Operating Systems

Real-Time Operating Systems

Chalermek Intanagonwiwat

Slides courtesy of Subhashis Banerjee

Real-Time Systems

- Result in severe consequences if logical and timing correctness are not met
- Two types exist
  - Soft real-time
    - Tasks are performed as fast as possible
    - Late completion of jobs is undesirable but not fatal.
    - System performance degrades as more & more jobs miss deadlines
    - Example:
      - Online Databases

Real-Time Systems (cont.)

- Hard real-time
  - Tasks have to be performed on time
  - Failure to meet deadlines is fatal
  - Example :
    - Flight Control System
- Qualitative Definition

Hard and Soft Real Time Systems

(Operational Definition)

- Hard Real Time System
  - Validation by provably correct procedures or extensive simulation that the system always meets the timings constraints
- Soft Real Time System
  - Demonstration of jobs meeting some statistical constraints suffices.
- Example – Multimedia System
  - 25 frames per second on an average

Most Real-Time Systems are embedded

- An embedded system is a computer built into a system but not seen by users as being a computer
- Examples
  - FAX machines
  - Copiers
  - Printers
  - Scanners
  - Routers
  - Robots

Role of an OS in Real Time Systems

- Standalone Applications
  - Often no OS involved
  - Micro controller based Embedded Systems
- Some Real Time Applications are huge & complex
  - Multiple threads
  - Complicated Synchronization Requirements
  - File system / Network / Windowing support
  - OS primitives reduce the software design time

Features of Real Time OS (RTOS)

- Scheduling.
- Resource Allocation.
- Interrupt Handling.
- Other issues like kernel size.

Foreground/Background Systems

- Small systems of low complexity
- These systems are also called "super-loops"
- An application consists of an infinite loop of desired operations (background)
- Interrupt service routines (ISRs) handle asynchronous events (foreground)

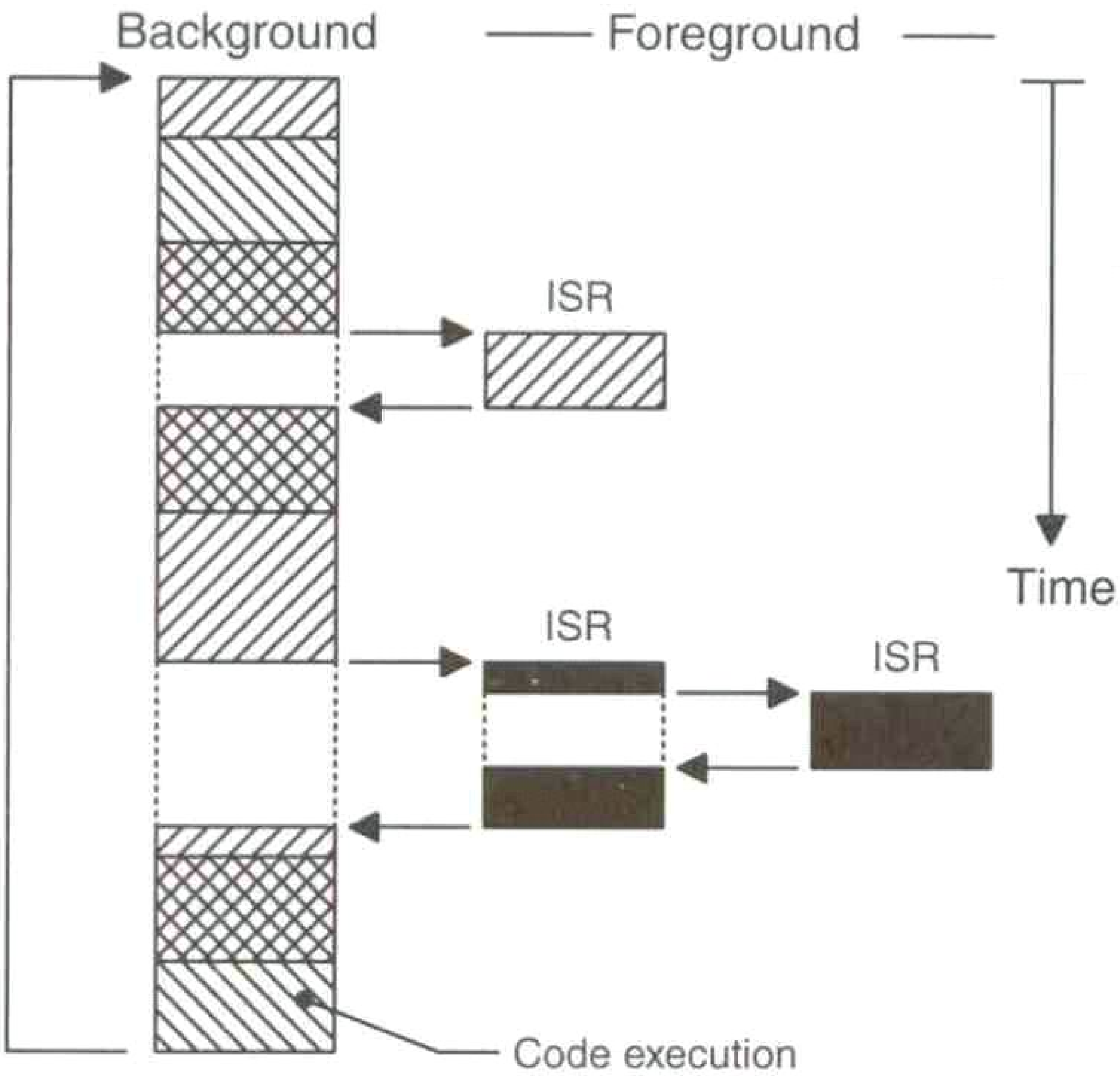
Foreground/Background Systems (cont.)

- Critical operations must be performed by the ISRs to ensure the timing correctness
- Thus, ISRs tend to take longer than they should
- Task-Level Response
  - Information for a background module is not processed until the module gets its turn

Foreground/Background Systems (cont.)

- The execution time of typical code is not constant
- If a code is modified, the timing of the loop is affected
- Most high-volume microcontroller-based applications are F/B systems
  - Microwave ovens
  - Telephones
  - Toys

Foreground/Background Systems (cont.)

