

# ELEC3730

## Embedded Systems

Real-time kernels  
How MicroC/OS-II does it!

Adrian Wills, EA-204, Ext. 16028

# Previous Lecture

- Real-time embedded systems challenge
  - hard deadlines, many tasks, one CPU
- Methods to handle multiple tasks
  - Polled
  - Foreground/background
  - Real-time kernels - **schedule tasks**
- MicroC/OS-II
  - Example program with two tasks

# Topics Covered

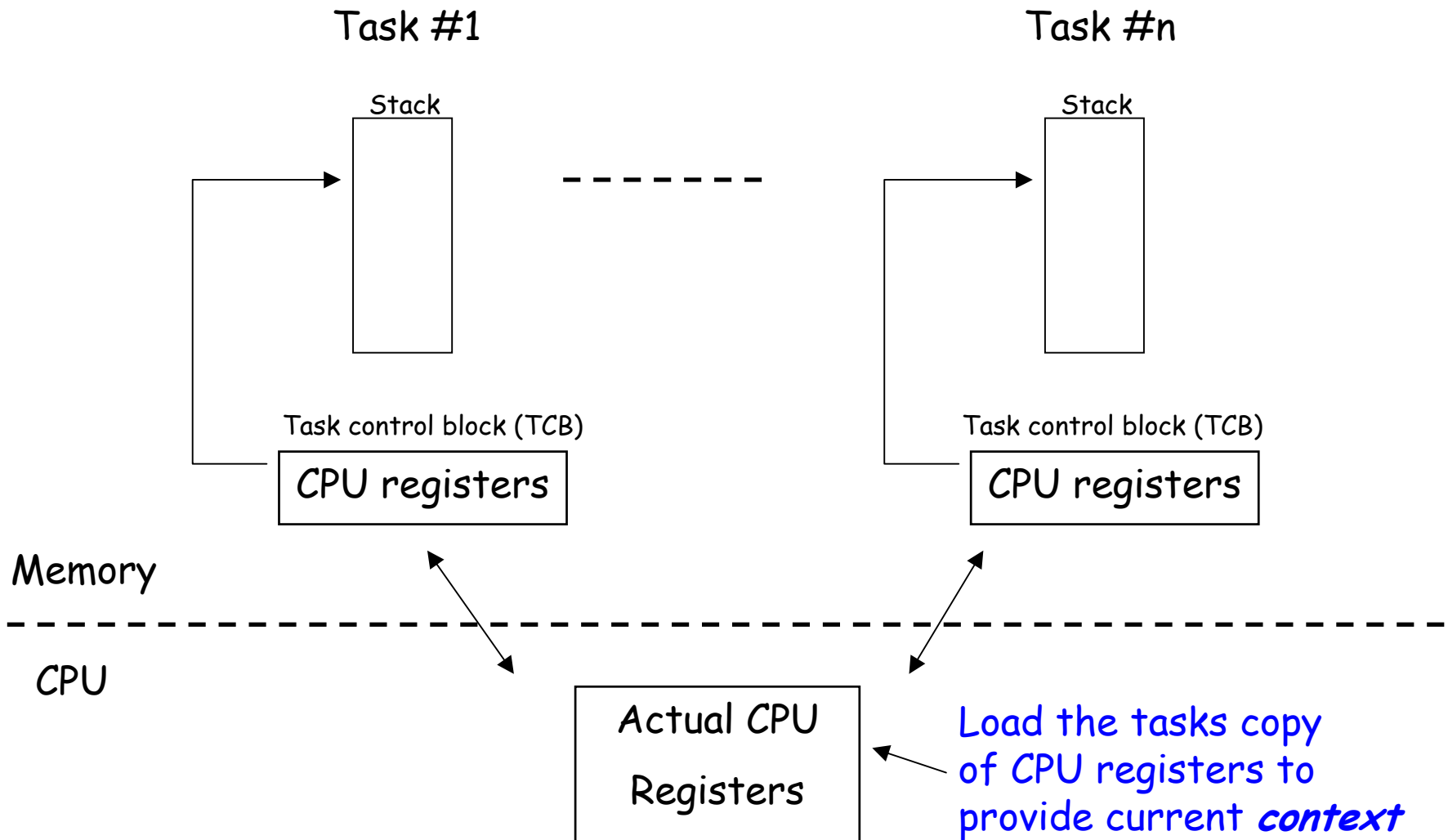
- Task states
  - dormant, ready, running, waiting
- Scheduler
  - **When:** preemptive & non-preemptive kernels
  - **Which:** assigning priorities
  - **How:** context switching
- Shared Resources (more next lecture)
  - critical sections
  - schedule locking
  - semaphores

# Recall: Task structure

- Infinite loop
  - *runs* on CPU when "conditions" are satisfied and kernel *schedules* it

```
void Task()  
{  
    //Initialisation code here  
  
    while (1)  
    {  
        //Perform specific task here  
    }  
}
```

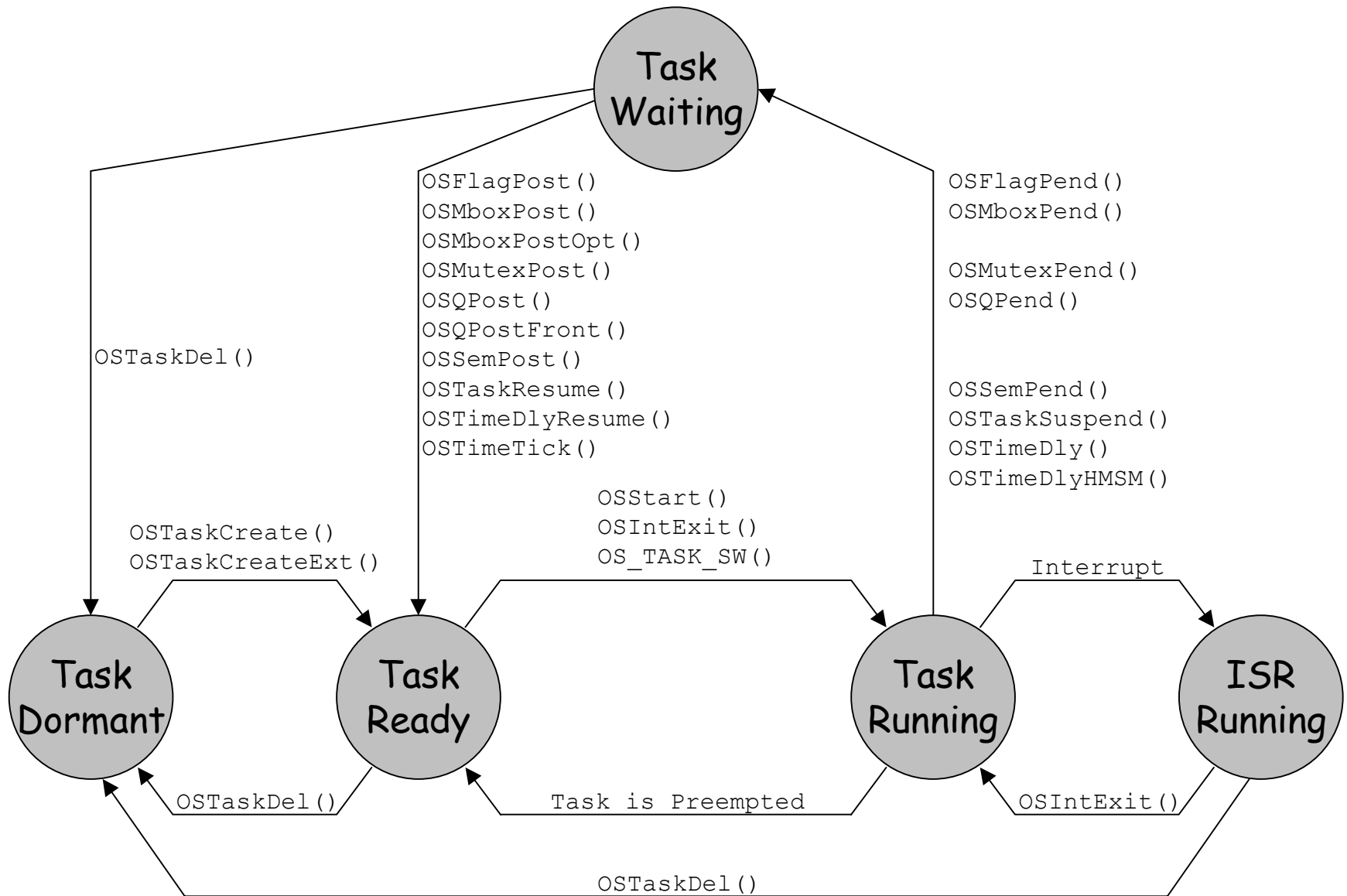
# Recall: Real-time kernel



# Task States

- ***Dormant***
  - Exists in memory, but is not known by kernel
- ***Ready***
  - Task is in queue to use CPU
- ***Running***
  - Task has control of CPU and is executing
- ***Waiting***
  - Task is waiting for something before it continues

