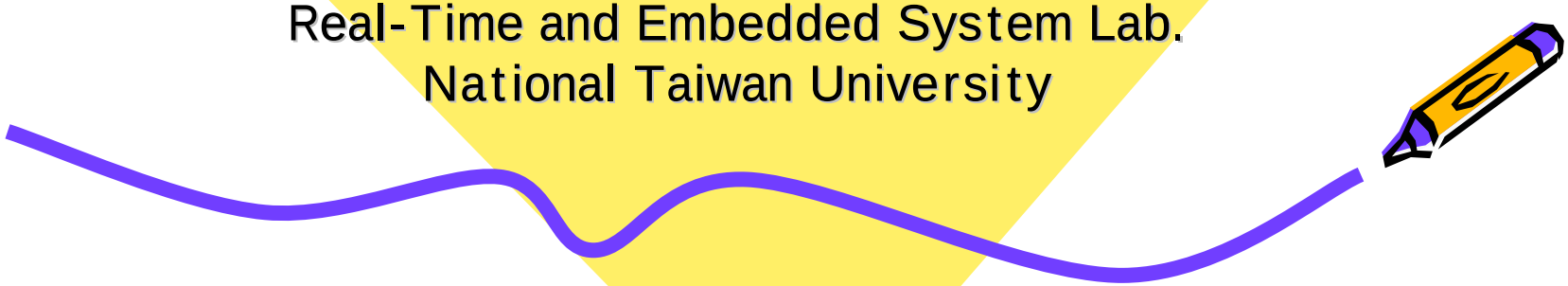
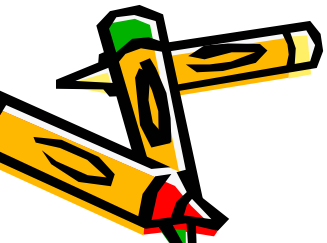
Chapter-2 Real-Time System Concepts

Dr. Li-Pin Chang
Real-Time and Embedded System Lab.
National Taiwan University



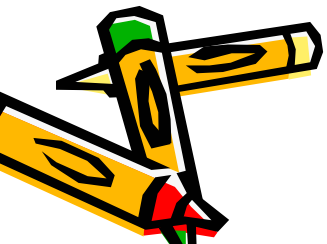
Objectives

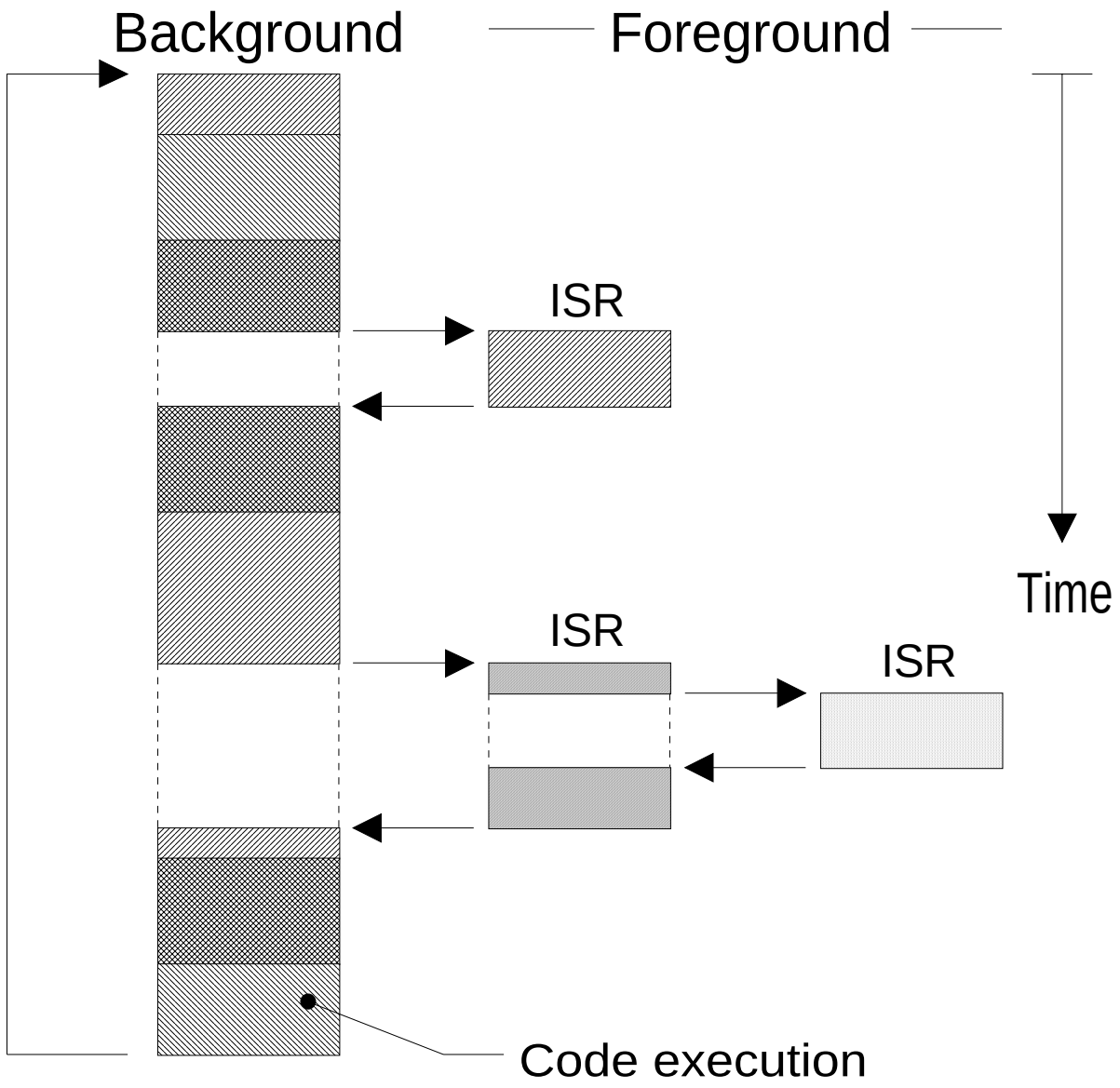
- To know essential topics on the design of:
  - Embedded operating systems
  - Real-time systems



Foreground/Background Systems

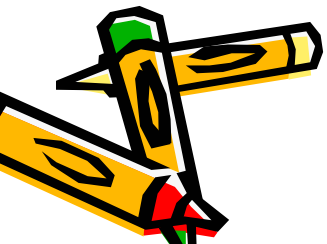
- The system consists of an infinite loop which calls modules to perform jobs. (a super loop)
  - The background (or, task) level.
- Critical events are handled in ISR's.
  - The foreground (or, interrupt) level.
- An event-based approach. The system could be in an idle state if there is no event.
- Similar to Cyclic Executive.




Background
Foreground
ISR
ISR
ISR
Time
Code execution

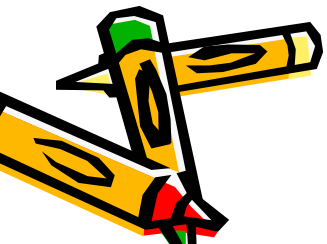
Critical Sections

- A portion of code must be treated indivisibly.
  - To protect shared resources from corrupting due to race conditions.
  - Could be implemented by using interrupt enable/disable, semaphores, events, mailboxes, etc.



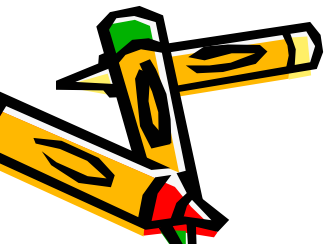
Resources

- An entity used by a task.
  - Memory objects
    - Such as tables, global variables …
  - I/O devices.
    - Such as disks, communication transceivers.
- A task must gain exclusive access to a shared resource to prevent data (or I/O status) from being corrupted.
  - Mutual exclusion.



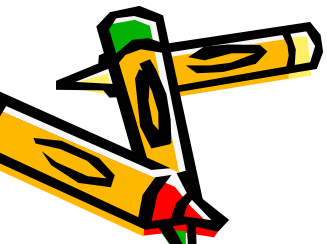
Multitasking

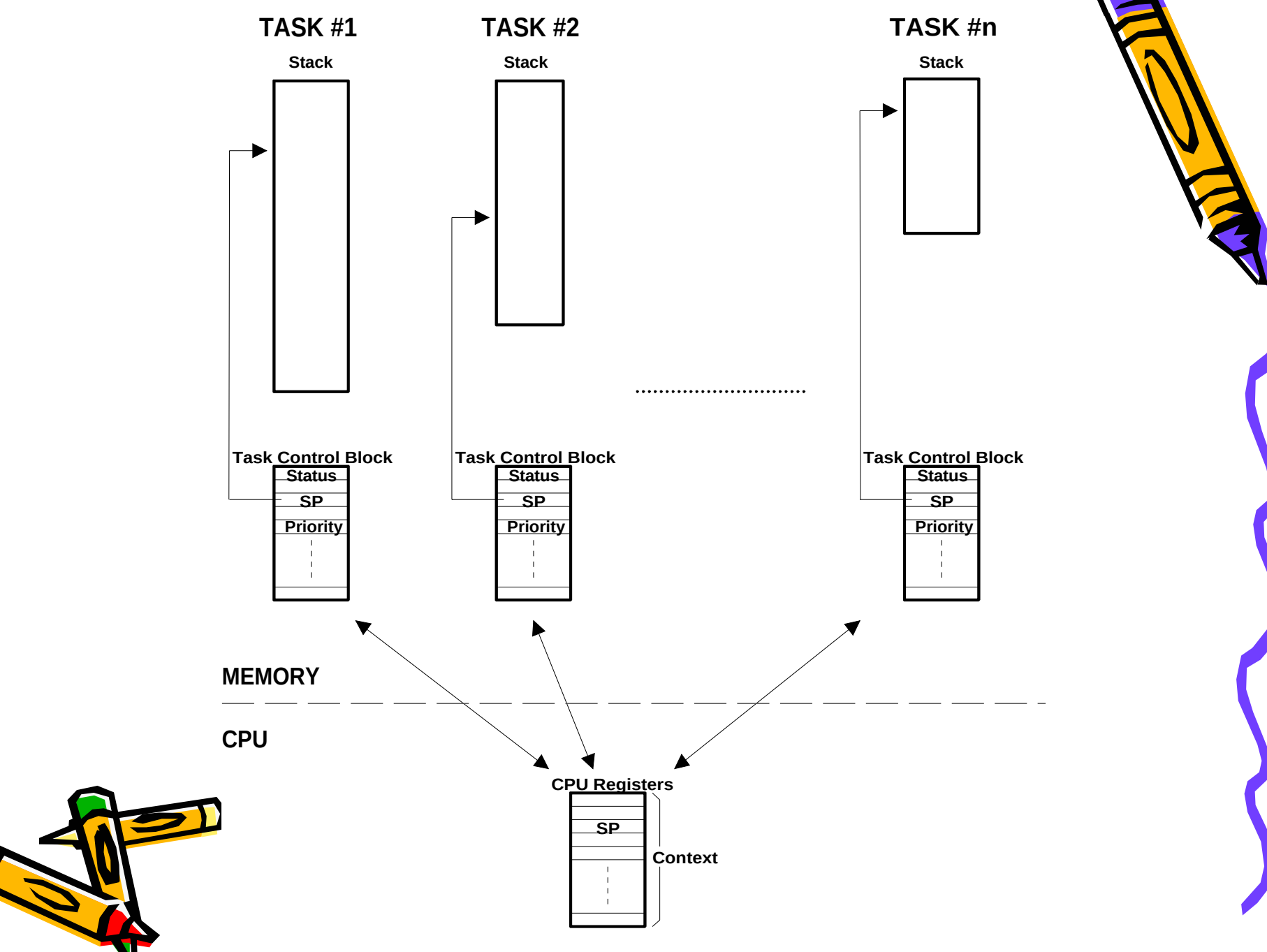
- The scheduler of an operating system switch the attention of CPU among several tasks.
  - Tasks logically share the computing power of a CPU.
  - Tasks logically execute concurrently.
  - How much CPU share could be obtained by each task depends on the scheduling policy adopted.



Task

- Also called a thread or a process in practice. It is considered as an active/executable entity in a system.
- From the perspective of OS, a task is of a priority, a set of registers, its own stack area, and some housekeeping data.
- From the perspective of scheduler, a task is of a series of consecutive jobs with regular ready time (for periodic tasks).

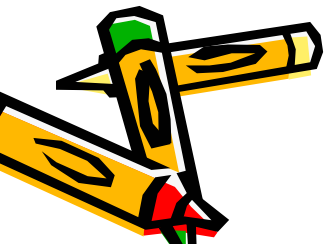


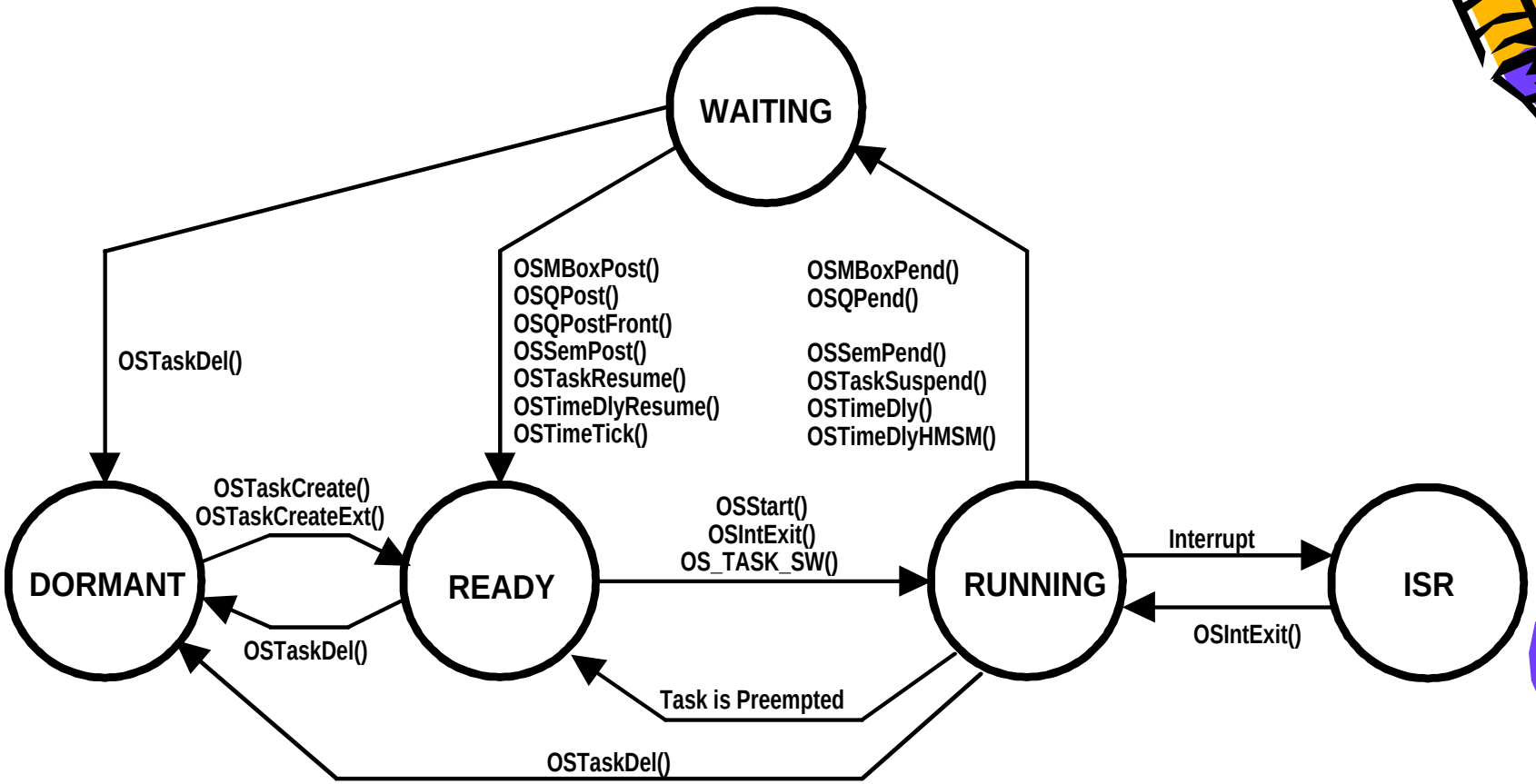

TASK #1
TASK #2
TASK #n
Stack
Stack
Stack
Task Control Block
Status
SP
Priority
Task Control Block
Status
SP
Priority
Task Control Block
Status
SP
Priority
MEMORY
CPU
CPU Registers
SP
Context

Task

- A task is basically an infinite loop for a real-time system.