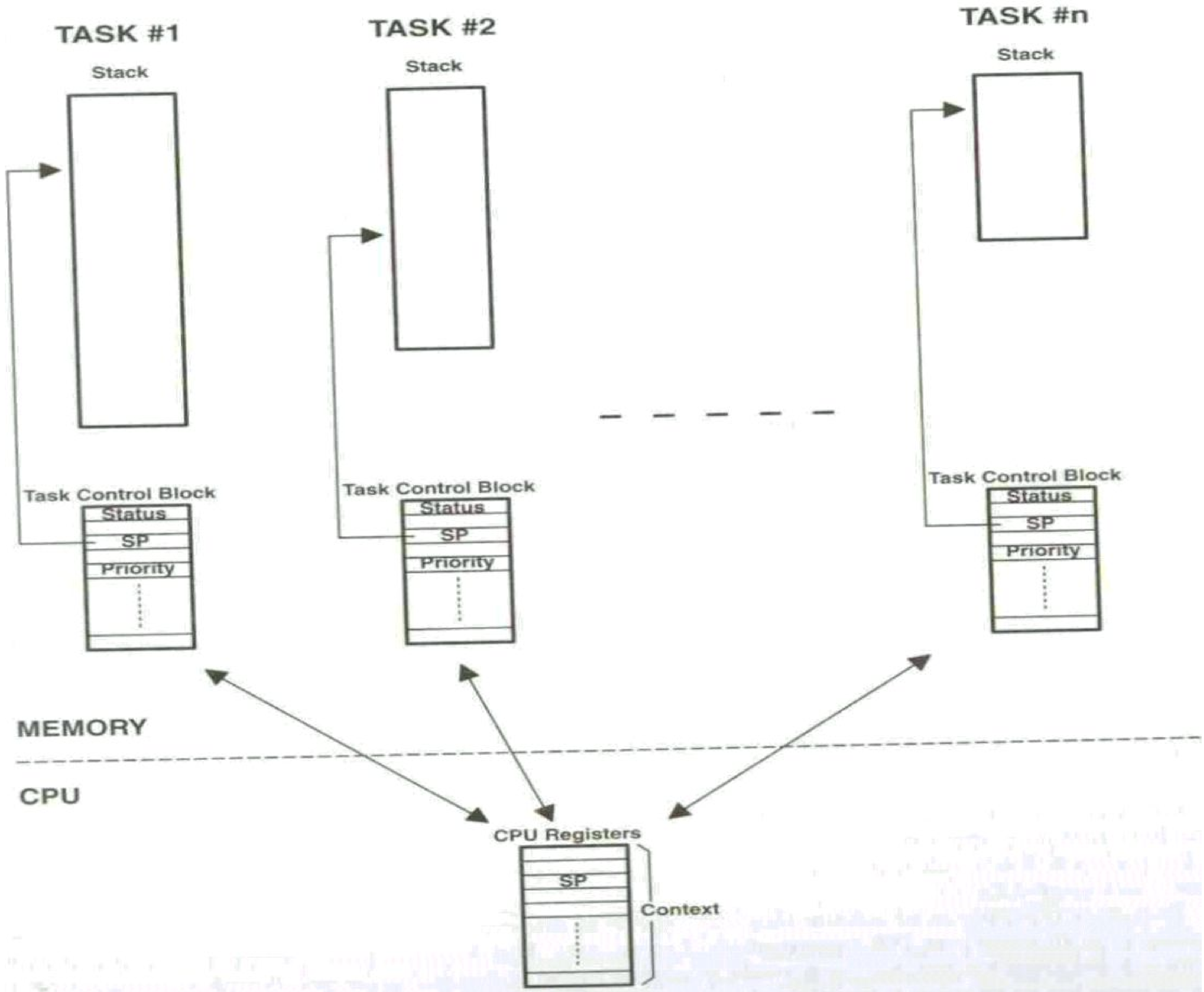


- From a power consumption point of view, it might be better to halt and perform all processing in ISRs

Multitasking Systems

- Like F/B systems with multiple backgrounds
- Allow programmers to manage complexity inherent in real-time applications

Multitasking Systems (cont.)



Scheduling in RTOS

- More information about the tasks are known
  - Number of tasks
  - Resource Requirements
  - Execution time
  - Deadlines
- Being a more deterministic system better scheduling algorithms can be devised.

Scheduling Algorithms in RTOS

- Clock Driven Scheduling
- Weighted Round Robin Scheduling
- Priority Scheduling

Scheduling Algorithms in RTOS (cont.)

- Clock Driven
  - All parameters about jobs (execution time/deadline) known in advance.
  - Schedule can be computed offline or at some regular time instances.
  - Minimal runtime overhead.
  - Not suitable for many applications.

Scheduling Algorithms in RTOS (cont.)

- Weighted Round Robin
  - Jobs scheduled in FIFO manner
  - Time quantum given to jobs is proportional to it's weight
  - Example use : High speed switching network
    - QOS guarantee.
  - Not suitable for precedence constrained jobs.
    - Job A can run only after Job B. No point in giving time quantum to Job B before Job A.

Scheduling Algorithms in RTOS (cont.)

- Priority Scheduling
  - Processor never left idle when there are ready tasks
  - Processor allocated to processes according to priorities
  - Priorities
    - Static - at design time
    - Dynamic - at runtime

Priority Scheduling

- Earliest Deadline First (EDF)
  - Process with earliest deadline given highest priority
- Least Slack Time First (LSF)
  - slack = relative deadline – execution left
- Rate Monotonic Scheduling (RMS)
  - For periodic tasks
  - Tasks priority inversely proportional to it's period

Schedulers

- Also called “dispatchers”
- Schedulers are parts of the kernel responsible for determining which task runs next
- Most real-time kernels use priority-based scheduling
  - Each task is assigned a priority based on its importance
  - The priority is application-specific

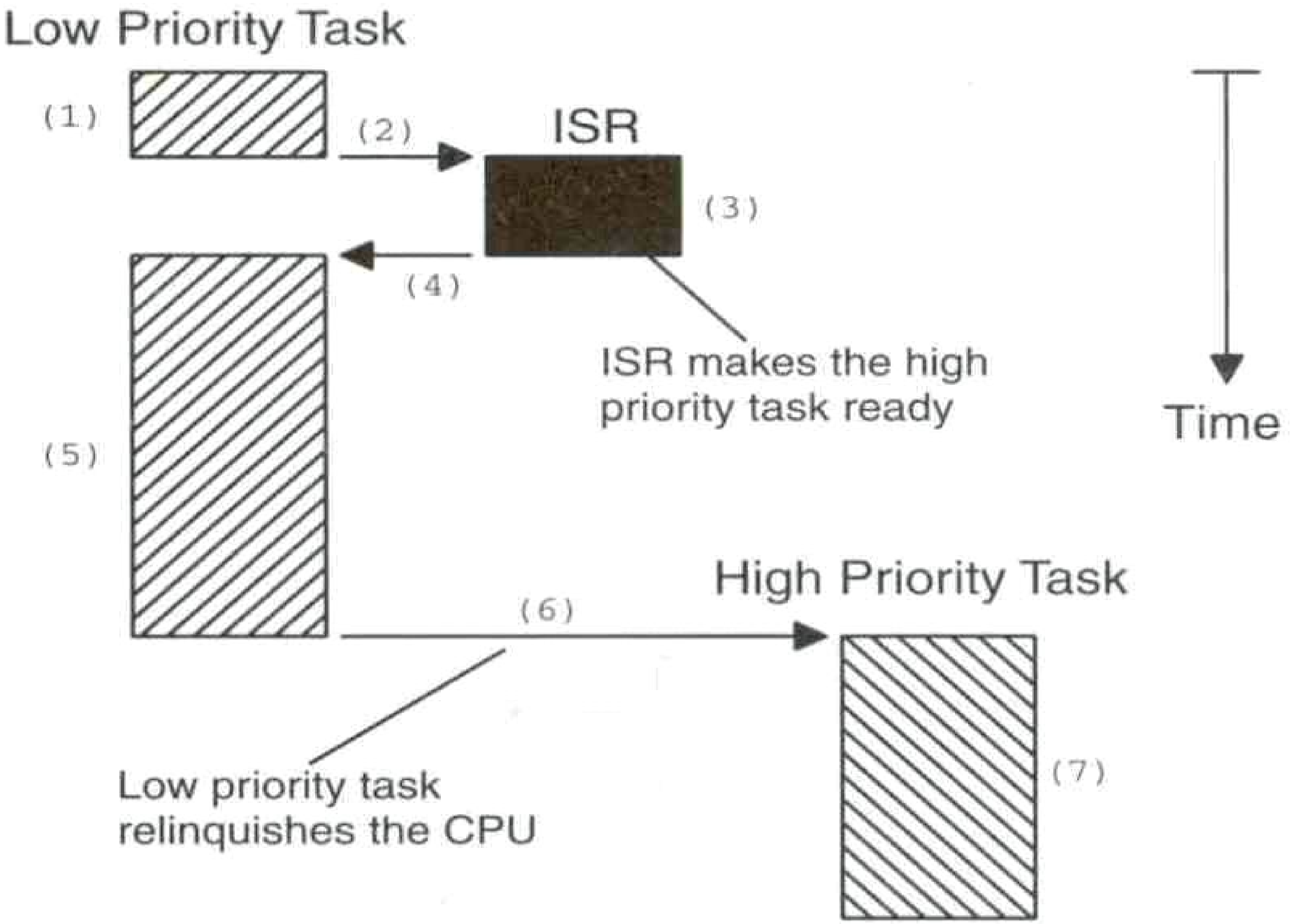
Priority-Based Kernels

- There are two types
  - Non-preemptive
  - Preemptive

Non-Preemptive Kernels

- Perform "cooperative multitasking"
  - Each task must explicitly give up control of the CPU
  - This must be done frequently to maintain the illusion of concurrency
- Asynchronous events are still handled by ISRs
  - ISRs can make a higher-priority task ready to run
  - But ISRs always return to the interrupted tasks

Non-Preemptive Kernels (cont.)