# Task states



# Example 1 revisited

```
int main()
{
    // Display a banner.
    printf("##### uCOS-II ELEC3730 Example 1\n");

    // Initialize uCOS-II.
    OSInit();

    // Create some tasks
    OSTaskCreate(Task1, (void *) NULL, &Task1Stk[TASK_STK_SIZE], 5);
    OSTaskCreate(Task2, (void *) NULL, &Task2Stk[TASK_STK_SIZE], 6);

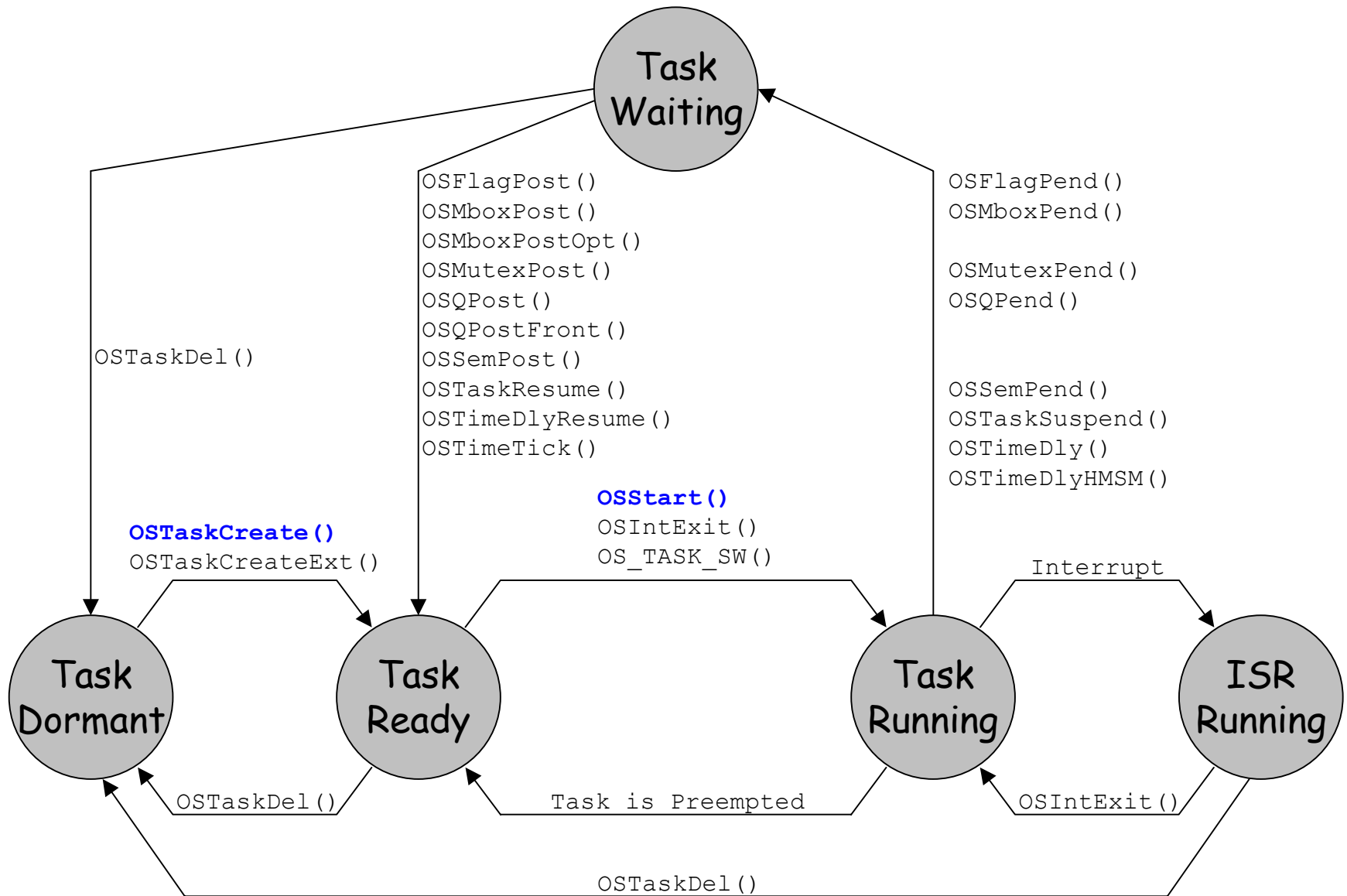
    // Start multitasking.
    OSStart();

    /* NEVER EXECUTED */
    printf("main(): We should never execute this line\n");
}
```

Kernel moves task state from dormant to ready

Start the kernel running moves highest priority ready task into running

# Task states



```
/******  
 * First Task *  
*****/
```

```
void Task1(void *pdata)  
{  
    printf("(II) Task1 initialised\n");  
    while (1)  
    {  
        OSTimeDly(100);  
        printf("1 ");  
    }  
}
```

