Spring 2022
ECE568:
Software for Embedded Systems

Lecture #5: Embedded Software Architectures

Vijay Raghunathan
School of ECE, Purdue University
vr@purdue.edu

(c) 2022, All rights reserved.

Outline

- Single Program Approach (SPA)
- Foreground-Background Systems
- Multi-tasking Systems
  - Event driven architectures
  - Thread based architectures
- An anatomy of two simple embedded operating systems
- How can we organize / schedule computation in these various architectures?

A Simple Embedded System

- As an example, let's consider a simple embedded system with a keypad, an LCD, and an RS-232 port that runs some communications. The system also has some GPIO and a parallel printer. Each change of state of an input or output results in an RS-232 message sent out and an LCD update. Received RS-232 messages can result in printouts, LCD updates, and output status updates. We may have to start a flash pattern on a particular LED as a result of:
  - An input or output becoming active
  - Keypad entry
  - Communications message received

[Melkonian00]

Single Program Approach:


```
int main(void)   {
   Init_All();
   for (;;) {
      IO_Inputs_Scan();
      KBD_Scan();
      RS232_Receive();
      IO_Process_Outputs();
      LCD_Update();
      RS232_Send();
      PRN_Print();
   Maintenance_Process();
   }
  // should never get here; put error
  // handling here, just in case
  return (0);
}
```

[Melkonian00]

Observations on the SPA

- Essentially an infinite loop
  - Each function called in the loop represents an independent task
- Each of these tasks must return in a reasonable time, no matter what code is being executed (who ensures this?)
- We have no idea at what frequency our main loop runs
  - The frequency is not constant and can significantly change with changes in system status (e.g., displaying a large image, printing a long document)
- Mix of periodic and event-driven tasks
  - Most tasks are event driven (have dedicated input event queues)
    - e.g., IO_Process_Outputs receives events from RS232_Receive, IO_Inputs_Scan, and KBD_Scan when an output needs to be turned on
  - Others are periodic in nature
    - No trigger event, but may have different (time-varying) periods

Observations on the SPA (contd.)

- Need some simple means of inter-task communications
  - e.g., may want to stop scanning the inputs after a particular keypad entry and restart the scanning after another entry
    - require a call from the keypad scanner to stop the I/O scanner task
  - e.g., may also want to slow down the execution of some tasks depending on the circumstances
    - say we detect an avalanche of input state changes, and our RS-232 link can no longer cope with sending all these messages

Observations on the SPA (contd.)

- Usually, each task in the loop checks for the occurrence of asynchronous events using “polling”
  - i.e., must explicitly check if an event has occurred or not
  - Depending on result of the poll, task may do different things (e.g., if keyboard input received, then process it. If no keyboard input, just return)
  - Adds to timing uncertainty of the infinite loop
  - Not very resource efficient

Foreground/Background (FB) Sys.

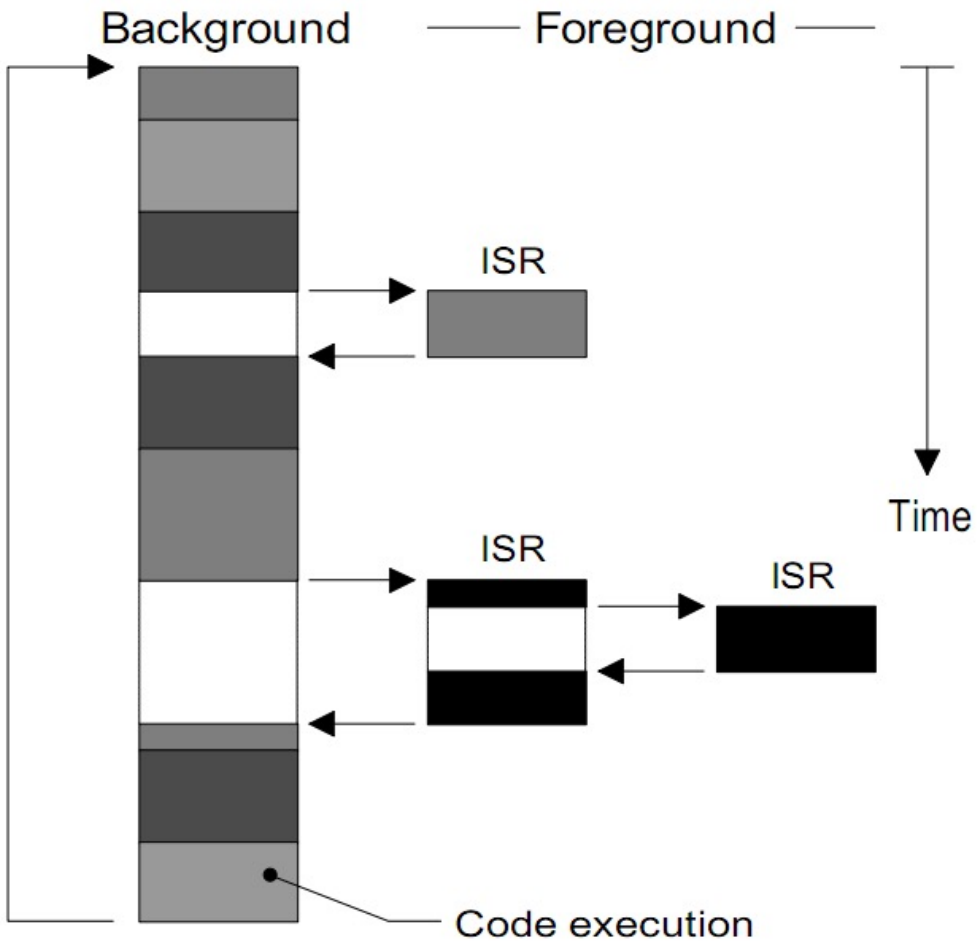
- Break workload into two levels
  - Critical tasks (tight timing constraints) form the foreground while other tasks (best-effort) form the background

```
Background:
  while True:
    if (time for display update):
      do display
    else if (time for operator input):
      do operator
    else if (time for mgmnt. request):
      do mgmnt.

Foreground (interrupt):
  on interrupt
    do clock module
    if (time for control):
      do control
```

- Decoupling relaxes timing constraint

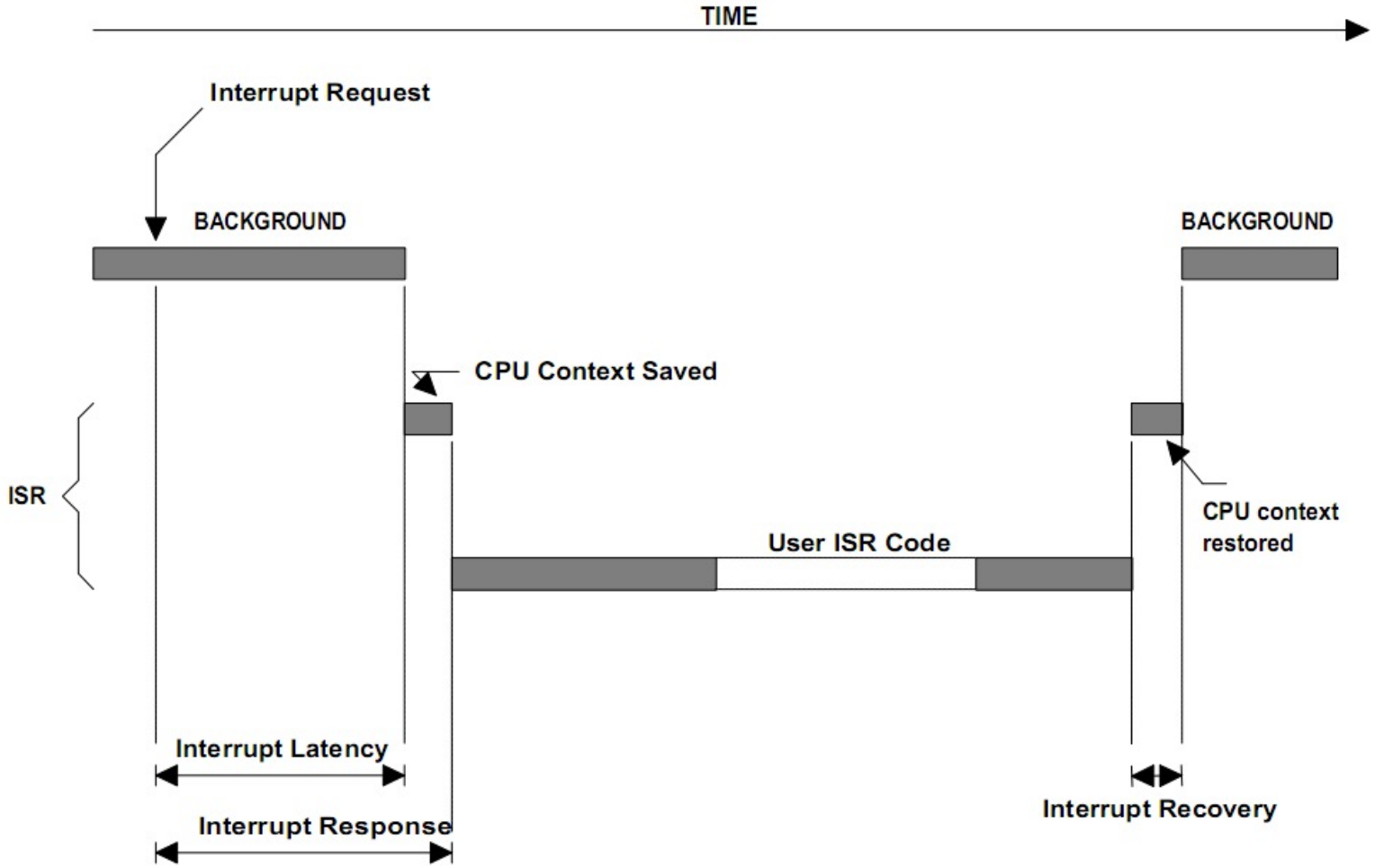
Interrupts in FB Systems

- Task-level response time depends on how fast the background loop runs and when interrupt occurs



Interrupt-related Timing Metrics

- Disabling interrupts affects interrupt latency
- Context switch time depends on number of registers



