CENG 383

Real-Time Systems

Lecture 2

General Concepts of RTOS (Real-Time Operating System)

Asst. Prof. Tolga Ayav, Ph.D.

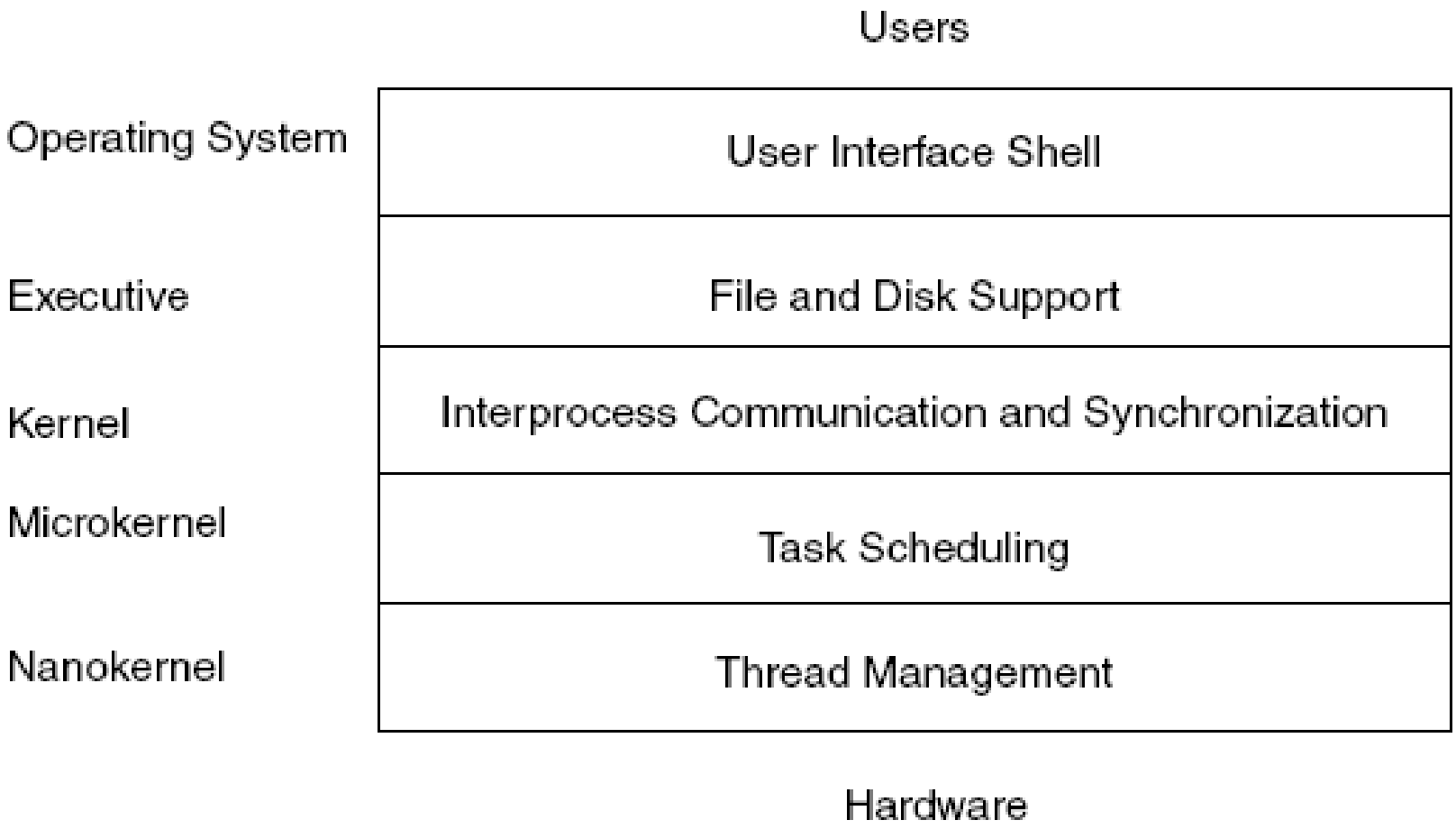
Department of Computer Engineering

İzmir Institute of Technology

Operating System

- Specialized collection of system programs is called operating system.
- Must provide at least three specific functions:
  - Task scheduling
  - Task dispatching
  - İntertask communication
- Kernel (Nucleus): The smallest portion of OS that provides those three essential functions.