Kernel moves task from running to waiting state

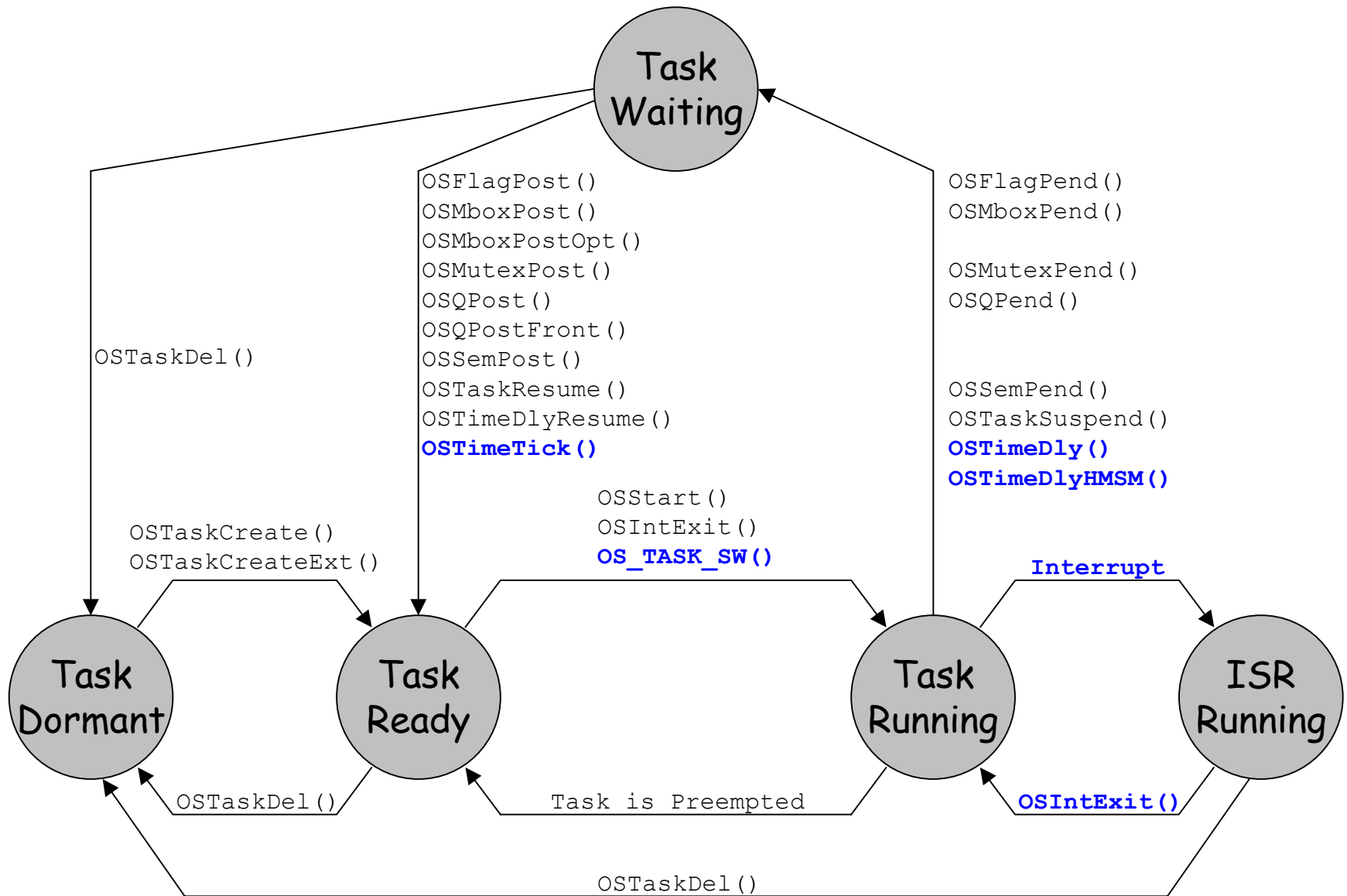
```
/******  
 * Second Task *  
*****/
```

```
void Task2(void *pdata)  
{  
    printf("(II) Task2 initialised\n");  
    while (1)  
    {  
        OSTimeDlyHMSM(0,0,4,0);  
        printf("2 ");  
    }  
}
```