Advantages of Non-Preemptive Kernels

- Interrupt latency is typically low
- Can use non-reentrant functions without fear of corruption by another task
  - Because each task can run to completion before it relinquishes the CPU
  - However, non-reentrant functions should not be allowed to give up control of the CPU
- Task-response is now given by the time of the longest task
  - much lower than with F/B systems

Advantages of Non-Preemptive Kernels (cont.)

- Less need to guard shared data through the use of semaphores
  - However, this rule is not absolute
  - Shared I/O devices can still require the use of mutual exclusion semaphores
  - A task might still need exclusive access to a printer

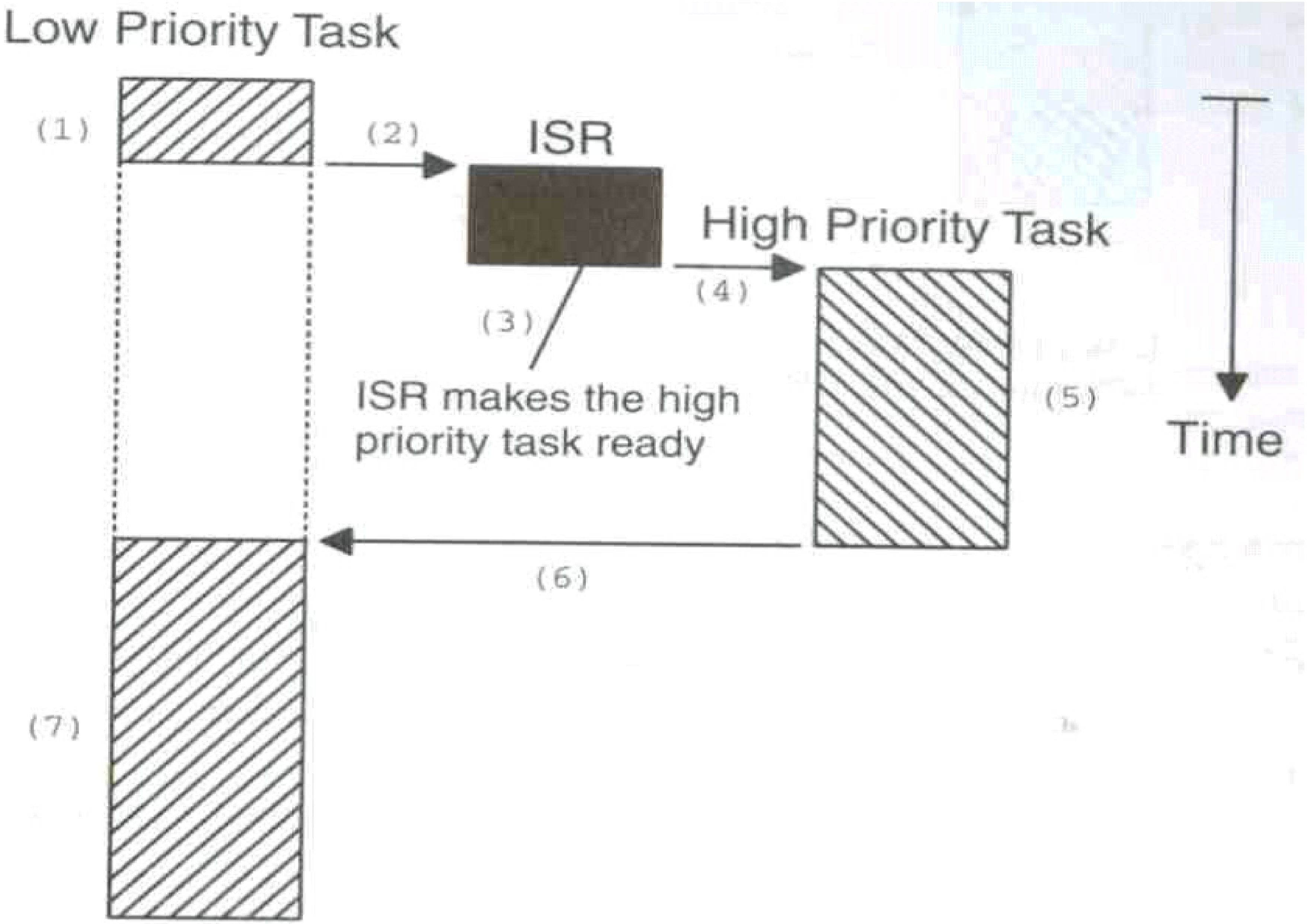
Disadvantages of Non-Preemptive Kernels

- Responsiveness
  - A higher priority task might have to wait for a long time
  - Response time is nondeterministic
- Very few commercial kernels are non-preemptive

Preemptive Kernels

- The highest-priority task ready to run is always given control of the CPU
  - If an ISR makes a higher-priority task ready, the higher-priority task is resumed (instead of the interrupted task)
- Most commercial real-time kernels are preemptive

Preemptive Kernels (cont.)



Advantages of Preemptive Kernels

- Execution of the highest-priority task is deterministic
- Task-level response time is minimized

Disadvantages of Preemptive Kernels

- Should not use non-reentrant functions unless exclusive access to these functions is ensured

Reentrant Functions

- A reentrant function can be used by more than one task without fear of data corruption
- It can be interrupted and resumed at any time without loss of data
- It uses local variables (CPU registers or variables on the stack)
- Protect data when global variables are used

Reentrant Function Example

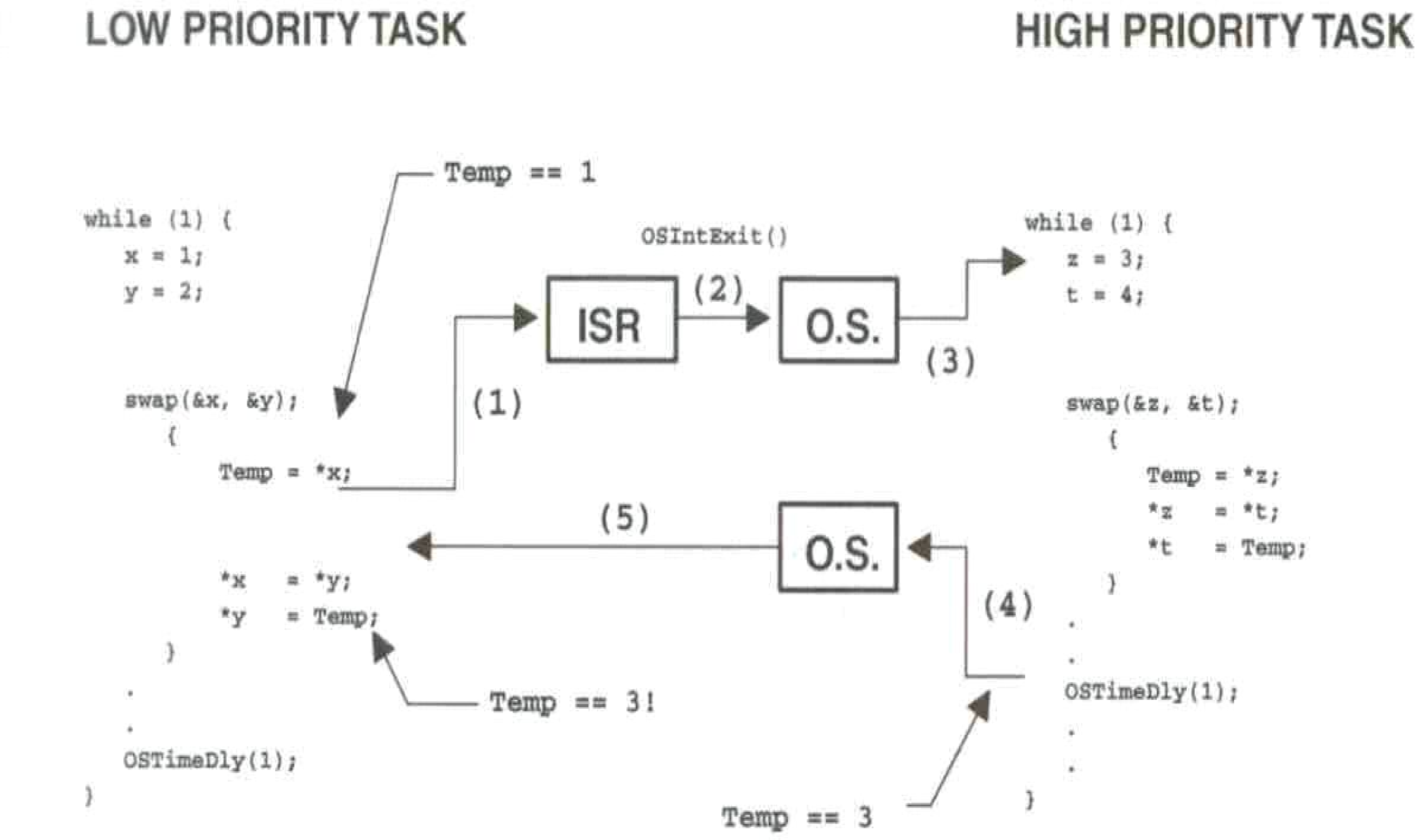
```
void strcpy(char *dest, char *src)
{
  while (*dest++ = *src++) {
      …
  }
  *dest = NULL;
}
```