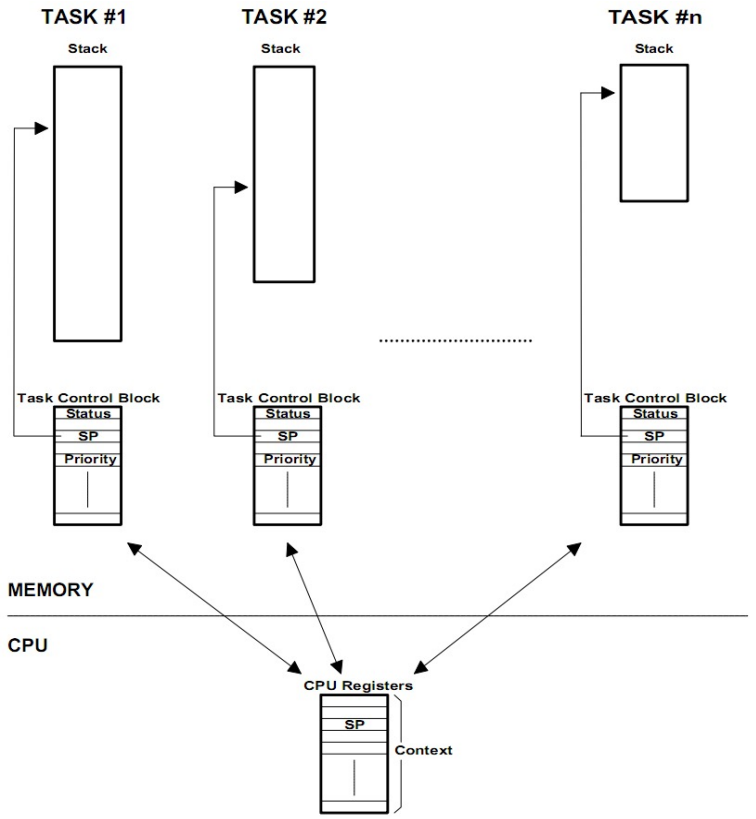
Multi-tasking Approach

- Single program approach: one “task”
- Foreground/background: two tasks (or levels)
- Generalization: multiple tasks (processes, threads, etc.)
  - Time share the CPU between these tasks
  - Similar to FB system with multiple backgrounds
  - Each task can be carried out in parallel
    - Tasks simultaneously interact with external elements
      - Monitor sensors, control actuators, interrupts, I/O, etc.
    - Makes writing applications easier
  - Requires
    - Scheduling of these tasks (When? Which? How?)
    - Sharing data between concurrent tasks

Multi-tasking Approach

- Context Switch: Process of switching execution from one task to another
  - Context of a task represents snapshot of everything needed to restart the task later from where it was stopped
- When done fast, creates the illusion of concurrency

Context usually consists of processor registers (including program counter, stack pointer)

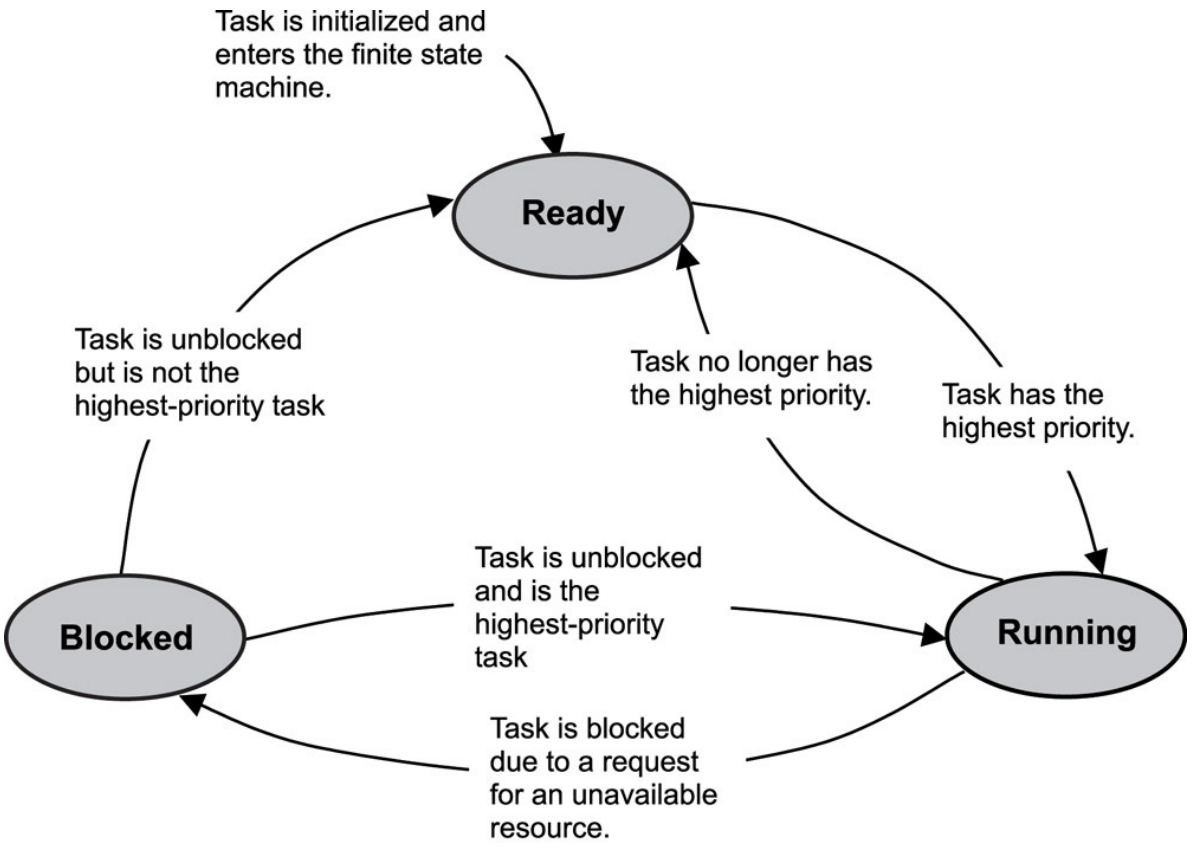


Task Characteristics

- New instances of tasks may be activated over time
  - Periodic vs. aperiodic (sporadic) activation
  - Also called “jobs”
- Tasks may have:
  - Resource requirements (e.g., need X clock cycles of CPU time)
  - Importance levels (e.g., priorities or criticality)
  - Precedence relationships
  - Communication requirements
  - Of course, timing constraints!
    - specify time at which task is to be performed, or is to be completed
    - e.g., period of a periodic task, deadline of a task

Task States

- A task can be in one of several states
  - Inactive: Not active or dormant
  - Ready to run: Waiting for CPU
    - Maintained in a ready queue
  - Running: Currently in execution
  - Blocked: Waiting for some event or shared resource (not CPU)
  - Interrupted: Has been interrupted to service an interrupt



Who manages the multiple tasks?

- Application tasks should not have to worry about
  - What happens to other tasks
  - Low-level hardware access
  - Coordinating the use of shared resources
- Have a layer of software to handle all of these and hide them from application tasks
  - Called by many names: operating system (OS), kernel, runtime, etc.
  - Provides execution environment for application tasks and various services through a well-defined interface (API)
  - Real-time OS (RTOS): Emphasis is on satisfying timing constraints (to construct real-time systems)

Separation of concerns: Let tasks focus on functionality and let the OS take care of orchestrating task execution

How to schedule multiple tasks?

- Cyclic executive model
  - Simple, static schedulability analysis
  - Resulting schedule or table can be used at run time
  - Time Division Multiple Access (TDMA)-like scheduling
- Event-driven model
  - Tasks are represented by functions that are ***handlers for events***
  - Next event processed after function for previous event finishes
- Thread-based model
  - Preemptive vs. non-preemptive
  - Priority based scheduling: *Static* vs. *dynamic* priority
  - Often, static schedulability analysis can be done
  - No explicit schedule constructed offline. at run time tasks are executed “highest priority first”

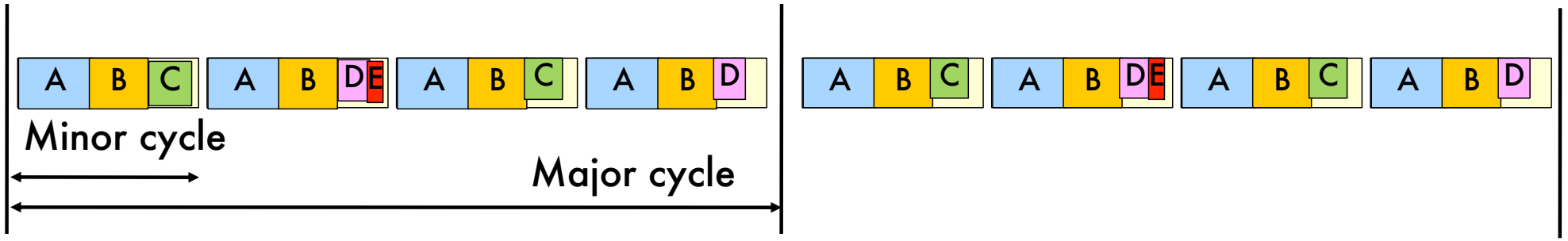
Cyclic Executive Scheduling

- Applicable to tasks that are periodic in nature
  - For aperiodic tasks, use worst case inter-arrival time
- Basic approach: time slices of equal length organized into major and minor cycles
  - Major cycle = LCM, Minor cycle = GCD
  - A timer with period = minor cycle interrupts a scheduler, which activates the task according to order in major cycle
  - Sum of executions within a time slice ≤ minor cycle
  - May need to split tasks
  - Handling deadlines may be difficult

Cyclic Executive Example

Task	Time Period (P)	Exec. time (C)
A	25 ms	10 ms
B	25 ms	8 ms
C	50 ms	5 ms
D	50 ms	4 ms
E	100 ms	2 ms

- Major cycle = LCM (25, 25, 50, 50, 100) = 100ms
- Minor cycle = GCD (25, 25, 50, 50, 100) = 25ms



Cyclic Executive Example


```
start_up;

loop
        wait_for_minor_cycle_begin;

        procedure_for_A;
        procedure_for_B;
        procedure_for_C;

        wait_for_minor_cycle_begin;

        procedure_for_A;
        procedure_for_B;
        procedure_for_D;
        procedure_for_E;

        wait_for_minor_cycle_begin;

        procedure_for_A;
        procedure_for_B;
        procedure_for_C;

        wait_for_minor_cycle_begin;

        procedure_for_A;
        procedure_for_B;
        procedure_for_D;

end loop;
```