Task 2 is now highest priority ready task, kernel moves from ready to running

Kernel moves task from running to waiting state

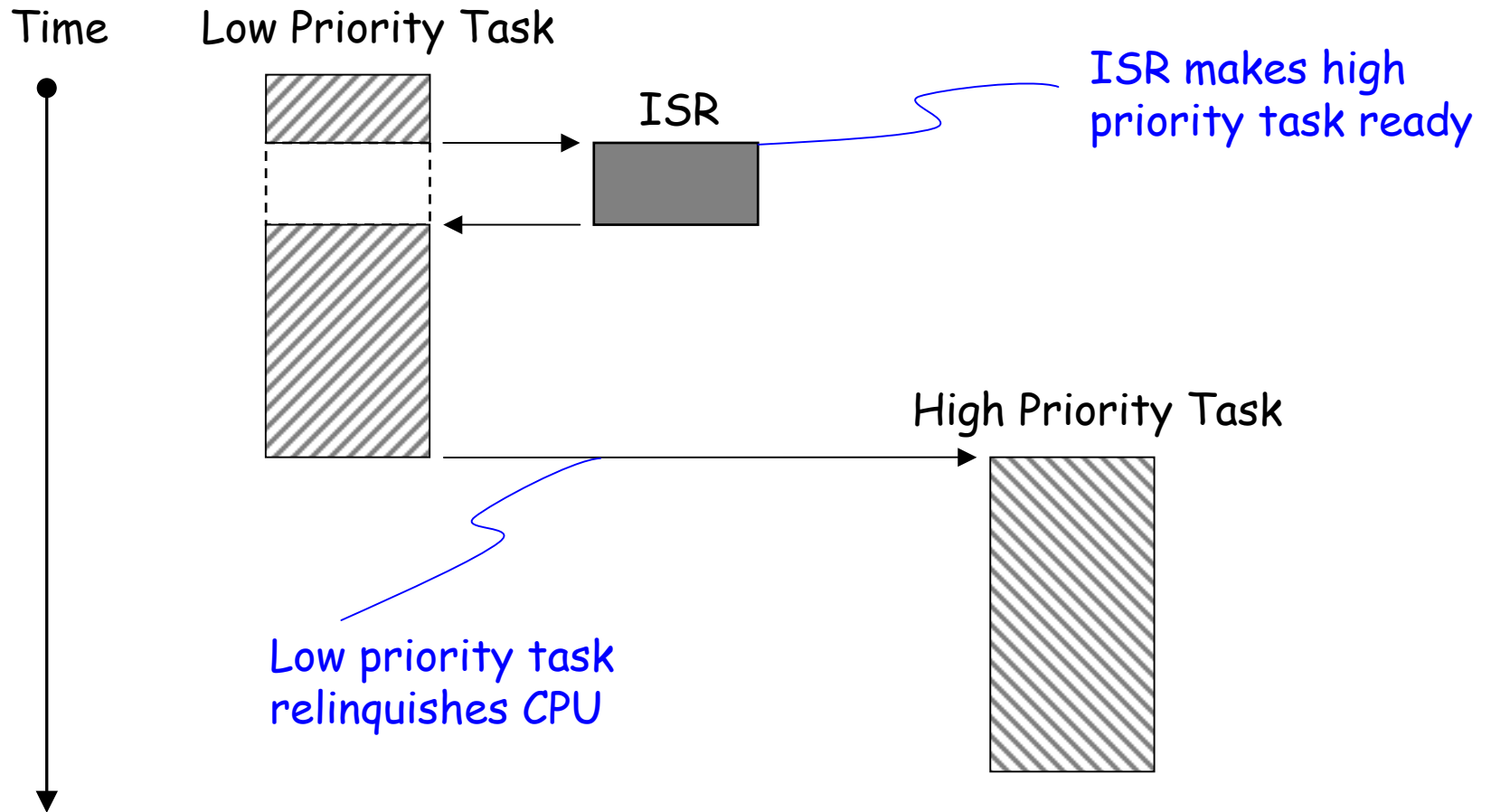
# Task states



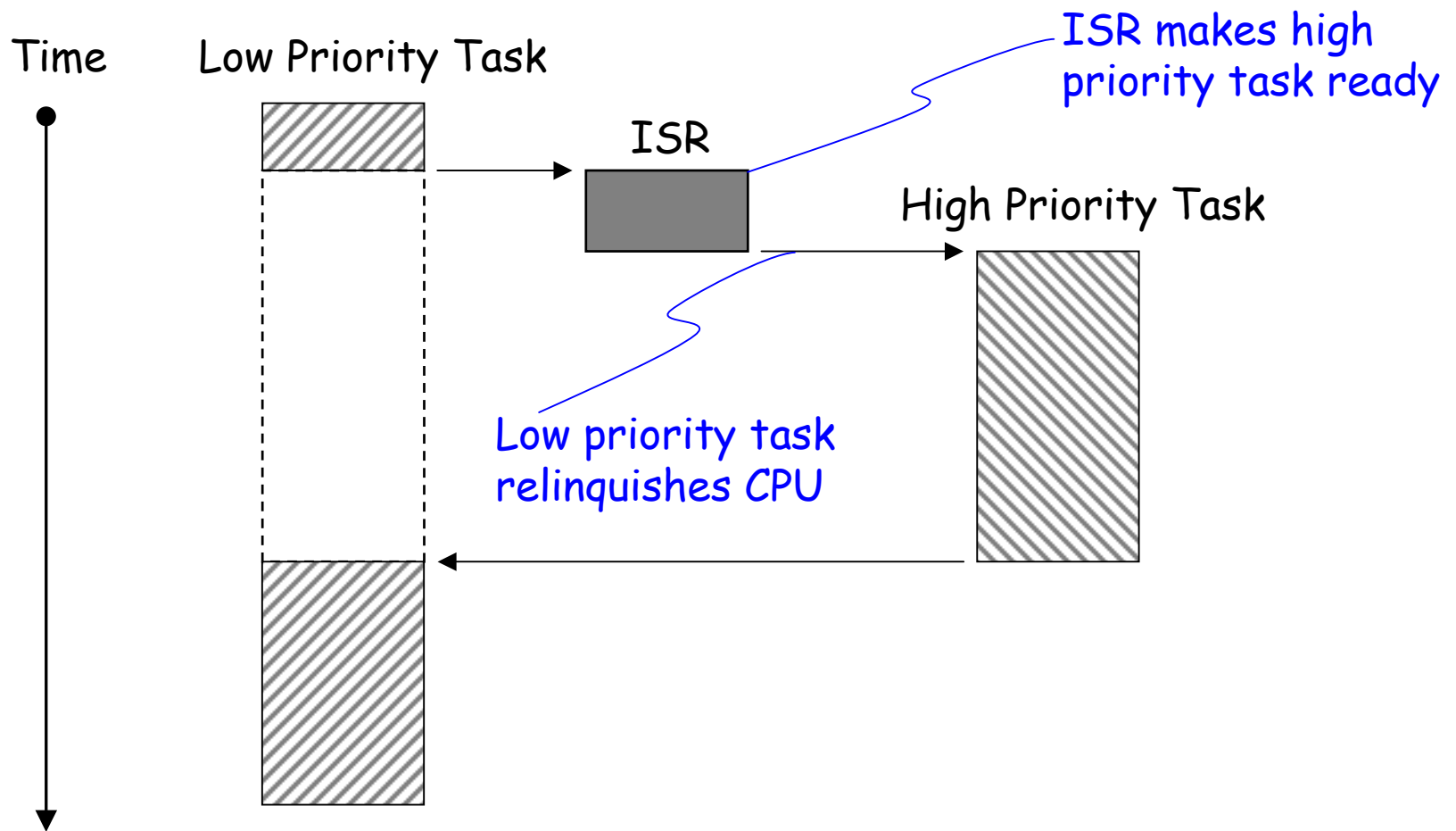
# Scheduler

- When and how do tasks move from *ready* to *running*, and which task?
- When
  - depends on kernel (preemptive or not)
  - MicroC/OS-II is a preemptive kernel
- Which task
  - highest priority ready task
- How
  - kernel performs a context switch