Role of the Kernel



Three Kernel Functions

OS Kernel - 3 functions:

- **Task Scheduler :** To determine which task will run next in a multitasking system
- **Task Dispatcher:** To perform necessary bookkeeping to start a task
- **Intertask Communication:** To support communication between one process (i.e. task) and another

Kernel Design Strategies

- *Polled Loop Systems*
- *Cyclic Executive*
- *Cooperative Multitasking*
- *Interrupt-Driven Systems*
- *Foreground/Background Systems*
- *Full featured RTOS*

1. Polled Loop Systems

Polled loops are used for fast response to single devices.
In a polled-loop system, a single and a repetitive instruction is used to test a flag that indicates whether or not some event has occurred. If the event has not occurred, then the polling continues.

Example: suppose a software system is needed to handle packets of data that arrive at a rate of no more than 1 per second. A flag named packet_here is set by the network, which writes the data into the CPU's memory via direct memory access (DMA). The data are available when packet_here = 1. Using a C code fragment, such a polled loop to handle such a system is:

```
for(;;) { /* do forever */
   if (packet_here) { /* check flag */

      process_data(); /* process data */
      packet_here=0; /* reset flag */

   }
}
```

Synchronized Polled Loop (1)

A variation on the polled loop uses a fixed clock interrupt to pause between the time when the signaling event is triggered and then reset. Such a system is used to treat events that exhibit switch bounce.

Example: suppose a polled-loop system is used to handle an event that occurs randomly, but no more than once per second. The event is known to exhibit a switch-bounce effect that disappears after 20 milliseconds. A 10-millisecond fixed-rate interrupt is available for synchronization. The event is signaled by an external device that sets a memory location via DMA. The C code looks like the following:

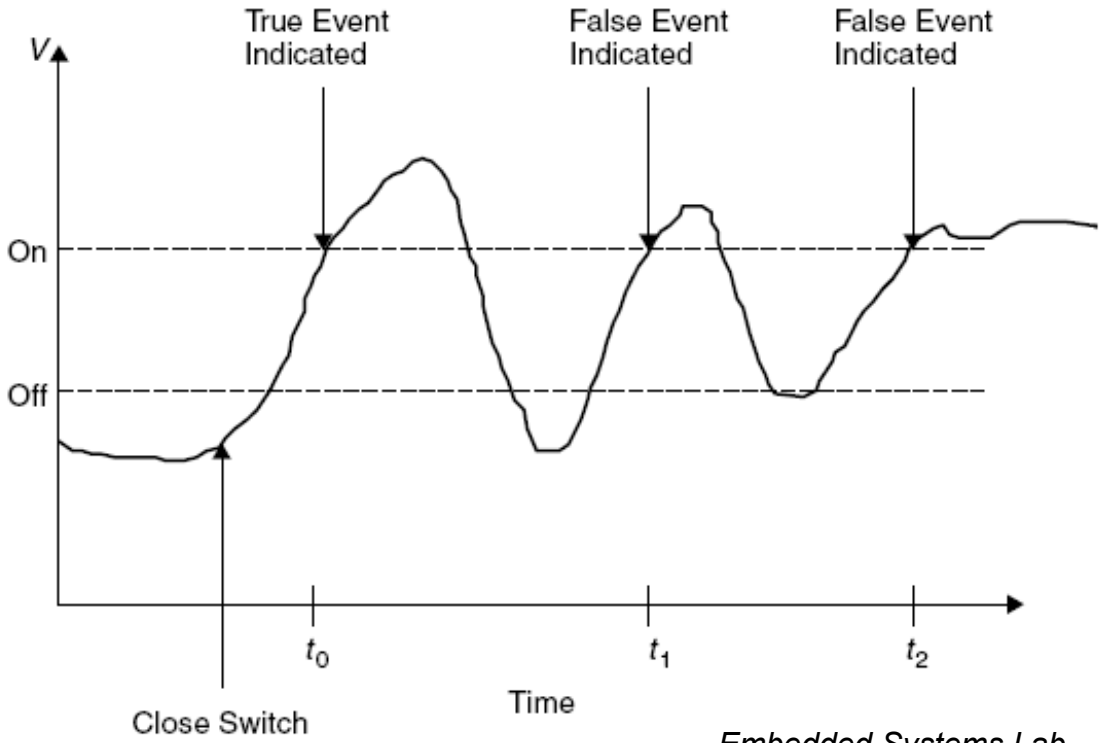
```
for(;;) { /* do forever */
   if(flag) { /* check flag */

      pause(20); /* wait 20 ms */
      process_event(); /* process event */
      flag=0; /* reset flag */
   }
}
```