Non-Reentrant Function Example

```
int Temp;

void swap(int *x, int *y)
{
   Temp = *x;
   *x = *y;
   *y = Temp;
}
```

Non-Reentrant Function Example (cont.)



LOW PRIORITY TASK

```
while (1) {
    x = 1;
    y = 2;


    swap(&x, &y);
        {
            Temp = *x;


            *x    = *y;
            *y    = Temp;
        }
    .
    .
    OSTimeDly(1);
}
```

HIGH PRIORITY TASK

```
while (1) {
    z = 3;
    t = 4;


    swap(&z, &t);
        {
            Temp = *z;
            *z   = *t;
            *t   = Temp;
        }
    .
    .
    OSTimeDly(1);
    .
    .
}
```

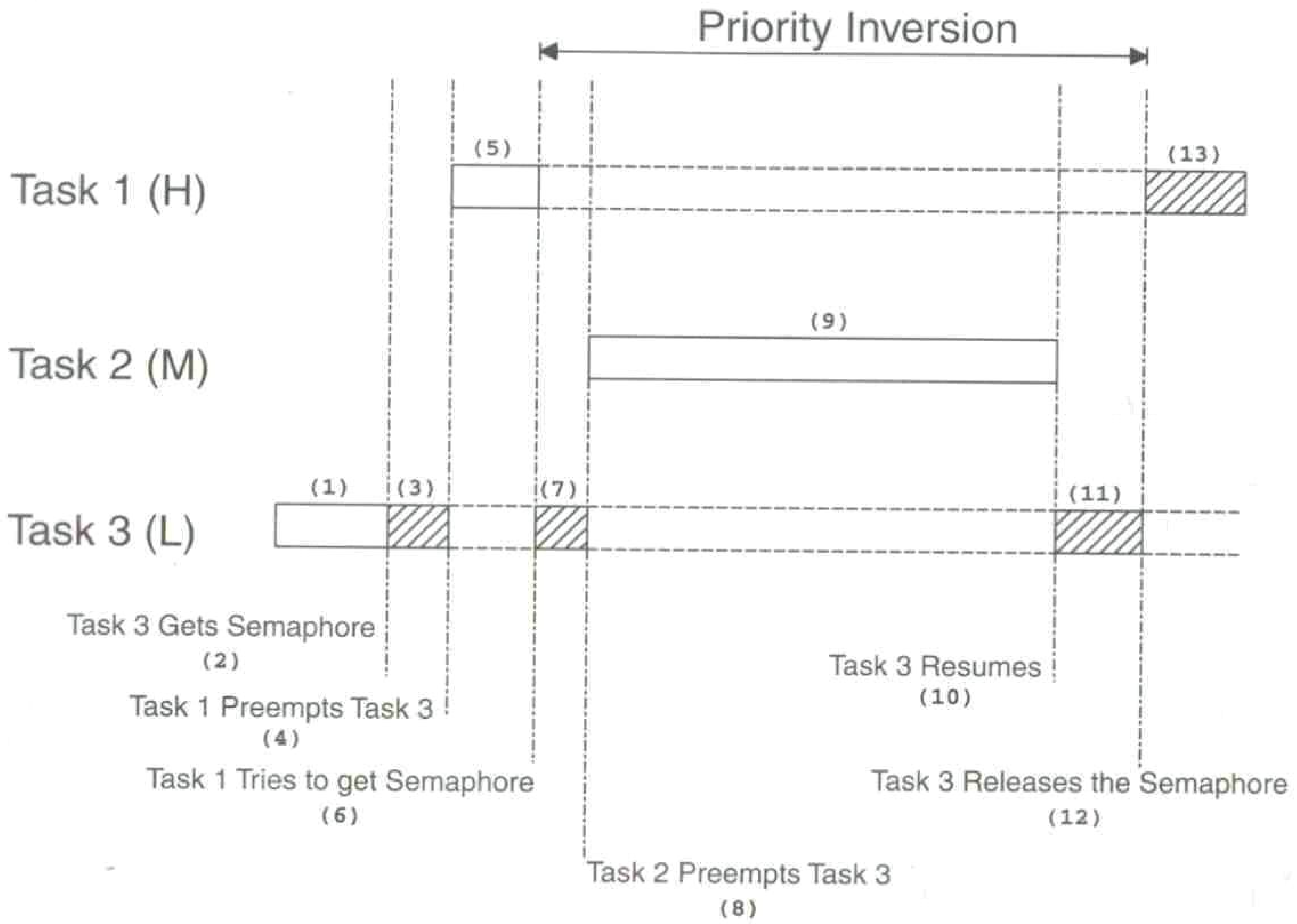
Resource Allocation in RTOS

- Resource Allocation
  - The issues with scheduling applicable here.
  - Resources can be allocated in
    - Weighted Round Robin
    - Priority Based
- Some resources are non preemptible
  - Example: semaphores
- Priority inversion problem may occur if priority scheduling is used

Priority Inversion Problem

- Common in real-time kernels
- Suppose task 1 has a higher priority than task 2
- Also, task 2 has a higher priority than task 3
- If mutual exclusion is used in accessing a shared resource, priority inversion may occur