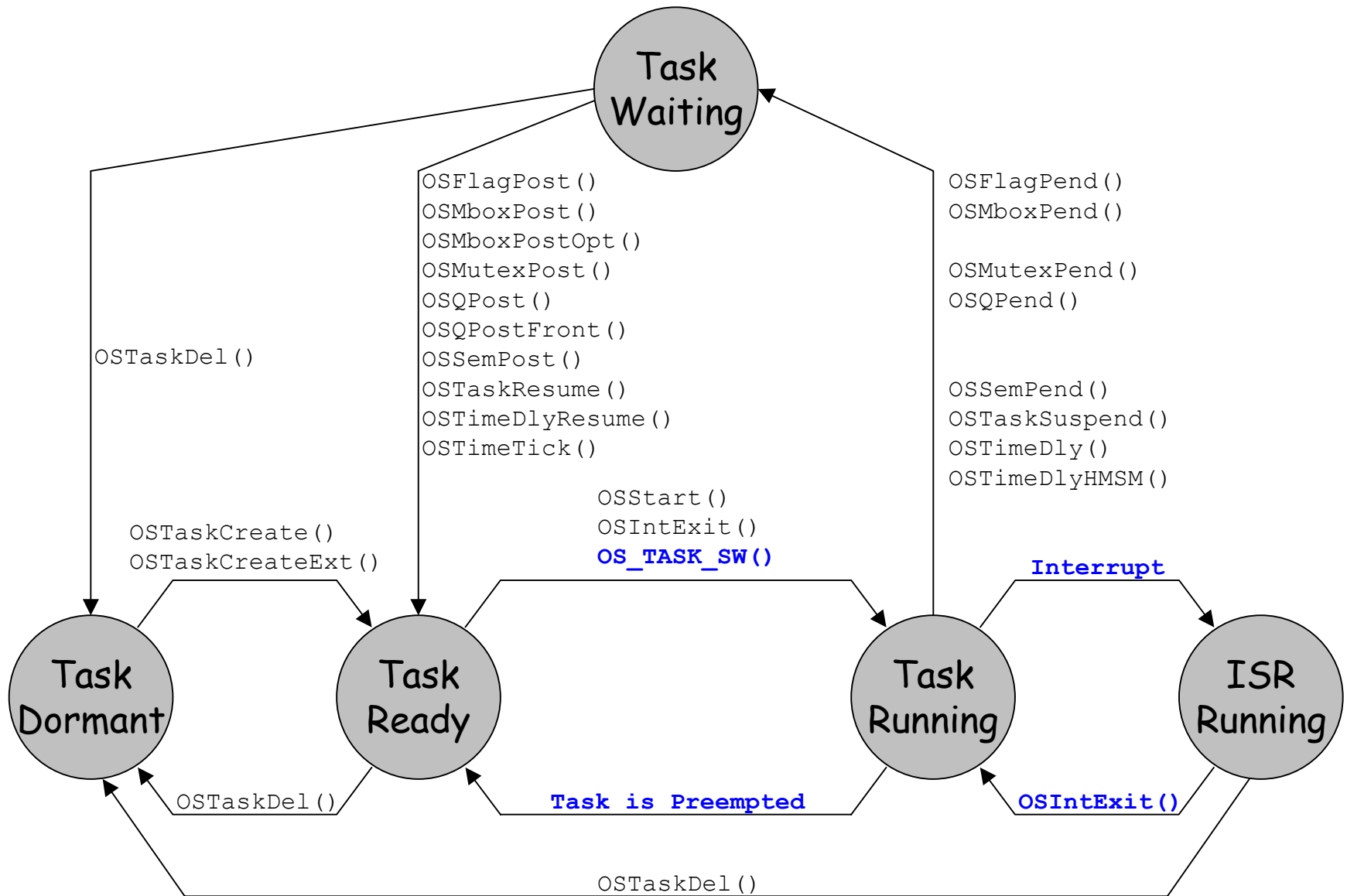
# When: Non-Preemptive Kernels

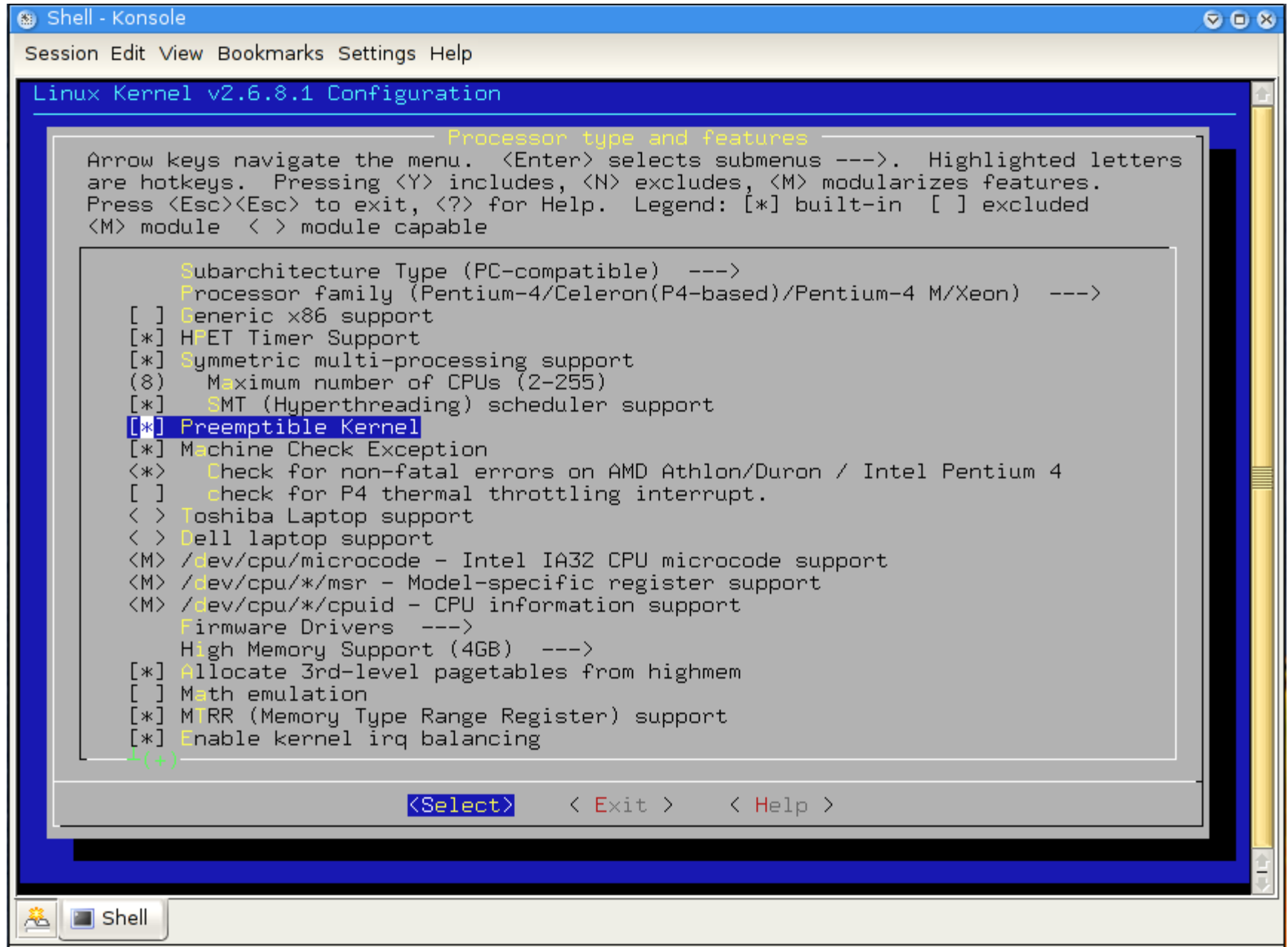


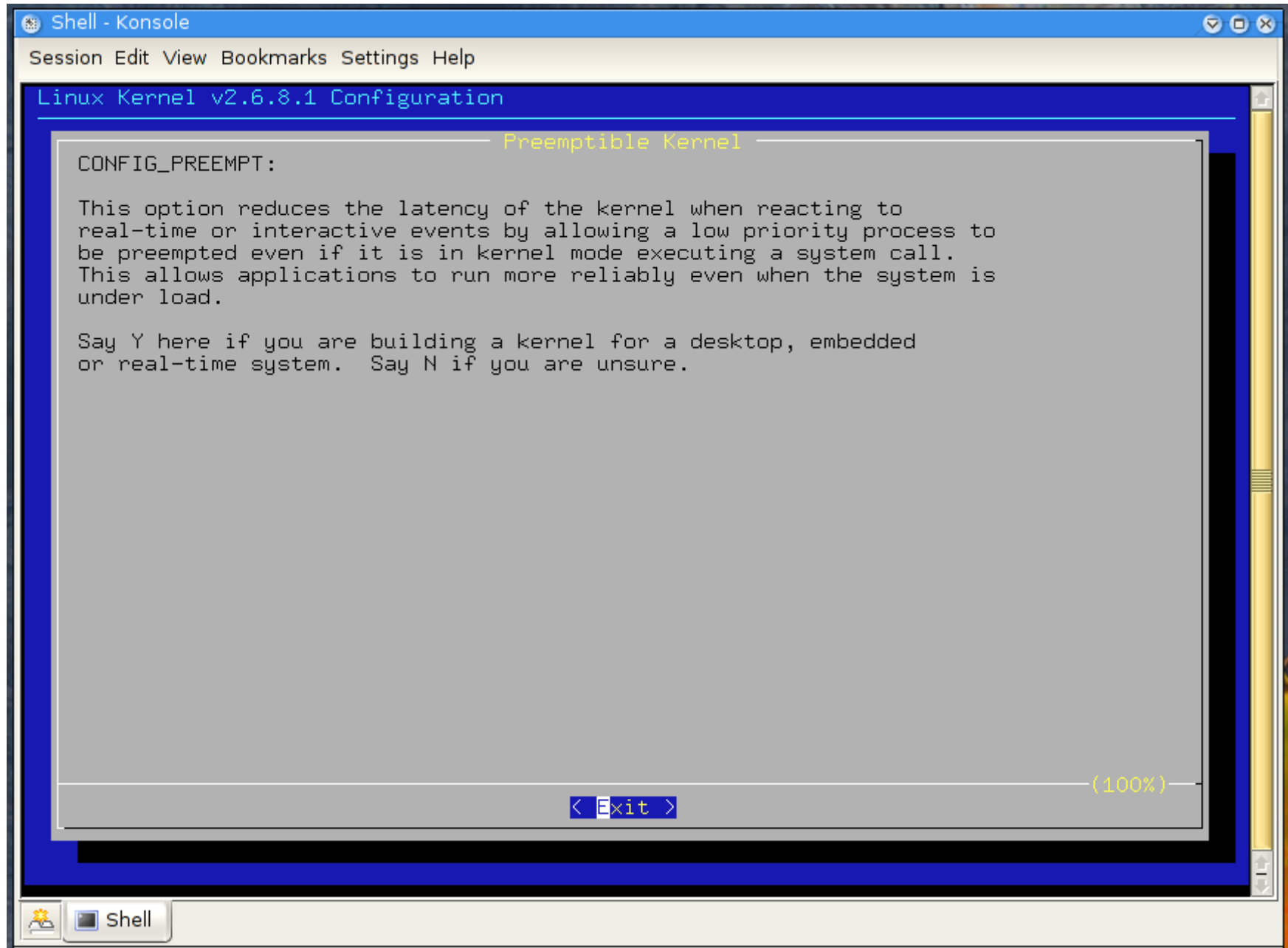
# When : Preemptive Kernels



# Task states







# Which Task

- MicroC/OS-II selects highest priority task on ready list
  - can have a maximum of 64 distinct priorities 0 - 63
  - highest priority is 0
  - recommended not to use the first four and last four priority levels
  - thus, 56 distinct tasks
  - priority can be changed dynamically

# Which Task

- Assigning priorities - difficult job
  - time critical (high priority), less time critical (lower priority)
- Rate Monotonic Scheduling (RMS)
  - Task with highest execution rate gets highest priority

# Which Task: RMS

- Assumptions:
  - tasks are periodic
  - no synchronisation or sharing of resources
  - CPU must execute highest priority task ready
- Then given  $n$  tasks, each assigned RMS priorities, then all *hard* real-time deadlines are met if

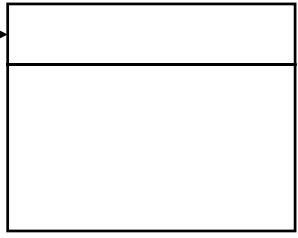
$$\sum \frac{E_i}{T_i} \leq n(2^{1/n} - 1)$$

where  $E_i$  is maximum execution time of task  $i$   
and  $T_i$  is the period of task  $i$