Cyclic Executive Scheduling

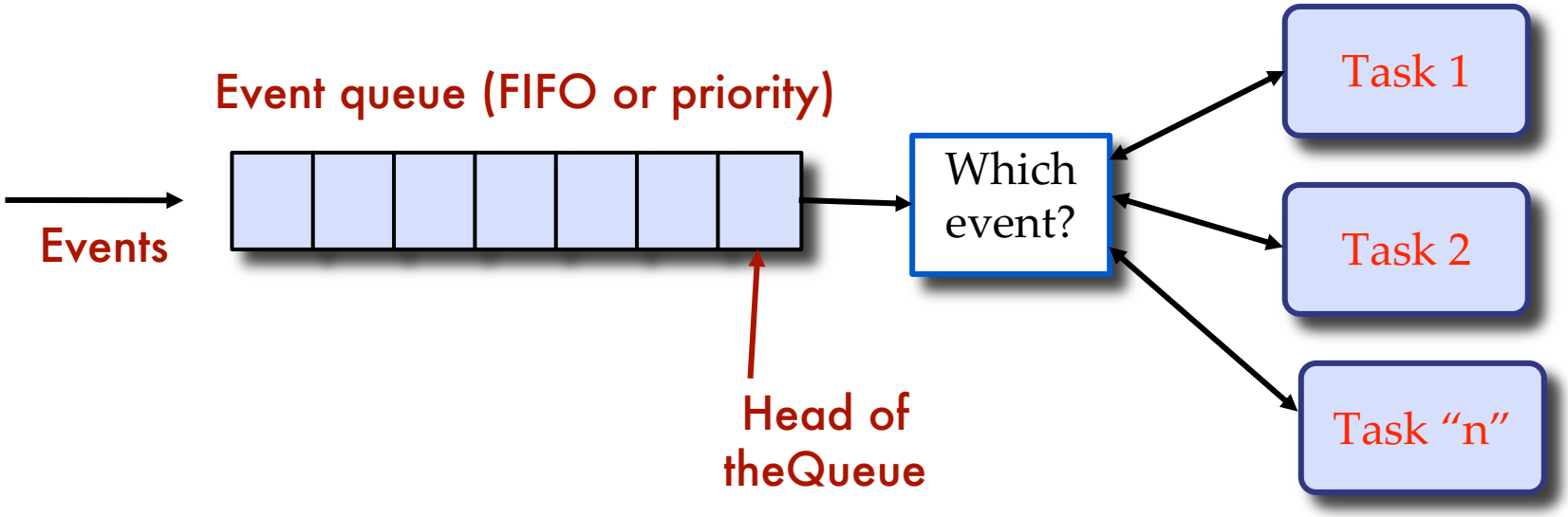
- Advantages
  - Simple to implement
  - Predictable: sequence of tasks is always same
  - No context switches
- Disadvantages
  - Fitting code segments into slots
    - Time consuming iterative process
  - Inflexible, sensitivity to application changes
    - Table is completely overhauled when tasks or their characteristics change
    - Task splitting may change
  - Fragile during overload conditions
    - If a task doesn't finish on time, both letting it continue or aborting it have problems (disruption to scheduling vs. inconsistent system state)
  - Modification to add functions can be expensive in redesign time > fragile design

Outline

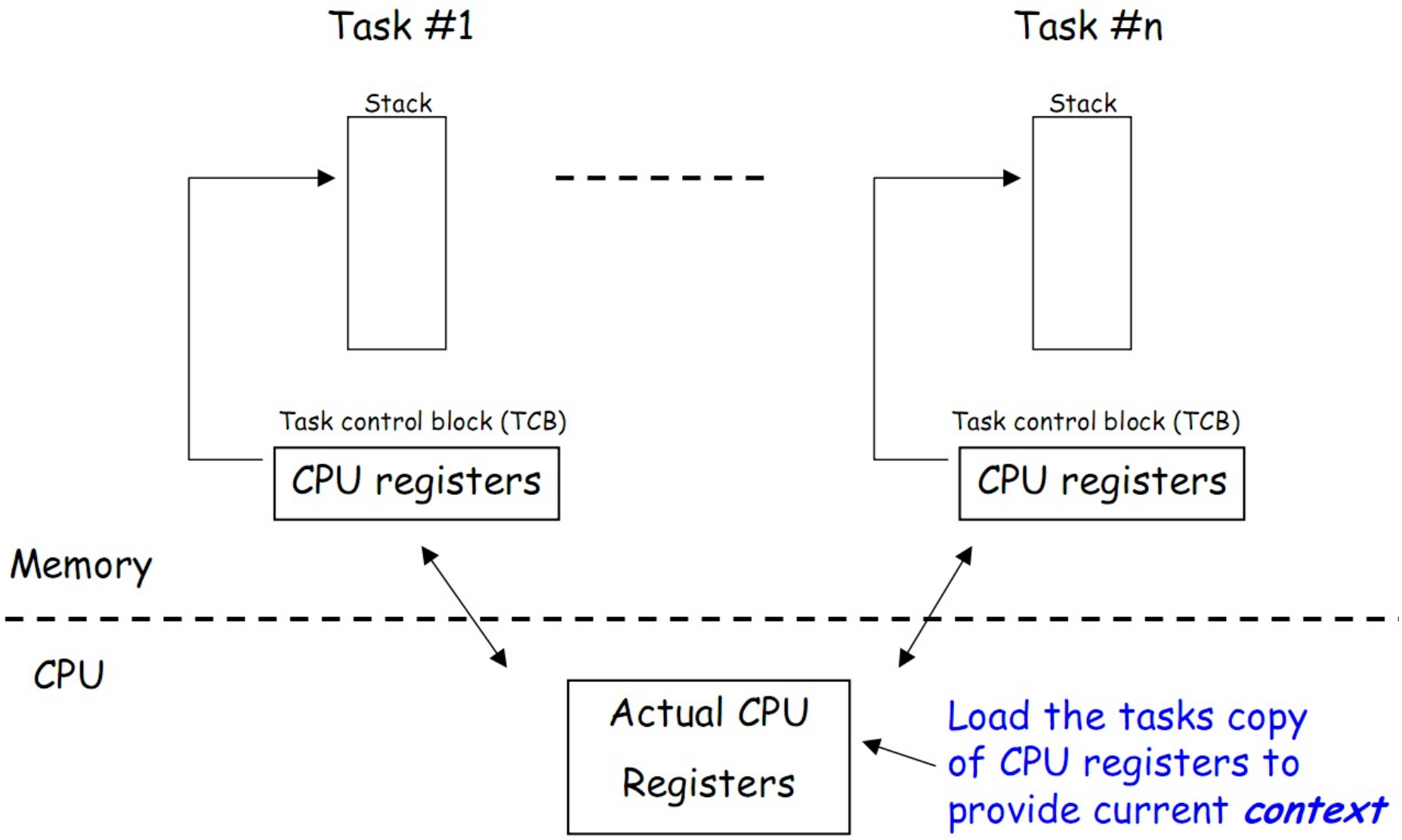
- Single Program Approach (SPA)
- Foreground-Background Systems
- Multi-tasking Systems
  - Event driven architectures
  - Thread based architectures
- An anatomy of two simple embedded operating systems
- How can we organize / schedule computation in these various architectures?

Event-driven Scheduling

- System consists of tasks that respond to events
  - Events can be interrupts or software events (e.g., messages from other tasks)
- Kernel is a main loop that does the following
  - Check if event has occurred. If so, select an event to process
  - Handle the event (i.e., call task that corresponds to the event)
- Each task is an "event handler" function
  - Programmers need to create these handlers and bind them to events



Recap: Multi-tasking model



Preemption

- Preemption: Once a task starts, can it be stopped to execute another task before it completes?
  - Interrupts (when not disabled) can always stop tasks
- Non-preemptive systems: task, once started, runs until it ends or has to do some I/O
  - Lower number of context switches
- Preemptive system: task may be stopped to run another
  - Incurs overhead and implementation complexity
  - But has better schedulability and easier analysis

Illustration: Non-preemptive system

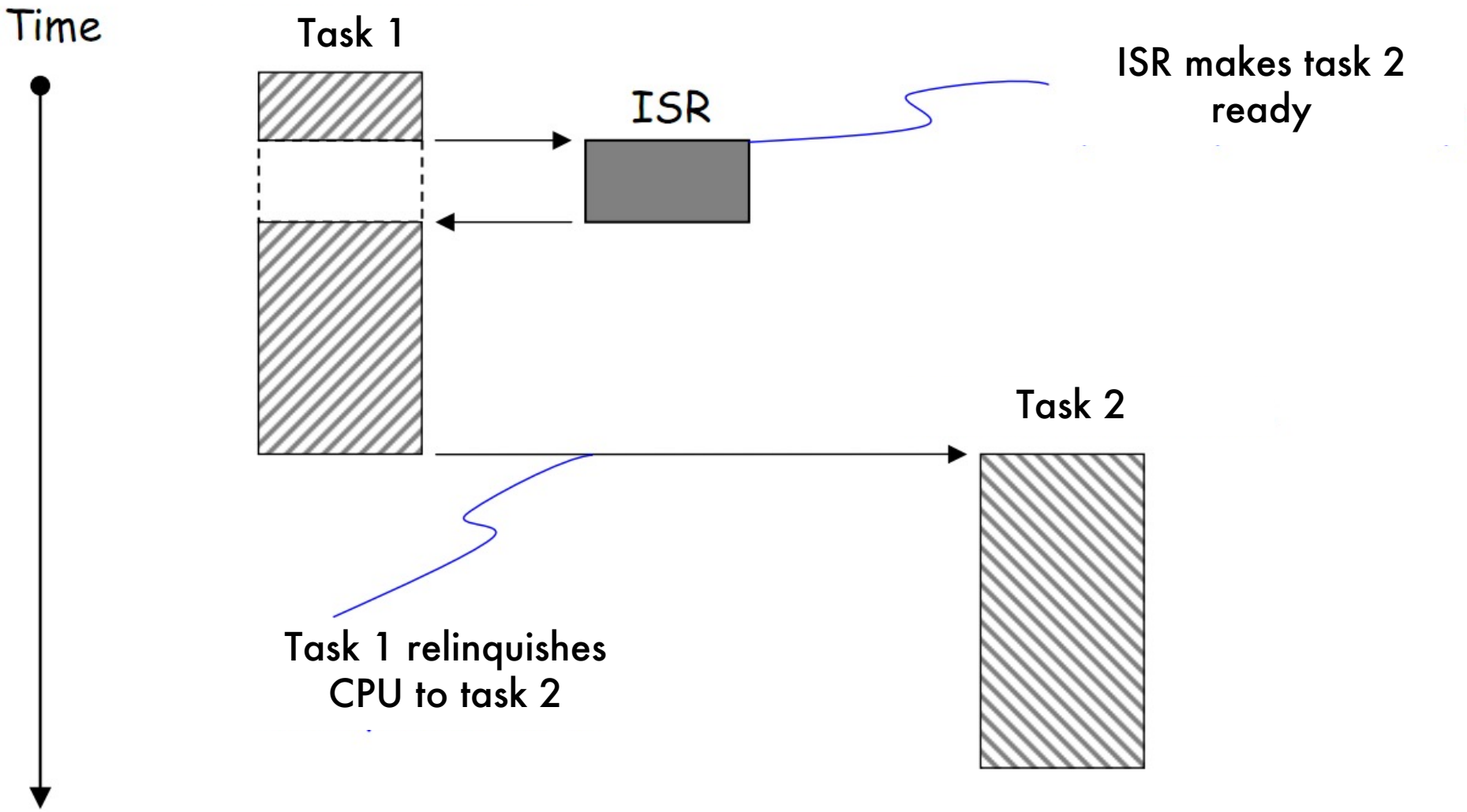
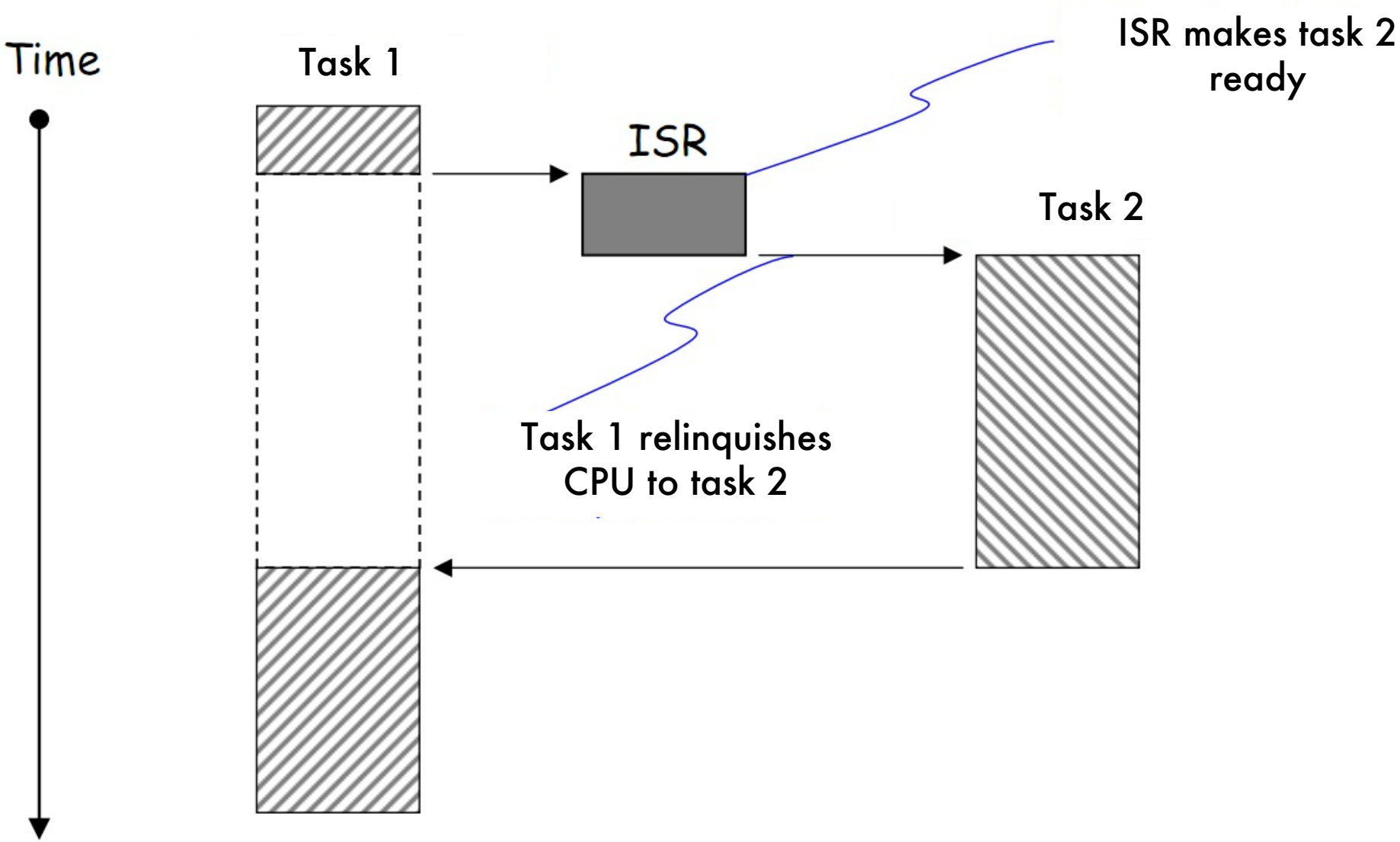