- There are 5 states under uC/OS-2:
  - Dormant, ready, running, waiting, interrputed.



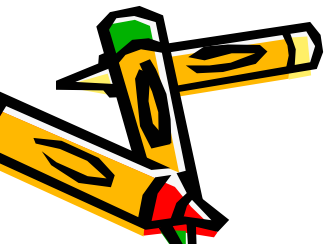

WAITING
OSTaskDel()
OSMBoxPost()
OSQPost()
OSQPostFront()
OSSemPost()
OSTaskResume()
OSTimeDlyResume()
OSTimeTick()
OSMBoxPend()
OSQPend()
OSSemPend()
OSTaskSuspend()
OSTimeDly()
OSTimeDlyHMSM()
OSTaskCreate()
OSTaskCreateExt()
DORMANT
READY
OSStart()
OSIntExit()
OS_TASK_SW()
RUNNING
Interrupt
ISR
OSIntExit()
OSTaskDel()
Task is Preempted
OSTaskDel()

Context Switch

- It occurs when the scheduler decides to run a different task.
- The scheduler must save the context of the current task and then load the context of the task-to-run.
  - The context is of a priority, the contents of the registers, the pointers to its stack, and the related housekeeping data.
- Context-switches impose overheads on the task executions.
  - A practicable scheduler must not cause intensive context switches. Because modern CPU's have deep pipelines and many registers.
- For a real-time operating system, we must know how much time it takes to perform a context switch.
  - The overheads of context switch are accounted into high priority tasks. (blocking time, context switch time...)

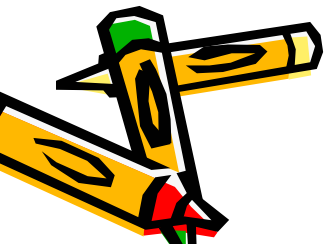
Kernels

- The kernel is a part of a multitasking system, it is responsible for:
  - The management of tasks.
  - Inter-task communication.
- The kernel imposes additional overheads to task execution.
  - Kernel services take time.
    - Semaphores, message queues, mailboxes, timing controls, and etc…
  - ROM and RAM space are needed.
- Single-chip microcontrollers generally are not suitable to run a real-time kernel because they mostly have little RAM (e.g., 8KB of RAM).



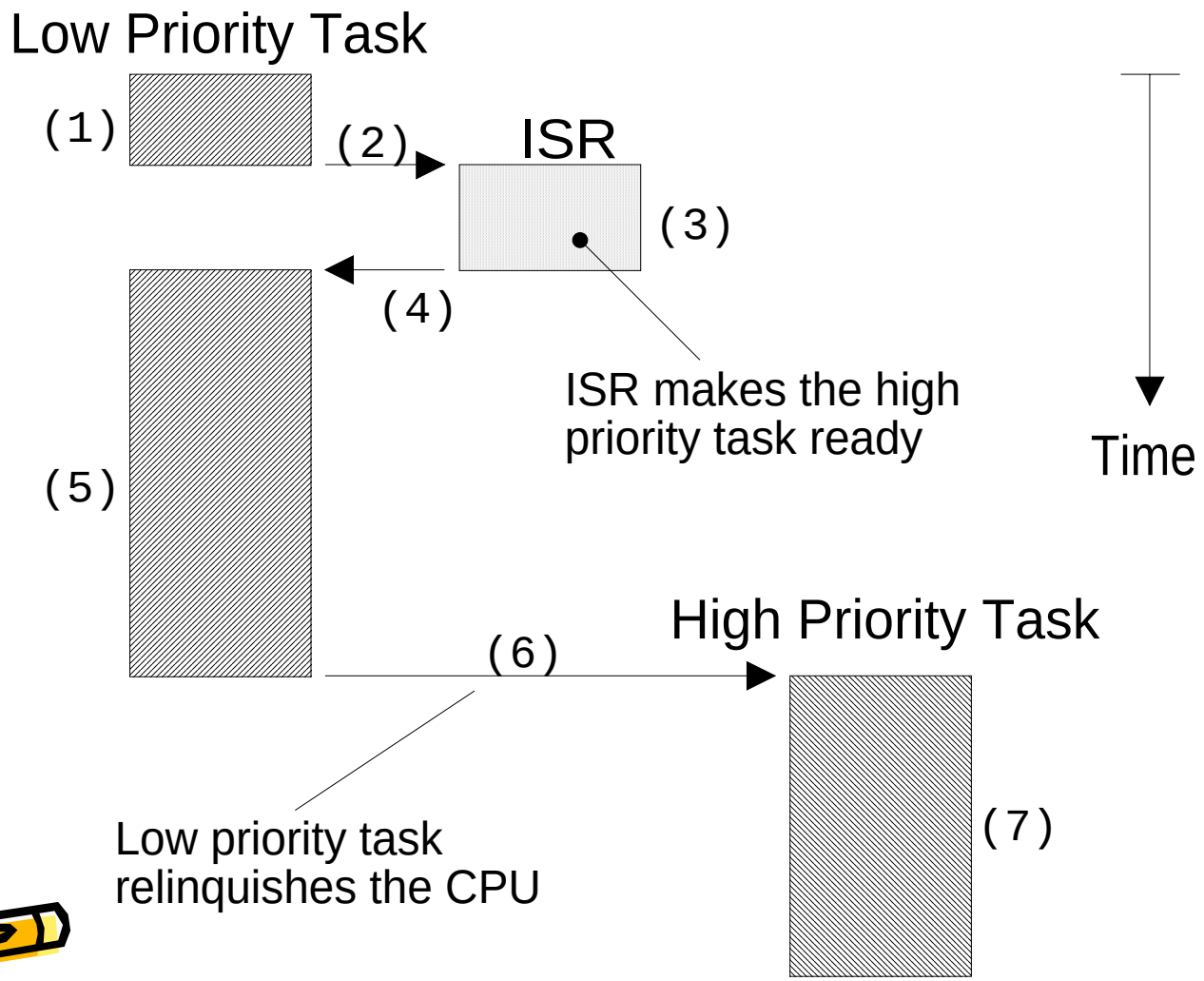
Schedulers

- A scheduler is a part of the kernel. It is responsible for determining which task should run next.
  - Preemptible or non-preemptible.
- Most real-time systems are priority based.
  - Priorities are application-specific.
- The scheduler always gives the CPU to the highest-priority task which is ready to run.



Non-Preemptive Kernels

- Context switches occur only when tasks explicitly give up control of the CPU.
  - High-priority tasks gain control of the CPU.
  - This procedure must be done frequently to improve the responsiveness.
- Events are still handled in ISR's.
  - ISR's always return to the interrupted task.
- Most tasks are race-condition free.
  - Non-reentrant codes can be used without protections.
  - In some cases, synchronizations are still needed.
- Pros: simple, robust.
- Cons: Not very responsive. There might be lengthy priority inversions.


Low Priority Task
(1)
(2)
ISR
(3)
(4)
ISR makes the high
priority task ready
Time
(5)
High Priority Task
(6)
Low priority task
relinquishes the CPU
(7)

(1) A task is executing but gets interrupted.

(2) If interrupts are enabled, the CPU vectors (i.e. jumps) to the ISR.

(3) The ISR handles the event and makes a higher priority task ready-to-run.

(4) Upon completion of the ISR, a *Return From Interrupt* instruction is executed and the CPU returns to the interrupted task.

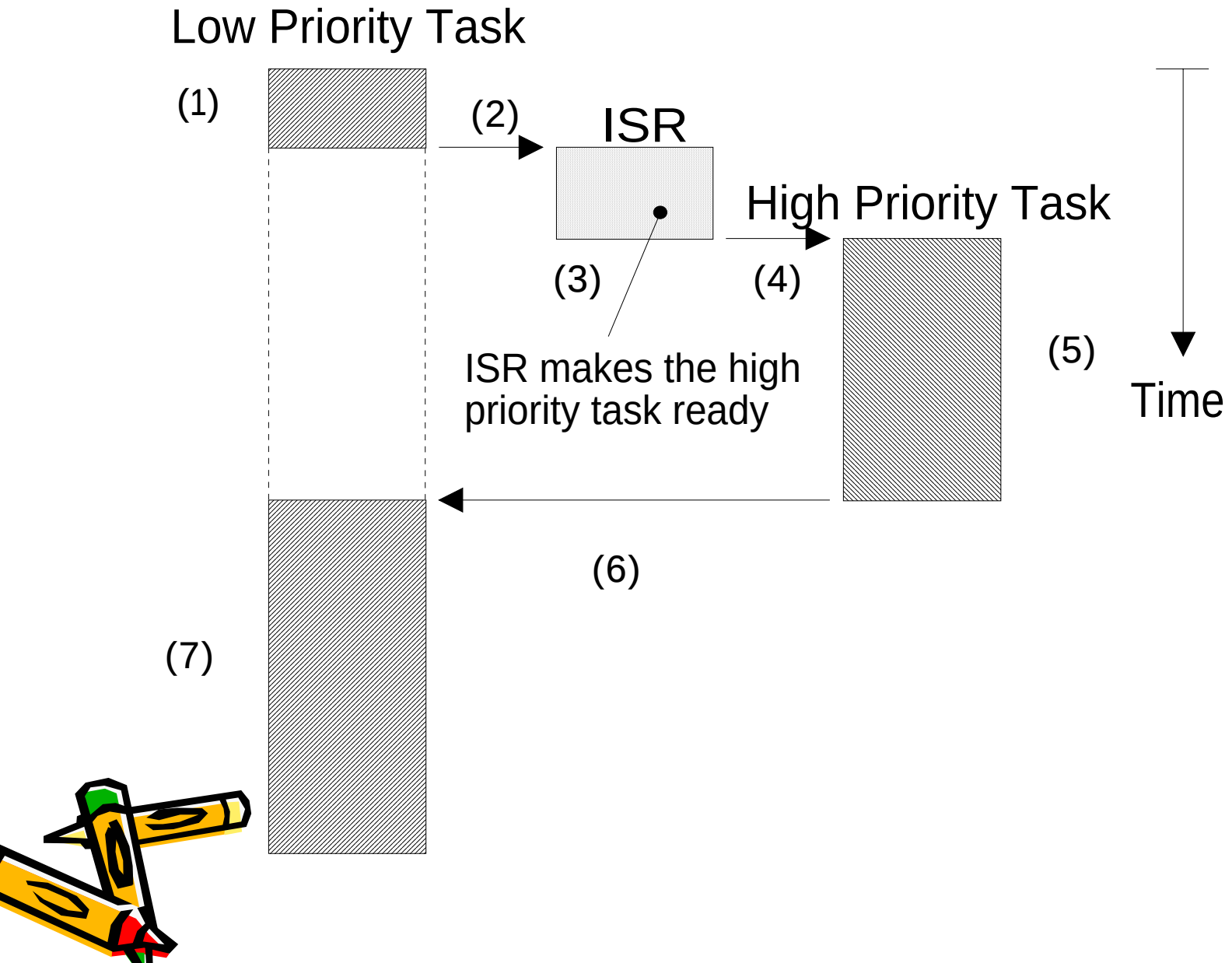
(5) The task code resumes at the instruction following the interrupted instruction.

(6) When the task code completes, it calls a service provided by the kernel to relinquish the CPU to another task.

(7) The new higher priority task then executes to handle the event signaled by the ISR.

Preemptive Kernels

- The benefit of a preemptive kernel is the system is more responsive.
  - uC/OS-2 (and most RTOS) is preemptive.
  - The execution of a task is deterministic.
    - A high-priority task gain control of the CPU instantly when it is ready (if no resource-locking is done).
- ISR might not return to the interrupted task.
  - It might return a high-priority task which is ready.
- Concurrency among tasks exists. As a result, synchronization mechanisms (semaphores…) must be adopted to prevent from corrupting shared resources.
  - Preemptions, blocking, priority inversions.


Low Priority Task
(1)
(2)
ISR
High Priority Task
(3)
(4)
(5)
Time
ISR makes the high priority task ready
(6)
(7)

(1) A task is executing but interrupted.

(2) If interrupts are enabled, the CPU vectors (jumps) to the ISR.

(3) The ISR handles the event and makes a higher priority task ready to run. Upon completion of the ISR, a service provided by the kernel is invoked. (i.e., a function that the kernel provides is called).

(4)

(5) This function knows that a more important task has been made ready to run, and thus, instead of returning to the interrupted task, the kernel performs a context switch and executes the code of the more important task. When the more important task is done, another function that the kernel provides is called to put the task to sleep waiting for the event (i.e., the ISR) to occur.

(6)

(7) The kernel then sees that a lower priority task needs to execute, and another context switch is done to resume execution of the interrupted task.

Reentrant Functions

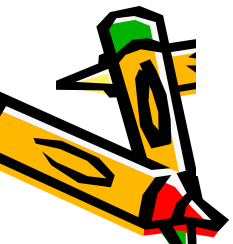
- Reentrant functions can be invoked simultaneously without corrupting any data.
  - Reentrant functions use either local variables (on stacks) or synchronization mechanisms (such as semaphores).

