger a full system restart.
- NASA eventually found the bug and sent an update patch to the lander.

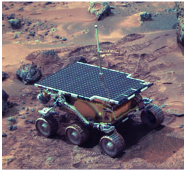


FIG 14. Mars Pathfinder landed a base station with a roving probe on Mars in 1997. Priority inversion nearly ended the Mars Pathfinder mission in 1997

[17] What really happened on Mars Rover Pathfinder?

[18] Mutexes and Semaphores Demystified

Ways to Protect Shared Data

- Disabling interrupts
  - Most drastic, affects all other tasks
  - Only method if task & interrupts share data
  - Fast (single instruction)
- Using 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

The end