# How: Context Switch

Low Priority Task

OS\_TCB

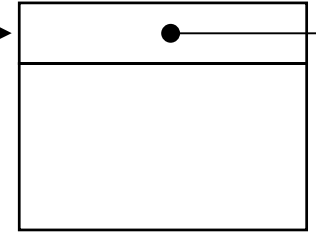


OSTCBCur

Points to TCB of task being suspended

High Priority Task

OS\_TCB

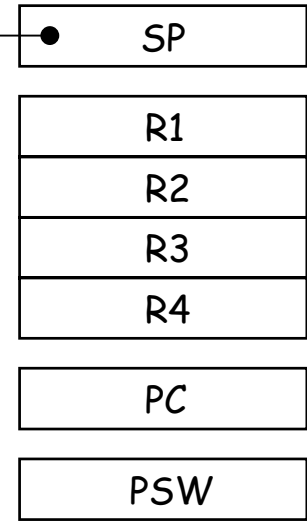


OSTCBHighRdy

Points to TCB of highest priority task

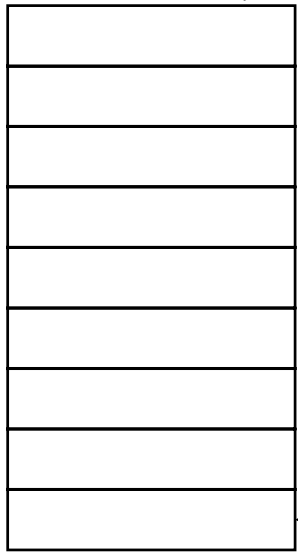
Points to TOS of task to resume

CPU



CPU SP points to TOS of task being suspended

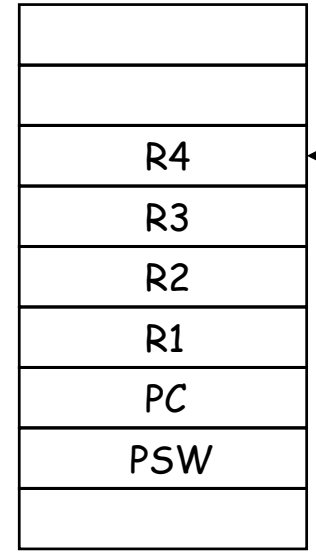
Low Memory



Stack Growth

High Memory

Low Memory



High Memory

# How: Context Switch

Low Priority Task

High Priority Task

OS\_TCB

OS\_TCB

OSTCBCur

OSTCBHighRdy

SP saved into TCB

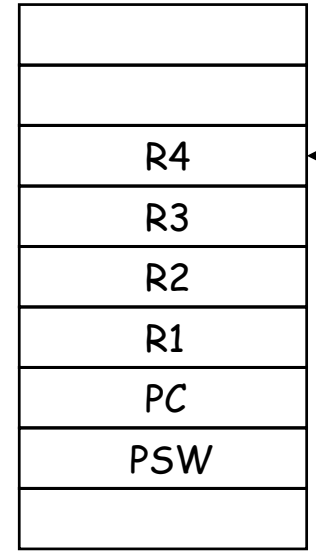
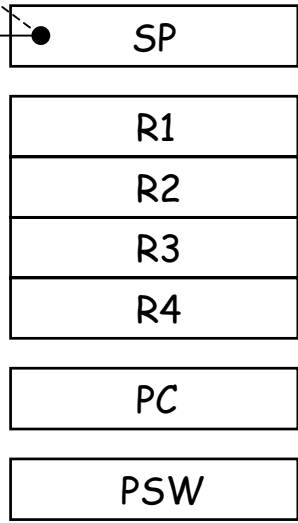
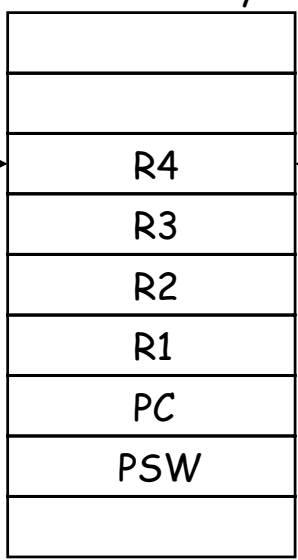
Software int. handler saves R1 - R4