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

Non-Reentrant Functions

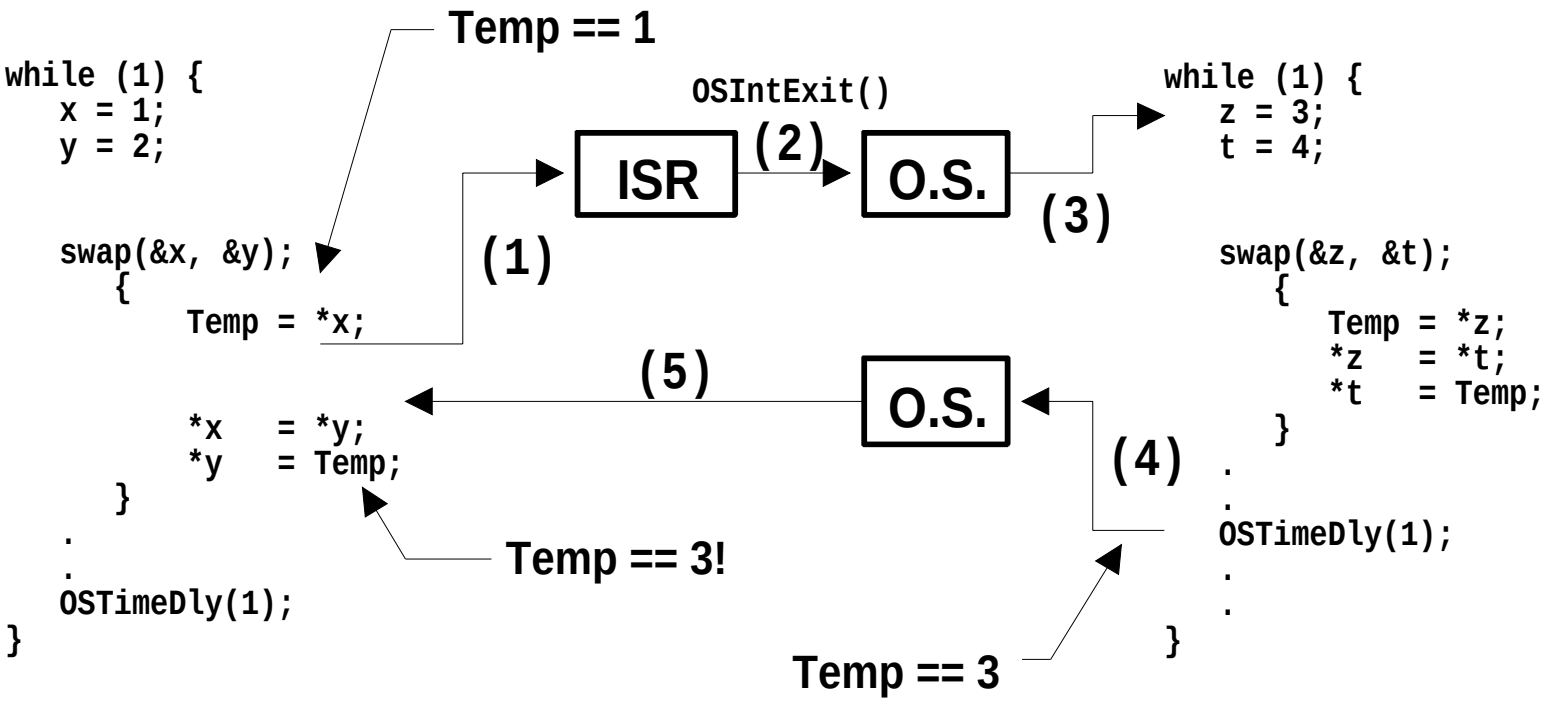
- Non-Reentrant functions might corrupt shared resources under race conditions.

```
int Temp;

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



LOW PRIORITY TASK

HIGH PRIORITY TASK



(1) When swap() is interrupted, TEMP contains 1.

(2)

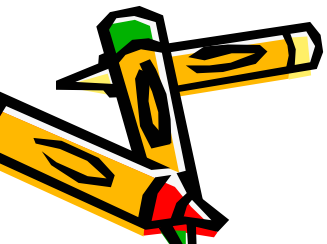
(3) The ISR makes the higher priority task ready to run, so at the completion of the ISR, the kernel is invoked to switch to this task. The high priority task sets TEMP to 3 and swaps the contents of its variables correctly. (i.e., z=4 and t=3).

(4) The high priority task eventually relinquishes control to the low priority task be calling a kernel service to delay itself for one clock tick.

(5) The lower priority task is thus resumed. Note that at this point, TEMP is still set to 3! When the low priority task resumes execution, the task sets y to 3 instead of 1.

Non-Reentrant Functions

- There are several ways to make the code reentrant:
  - Declare TEMP as a local variable.
  - Disable interrupts and then enable interrupts.
  - Use a semaphore.



Round-Robin Scheduling

- Adopted when no “priority” is adopted or tasks have the same priority.
  - Most traditional operating systems utilize RR scheduling.
- The scheduler checks if a context switch should be made every quantum.
  - A quantum is a pre-determined amount of time.
- Context switch occurs when:
  - The current task has no work to do.
  - The current task completes.
  - The quantum for the current task is exhausted.
- Most real-time operating systems require every task has an unique priority. As a result, RR scheduling is not adopted by most RTOS.

Priorities

- Priorities reflect the criticality (importance) of tasks.
  - The higher the priority, the lower the number is.
- Priorities are assigned by programmers (for most real-time schedulers).
- Priorities of tasks do not change under a static priority system.
  - For example, under the RM scheduler.
- Priorities of tasks might dynamically reflect certain run-time criteria (and change) under a dynamic priority system.
  - For example, under the EDF scheduler.
- Due to the adoption of resource-synchronization protocol, priorities might change even under a RM scheduler.

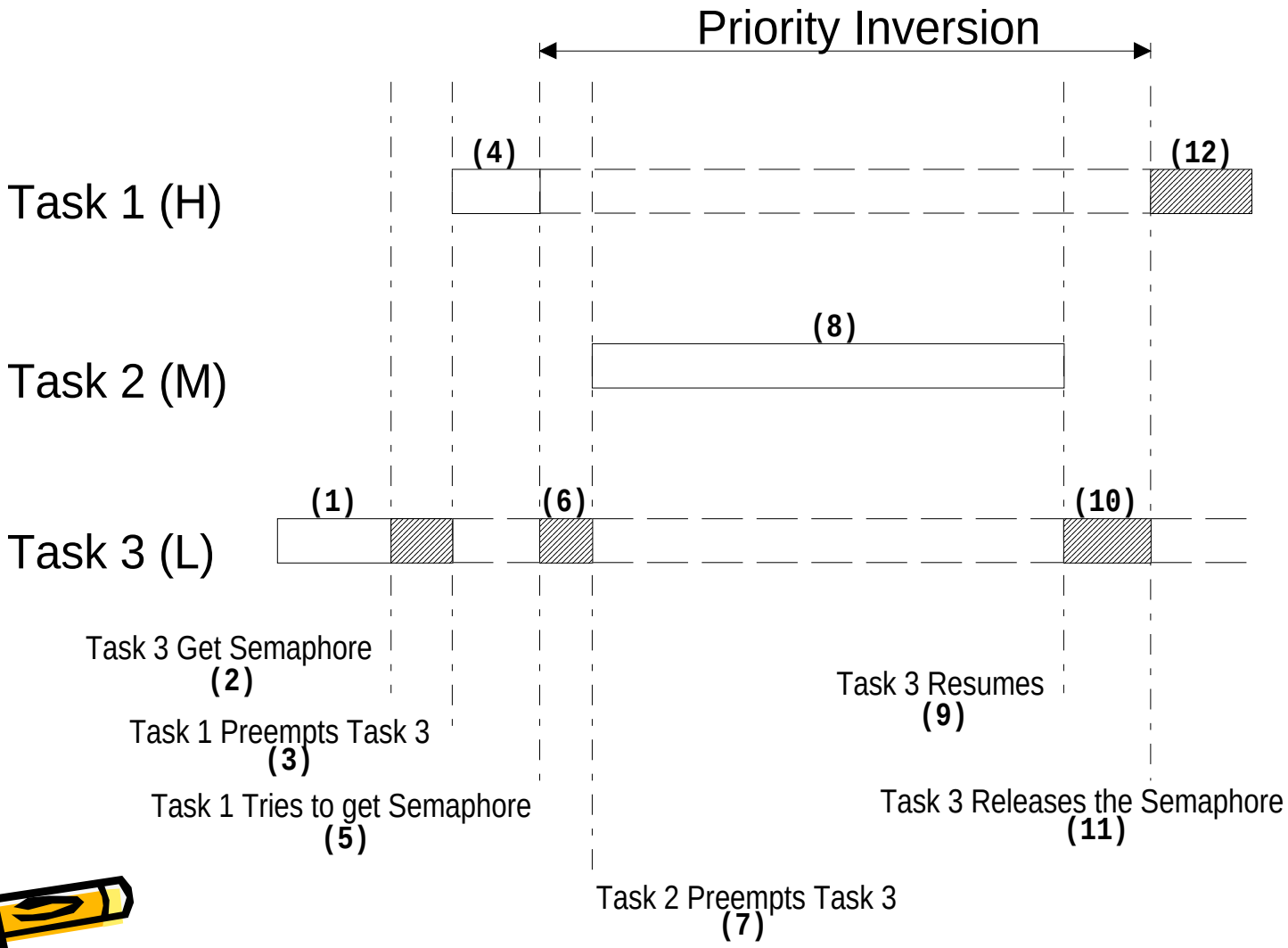
Priority Inversions

- A high-priority task is blocked if:
  - It is currently running or ready-to-run.
  - It can not gain control of the CPU because of a low-priority task.
- Such a phenomenon is also called a priority inversion.
- Low-priority tasks won't be "blocked" by high-priority tasks.
- It is essential to properly control the number and interval of priority inversions.
  - Priority inversion must be accounted into the schedulability test.

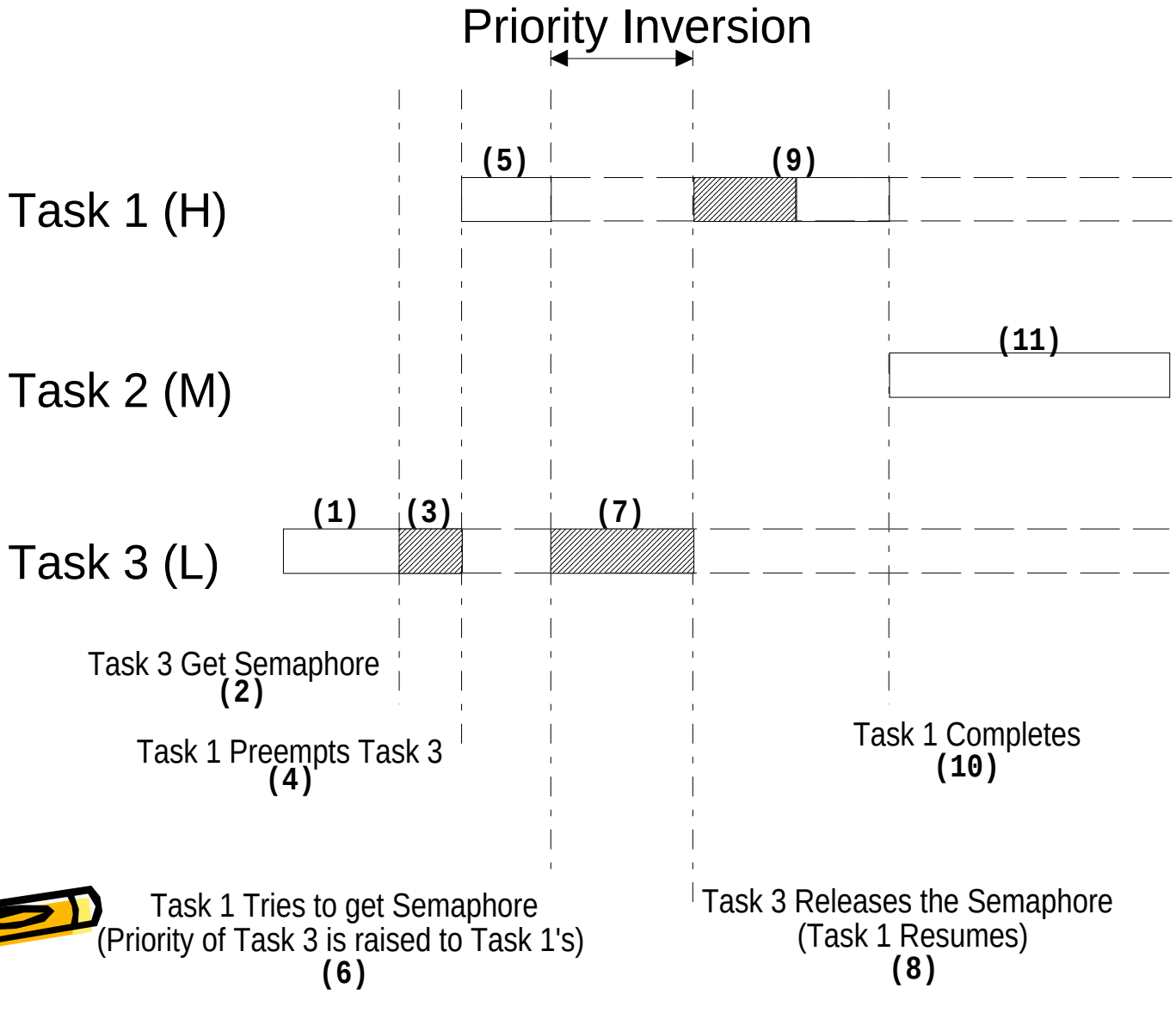
$$\forall j, \sum_{i \neq j} \frac{c_i}{p_i} + \frac{c_j + b_j}{p_j} \leq U(n)$$

Priority Inversions

- Three major problems caused by resources-sharing must be properly handled under real-time systems.
  - Unbounded priority inversions.
  - Chain (and transitive) blocking.
  - Deadlocks.
- Priority inheritance protocol (PIP) can avoid unbounded priority inversions.
  - Priorities of tasks might change to reflect that high-priority tasks are blocked.
- Priority ceiling protocol (PCP) is a super-set of PIP, and PCP can avoid chain blocking and deadlocks.
  - The blocking time is deterministic under the adoption of PCP.
  - However, it is extremely hard to implement PCP in a RT scheduler.


Priority Inversion
Task 1 (H)
Task 2 (M)
Task 3 (L)
(4)
(12)
(8)
(1)
(6)
(10)
Task 3 Get Semaphore
(2)
Task 1 Preempts Task 3
(3)
Task 1 Tries to get Semaphore
(5)
Task 2 Preempts Task 3
(7)
Task 3 Resumes
(9)
Task 3 Releases the Semaphore
(11)

Priority Inversion



Assigning Task Priorities

- Priorities of tasks are assigned by programmers.
- For fixed-priority systems, RMS is optimal.
  - On-line admission control exists for RMS:
- If deadline <> period, DMS is optimal.
  - We have little knowledge on any efficient admission control for DMS.
- Pseudo-polynomial schedulability tests for RMS/DMS exist.
  - However, they are too time-consuming for on-line implementations.
- Since RTOS's rely on programmers to set priorities, as a result, admission control are mostly done by programmers.

Assigning Task Priorities

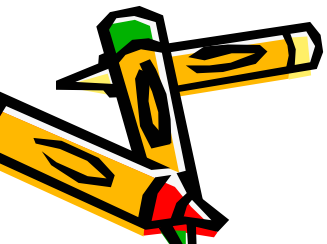
- An efficient schedulability test for RMS only takes O(n) to complete for n tasks.
  - O(1) for on-line admission control.
- As a rule of thumb, a real-time system should bound the total utilization under 70%.

$$\sum \frac{c_i}{p_i} \leq U(n)$$

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

Mutual Exclusion

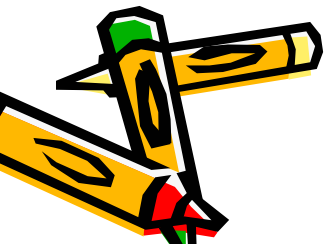
- Mutual exclusion must be adopted to protect shared resources.
  - Global variables, linked lists, pointers, buffers, and ring buffers.
  - I/O devices.
- When a task is using a resource, the other tasks which are also interested in the resource must not be scheduled to run.
- Common techniques used are disable/enable interrupts, performing a test-and-set instruction, disabling scheduling, and using synchronization mechanisms (such as semaphores).



Mutual Exclusion

- Disabling/enabling interrupts:
  - OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL()
  - All events are masked since interrupts are disabled.
  - Tasks which do not affect the resources-to-protect are also postponed.
  - Must not disable interrupt before calling system services.

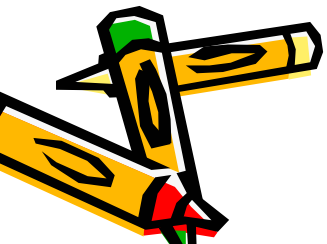
```
void Function (void)
{
    OS_ENTER_CRITICAL();
    .
    .     /* You can access shared data in here */
    .
    OS_EXIT_CRITICAL();
}
```



Mutual Exclusion

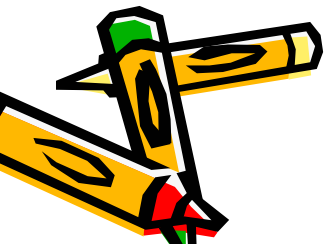
- The test-and-set instruction:
  - An atomic operation (a CPU instruction) to lock a guarding variable.
  - It is equivalent to the SWAP instruction.
  - Starvation might happen.