Illustration: Preemptive system



Priority-based Scheduling

- Tasks assigned priorities, statically or dynamically
  - Priority assignments relate to timing constraints
  - Static priority attractive… no recalculation, cheap
- Can be preemptive, non-preemptive, or time-sliced
- Appropriate assignment of priorities enable satisfaction of timing constraints
- Example
  - Task 1: Time Period = 2s, Requires 1s of CPU; Deadline = 2s
  - Task 2: Time Period = 5s, Requires 2s of CPU; Deadline = 5s
  - Consider: Priority1 > Priority2
  - Consider: Priority 2 > Priority 1
  - Which works?

Some Priority-based Approaches

- Rate Monotonic (RM) algorithm
  - Static priorities based on task periods
    - higher priorities to shorter periods
    - Optimal among all static priority schemes
- Earliest Deadline First (EDF) algorithm
  - Dynamic priority assignment
  - Closer a task's deadline, higher is it priority
  - Applicable to both periodic and aperiodic tasks
  - Optimal among all scheduling schemes for 1 CPU
  - Need to calculate priorities when new tasks arrives
    - more expensive in terms of run-time overheads
- For both, key results on schedulability exist

Embedded and Real Time OSs

- Usually one of four varieties
  - Small, fast, proprietary kernels
  - Real-time extensions to commercial OSs
  - Research OSs
  - Part of language run-time environments, e.g., Java (embedded, real-time Java)
- Mainly differ with respect to the following main issues
  - Concurrency model: event, thread, scheduling, etc.
  - Memory model: static vs. dynamic
  - Component model, if any

Some RTOS Examples

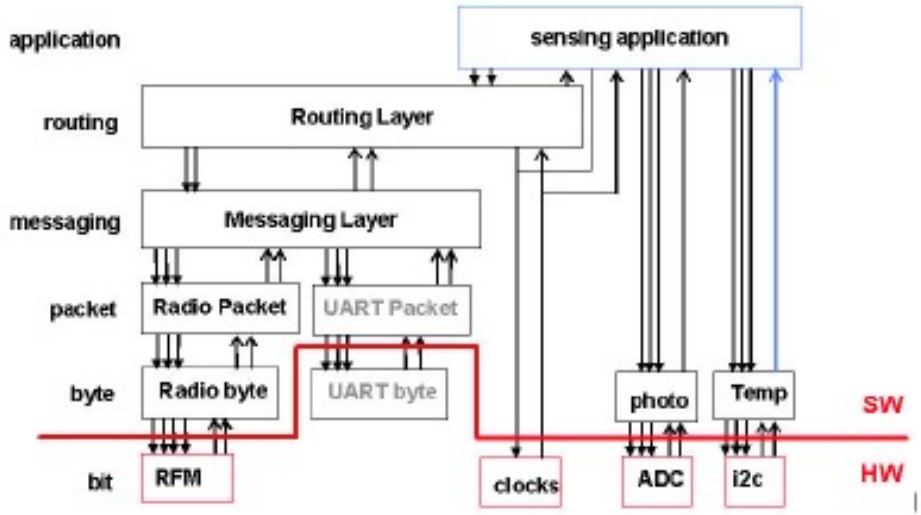
- For tiny systems
  - TinyOS, SOS, Contiki, NuttOS, ERIKA, FreeRTOS
- For mid-size systems
  - µCOS-III, eCos, OSE
  - Embedded Linuxes: µCLinux
  - Inferno
- For large-size systems
  - Windriver VxWorks
  - QNX Neutrino
  - RTEMS
  - Real-time Linuxes: RTLinux, RTAI, Linux/RK
  - Research kernels: SHARK, MARTE

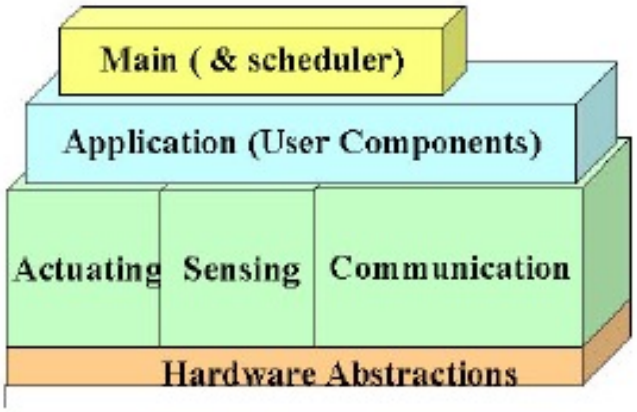
Next: Two RTOS examples

- A tour through the internals of two contrasting OSs
  - TinyOS
    - Event driven model
  - uC/OS-II
    - Thread based model

Example: TinyOS

- Dominant OS for resource-constrained wireless sensor nodes
- Not really an OS in the traditional sense
  - More like an application-specific operating system
  - Application = Scheduler + Graph of Components
- Event-driven programming model
- Static memory allocation, program analysis



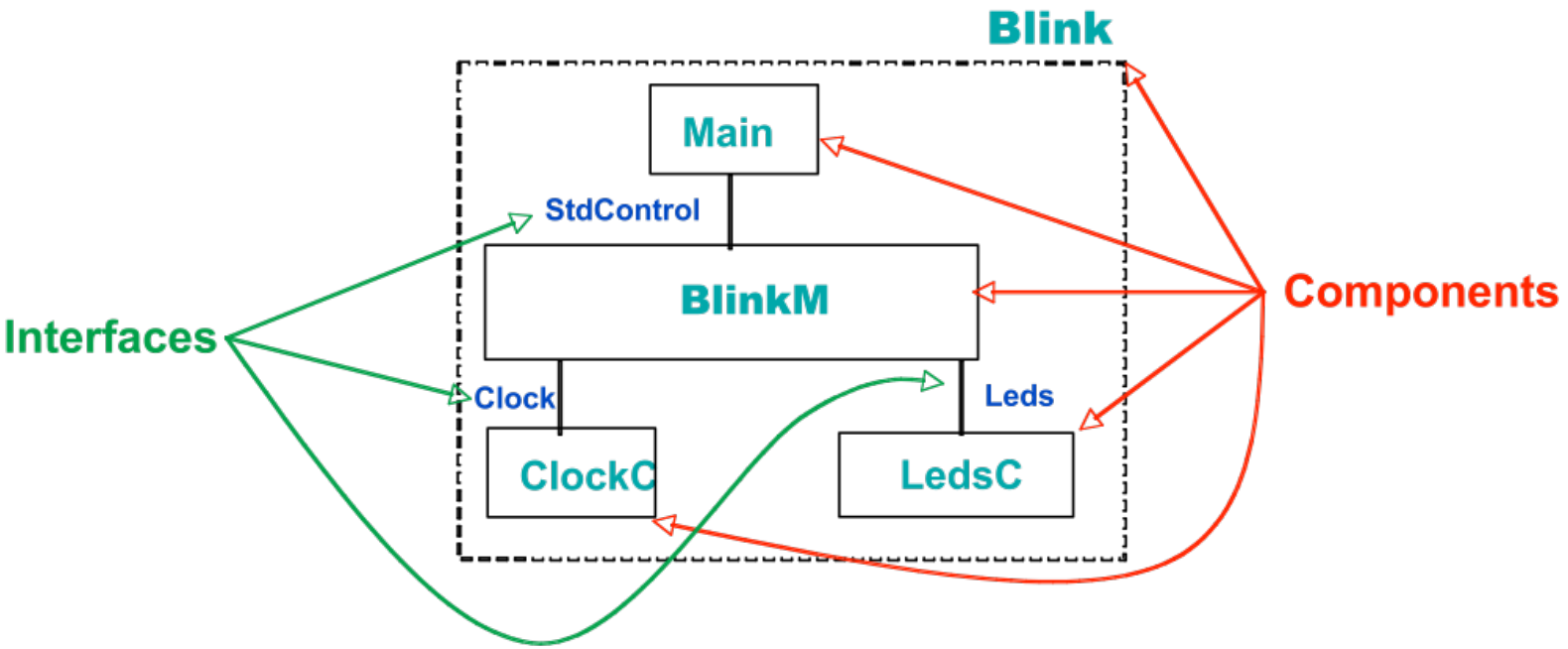


Building Blocks of TinyOS

- Application consists of one or more components linked together to form an executable
- Two types of components
  - Modules: Implement functionality
  - Configurations: Just wire up other components
- Wiring (linking up) is done using interfaces
  - Every component provides and uses interfaces
  - Interfaces are the only points of access to a component
- Interfaces consist of two sets of functions
  - Commands: Providers of interfaces implement these functions
  - Events: Users of interfaces implement these functions
- A component may use multiple interfaces of the same or different type