Synchronized Polled Loop (2)

```
for(;;) { /* do forever */
   if(flag) { /* check flag */
      counter = 0;
      while(counter<3);
      process_event(); /* process event */
      flag=0; /* reset flag */
   }
}
```

10ms interrupt service routine might look like:

```
counter=counter+1;
```



2. Cyclic Executive

Cyclic executives are noninterrupt-driven systems that can provide the illusion of simultaneity by taking advantage of relatively short processes on a fast processor in a continuous loop.

```
for(;;) {
   task_1();
   task_2();
   ...
   task_n();
}
```

```
for(;;) {
   task_1();
   task_2();
   task_1();
   task_3();
}
```

If each process is relatively short and uniform in size, then reactivity and simultaneity can be achieved without interrupts. Moreover, if each process is carefully constructed including synchronization through messaging or global variables, complete determinism and schedulability can be achieved.

3. Cooperative Multitasking

Two or more processes are coded in the state-driven fashion just discussed, and after each phase is complete, a call is made to a central dispatcher. The dispatcher holds the program counter for a list of processes that are executed in round-robin fashion; that is, it selects the next process to execute. Known as code-driven finite state automata approach.

```
void process_a(void)
{
   for(;;){
    switch(state_a){
      case 1: phase_a1();
              break;
      case 2: phase_a2();
              break;
      case 3: phase_a3();
              break;
      case 4: phase_a4();
              break;
      case 5: phase_a5();
              break;
    }
   }
}
```

```
void process_b(void)
{
   for(;;){
    switch(state_b){
      case 1: phase_b1();
              break;
      case 2: phase_b2();
              break;
      case 3: phase_b3();
              break;
      case 4: phase_b4();
              break;
      case 5: phase_b5();
              break;
    }
   }
}
```

"break" commands call dispatcher. State_a and state_b are global variables managed by the dispatcher. This is simpliest form of fairness scheduling.

4. Interrupt-Driven Systems

In interrupt-driven systems, the main program is a single jump-to-self instruction. The various tasks in the system are scheduled via either hardware or software interrupts, whereas dispatching is performed by the interrupt-handling routines.

```
void main(void)
{
   for(;;);
}
```

```
void _isr_1(void)
{
   ...
}
```

...

```
void _isr_n(void)
{
   ...
}
```

Hardware Interrupt: A signal generated by a peripheral device and sent to the CPU. In turn, the CPU executes an interrupt service routine (ISR), which takes action in response to the interrupt.

Software Interrupt: Similar to the hardware interrupt, in that it causes one code module to pass control to another.

Hardware Interrupts and ISRs

- An interrupt can happen at any time (Asynchronous).
- CPU invokes the ISR.
- Around the code that reads/writes to the shared resources interrupts are disabled in the application.
- Synchronization mechanisms cannot be used in ISR, because ISR should not wait indefinitely.
- When interrupts are disabled, the system's ability to receive stimuli from the outside world is minimal.
- It is important to keep the critical sections of code in which the interrupts are disabled as short as possible.
- ISRs can be reentrant where applicable
- A snapshot of the machine – called the context – must be preserved upon switching tasks so that it can be restored upon resuming the interrupted process.

Context Switch

Saving the minimume amount of information necessary to safely restore any process after it has been interrupted. This information ordinarily includes:

- *Contents of general purpose registers*
- *Contents of the program counter*
- *Contents of special registers (e.g. Flags)*
- *Memory page register*
- *Images of memory mapped I/O locations (mirror images)*

```
void main(void)
{
   for(;;);
}
```