```
int lock=1;

swap(&flag,&lock);  /* corresponds to SWAP instruction */

if(flag == 1)
   Locking is failed.
   flag remains 1.
else
   Locking is success.
   flag is set as 1 after the swapping.
   ... critical section ...
```



Mutual Exclusion

- Disabling/Enabling Scheduling:
  - No preemptions could happen while the scheduler is disabled.
  - However, interrupts still happen.
    - ISR's could still corrupt shared data.
    - Once an ISR is done, the interrupted task is always resumed even there are high priority tasks ready.
  - Rescheduling might happen right after the scheduler is re-enabled.
  - Higher overheads and weaker effects than enabling/disabling interrupts.

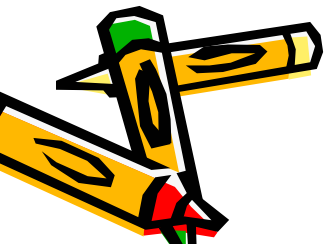


```
void Function (void)
{
    OSSchedLock();
    .     /* You can access shared data
    .        in here (interrupts are recognized) */
    OSSchedUnlock();
}
```

Mutual Exclusion

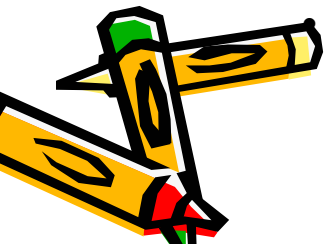
- Semaphores:
  - Provided by the kernel.
  - Semaphores are used to:
    - Control access to a shared resource.
    - Signal the occurrence of an event.
    - Allow tasks to synchronize their activities.
  - Higher priority tasks which does not interested in the protected resources can still be scheduled to run.

```
OS_EVENT *SharedDataSem;
void Function (void)
{
    INT8U err;
    OSSemPend(SharedDataSem, 0, &err);
    .     /* You can access shared data
    .        in here (interrupts are recognized) */
    OSSemPost(SharedDataSem);
}
```



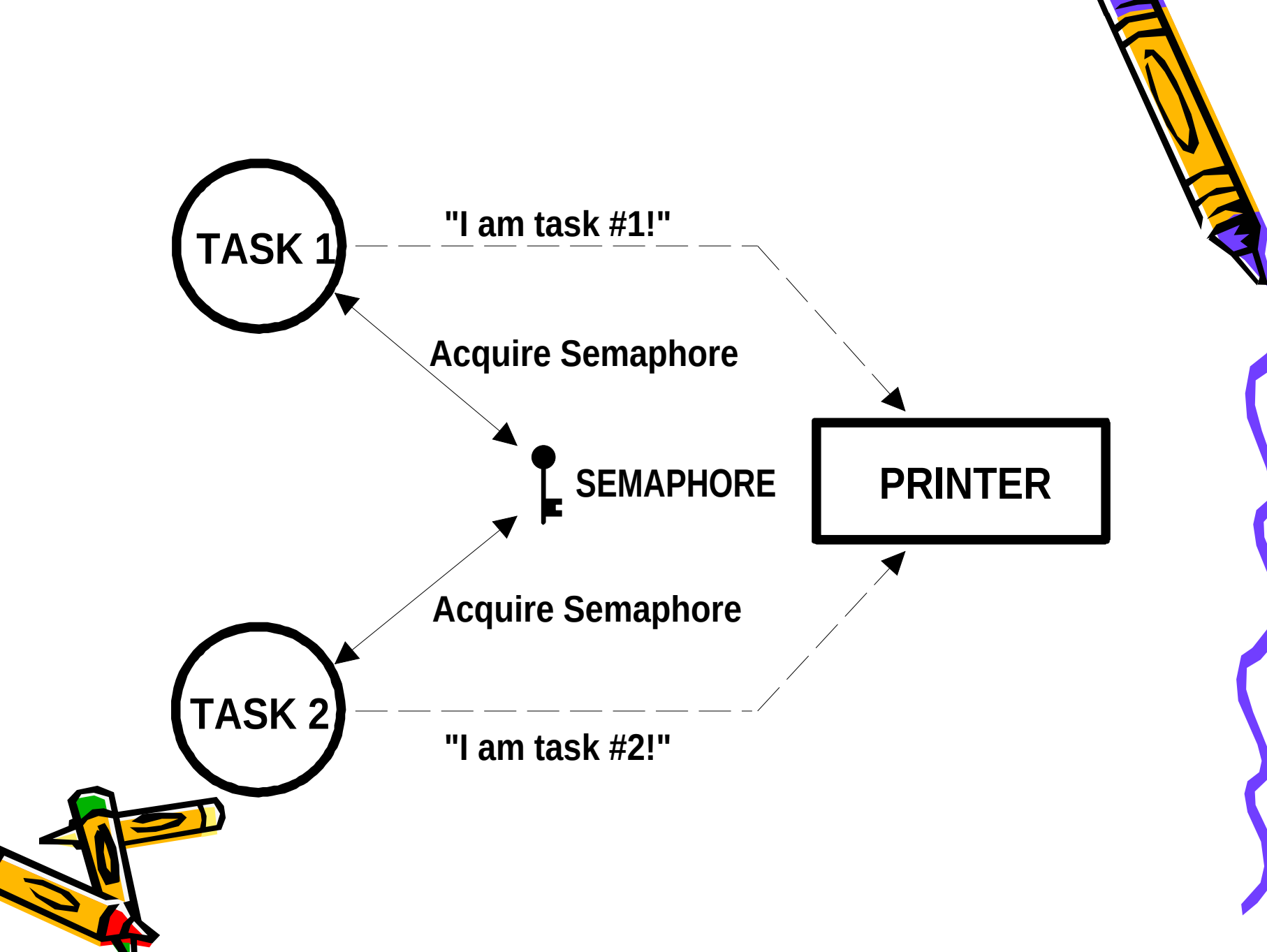
Mutual Exclusion

- Semaphores:
  - OSSemPend() / OSSemPost()
  - A semaphore consists of a wait list and an integer counter.
  - OSSemPend:
    - Counter--;
    - If the value of the semaphore <0, the task is blocked and moved to the wait list immediately.
    - A time-out value can be specified .
  - OSSemPost:
    - Counter++;
    - If the value of the semaphore >= 0, a task in the wait list is removed from the wait list.
      - Reschedule if needed.



Mutual Exclusion

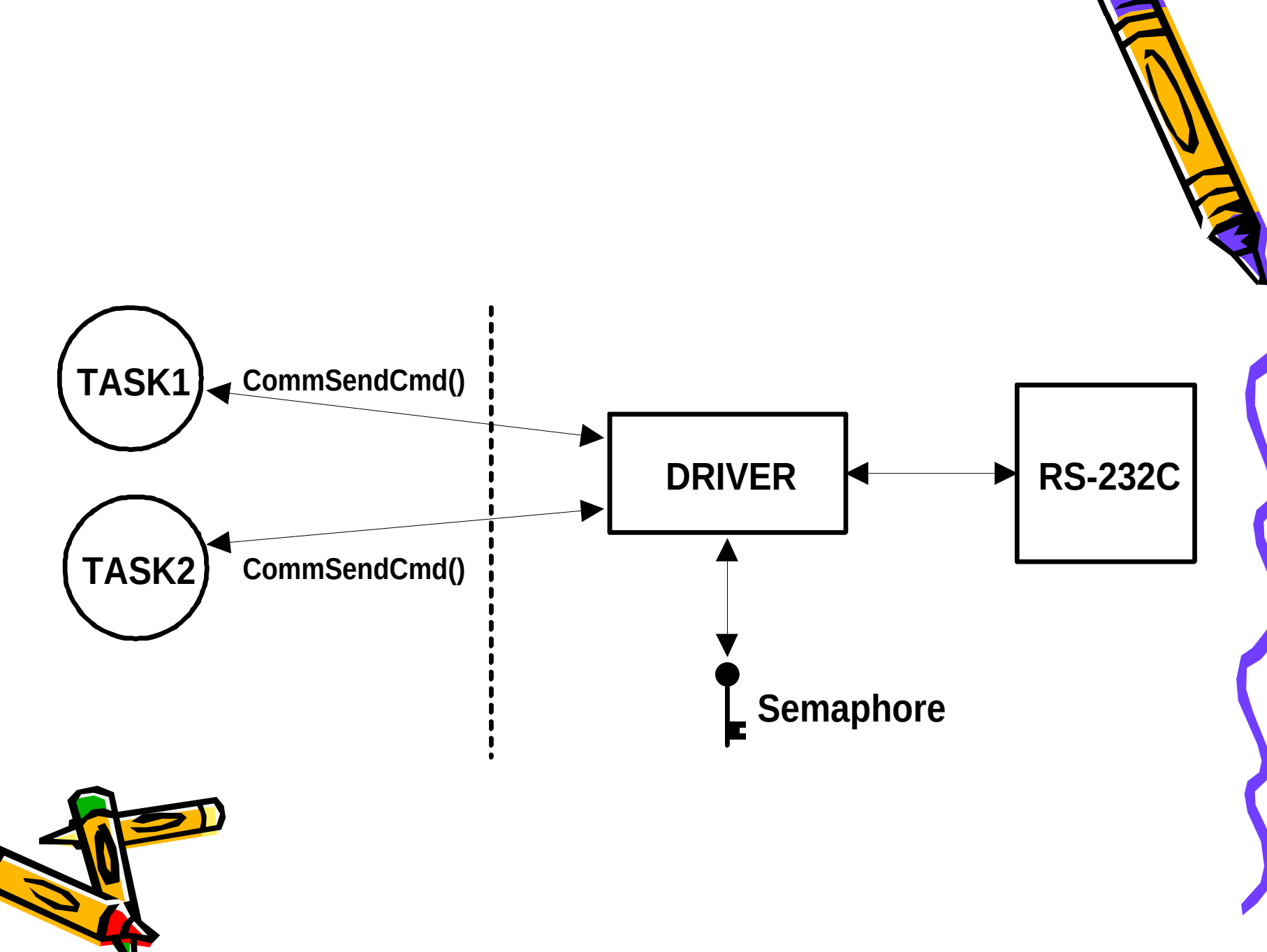
- Semaphores:
  - **Three** kinds of semaphores:
    - Counting semaphore (init >1)
    - Binary semaphore (init = 1)
    - Rendezvous semaphore (init = 0)
  - On event posting, a waiting task is released from the waiting queue.
    - The highest-priority task.
    - FIFO (not supported by uC/OS-2)
  - Interrupts and scheduling are still enabled under the use of semaphores.


TASK 1
"I am task #1!"
Acquire Semaphore
SEMAPHORE
PRINTER
Acquire Semaphore
TASK 2
"I am task #2!"

Mutual Exclusion

- Example: use a semaphore to protect a RS-232 communication port→