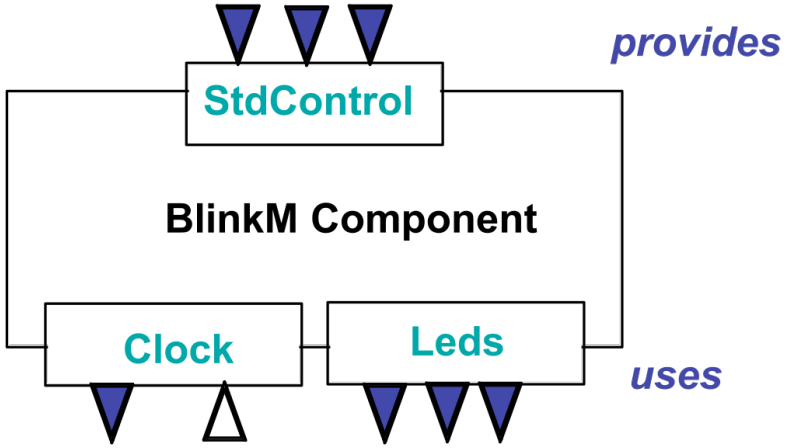
Simple Application: Blink LEDs

- Blink the red LED at 1Hz
- Application contains 5 components and 3 interfaces



Module Component

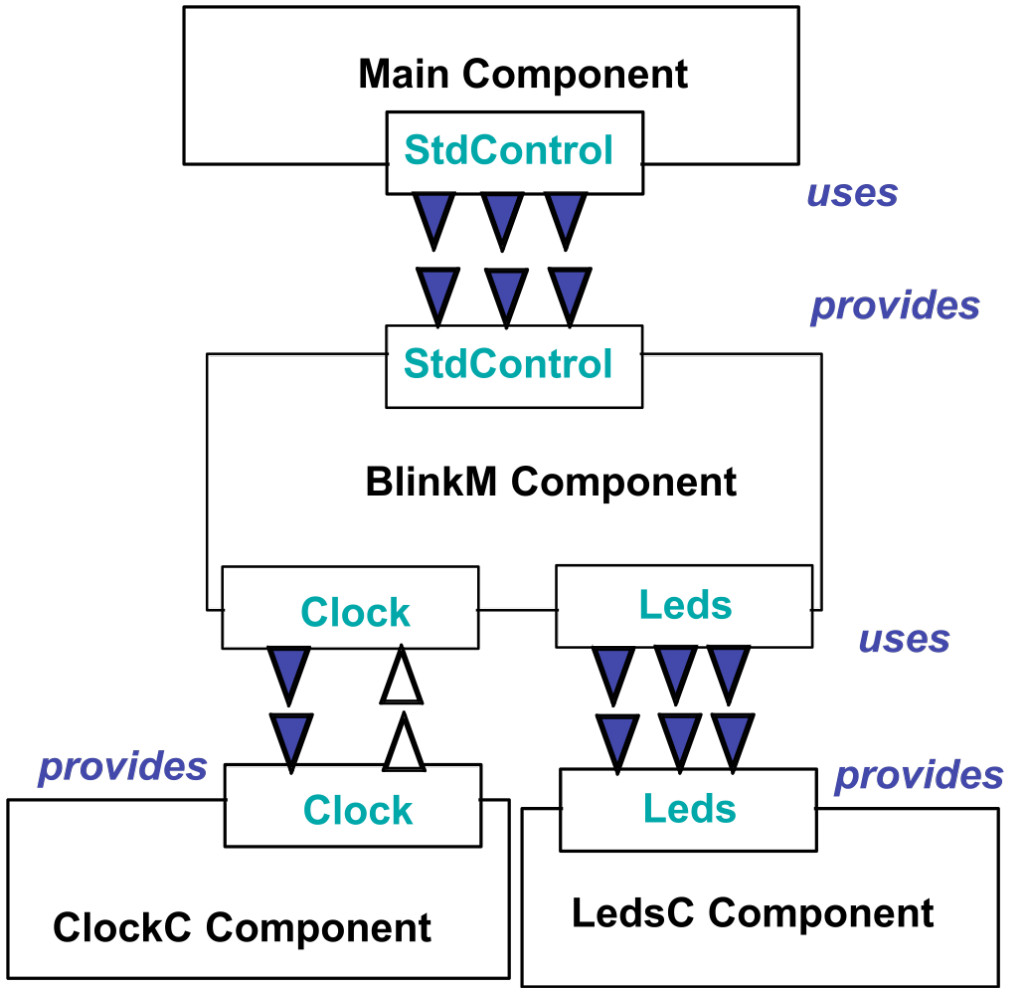

```
module BlinkM {
provides {
    interface StdControl; }
uses {
    interface Clock;
    interface Leds; }
}

implementation {
  bool state;
  command result_t StdControl.init() {
    state = FALSE;
    return SUCCESS;
  }
  command result_t StdControl.start() {
    return call Clock.setRate(TOS_I1PS, TOS_S1PS);
  }
  command result_t StdControl.stop() {
    return call Clock.setRate(TOS_I0PS, TOS_S0PS);
  }
  event result_t Clock.fire()  {
    state = !state;
    if (state)
      call Leds.redOn();
    else
      call Leds.redOff();
    return SUCCESS;
  }
```



- Component is implementing functions
- Component is NOT wiring other components together
- Therefore, component is a module

The Complete Picture

```
Configuration Main {
        uses {
 interface StdControl;}
        }
   module BlinkM {
       provides {
 interface StdControl;}
        uses {
    interface Clock;
    interface Leds;}
        }
Configuration ClockC {
       provides {
    interface Clock;}
        }
Configuration LedsC {
       provides {
    interface Leds;}
        }
```



TinyOS Concurrency Model

- Event-driven execution
  - Non-preemptive run-to-completion
- Two contexts
  - Foreground - Commands, Events
  - Background - Tasks
  - Background context can be interrupted by foreground

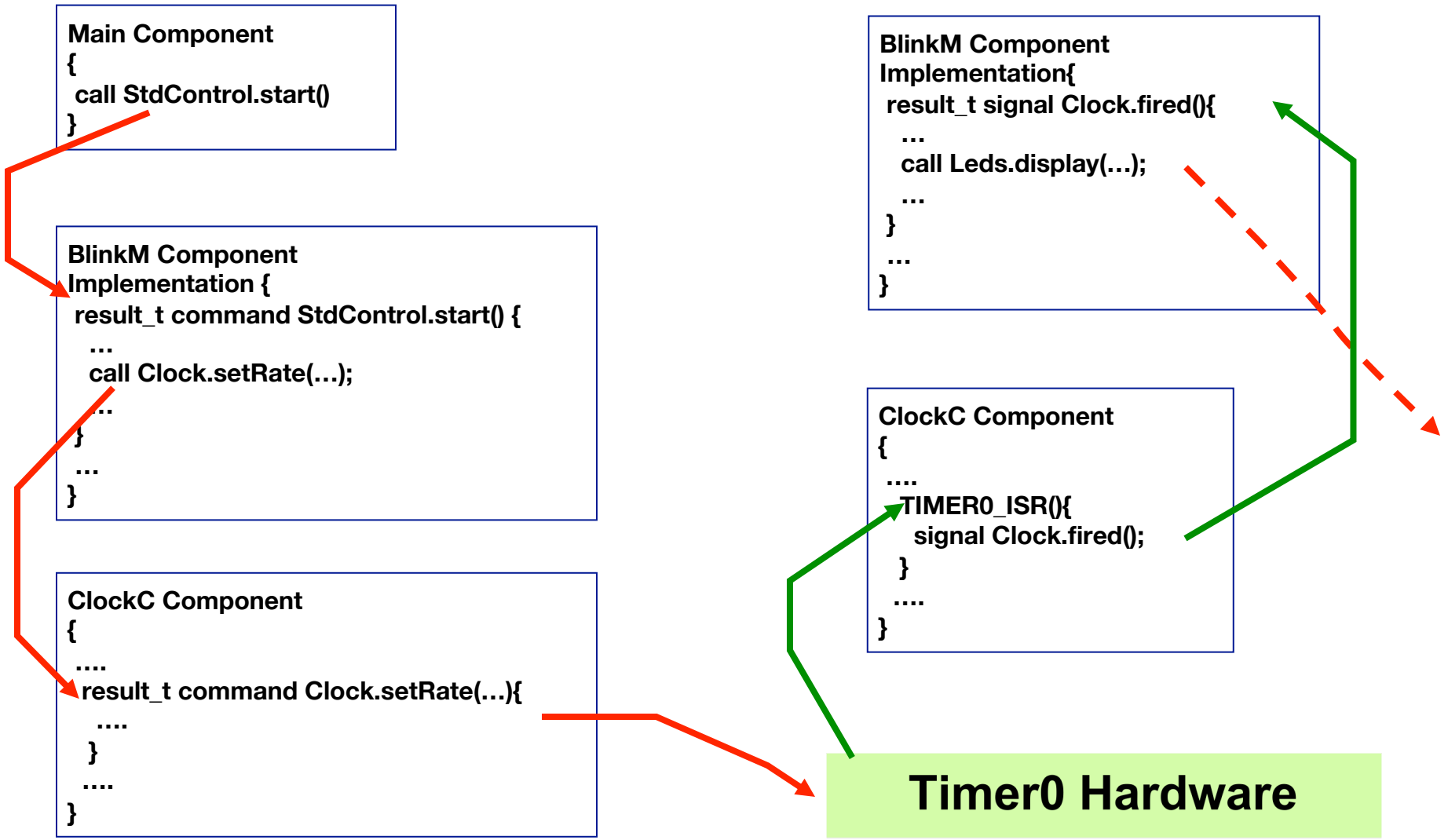
Clock.nc

```
interface Clock {
  event result_t fire();
  command result_t setRate(char interval,
char scale);
}
```