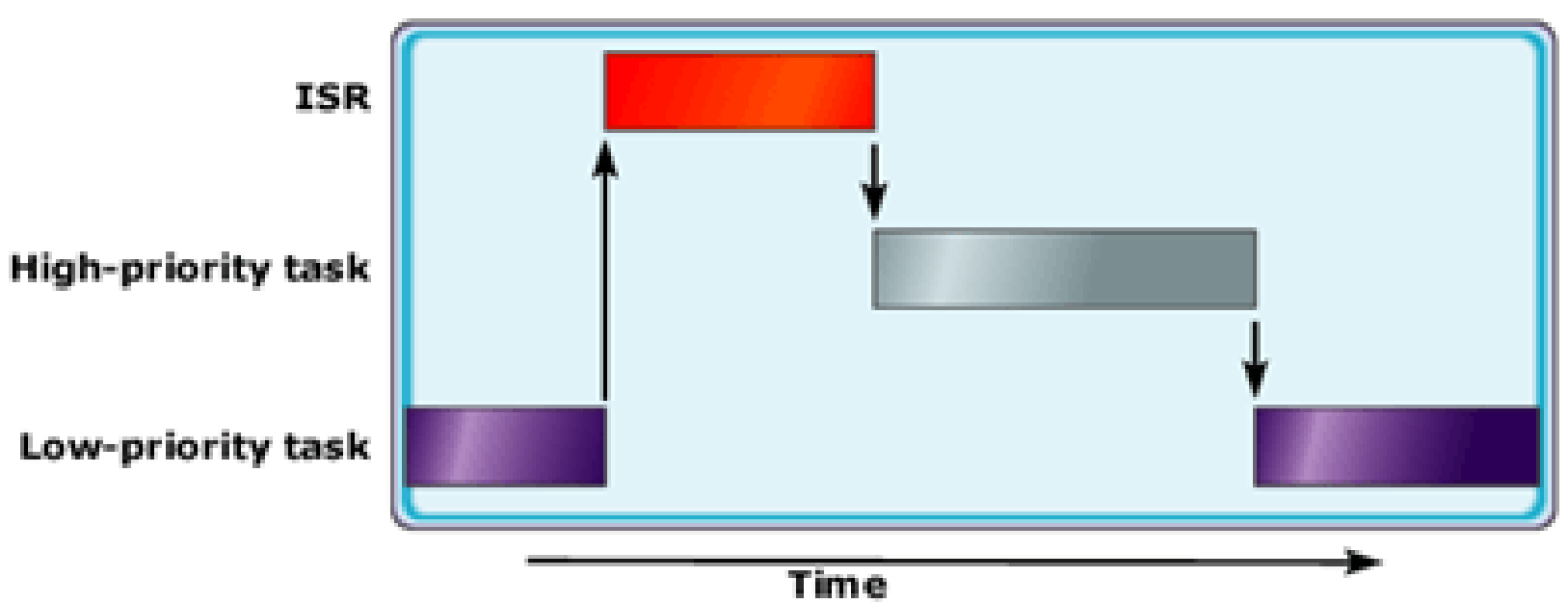
```
void _isr_1(void)
{
  save(context);
  task_1();
  restore(context);
}
```

...

```
void _isr_n(void)
{
  save(context);
  task_n();
  restore(context);
}
```