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


TASK1
CommSendCmd()
TASK2
CommSendCmd()
DRIVER
RS-232C
Semaphore

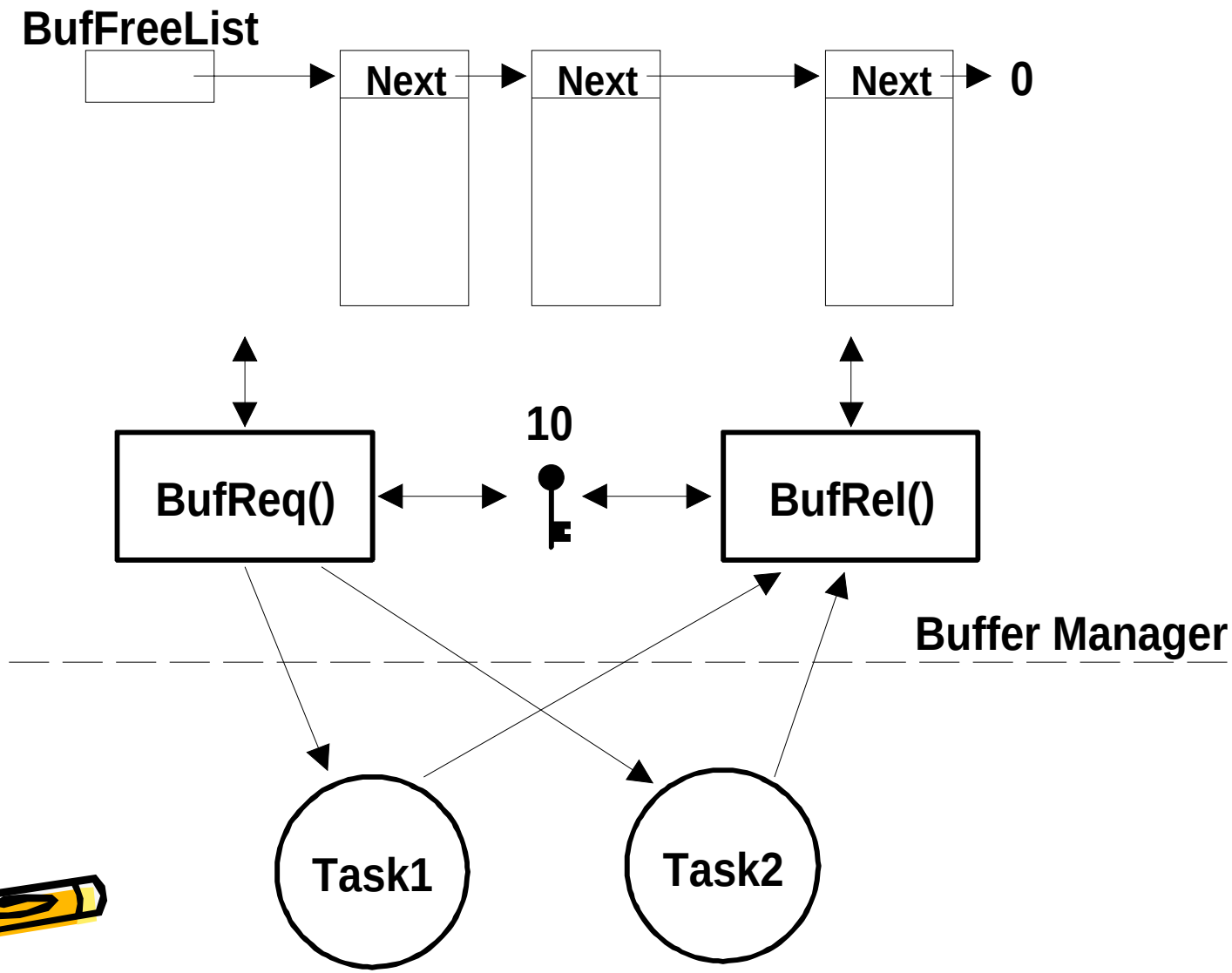
Mutual Exclusion

- Using a counting semaphore to synchronize the use of a buffer.

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

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

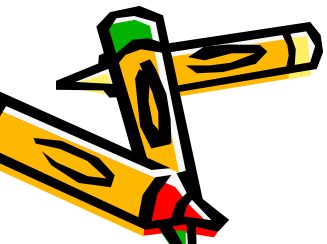
```
void BufRel(BUF *ptr)
{
  Disable interrupts;
  ptr->BufNext = BufFreeList;
  BufFreeList  = ptr;
  Enable interrupts;
  Release semaphore;
}
```

**Red statements can be replaced by a binary semaphore. Here we disable/enable interrupts for the consideration of efficientcy.


BufFreeList
Next
Next
Next
0
10
BufReq()
BufRel()
Buffer Manager
Task1
Task2

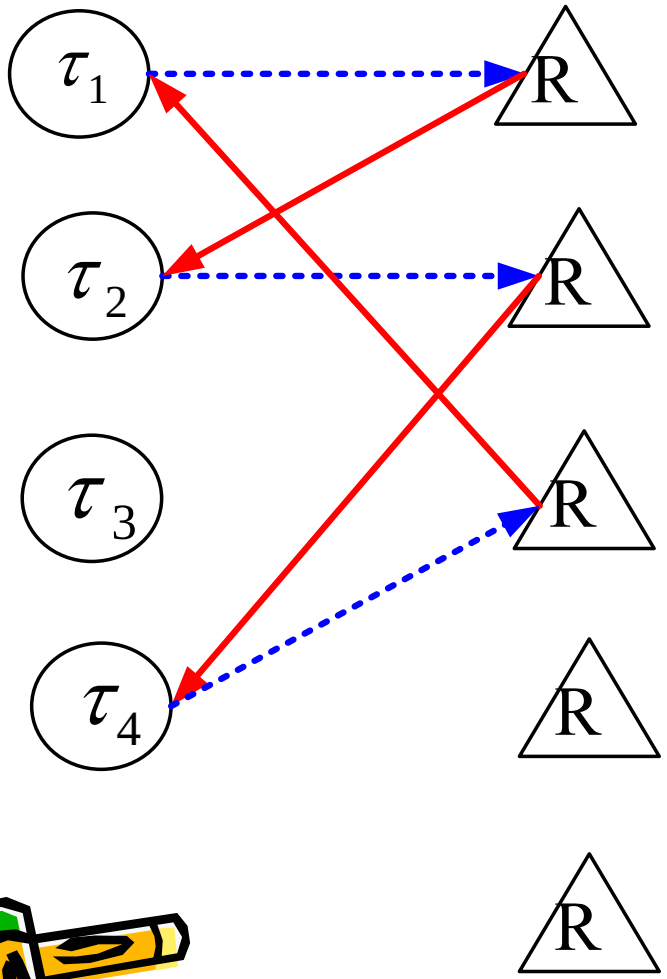
Mutual Exclusion

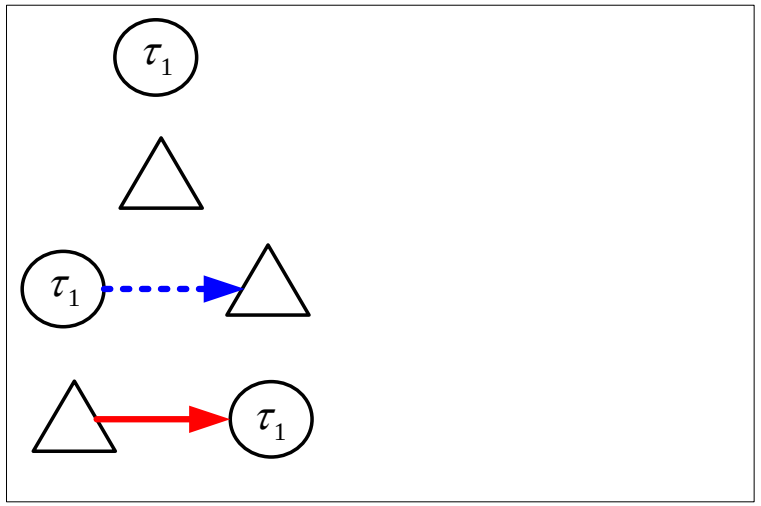
- Summary:
  - Semaphores are versatile while concurrency is still guaranteed
    - However, the overheads are relatively high.
  - Interrupt enabling/disabling are suitable for very short critical sections since the overheads are very low.
    - However, it kills the parallelism.



Deadlocks

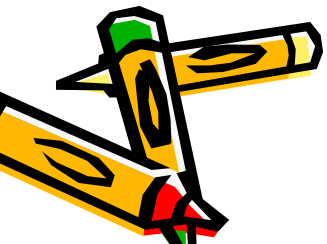
- Tasks circularly wait for certain resources which are already locked by another tasks.
  - No task could finish executing under such a circumstance.
- Deadlocks are intolerable in real-time systems since a bunch of tasks will miss their deadlines.
- Deadlocks in static systems can be detected and resolved in advance.
- Deadlocks are not easy to detect and resolve in a on-line fashion.
  - A brute-force way to avoid deadlocks is to set a timeout when acquiring a semaphore.
  - The elegant way is to adopt resource synchronization protocols.
    - Priority Ceiling Protocol (PCP), Stack Resource Policy (SRP)

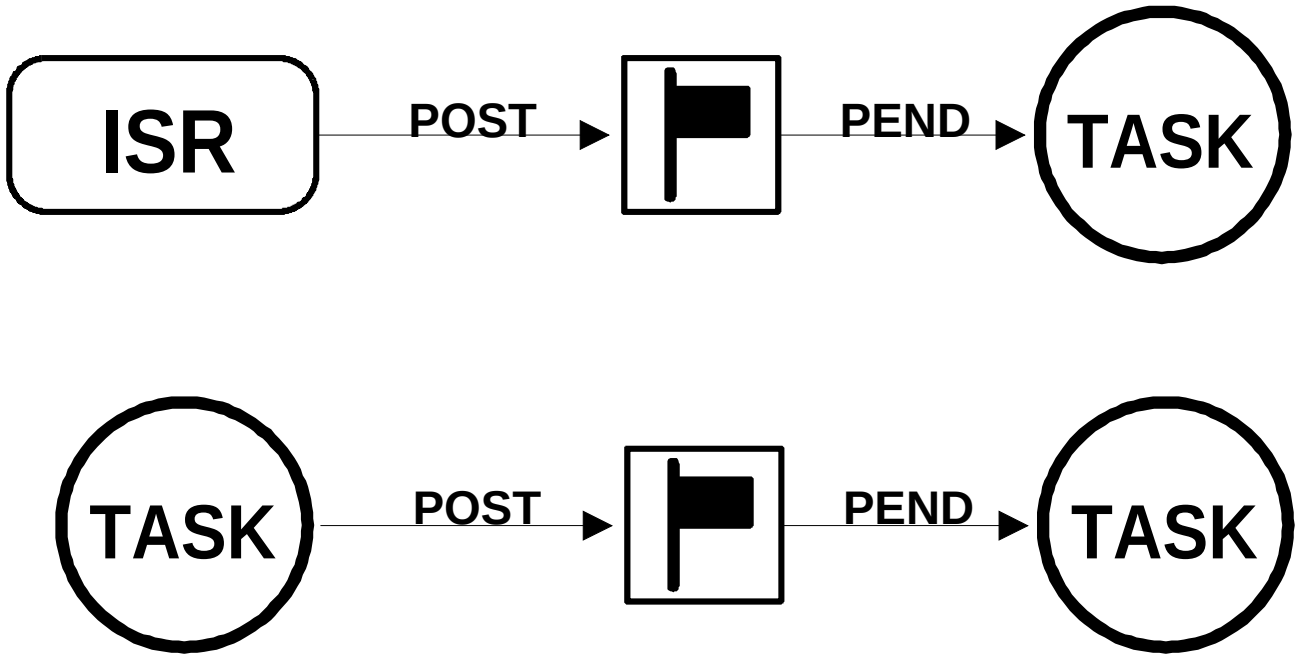

τ_1
τ_2
τ_3
τ_4
R
R
R
R
R


τ_1
τ_1
τ_1

Synchronization

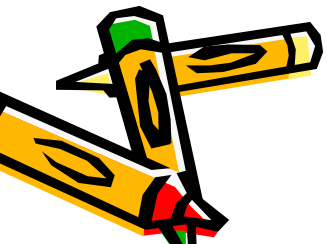
- Different from mutual exclusion, it is much like waiting for an event.
- If a semaphore is used, it must be initialized to 0.
  - It is called unilateral rendezvous.
  - Task←→Task, ISR→Task
    - Note that an ISR never cause itself blocked.




ISR
POST
PEND
TASK
TASK
POST
PEND
TASK

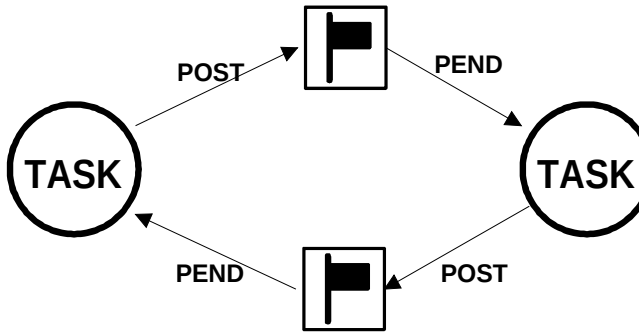
Synchronization

- Two semaphores could be used to rendezvous two tasks.
  - It can not be used to synchronize between ISR's and tasks.
  - For example, a kernel-mode thread could synchronize with a user-mode worker thread which performs complicated jobs.

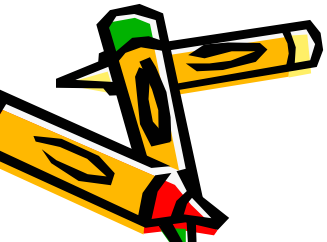


Synchronization