StdControl.nc

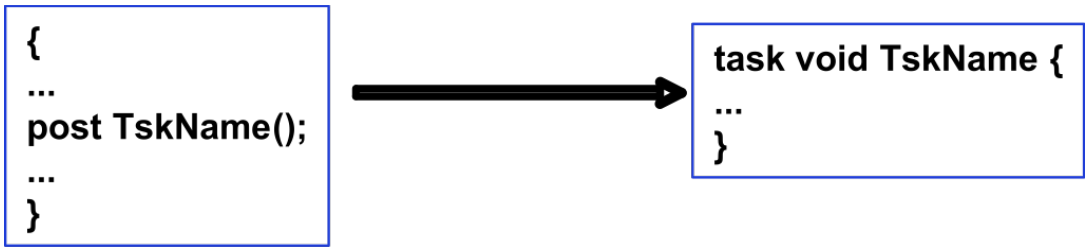
```
interface StdControl {
  command result_t init();
  command result_t start();
  command result_t stop();
}
```

Understanding Blink Execution

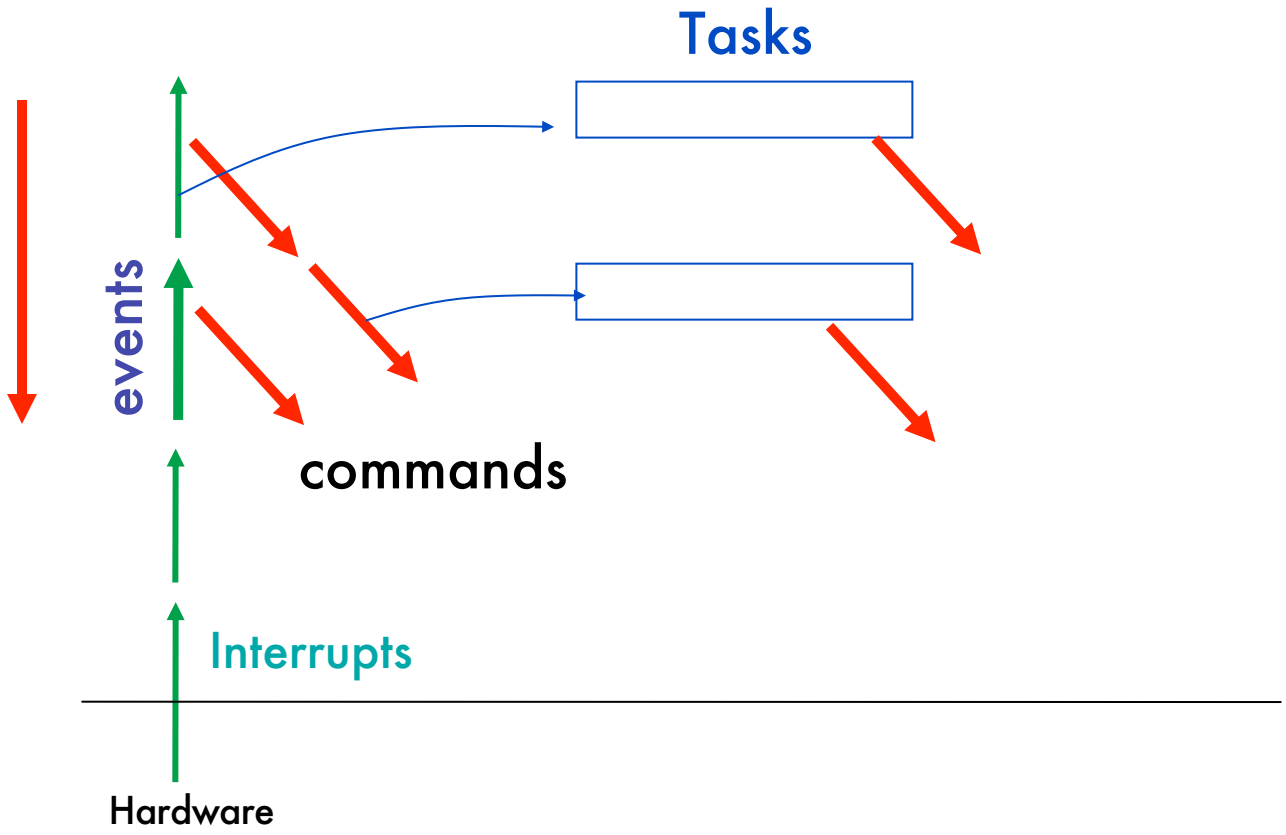


Tasks

- To provide concurrency internal to a component
  - longer running operations (background)
- Can be preempted by events
- Able to perform operations beyond event context
  - e.g., process/transform data that is sampled
- May call commands
- May signal events
- Not preempted by other tasks
  - Run to completion model



TinyOS Execution Contexts



- Events generated by interrupts preempt tasks
- Tasks do not preempt tasks

TinyOS: Pros and Cons

- Pros
  - Static memory allocation => resource guarantee
  - Non-preemptive scheduling => minimal memory requirement
  - Modular nature => independent software development
- Cons
  - A new programming language
  - Static memory allocation => resource over subscription
  - Not really an OS
    - No resource management
  - Simple concurrency model
  - Monolithic firmware image hard to reconfigure dynamically

Example 2: µC/OS-II

- Portable, scalable, preemptive, multitasking RTOS
- Provides several services to applications
  - Semaphores, mutexes, event flags, mailboxes, message queues, task management, fixed-size memory block management, time management
- Source freely available for academic non-commercial usage for many platforms
  - Written almost entirely in C with some assembly
- In 2000, certified by Federal Aviation Administration (FAA) for use in avionics systems

µC/OS-II task model

- Users write applications in the form of tasks

- Each task is an infinite loop that runs on the CPU when “conditions” are satisfied and the kernel schedules it

```
void Task()
{
  //Initialisation code here

  while (1)
  {
    //Perform specific task here
  }
}
```

Task states

- Dormant
  - Exists in memory, but not known to kernel
- Ready
  - Task is in queue to use the CPU
- Running
  - Task has control of CPU and is executing
- Waiting
  - Task is suspended/blocked, waiting for something to happen

Simple Example

```
/*********************************
 * uC/OS-II Win32 Example Program *
 *********************************/

// Include ucosII header file
#include    "includes.h"

// Define stack size for each task
#define  TASK_STK_SIZE     512

// Define 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);
```

MicroC/OS-II specific header

Stack size

Each task has a stack

Two task (function) prototypes

Simple Example

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

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

Normal function declaration

Enter infinite loop

Ask kernel to delay this task

Different delay function

Simple Example

Normal old main routine

```
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");
}
```

Initialise the kernel (or OS)

Let kernel create TCB

Start the kernel running

What happens after OSStart()?

- Kernel picks highest priority task and starts running it
  - Changes its state to “Running”
- When task asks kernel to delay its execution
  - Task goes into the “Waiting” state
  - Each clock “tick” event is captured by the kernel
    - Using a timer or clock ISR
  - At each tick, kernel decrements counter for each task and if the counter == 0, task state is changed to “Ready” state
  - If a task is highest priority in the “Ready” state, it gets selected for execution
  - Runs till the next time it asks kernel to delay it

Back to Example 1


```
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

Back to Example 1

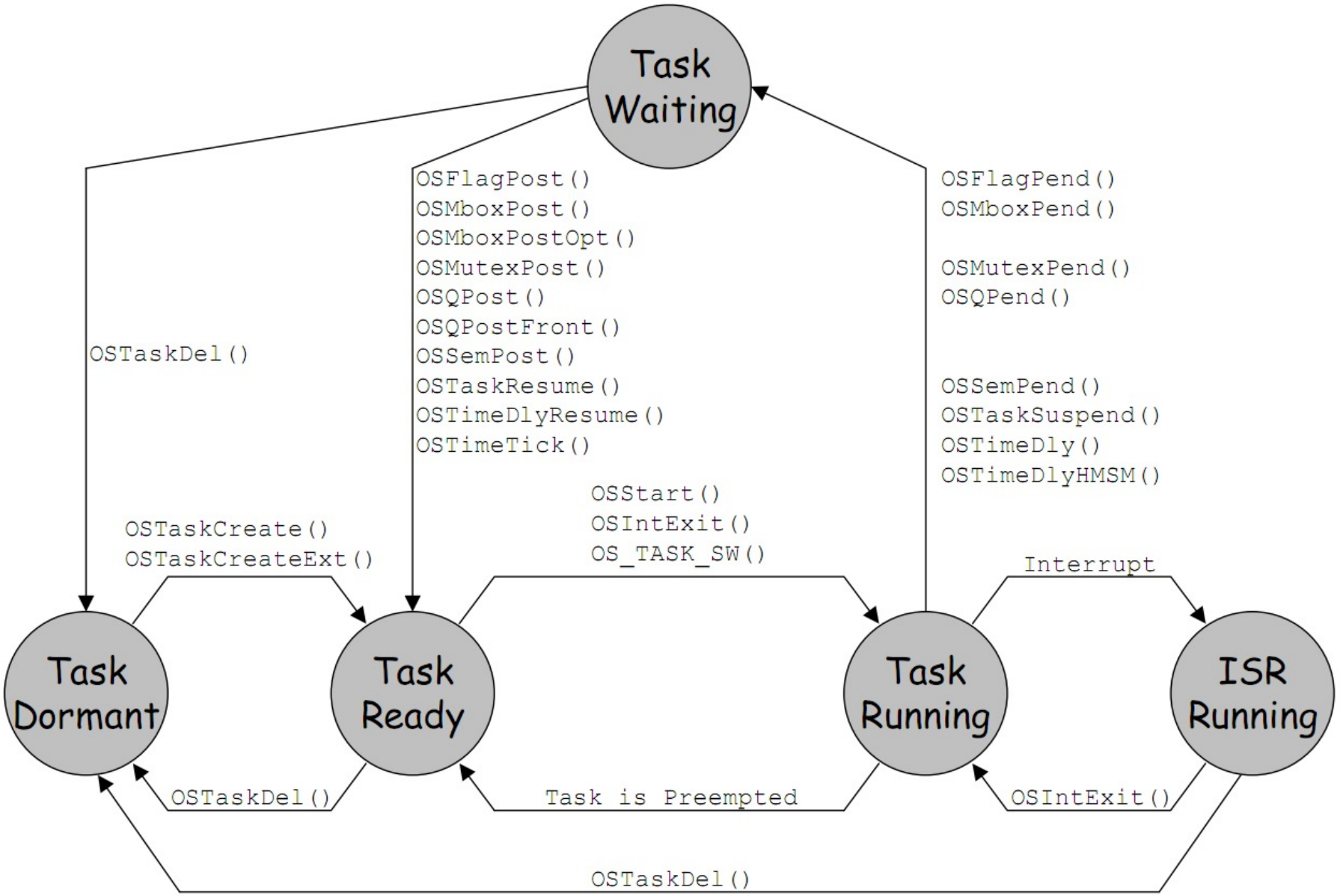