Priority Inversion Example



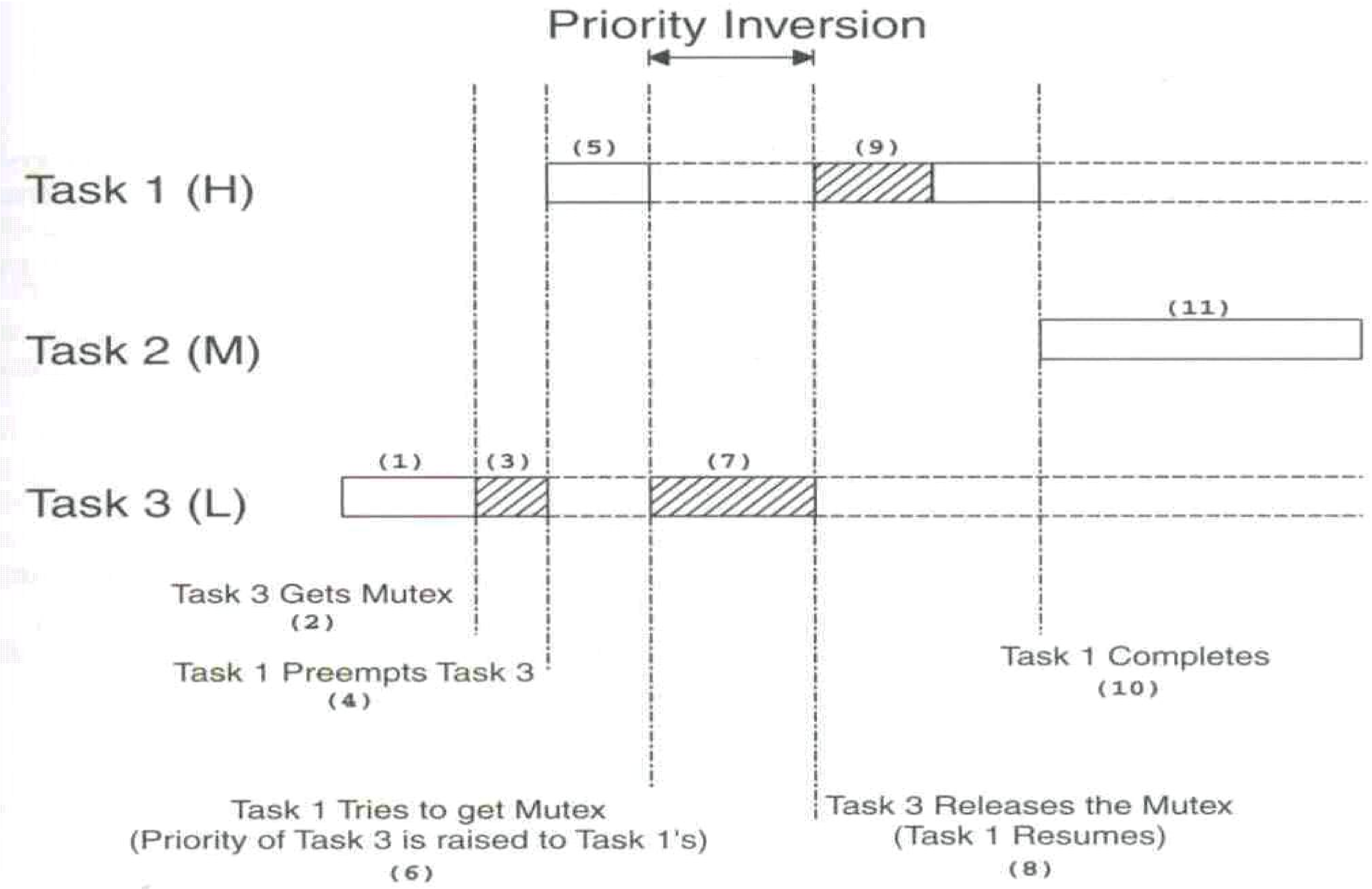
A Solution to Priority Inversion Problem

- We can correct the problem by raising the priority of task 3
  - Just for the time it accesses the shared resource
  - After that, return to the original priority
  - What if task 3 finishes the access before being preempted by task 1?
    - incur overhead for nothing

A Better Solution to the Problem

- Priority Inheritance
  - Automatically change the task priority when needed
  - The task that holds the resource will inherit the priority of the task that waits for that resource until it releases the resource

Priority Inheritance Example



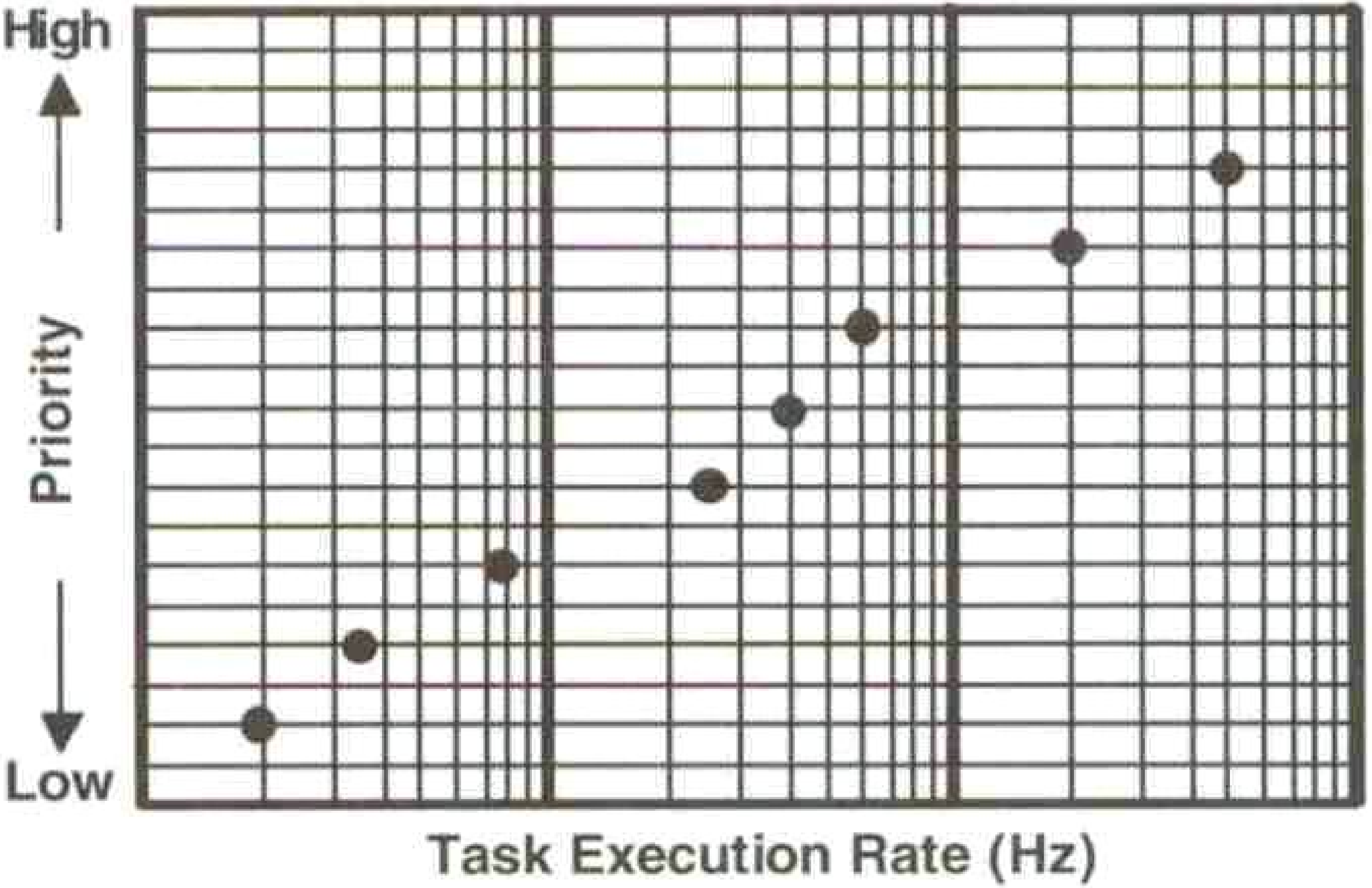
Assigning Task Priorities

- Not trivial
- In most systems, not all tasks are critical
  - Non-critical tasks are obviously low-priorities
- Most real-time systems have a combination of soft and hard requirements