```
Task1()
{
    for (;;) {
            Perform operation;
            Signal task #2;                 (1)
            Wait for signal from task #2;   (2)
            Continue operation;
    }
}
```



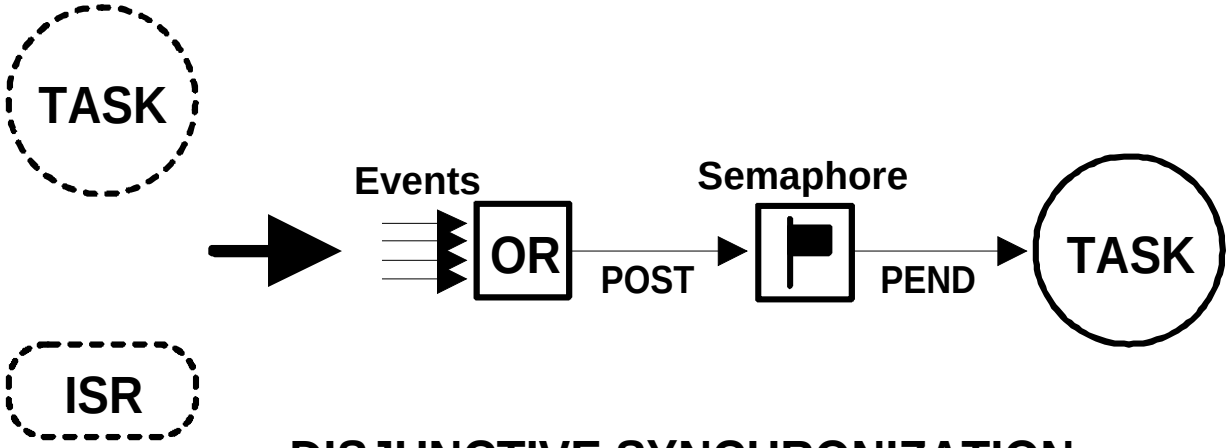
** Semaphores are both initialized to 0



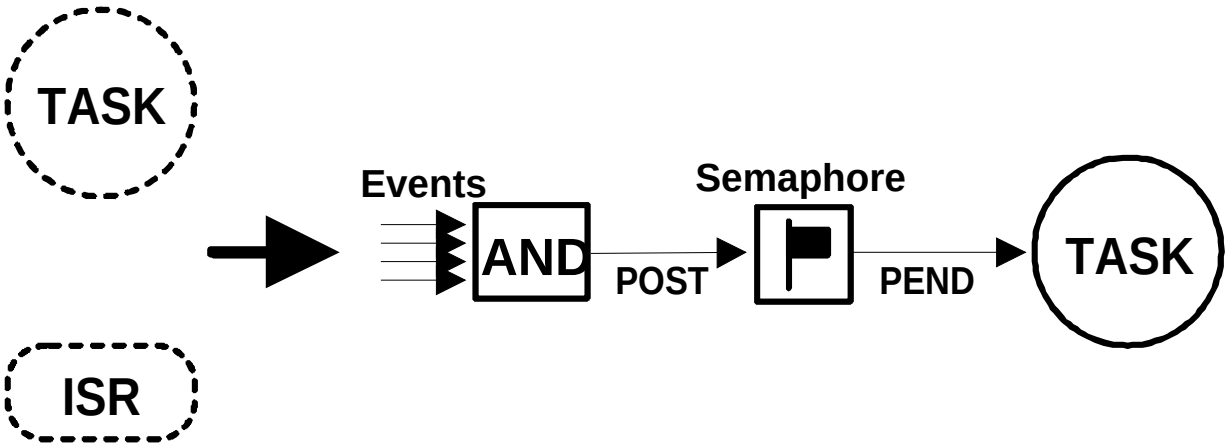
```
Task2()
{
    for (;;) {
            Perform operation;
            Signal task #1;                 (3)
            Wait for signal from task #1;   (4)
            Continue operation;
    }
}
```

Event Flags

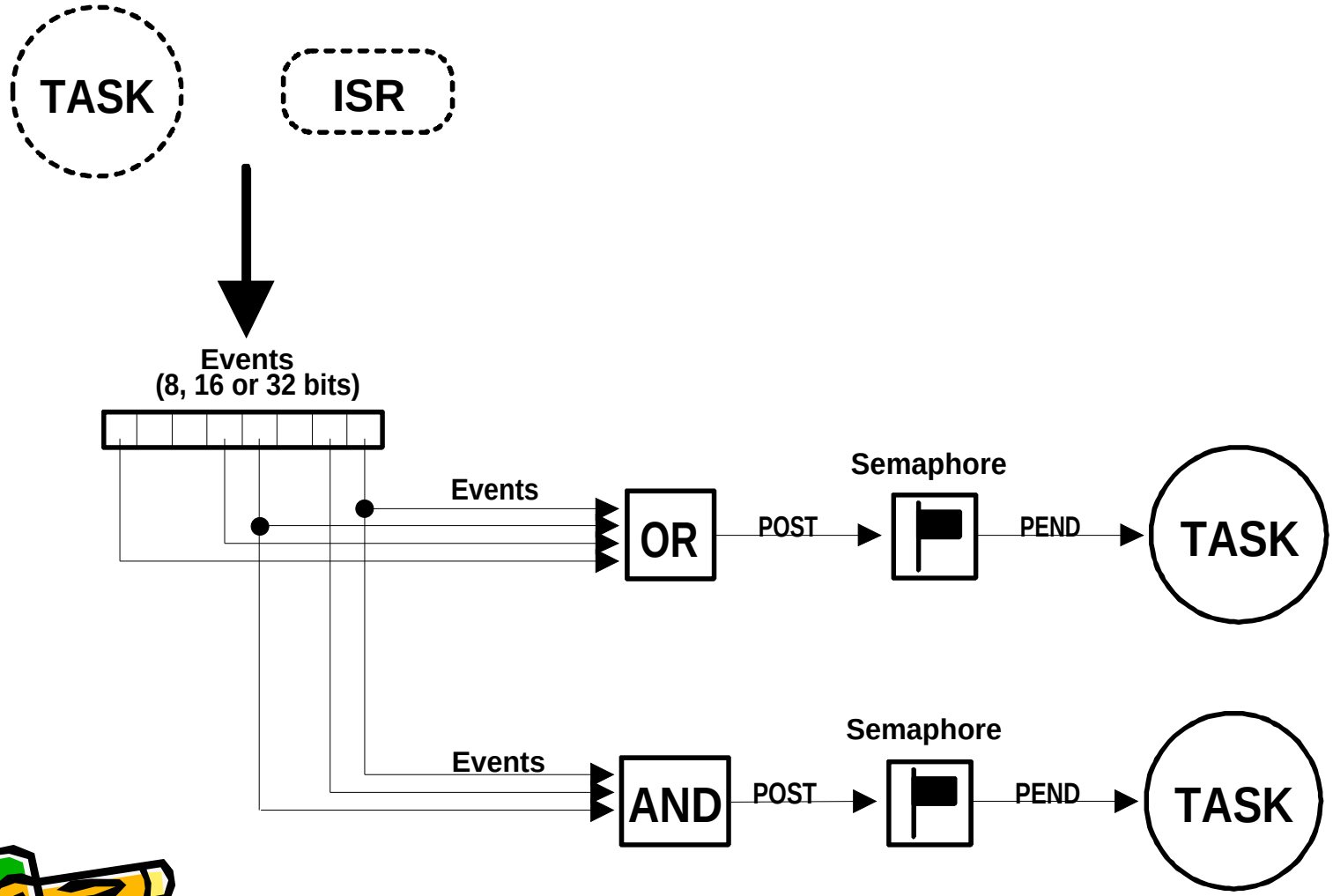
- Event flags are used when a task needs to synchronize with the occurrence of one or more events.
- A set of event can be defined by programmers, represented as a bitmap. (8,16, or 32 bits)
- A task can wait for anyone of (disjunctive, OR) or all of (conjunctive, AND) the defined events.
- An event can notify multiple tasks.
- If any high-priority task becomes ready, context-switch occurs (the highest-priority task is scheduled to run).
- uC/OS-2 supports SET/CLEAR/WAIT for event flags.



DISJUNCTIVE SYNCHRONIZATION



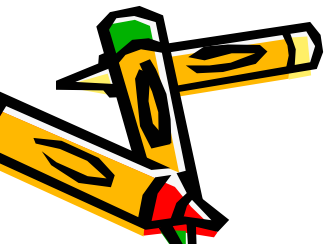
CONJUNCTIVE SYNCHRONIZATION


TASK
ISR
Events
(8, 16 or 32 bits)
Events
OR
POST
Semaphore
PEND
TASK
Events
AND
POST
Semaphore
PEND
TASK

Intertask communication

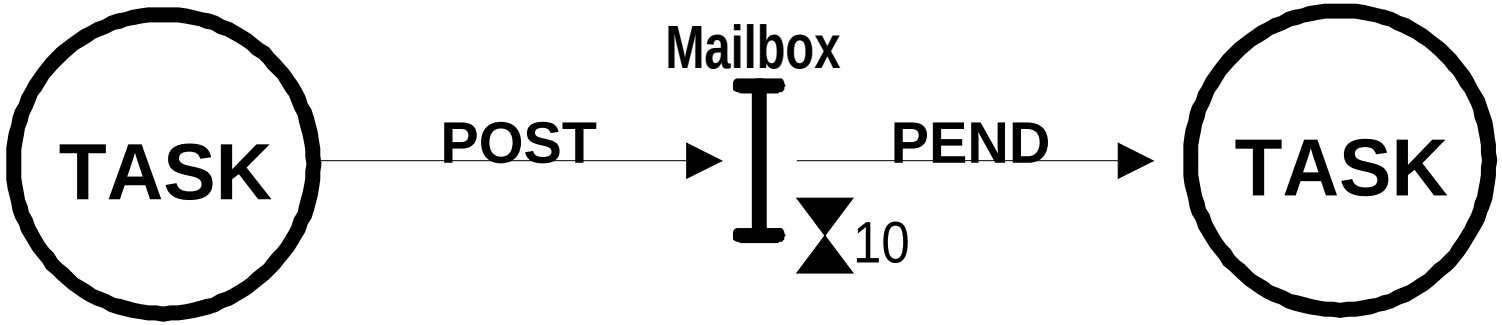
- A task/ISR might want to exchange data with another task/ISR.
  - Mutual exclusion is needed for shared variable.

- If an ISR is involved in intertask communication, the only way is to enable/disable interrupts.
  - Why?



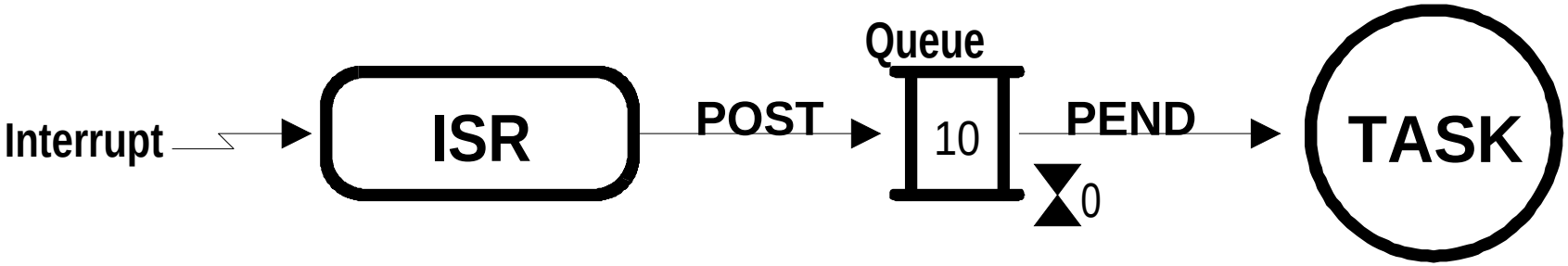
Message Mailboxes

- A mailbox is a data exchange between tasks.
  - A mailbox consists of **a data pointer** and a wait-list.
- **OSMboxPend**():
  - The message in the mailbox is retrieved.
  - If the mailbox is empty, the task is immediately **blocked** and moved to the wait-list.
  - A time-out value can be specified.
- **OSMboxPost**():
  - A message is posted in the mailbox.
  - If there is already a message in the mailbox, **an error is returned (not overwritten).**
  - If tasks waiting for a message from the mailbox, the task with the highest priority is removed from the wait-list and scheduled to run.
- **OSMboxAccept**():
  - If there is no message, return immediately instead of being blocked.


TASK
POST
Mailbox
10
PEND
TASK

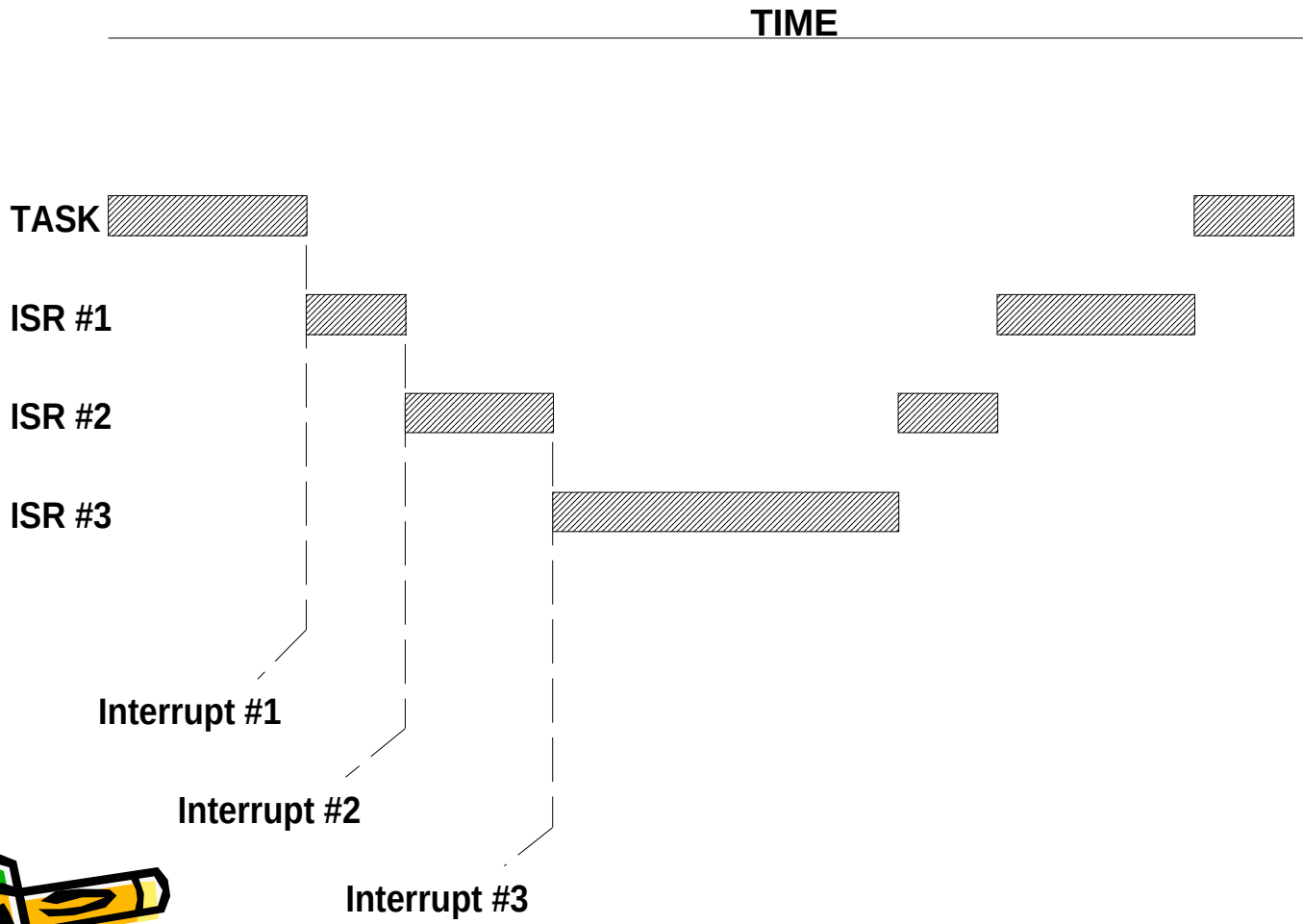
Message Queues

- A message queue consists an array of elements and a wait-list.
- Different from a mailbox, a message queue can hold many data elements (in a FIFO basis).
- As same as mailboxes, there can be multiple tasks pend/post to a message queue.
- **OSQPost**(): a message is appended to the queue. The highest-priority pending task (in the wait-list) receives the message and is scheduled to run, if any.
- **OSQPend**(): a message is removed from the array of elements. If no message can be retrieved, the task is moved to the wait-list and becomes blocked.
- **OSQAccept**(): if there is no messages, return immediately instead of being blocked.


Queue
Interrupt
ISR
POST
10
PEND
TASK
0

Interrupts

- An interrupt is a hardware mechanism used to inform the CPU that an asynchronous event had occurred.
- The CPU saves the context of the current running task and jumps to the corresponding service routine (ISR).
- Common interrupts: clock tick (triggering scheduling), I/O events, hardware errors.
- Disabling interrupts affects interrupt latency.
- The ISR processes the event, and upon completion of the ISR, the program returns to
  - The background for a foreground/background system
  - The interrupted task for a non-preemptive kernel