```
/*************************
 * First Task            *
 *************************/
void Task1(void *pdata)
{
  printf("(II) Task1 initialised\n");
  while (1)
  {
    OSTimeDly(100);
    printf("1 ");
  }
}

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

Kernel moves task from running to waiting state

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

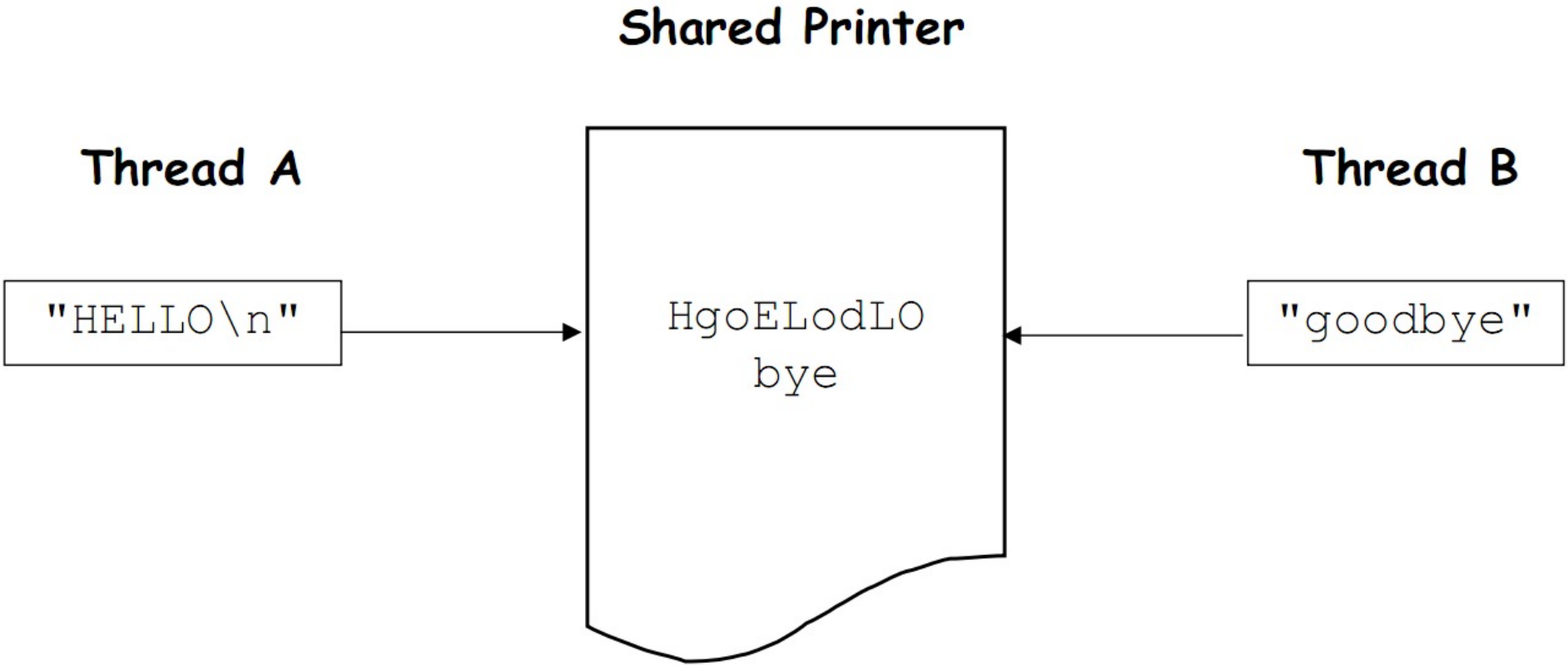
Kernel moves task from running to waiting state

Tasks migrating between states



How about shared resources?

- Tasks often need exclusive access to shared resources
  - e.g., I/O devices, memory, network, etc.



Illustrating the problem in uC/OS-II


```
#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

Illustrating the problem in uC/OS-II


```
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

Dealing with shared resources

- Three widely used methods to obtain exclusive access to a shared resource
  - Critical sections: Interrupts are disabled during these
  - Disable scheduling: Do not schedule any other tasks (ISRs can still run)
  - Semaphores: Tasks must obtain a special key/token before accessing the shared resource

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

Disabling Scheduling


```
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

- ISRs can still execute
- Useful if resource is not shared between tasks & ISRs
- Same task resumed after ISR even if higher priority tasks exist

Semaphores

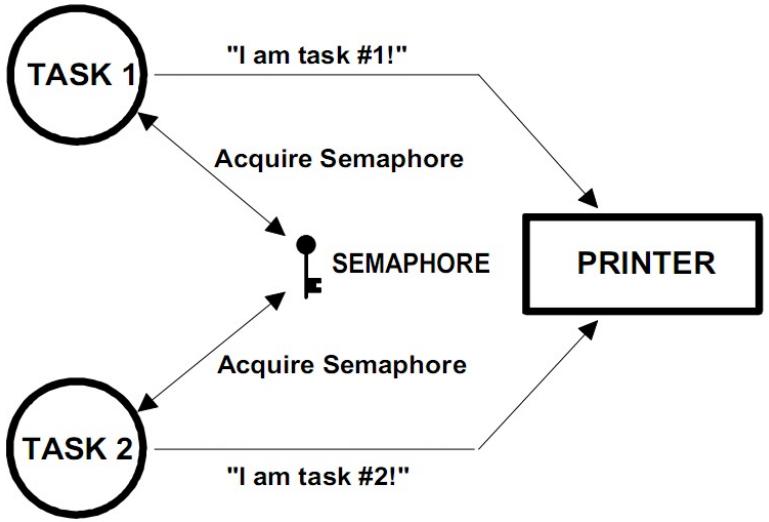
- Invented by Edgar Dijkstra in 1960s
- Elegant mechanism to
  - Control access to a shared resource (mutual exclusion)
  - Signal the occurrence of an event
  - Allow two tasks to synchronize their activities
- Semaphore is a key (permit) that a task has to acquire in order to use a resource
- If some other task is using the key, a task will wait till the key is available (placed into the “Blocked” state)
- Two types of semaphores
  - Binary semaphores (0 or 1)
  - Counting semaphores (range depends on no. of bits used)

Semaphore Primitives

- Can perform three operations on a semaphore
  - Initialize (also called Create) – provides initial value
  - Wait (also called Pend)
  - Signal (also called Post)
- If a task wants semaphore, it will perform a Wait (Pend)
  - If value > 0, semaphore value is decremented and task runs
  - If value == 0, task is blocked
- Can specify a timeout
  - If semaphore not available within a timeout, move task to “Ready” state and signal an error to the task
- Task releases semaphore by calling Signal (Post)
  - If no task waiting on semaphore, increment value
  - If task waiting, give semaphore to waiting task

Semaphores in uC/OS-II

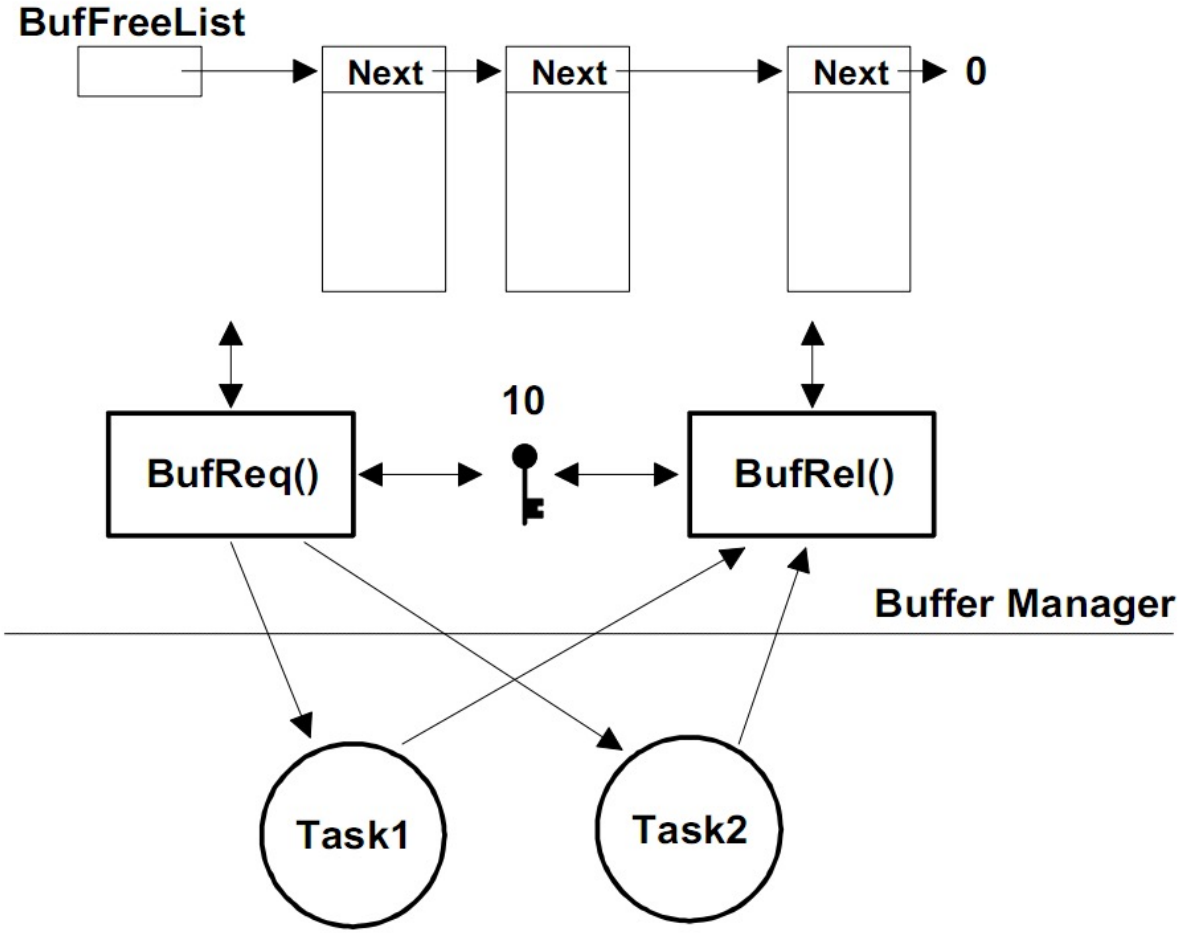

```
OS_EVENT *SharedDataSem;

void Function (void)
{
INT8U err;