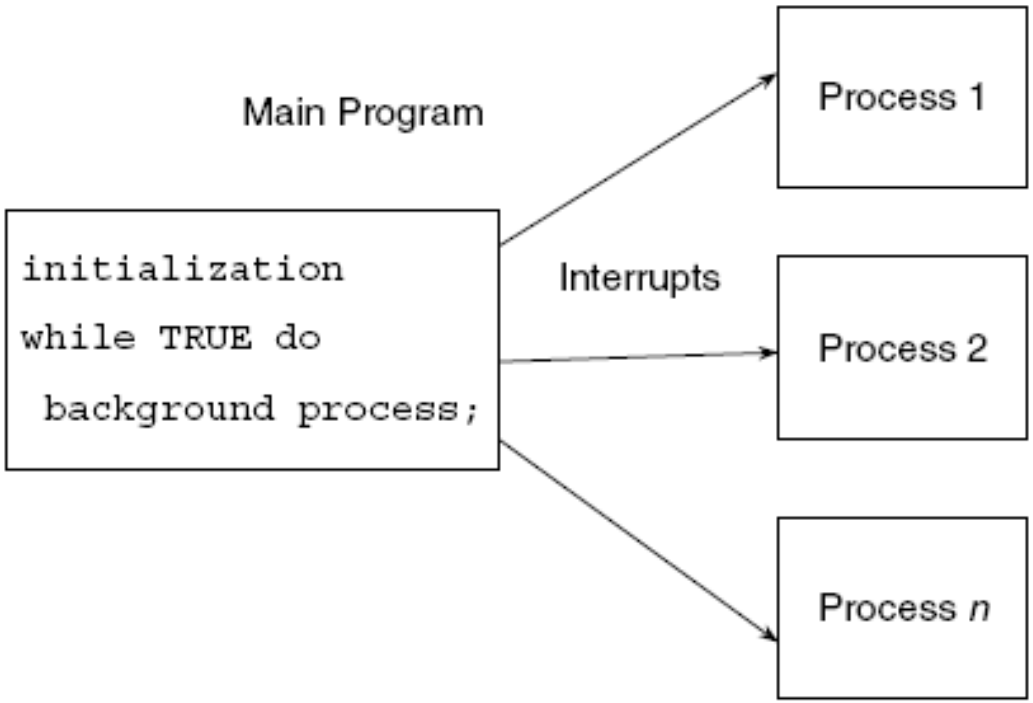
Preemptive-Priority Systems

A higher-priority task is said to preempt a lower-priority task if it interrupts the lower-priority task. Systems that use preemption schemes instead of round-robin or first-come-first-served scheduling are called preemptive-priority systems.



5. Foreground-Background Systems

Foreground/background systems are an improvement over the interrupt-only systems in that the polled loop is replaced by code that performs useful processing. Foreground/background systems are the most common architecture for embedded applications.



Initialization:

1. *Disable interrupts*
2. *Set up interrupt vectors and stacks*
3. *Perform self-test*
4. *Perform system initialization*
5. *Enable interrupts*

Background Process

- Not time critical
- What kind of funtions are appropriate ?

p: time-loading factor for all foreground tasks

e: execution time of background task

t: background process execution period

$$t = e / (1 - p)$$

Execution period can be very low!

Example Background Process: Watchdog Timer

```
void task_1(void)
{
    while(1){
        wdt_1=0;
        ...
        wdt_1=0;
        ...
        wait();
    }
}
```

```
void task2_n(void)
{
    while(1){
        wdt_n=0;
        ...
        wait();
    }
}
```

...

more frequently

```
void main(void)
{
    for(;;){

        wdt_1++; ... wdt_n++;
        if(wdt_1>10)
            reset_system();
            ...
        else if(wdt_n>500)
            reset_system();

    }
}
```

Example 1

```
void interrupt yourisr() /* Interrupt Service Routine (ISR) */
{
   disable();

   /* Body of ISR goes here */
   count--;

   oldhandler();

   outportb(0x20,0x20); /* Send EOI to PIC1 */
   enable();
}

#include <dos.h>
#define INTNO 0x08 // Timer Interrupt

void interrupt (*oldhandler)();
int count=30;

void main(void) {

  oldhandler = getvect(INTNO); /* Save Old Interrupt Vector */
  setvect(INTNO, yourisr); /* Set New Interrupt Vector Entry */


  /* Body of Program Goes Here */
  while(count>0);


  setvect(INTNO, oldhandler); /* Restore old Interrupt Vector */

}
```

Example 2

```
char far * scr=0xb8000000;
char flag=0;

void interrupt yourisr() /* Interrupt Service Routine (ISR) */
{
    disable();

    count++;
    if(count==10){count=0; flag=1; *scr='A';}

    oldhandler();

    outportb(0x20,0x20); /* Send EOI to PIC1 */
    enable();
}

#define INTNO 0x08 // Timer Interrupt

void interrupt (*oldhandler)();

void main(void) {

  oldhandler = getvect(INTNO); /* Save Old Interrupt Vector */
  setvect(INTNO, yourisr); /* Set New Interrupt Vector Entry */

  while(1)
    if(flag){ flag=0; *scr='A'; };


  setvect(INTNO, oldhandler); /* Restore old Interrupt Vector */

}
```

Real-Time

Background

Full-Featured RTOS

The foreground/background solution can be extended into an operating system by adding additional functions such as network interfaces, device drivers, and complex debugging tools. These types of systems are readily available as commercial products. Such systems rely on a complex operating system using round-robin, preemptivepriority, or a combination of both schemes to provide scheduling; the operating system represents the highest priority task, kernel, or supervisor.

- VxWorks
- Real-Time Unix
- Real-Time Linux
- ...