CPU

Low Memory

Low Memory

Stack Growth  
↑

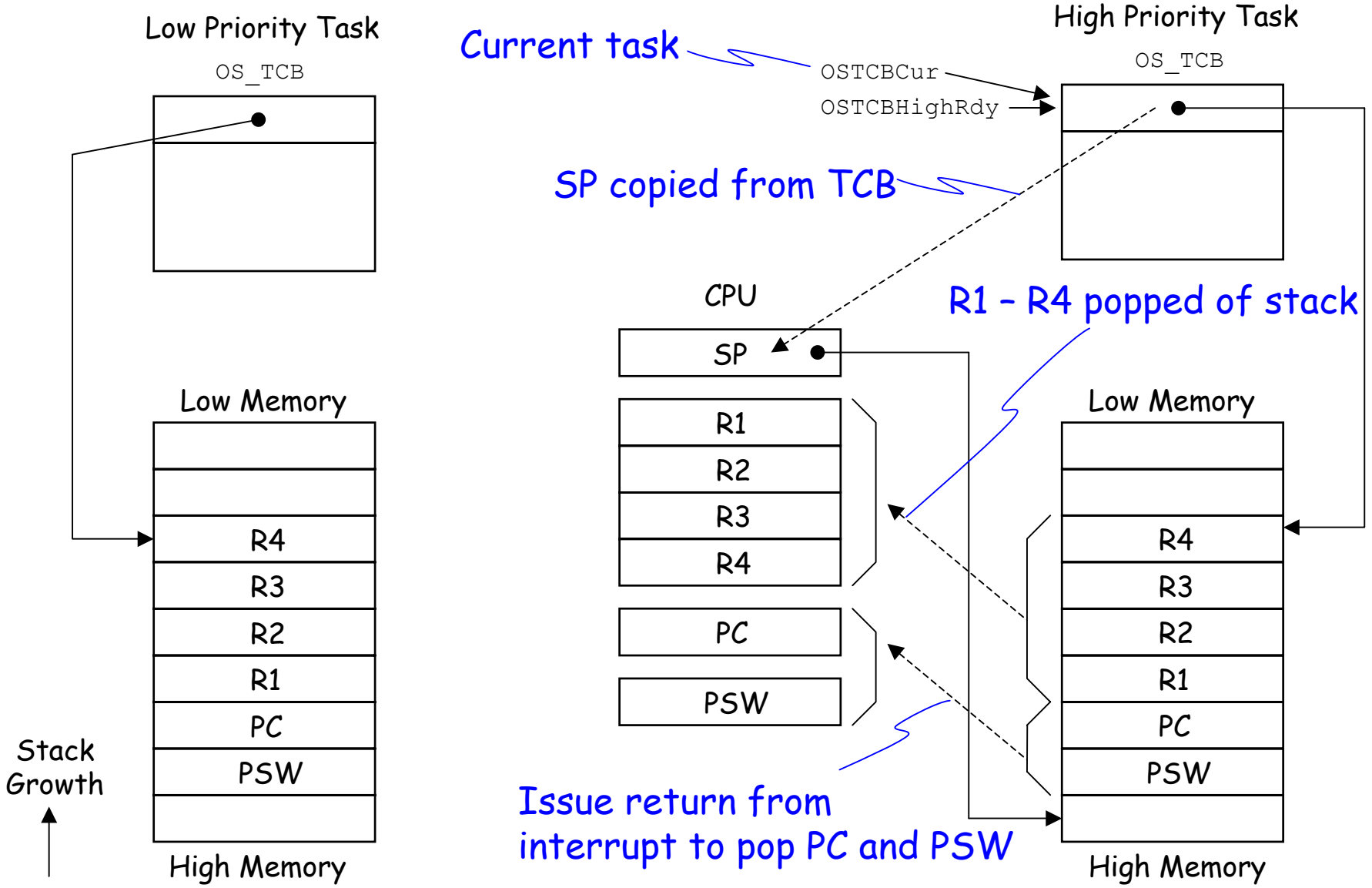


High Memory

High Memory

Software interrupt forces save of PC and PSW

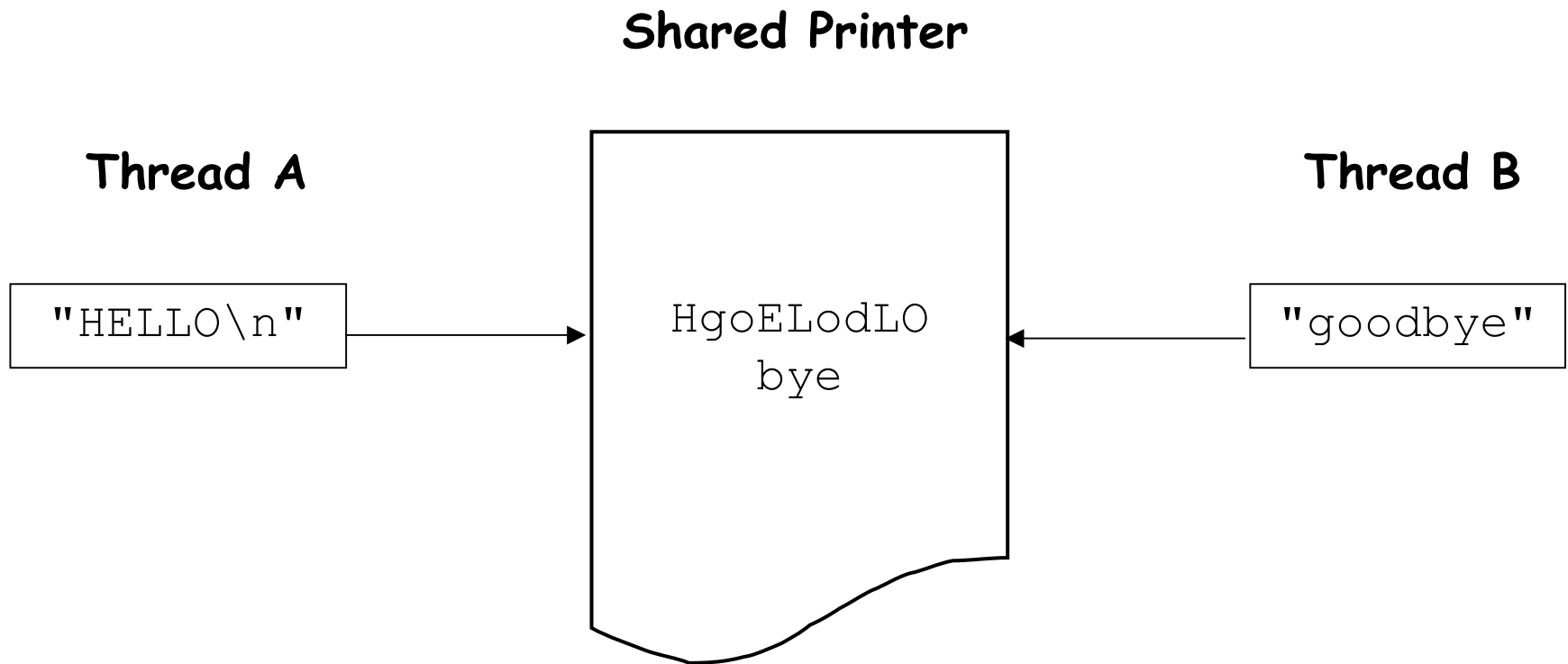
# How: Context Switch



# Shared Resources

- Tasks often need exclusive access to shared resources
- Examples include:
  - I/O devices
  - Memory
  - Network
  - etc.

# Uncontrolled Access to a Shared Resource (the Printer)



# Example

```
#include    "includes.h"
#define    TASK_STK_SIZE    512    // Stack size, in bytes
// Task stacks
OS_STK    Task1Stk[TASK_STK_SIZE];
OS_STK    Task2Stk[TASK_STK_SIZE];
// Task Function Prototypes
void Task1(void *pdata);
void Task2(void *pdata);

int SharedInteger;


int main()
{
    // Display a banner.
    printf("##### uCOS-II ELEC3730 Example 2\n");

    // Initialize uCOS-II.
    OSInit();

    // Create some tasks
    OSTaskCreate(Task1, (void *) NULL, &Task1Stk[TASK_STK_SIZE], 5);
    OSTaskCreate(Task2, (void *) NULL, &Task2Stk[TASK_STK_SIZE], 6);

    // Start multitasking.
    OSStart();

    /* NEVER EXECUTED */
    printf("main(): We should never execute this line\n");
}
```



Shared resource

# Example

```
void Task1(void *pdata)
{
    printf("(II) Task1 initialised\n");
    while (1)
    {
        SharedInteger = 1;
        OSTimeDly(50);
        printf("SharedInteger = %i\n", SharedInteger);
    }
}

void Task2(void *pdata)
{
    printf("(II) Task2 initialised\n");
    while (1)
    {
        OSTimeDly(75);
        SharedInteger = 2;
    }
}
```

Access to shared resource

Access to shared resource

# Demo Program

# Shared Resources

- Methods for obtaining exclusive access include
  - *Critical sections*: disable interrupts
  - *Disabling scheduler*: do not schedule any other tasks (ISR's can still run)
  - *Semaphores*: must obtain special key to access the resource

# Critical Sections

- Disable interrupts
  - no scheduling since no interrupts
- Perform read/write to resource
  - exclusive access guaranteed
- Enable interrupts
  - other tasks (and ISR's) can be scheduled (serviced)
- Micro-C/OS-II has two macros available
  - `OS_ENTER_CRITICAL()`
  - `OS_EXIT_CRITICAL()`

# Critical Sections

```
#define OS_ENTER_CRITICAL() asm {PUSHF; CLI} /* Disable interrupts */
```

```
#define OS_EXIT_CRITICAL() asm POPF /* Enable interrupts */
```

```
int Temp
```

```
void swap(int *x, int *y)
```

```
{
```

```
    OS_ENTER_CRITICAL();
```

```
    Temp = *x;
```

```
    *x = *y;
```

```
    *y = Temp;
```

```
    OS_EXIT_CRITICAL();
```

```
}
```

Interrupts disabled

Guaranteed execution order

Interrupts enabled

# Critical Sections

- Small overhead required
  - good for short critical sections
  - Micro-C/OS-II uses critical sections to access e.g. global variables
- System response time affected
  - long critical sections discouraged

# Disabling Scheduler

```
int Temp;
```

```
void swap(int *x, int *y)  
{
```

```
    OSSchedLock();
```

```
    Temp = *x;
```

```
    *x = *y;
```

```
    *y = Temp;
```

```
    OSShedUnlock();
```

```
}
```