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



- Task needs to know about semaphore existence

Counting Semaphores Example

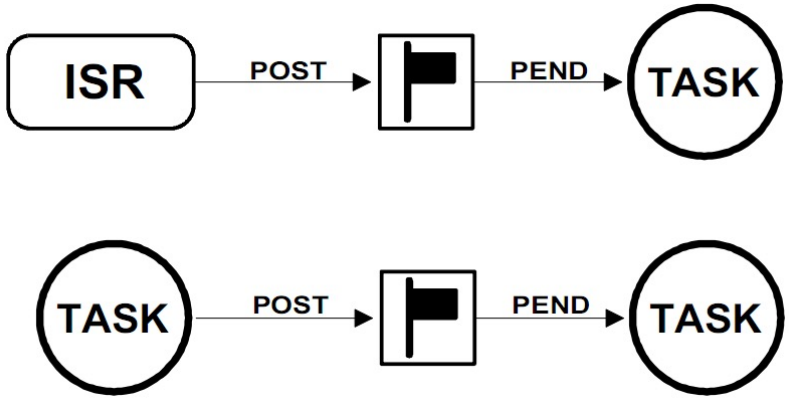
- Buffer management: You have a buffer pool (linked list) of 10 buffers that you want to give to tasks
  - Often used in networking protocols to store packets




```
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;
}
```

Synchronization

- A task can be synchronized with and ISR or another task, using a semaphore



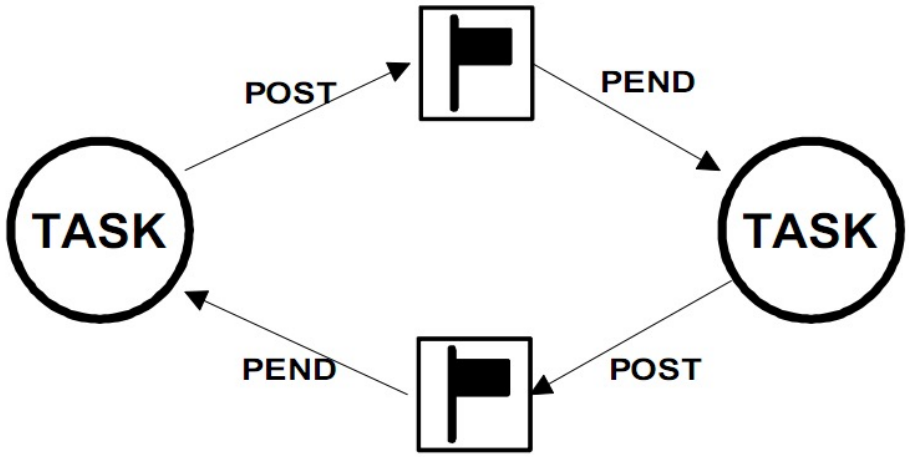
Unilateral rendezvous

- Task initiates I/O to a peripheral (data packets to radio) and waits. An ISR or task signals the semaphore when I/O is complete and the task continues
- Initialize semaphore to 0, not 1
- More than one task may signal occurrence of event
- More than one task may be waiting for event

Bilateral Rendezvous

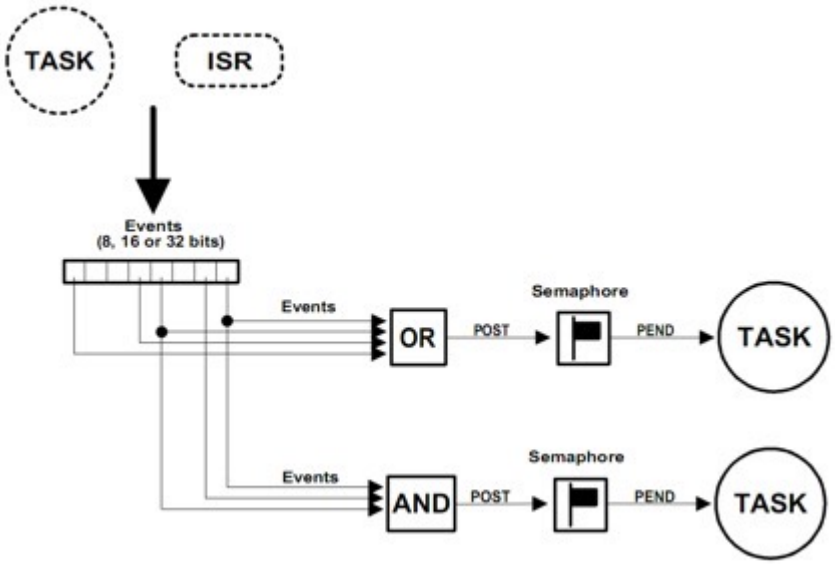
- Similar to unilateral rendezvous, but in both directions
  - Both tasks must synchronize with each other to continue
- Should not be used between task and ISRs

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


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



Event Flags

- Task may be waiting for multiple events to take place
  - To synchronize with different events
- Disjunctive synchronization (logical OR)
  - When any of the events have taken place
- Conjunctive synchronization (logical AND)
  - When all of the events have taken place
- Tasks can set, clear, or wait for event flags

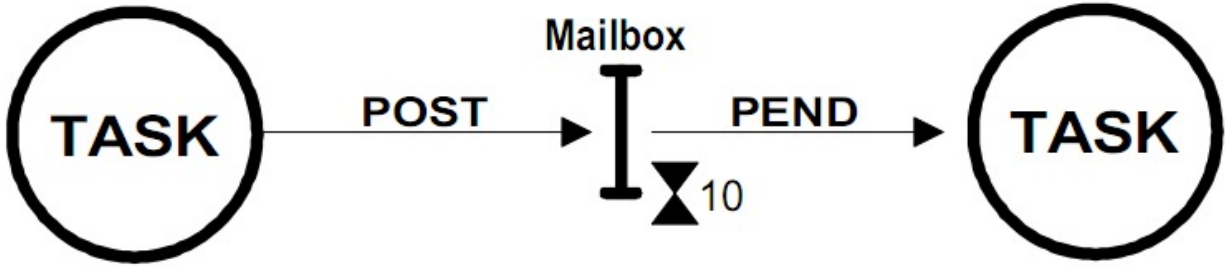


Intertask Communication

- Passing data (information) between ISRs/tasks can be done using global variables and data structures
  - Need to ensure mutual exclusion
  - ISR disabling or semaphore
- How does a task know that ISR has changed some global variable?
  - Option 1: Make sure ISR posts semaphore that task waits on
  - Option 2: Have task poll regularly to see if value changed
  - Option 3: Use a message mailbox or message queue

Message Mailboxes

- Usually, a pointer size variable
- Task/ISR can deposit a message (pointer) into mailbox
- One or more tasks can receive message from mailbox
  - Waiting list if multiple tasks wait for message in mailbox
  - Either priority or time used to arbitrate delivery
- Receivers of messages can specify timeouts
- Sender and receiver agree on what message represents
- Typical calls provided: Deposit, Wait, Accept



Reentrant Functions

- Reentrant function is one that can be called again before it completes
  - Multiple tasks can use function without data corruption
- Reentrant functions use only local variables or protect global variables when used

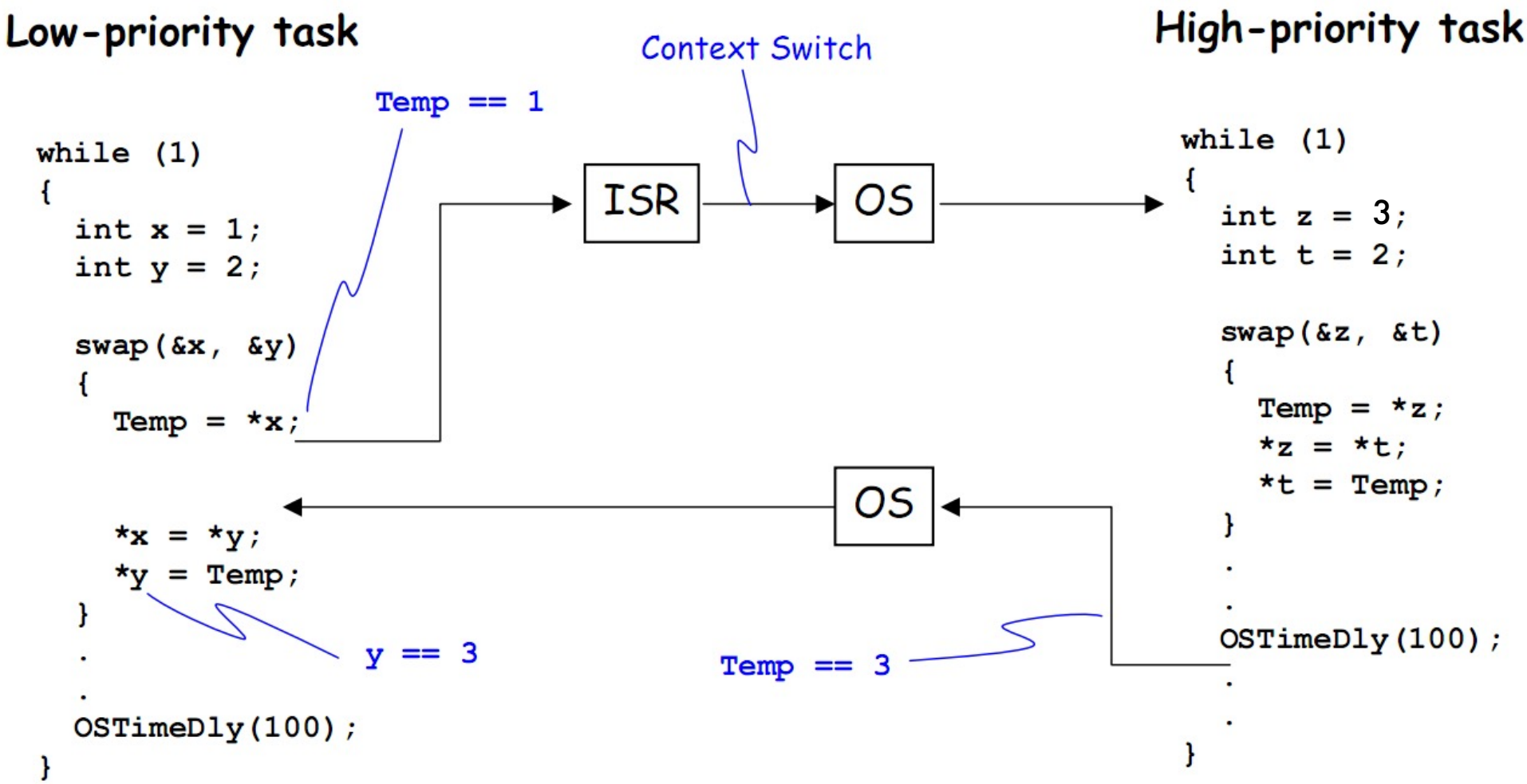
```
void strcpy(char *dest, char *src)
{
  while(*dest++ = *src++)
  {
    :
  }
  *dest = NULL;
}
```

Reentrant

```
int Temp

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

Non-Reentrant

Reentrant Functions



Summary

- Software organization
  - Single program approach
  - Foreground background approach
  - Multitasking
    - Event driven non-preemptive
    - Preemptive multitasking
- Examples of RTOSs and standards
- Two RTOS internals: TinyOS and *µc*/OS
- Dealing with shared resources, task communication