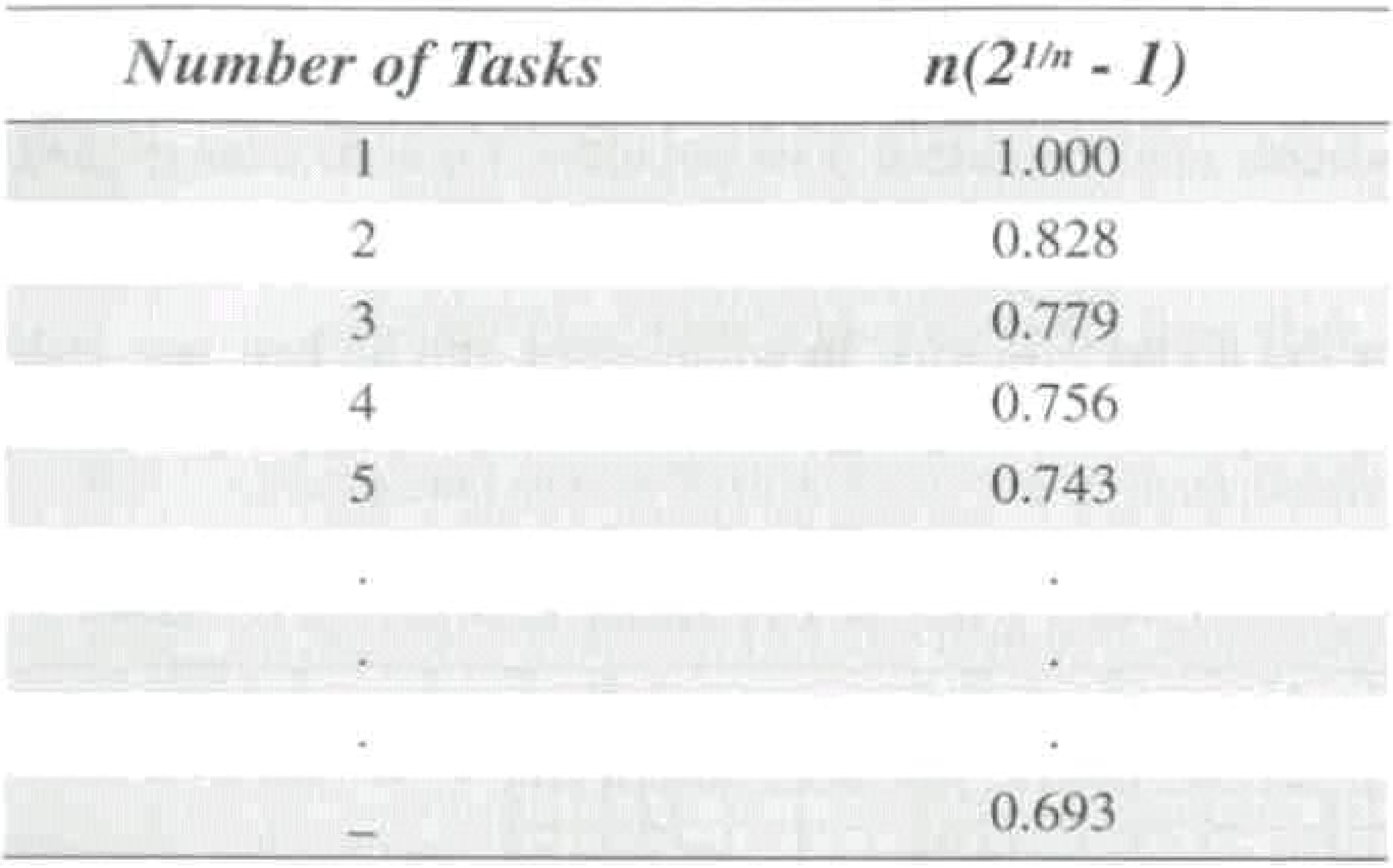
A Technique for Assigning Task Priorities

- Rate Monotonic Scheduling (RMS)
  - Priorities are based on how often tasks execute
- Assumption in RMS
  - All tasks are periodic with regular intervals
  - Tasks do not synchronize with one another, share data, or exchange data
  - Preemptive scheduling

RMS Example



RMS: CPU Time and Number of Tasks



Number of Tasks	$n(2^{1/n} - 1)$
1	1.000
2	0.828
3	0.779
4	0.756
5	0.743
.	.
.	.
.	.
–	0.693

RMS: CPU Time and Number of Tasks (cont.)

- The upper bound for an infinite number of tasks is 0.6973
  - To meet all hard real-time deadlines based on RMS, CPU use of all time-critical tasks should be less than 70%
  - Note that you can still have non-time-critical tasks in a system
  - So, 100% of CPU time is used
  - But not desirable because it does not allow code changes or added features later

RMS: CPU Time and Number of Tasks (cont.)

- Note that, in some cases, the highest-rate task might not be the most important task
    - Eventually, application dictates the priorities
    - However, RMS is a starting point

Other RTOS issues

- Interrupt Latency should be very small
  - Kernel has to respond to real time events
  - Interrupts should be disabled for minimum possible time
- For embedded applications Kernel Size should be small
  - Should fit in ROM
- Sophisticated features can be removed
  - No Virtual Memory
  - No Protection

Mutual Exclusion

- The easiest way for tasks to communicate is through shared data structures
  - Global variables, pointers, buffers, linked lists, and ring buffers
- Must ensure that each task has exclusive access to the data to avoid data corruption

Mutual Exclusion (cont.)

- The most common methods are:
  - Disabling interrupts
  - Performing test-and-set operations
  - Disabling scheduling
  - Using semaphores

Disabling and Enabling Interrupts

- The easiest and fastest way to gain exclusive access
- Example:

  Disable interrupts;

  Access the resource;

  Enable interrupts;

Disabling and Enabling Interrupts (cont.)

- This is the only way that a task can share variables with an ISR
- However, do not disable interrupts for too long
- Because it adversely impacts the “interrupt latency!”
- Good kernel vendors should provide the information about how long their kernels will disable interrupts

Test-and-Set (TAS) Operations

- Two functions could agree to access a resource based on a global variable value
- If the variable is 0, the function has the access
    - To prevent the other from accessing the resource, the function sets the variable to 1
- TAS operations must be performed indivisibly by the CPU (e.g., 68000 family)
- Otherwise, you must disable the interrupts when doing TAS on the variable

TAS Example

```
Disable interrupts;
if (variable is 0) {
    Set variable to 1;
    Enable interrupts;
    Access the resource;
    Disable interrupts;
    Set variable to 0;
    Enable interrupts;
} else {
    Enable interrupts;
}
```

Disabling and Enabling the Scheduler

- Viable for sharing variables among tasks but not with an ISR
- Scheduler is locked but interrupts are still enabled
  - Thus, ISR returns to the interrupted task
  - Similar to a non-preemptive kernel (at least, while the scheduler is locked)

Disabling and Enabling the Scheduler (cont.)

- Example:
  - Lock scheduler;
  - Access shared data;
  - Unlock scheduler;
- Even though this works well, you should avoid it because it defeats the purpose of having a kernel

Semaphores

- Invented by Edgser Dijkstra in the mid-1960s
- Offered by most multitasking kernels
- Used for:
  - Mutual exclusion
  - Signaling the occurrence of an event
  - Synchronizing activities among tasks

Semaphores (cont.)

- A semaphore is a key that your code acquires in order to continue execution
- If the key is already in use, the requesting task is suspended until the key is released
- There are two types
  - Binary semaphores
    - 0 or 1
  - Counting semaphores
    - >= 0

Semaphore Operations

- Initialize (or create)
  - Value must be provided
  - Waiting list is initially empty
- Wait (or pend)
  - Used for acquiring the semaphore
  - If the semaphore is available (the semaphore value is positive), the value is decremented, and the task is not blocked
  - Otherwise, the task is blocked and placed in the waiting list
  - Most kernels allow you to specify a timeout
  - If the timeout occurs, the task will be unblocked and an error code will be returned to the task

Semaphore Operations (cont.)