Scheduler disabled

Execution order, but interrupts enabled

Scheduler enabled

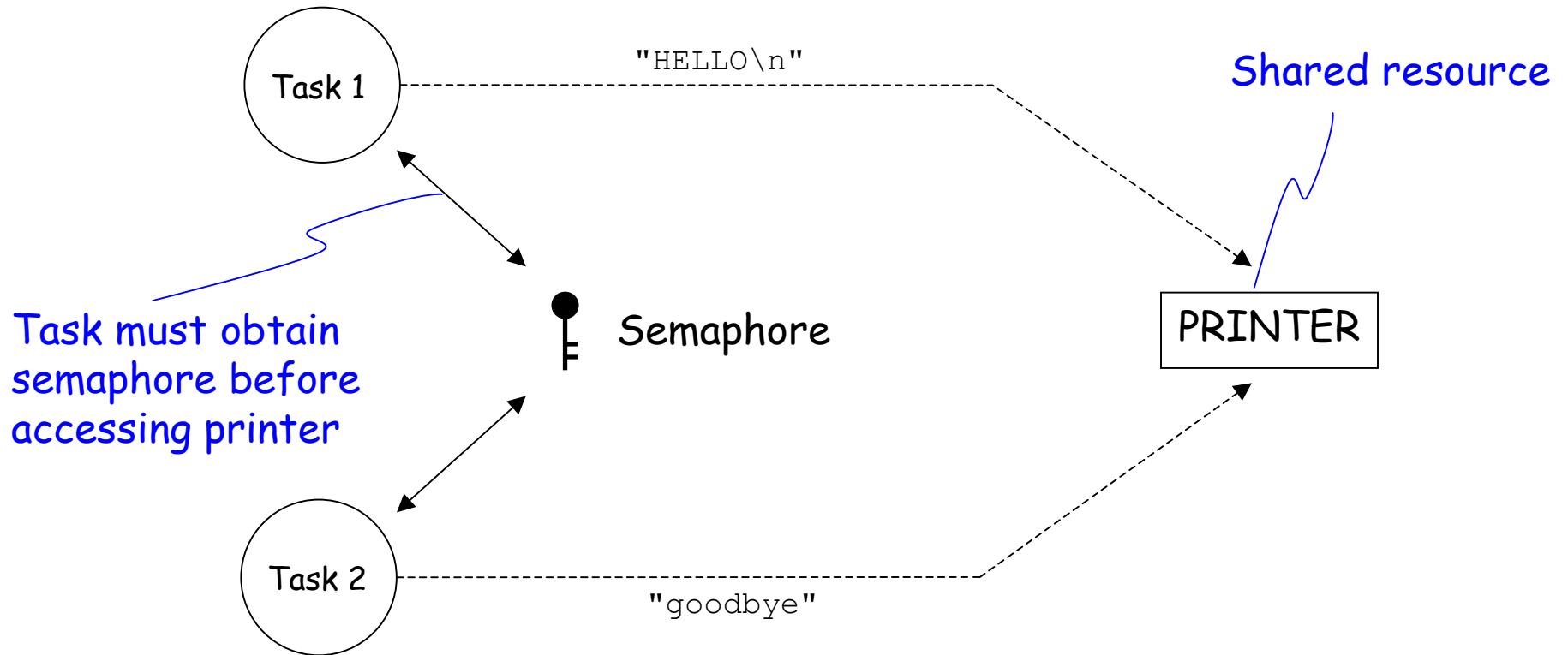
# Disabling Scheduler

- Useful if tasks and ISR's do not share a resource
  - disable scheduler
  - perform read/write operations
  - enable scheduler
- Interrupts are serviced immediately
  - better response time
  - execution is given back to interrupted task

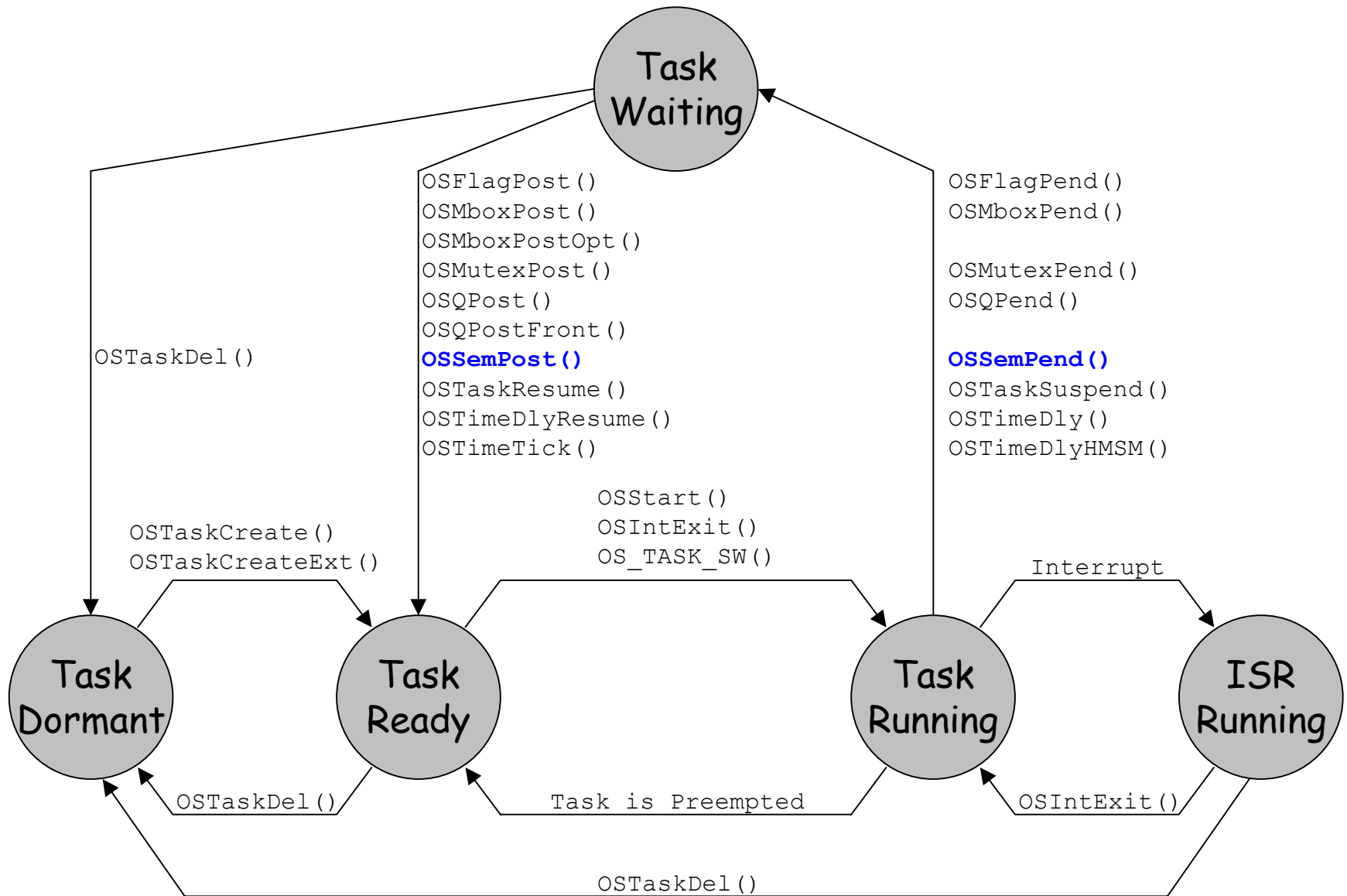
# Semaphores

- Concept been around since 1960's
  - task required to have a "key" before gaining access to resource
  - if "key" is unavailable, task is put on waiting list for that "key"
- Typically three operations with semaphores
  - create (or initialise)
  - pend (or wait)
  - post (or signal)

# Semaphores



# Task states: Semaphore Pend/Post



# Example revisited

```
#include    "includes.h"
#define    TASK_STK_SIZE    512    // Stack size, in bytes
// Task stacks
OS_STK    Task1Stk[TASK_STK_SIZE];
OS_STK    Task2Stk[TASK_STK_SIZE];
// Task Function Prototypes
void Task1(void *pdata);
void Task2(void *pdata);

OS_EVENT *SharedIntSem;

int SharedInteger;

int main()
{
    // Display a banner.
    printf("##### uCOS-II ELEC3730 Example 3\n");

    // Initialize uCOS-II.
    OSInit();

    // Create some tasks
    OSTaskCreate(Task1, (void *) NULL, &Task1Stk[TASK_STK_SIZE], 5);
    OSTaskCreate(Task2, (void *) NULL, &Task2Stk[TASK_STK_SIZE], 6);

    //Create a semaphore
    SharedIntSem = OSSemCreate(1);

    // Start multitasking.
    OSStart();
}
```

Declare global semaphore pointer

Create semaphore with count = 1

# Example revisited

```
void Task1(void *pdata)
{
    INT8U err;

    printf("(II) Task1 initialised\n");
    while (1)
    {
        OSSemPend(SharedIntSem, 0, &err);
        SharedInteger = 1;
        OSTimeDly(50);
        printf("SharedInteger = %i\n", SharedInteger);
        OSSemPost(SharedIntSem);
    }
}
```

Wait here for  
SharedIntSem  
semaphore

Signal release of  
SharedIntSem  
semaphore

```
void Task2(void *pdata)
{
    INT8U err;

    printf("(II) Task2 initialised\n");
    while (1)
    {
        OSSemPend(SharedIntSem, 0, &err);
        OSTimeDly(75);
        SharedInteger = 2;
        OSSemPost(SharedIntSem);
    }
}
```

Wait here for  
SharedIntSem  
semaphore

Signal release of  
SharedIntSem  
semaphore

# Demo Program