  - The highest priority task ready to run for a preemptive kernel

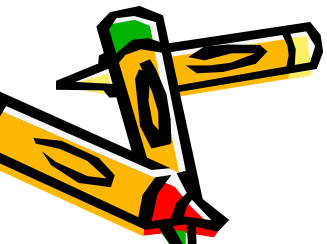

TIME
TASK
ISR #1
ISR #2
ISR #3
Interrupt #1
Interrupt #2
Interrupt #3

Interrupt Latency

- Real-time systems disable interrupts to manipulate critical sections of code and enable interrupts when critical section has executed.

- The longer interrupts disabled, the higher the interrupt latency is.

interrupt latency =
max. amount of interrupts are disabled +
Time to start executing the first instruction in the ISR



Interrupt Response

- Interrupt response: the time between the reception of the interrupt and the **start of the user code** that handles the interrupt – accounts for all the overhead involved in handling an interrupt

- For a foreground/background system and a non-preemptive kernel:

  Response time = Interrupt latency + Time to save the CPU's context

- For preemptive kernel

  Response time = Interrupt latency + Time to save the CPU's context + Execution time of the kernel ISR entry function

(to notify the kernel that an ISR is in progress and allows kernel to keep track of interrupt nesting, OSIntEnter( ) in uC/OS-2)

Interrupt Recovery

- The time required for the processor to return to the interrupted code.

- For a foreground/background system and a non-preemptive kernel:

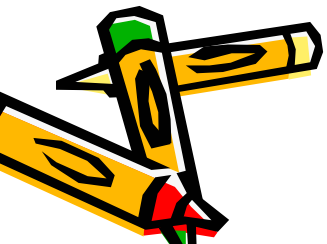
  Interrupt recovery
  = Time to restore the CPU's context
  + Time to execute the return from interrupt instruction

- For preemptive kernel:

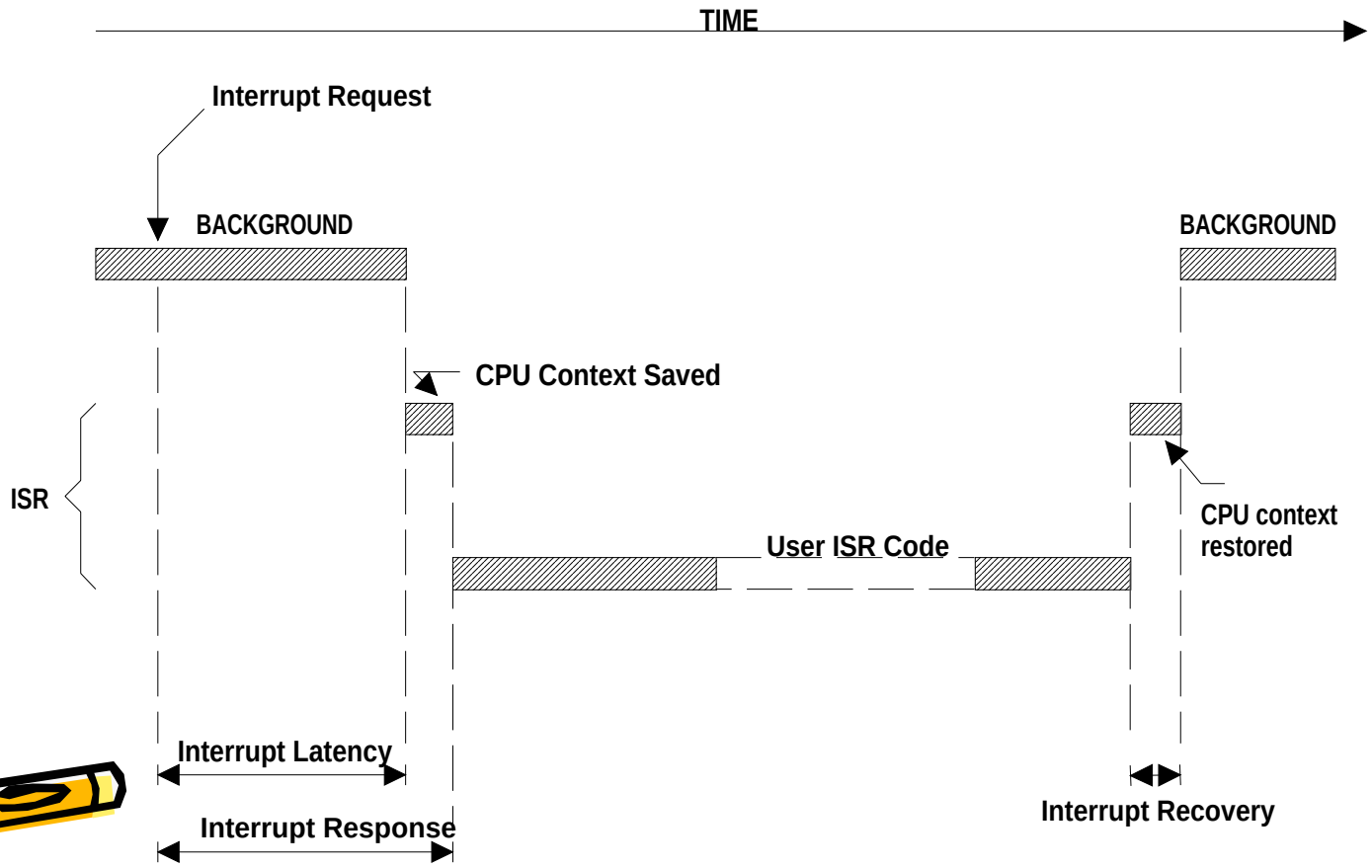
  Interrupt recovery
  = Time to determine if a higher priority task is ready
  + Time to restore the CPU's context of the highest priority task
  + Time to execute the return from interrupt instruction

ISR Processing Time

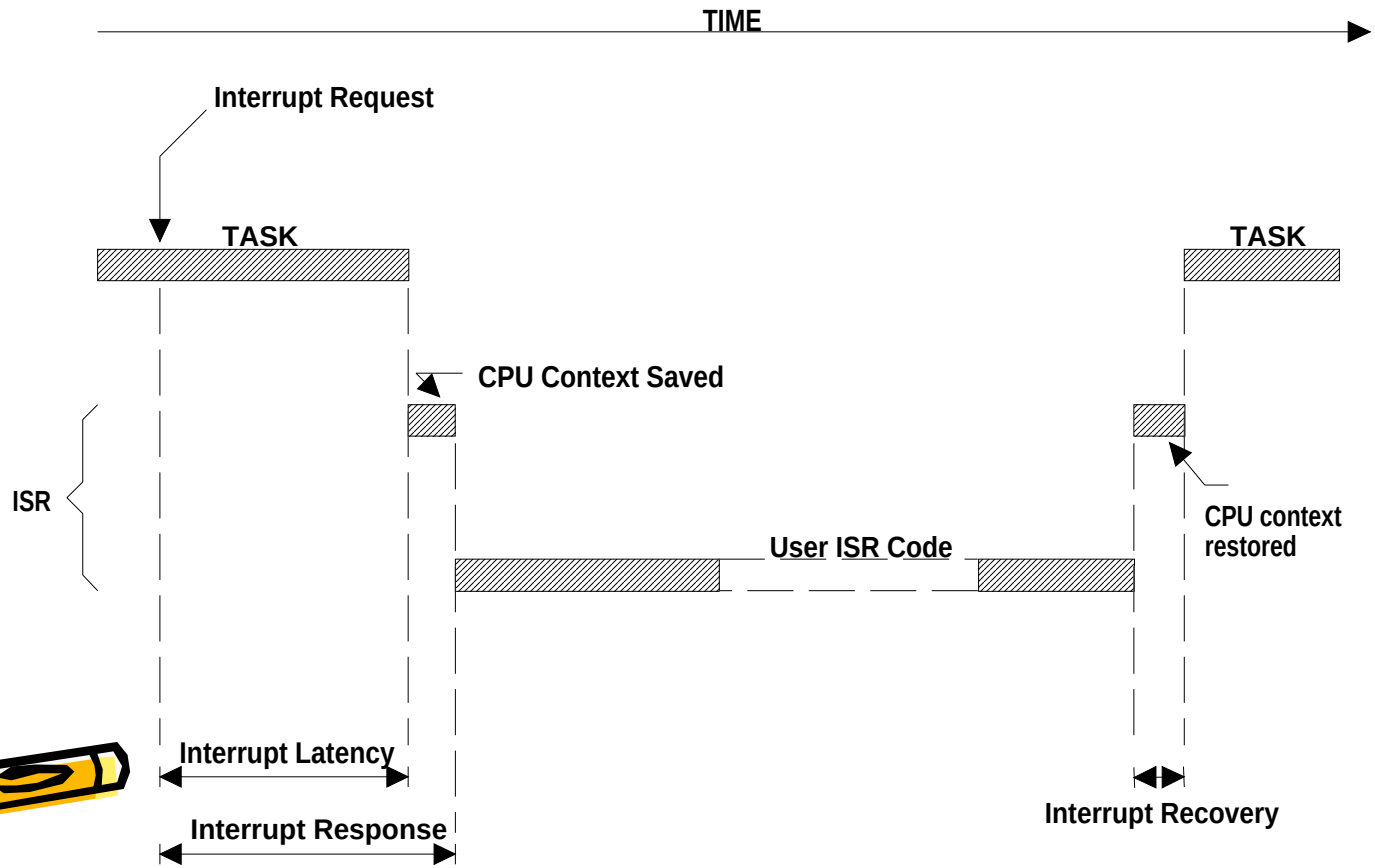
- ISRs should be as short as possible.
  - there are no absolute limits on the amount of time for an ISR.
- If the ISR code is the most important code that needs to run at any given time, it could be as long as it needs to be.
- In most cases, the ISR should
  - Recognize the interrupt
  - Obtain data or status from the interrupting device
  - Signal a task to perform a actual processing
- Overhead involved in signaling task
  - the processing of the interrupt



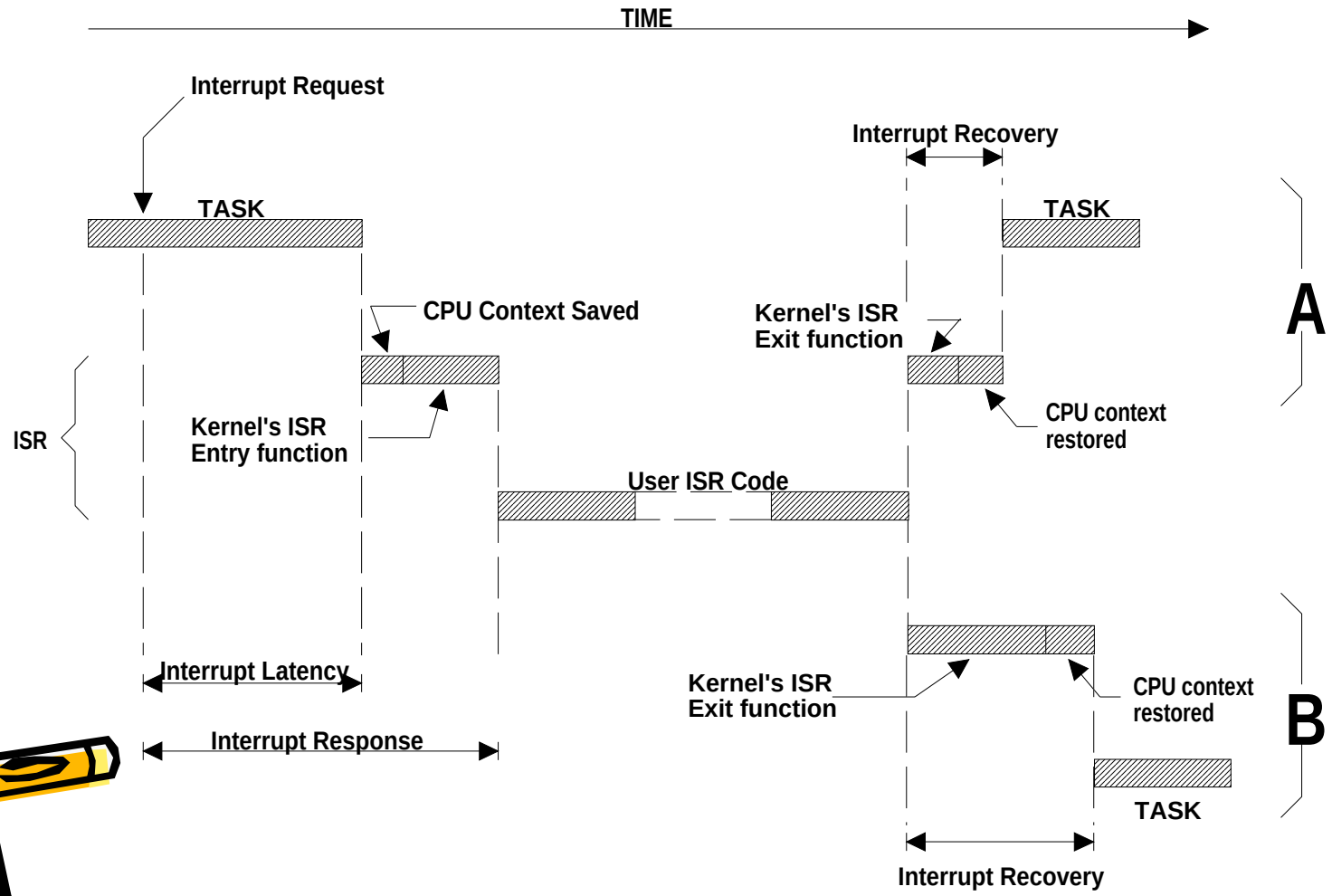
Interrupt latency, response, and recovery (Foreground/Background)



Interrupt latency, response, and recovery (Non-preemptive kernel)



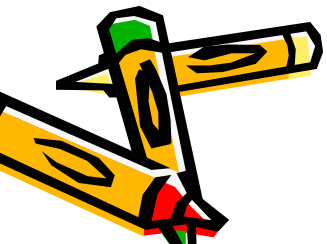
Interrupt latency, response, and recovery (Preemptive kernel)



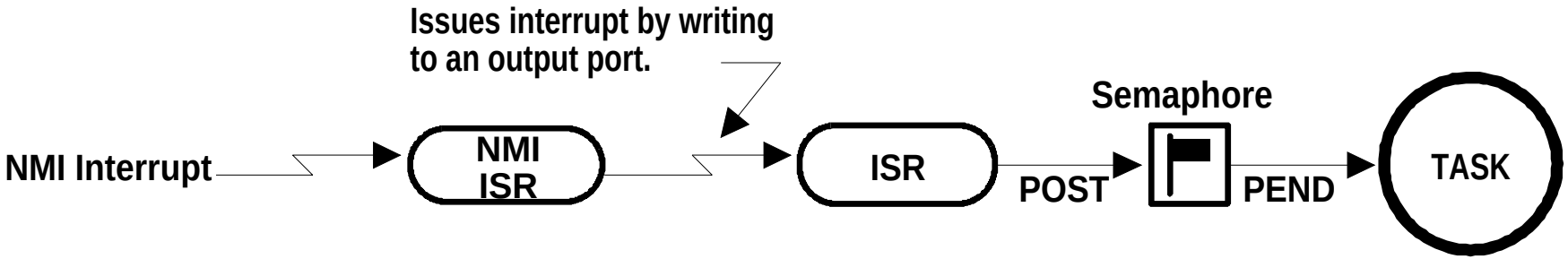
Non-Maskable Interrupts

- NMI's can not be disabled.
  - They are generally reserved for drastic events, such as the power-source is almost exhausted.

- You can not use kernel services to signal a task in ISR's of NMI's.
  - Since interrupts can not be disabled in the ISR of an NMI.
  - The size of global variable under this situation must be atomic. (i.e., byte, word, dword)
  - Or, we can trigger another hardware interrupt which's ISR uses kernel services to signal the desired task.



Signaling a task from the ISR of an NMI



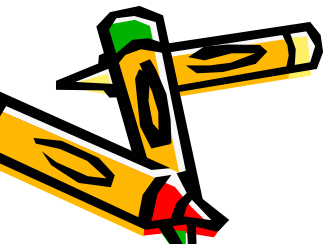
Non-Maskable Interrupts

Interrupt latency
= Time to execute longest instruction
+ Time to start executing the NMI ISR

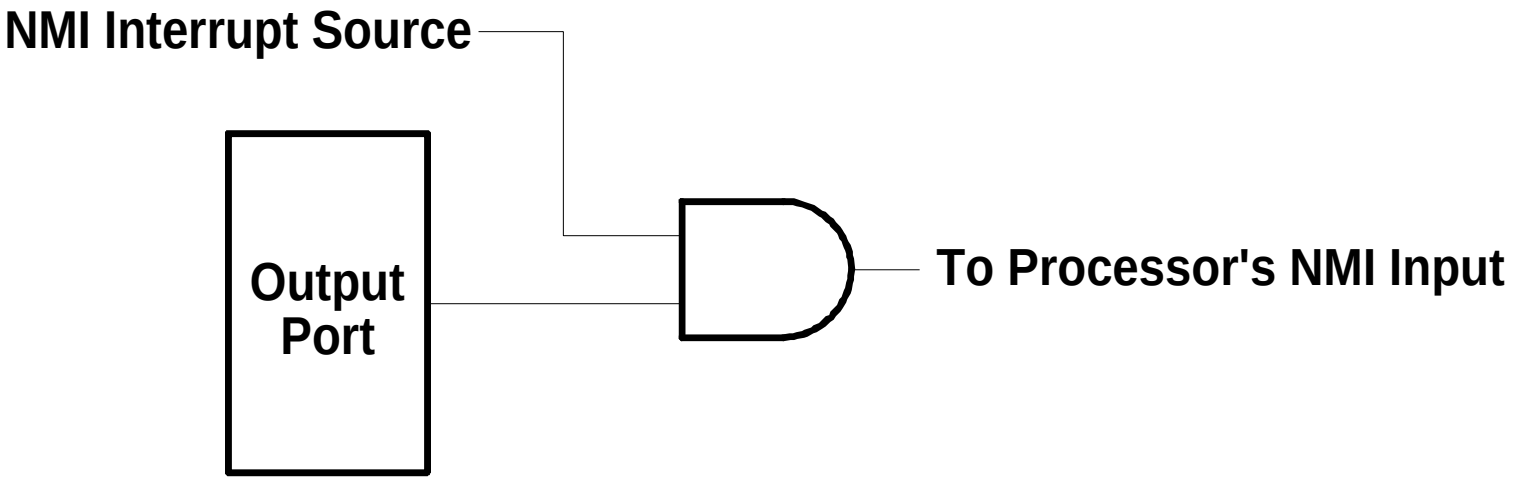
Interrupt response
= Interrupt latency
+ Time to save the CPU's context

Interrupt recovery
= Time to restore the CPU's context
+ Time to execute the return from interrupt instruction

- NMI can still be disable by adding external circuits.

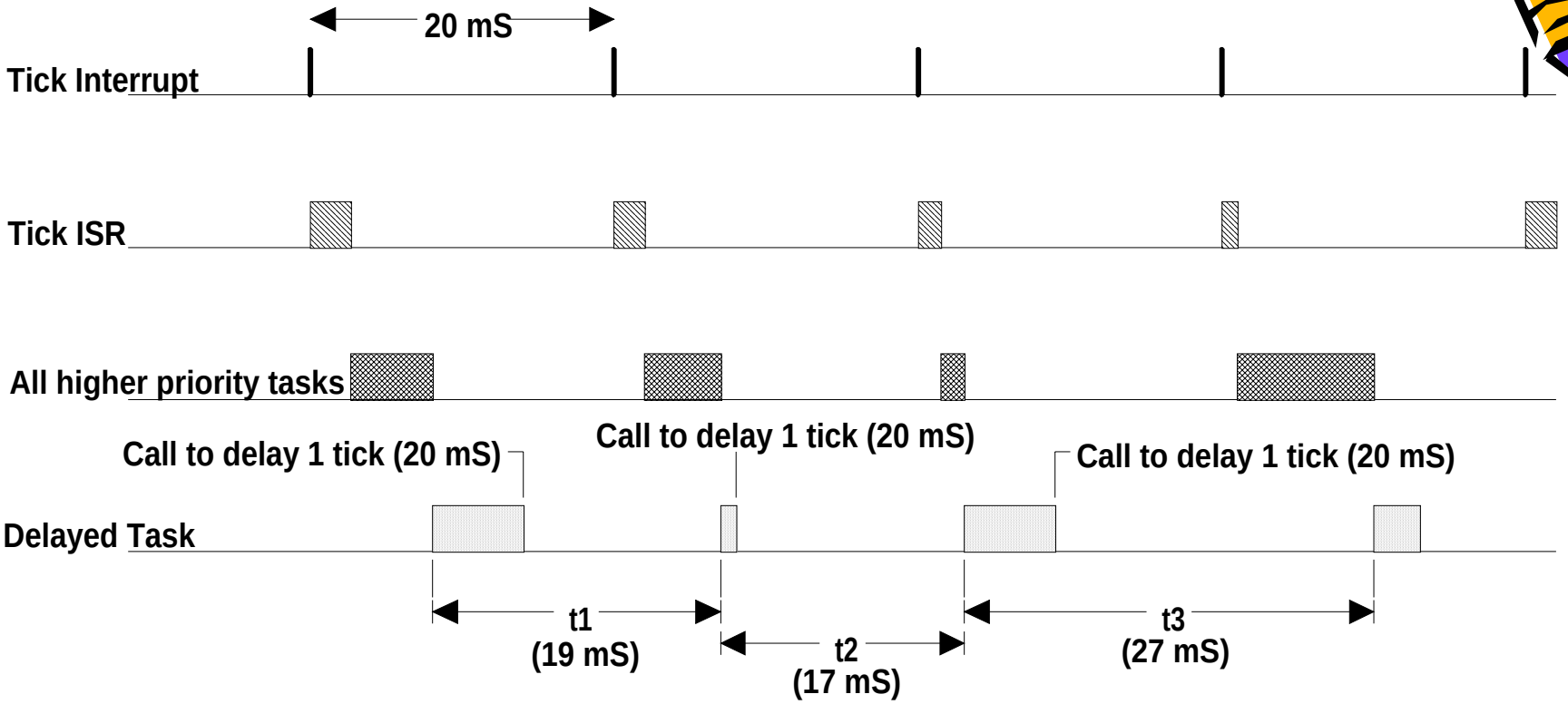


Disabling NMI's

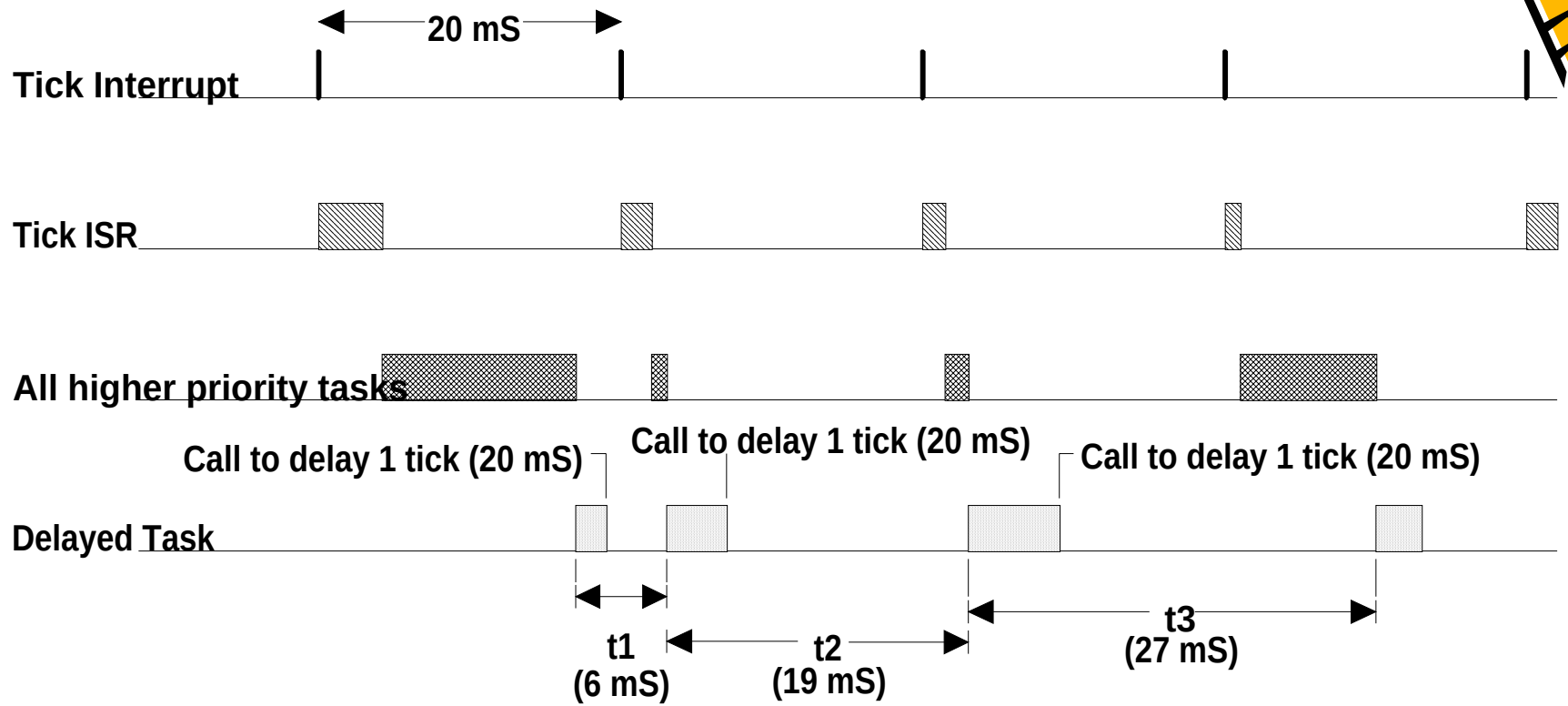


Clock Tick