- Signal (or post)
  - Used for releasing the semaphore
  - If no task is waiting, the semaphore value is incremented
  - Otherwise, make one of the waiting tasks ready to run but the value is not incremented
  - Which waiting task to receive the key?
    - Highest-priority waiting task
    - First waiting task

Semaphore Example

```
Semaphore *s;
Time timeout;
INT8U error_code;


timeout = 0;
Wait(s, timeout, &error_code);
Access shared data;
Signal(s);
```

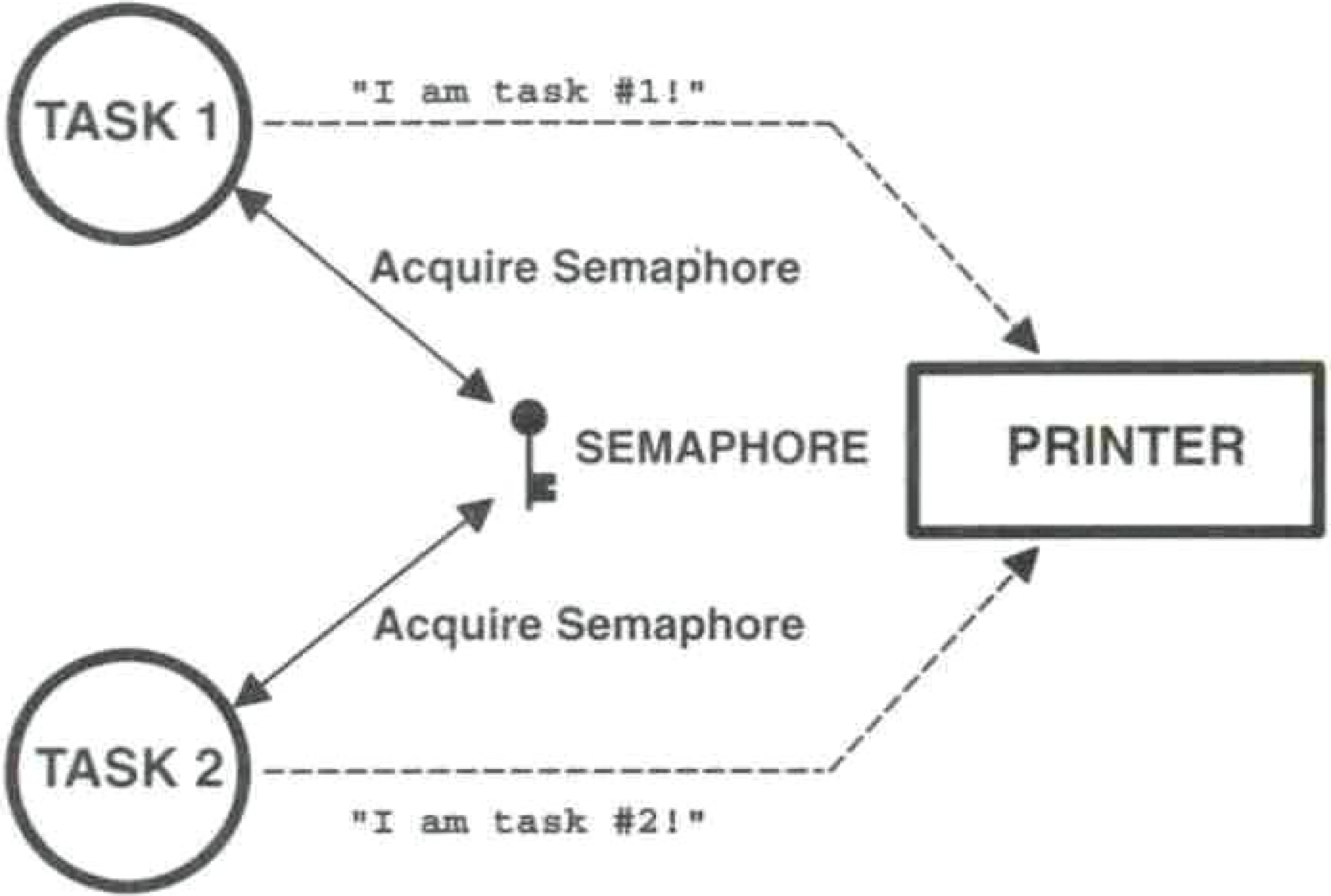
Applications of Binary Semaphores

- Suppose task 1 prints "I am Task 1!"
- Task 2 prints "I am Task 2!"
- If they were allowed to print at the same time, it could result in:

  I Ia amm T Tasask k1!2!

- Solution:
  - Binary semaphore

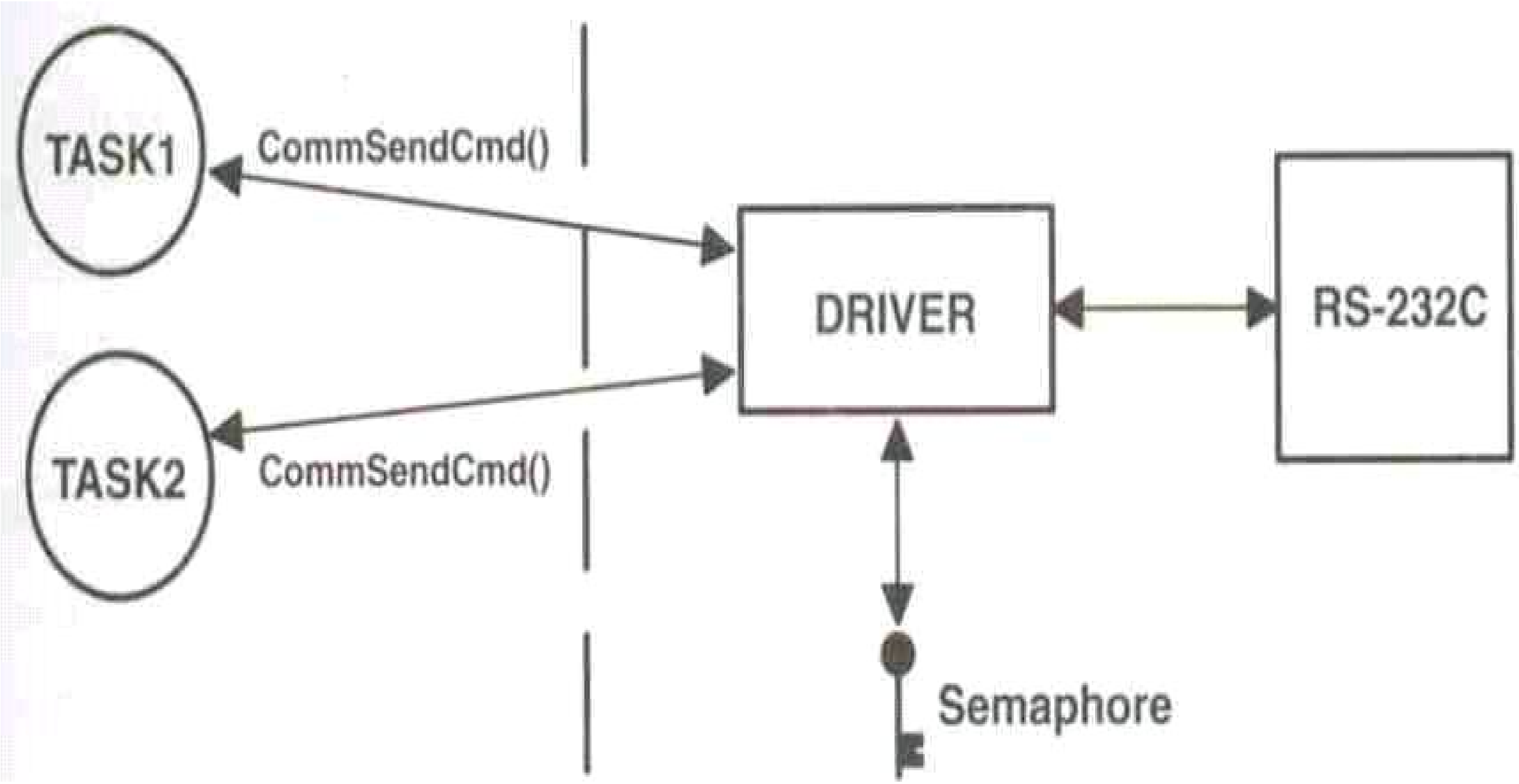
Sharing I/O Devices



Sharing I/O Device (cont.)