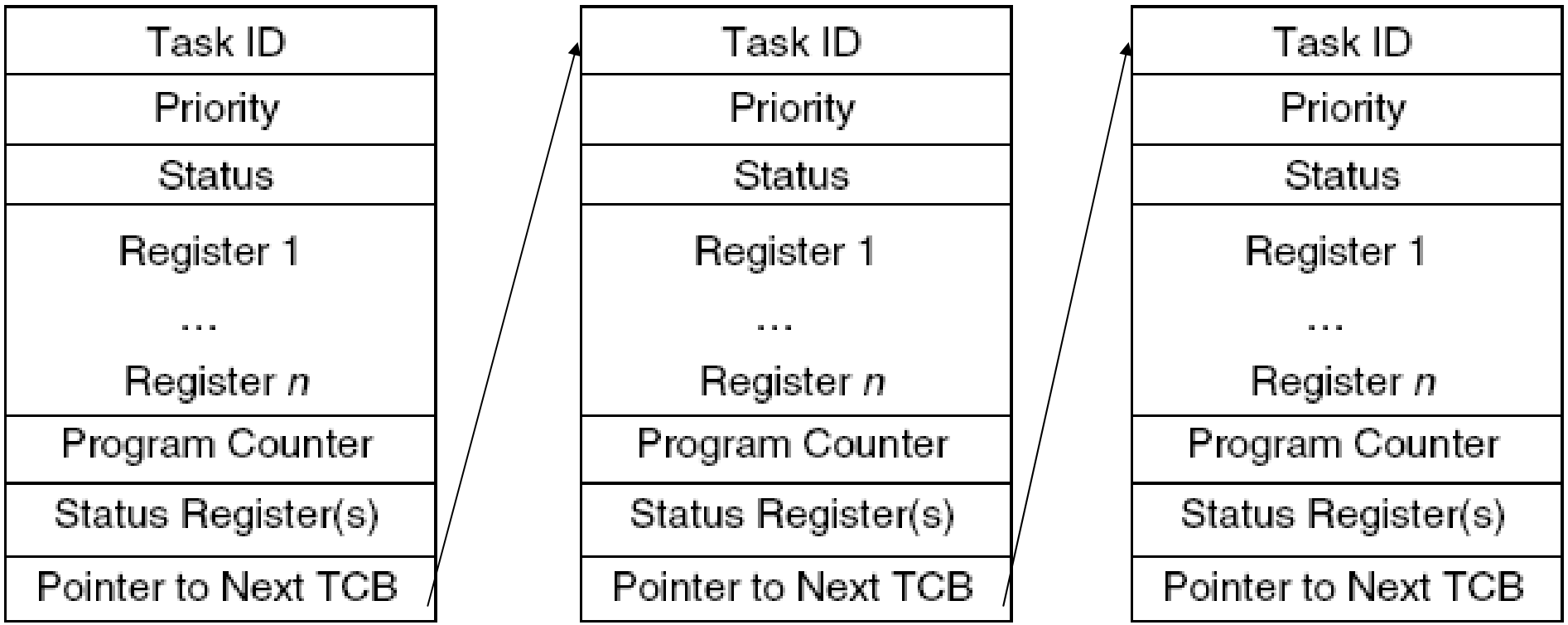
Real-Time Kernels

- A Kernel, executive or nucleus is the smallest portion of the OS that provides these functions
- Rea-Time kernels must provide:
  - A. Interrupt handling, guaranteed interrupt response
  - B. Process management (Support for scheduling of real-time processes and preemptive scheduling)
  - C. Interprocess communication and synchronization.
  - D. Time management.
  - E. Memory management
  - F. I/O support (Support for communication with peripheral devices via drivers)
  - G. High speed data acquisition
  - H. Resource management (User control of system resources)
  - I. Error and exception handling

Real-Time Kernel Features

- A real-time OS should provide support for the creation, deletion and scheduling of multiple processes
- A real-time OS must be able to response an event and take deterministic (well-defined in terms of function and time)
- action based on the event.
- A real-time OS should support interprocess communications via reliable and fast facilities, such as semaphores, shared memory and message passing.
- A real-time system must be able to handle very high burst rates in high speed data acquisition applications.

Task Control Block



Task States

- *Executing*
- *Ready*
- *Suspended (or blocked)*
- *Dormant (or sleeping)*

Task State Diagram