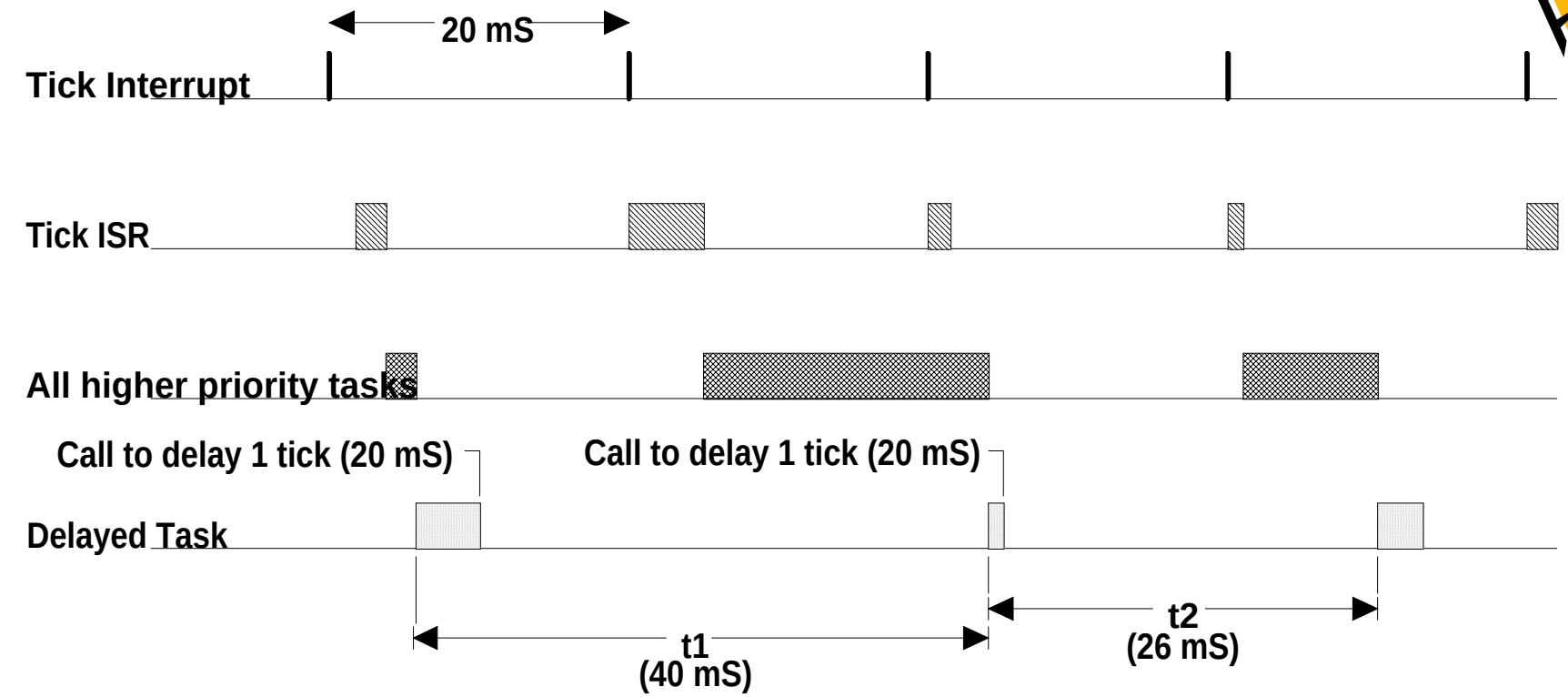
- Clock tick is a periodically hardware event (interrupt) generated by a timer.
- The kernel utilize the timer to delay tasks and to periodically perform scheduling.
- The higher the tick rate,
  - the better the responsiveness is.
  - the better the schedulability is.
    - Blocking due to clock tick resolution.
  - the higher the overhead is.



- A task delaying itself for one tick
- Higher priority tasks and ISRs execute prior to the task, which needs to delay for 1 tick
- A jitter occurs.



- The execution times of all higher priority tasks and ISRs are slightly less than 1 tick
- As a result, if you need to delay at least one tick, you must specify one extra tick



- The execution times of all higher priority tasks and ISRs are more than 1 clock tick.
- The task that tries to delay for 1 tick actually executes two ticks later and violates its deadline.

Memory Requirements

- Most real-time applications are embedded systems. Memory requirements must be analyzable.
- A preemptible kernel requires more RAM/ROM space.
- Code size (ROM) = kernel size + application size
- RAM requirements can be significantly reduced if
  - Stack size of every task can be differently specified
  - A separate stack is used to handle ISR's. (uC/OS-2 doesn't, DOS does)
- RAM requirement = application requirement + kernel requirement + SUM(task stacks + MAX(ISR nesting))
- RAM requirement = application requirement + kernel requirement + SUM(task stacks) + MAX(ISR nesting)
  - If a separate stack is prepared for ISR's.

Memory Requirements

- We must be careful on the usages of tasks' stacks:
  - Large arrays and structures as local variables.
  - Recursive function call.
  - ISR nesting.
  - Function calls with many arguments.

Advantages and Disadvantages of Real-Time Kernels

- A real-time kernel (RTOS) allows real-time applications to be designed and expanded easily.
  - Functions can be added without requiring major changes to the software.
- The use of RTOS simplifies the design process by splitting the application code into separate tasks.
- With a preemptive RTOS, all time-critical events are handled as quickly and as efficiently as possible.
- An RTOS allows you to make better use of your resources by providing you with valuable services – semaphores, mailboxes, queues, time delays, timeouts, etc.
  - Extra cost of the kernel.
  - More ROM/RAM space.
  - 2 to 4 percent additional CPU overhead.
  - Cost of the RTOS: $70 ~ $30,000 !
  - The maintenance cost: $100 ~ $5,000 per year !

Real-Time Systems Summary

	Foreground/Background	Non-Preemptive Kernel	Preemptive Kernel
Interrupt Latency (Time)	MAX(Longest instruction, User int. disable) + Vector to ISR	MAX(Longest instruction, User int. disable, Kernel int. disable) + Vector to ISR	MAX(Longest instruction, User int. disable, Kernel int. disable) + Vector to ISR
Interrupt response (Time)	Int. latency + Save CPU's context	Int. latency + Save CPU's context	Interrupt latency + Save CPU's context + Kernel ISR entry function
Interrupt recovery (Time)	Restore background's context + Return from int.	Restore task's context + Return from int.	Find highest priority task + Restore highest priority task's context + Return from interrupt
Task response (Time)	Background	Longest task + Find highest priority task + Context switch	Find highest priority task + Context switch
ROM size	Application code	Application code + Kernel code	Application code + Kernel code
RAM size	Application code	Application code + Kernel RAM + SUM(Task stacks + MAX(ISR stack))	Application code + Kernel RAM + SUM(Task stacks + MAX(ISR stack))
Services available?	Application code must provide	Yes	Yes