- In the example, each task must know about the semaphore in order to access the device
- A better solution:
  - Encapsulate the semaphore

Encapsulating a Semaphore



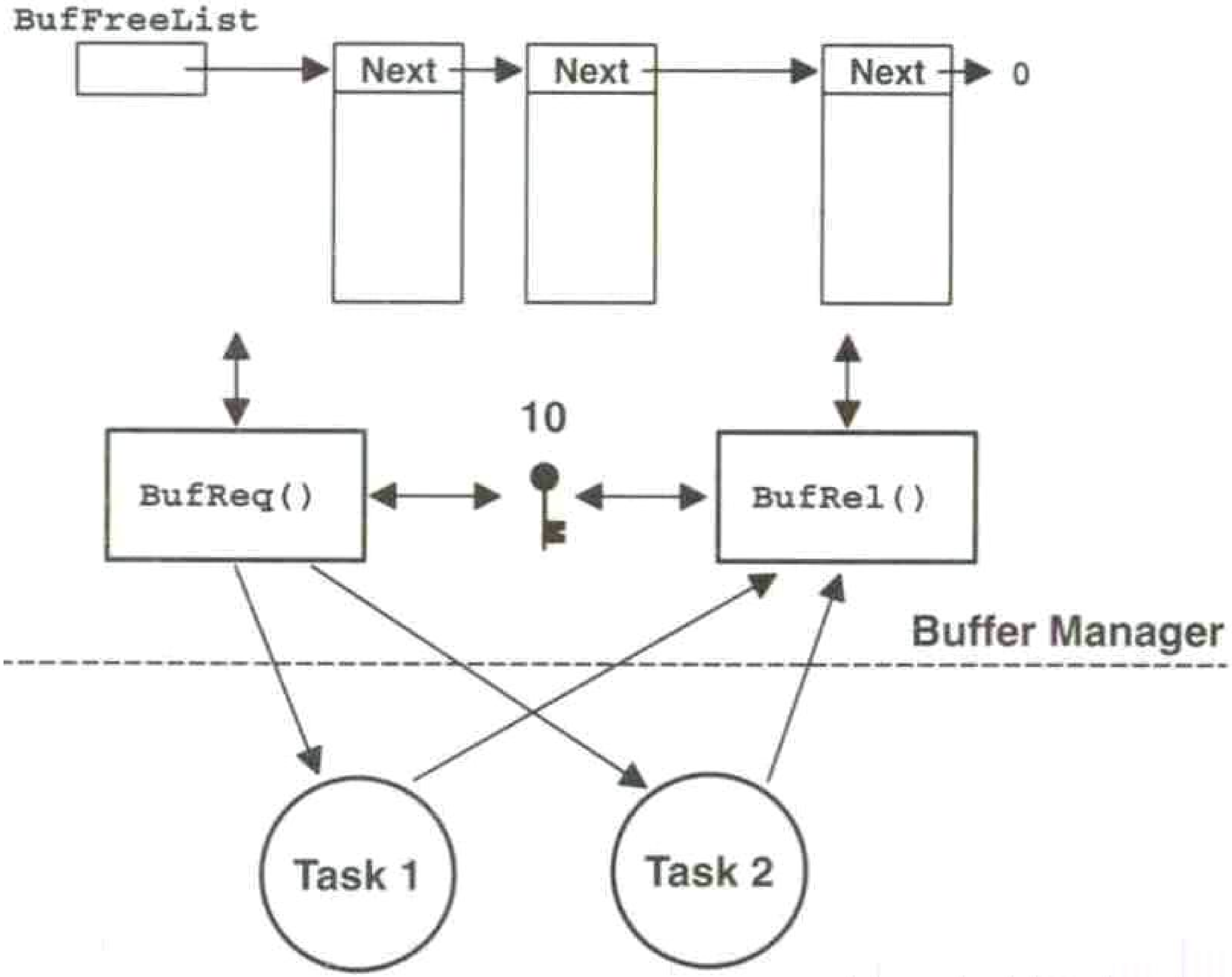
Encapsulating a Semaphore (cont.)

```
INT8U CommSendCmd(char *cmd, char *response,
    INT16U timeout)
{
    Acquire semaphore;
    Send command to device;
    Wait for response with timeout;
    if (timed out) {
        Release semaphore;
        return error code;
    } else {
        Release semaphore;
        return no error;
    }
}
```

Applications of Counting Semaphores

- A counting semaphore is used when a resource can be used by more than one task at the same time
- Example:
  - Managing a buffer pool of 10 buffers

Buffer Management



Buffer Management (cont.)

```
BUF *BufReq(void)
{
   BUF *ptr;

   Acquire a semaphore;
   Disable interrupts;
   ptr = BufFreeList;
   BufFreeList = ptr->BufNext;
   Enable interrupts;
   return (ptr);
}
```

Buffer Management (cont.)

```
void BufRel(BUF *ptr)
{
   Disable interrupts;
   ptr->BufNext = BufFreeList;
   BufFreeList = ptr;
   Enable interrupts;
   Release semaphore;
}
```

Buffer Management (cont.)

- Semaphores are often overused
- The use of semaphore to access simple shared variable is overkill in most situations
- For this simple access, disabling interrupts are more cost-effective
- However, if the variable is floating-point and CPU does not support floating-point in hardware, disabling interrupts should be avoided

Linux for Real Time Applications

- Scheduling
  - Priority Driven Approach
    - Optimize average case response time
  - Interactive Processes Given Highest Priority
    - Aim to reduce response times of processes
  - Real Time Processes
    - Processes with high priority
    - There was no notion of deadlines

Linux for Real Time Applications (cont.)

- Resource Allocation
  - There was no support for handling priority inversion
- Since version 2.6.18, priority inheritance available in both kernel and user space mutexes for preventing priority inversion

Interrupt Handling in Linux

- Interrupts were disabled in ISR/critical sections of the kernel
- There was no worst case bound on interrupt latency avaliable
  - eg: Disk Drivers might disable interrupt for few hundred milliseconds

Interrupt Handling in Linux (cont.)

- Linux was not suitable for Real Time Applications
  - Interrupts may be missed
- Since 2.6.18, Hard IRQs executed in kernel thread context
  - (where differing priority levels can be assigned)
  - allows developers to better insulate systems from external events

Other Problems with Linux

- Processes were not preemtible in Kernel Mode
  - System calls like fork take a lot of time
  - High priority thread might wait for a low priority thread to complete it's system call
- Processes are heavy weight
  - Context switch takes several hundred microseconds
- Linux 2.6.18 adds preemption points in kernel, with the goal of achieving microsecond-level latency;
  - all but "kernel-critical" portions of kernel code become preemptible involuntarily at any time

Why Linux?

- Coexistence of Real Time Applications with non Real Time Ones
  - Example http server
- Device Driver Base
- Stability

Why Linux? (cont.)

- Several real-time features are now integrated into the main distribution of Linux
  - Improved POSIX compliancy
    - including Linux's first implementation of message queues and of priority inheritance,
    - as well as an improved implementation of signals less apt to disregard multiple inputs
  - Hard IRQs executed in thread context

Why Linux? (cont.)

- Three levels of real-time preemptibility in kernel, configurable at compile time (throughput or real-time predictability):
  - Voluntary preemption
  - Preemptible kernel (ms level)
  - Full real-time preemption (microsecond level)
- Priority inheritance available in both kernel and user space mutexes

Why Linux? (cont.)

- High-resolution timers
  - Added in 2.6.21, this allows the kernel to actually use the high-resolution timers built into most processors, enabling,
  - For example, POSIX timers and nano_sleep calls to be "as accurate as hardware allows,"
  - The kernel's entire time management gets to the level of microseconds

Why Linux? (cont.)

- Various kernel config options for monitoring real-time behavior of kernel:
  - CONFIG_LATENCY_TRACE
    - Track the full trace of maximum latency
  - CONFIG_WAKEUP_TIMING
    - Tracking of the maximum recorded time between
      - » waking up for a high-priority task, and
      - » executing on the CPU, shown in microseconds