EMBEDDED SYSTEMS

CHAPTER – 6

REAL-TIME OPERATING SYSTEM

6. Real-Time Operating System [8 Hrs.]

6.1 Operating System Basics

6.2 Task, Process, and Threads

6.3 Multiprocessing and Multitasking

6.4 Task Scheduling

6.5 Task Synchronization

6.6 Device Drivers

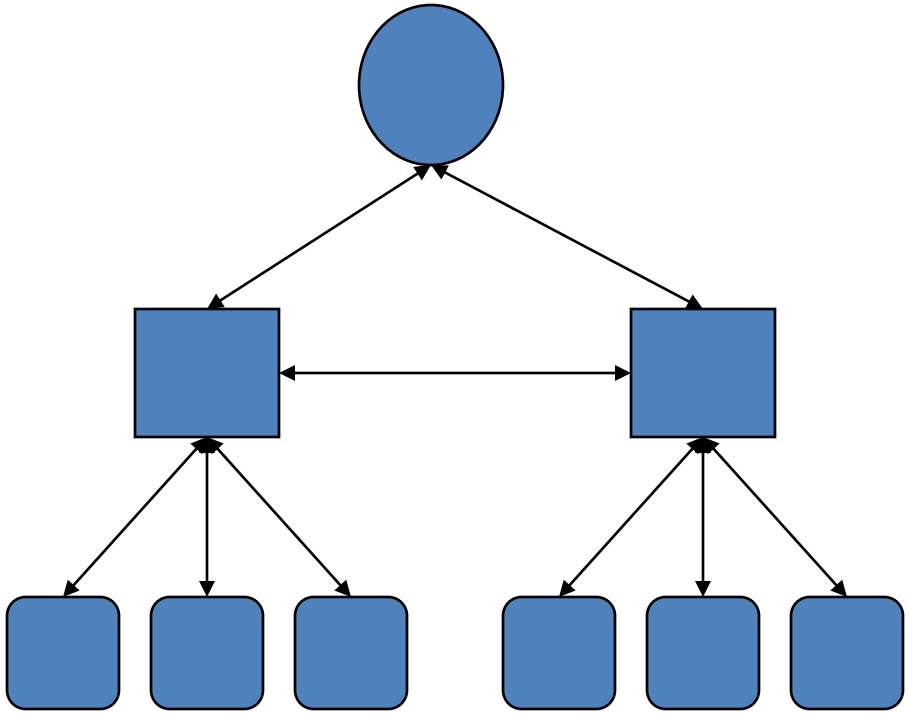
How the increasing need for time critical response for task/events is addressed in embedded applications?

- Assign priority to task & execute the high priority task when the task is ready to execute.
- Dynamically change the priorities of tasks if required on a need basis.
- Schedule the execution of tasks based on the priorities.
- Switch the execution of task when a task is waiting for an external event or a system resource including I/O device operation.

Operating System Basics

- acts as a bridge between the user application/ tasks & the underlying system resources through a set of system functionalities and services.
- Manages the system resources and makes them available to the user application/task on a need basis.
- Primary functions are:
  - Make the system convenient to use
  - Organize & manage the system resources efficiently and correctly.

Fire alarm system: an example



Central server

TCP/IP over radio

Controllers: ARM based

Low bandwidth radio links

Sensors: microcontroller based

Fire Alarm System

- Problem
  - Hundreds of sensors, each fitted with Low Range Wireless
    - Sensor information to be logged in a server & appropriate action initiated
- Possible Solution
  - Collaborative Action
    - Routing
      - Dynamic – Sensors/controllers may go down
      - Auto Configurable – No/easy human intervention.
      - Less Collision/Link Clogging
      - Less no of intermediate nodes
        - » Fast Response Time
      - Secure

RTOS: Target Architectures

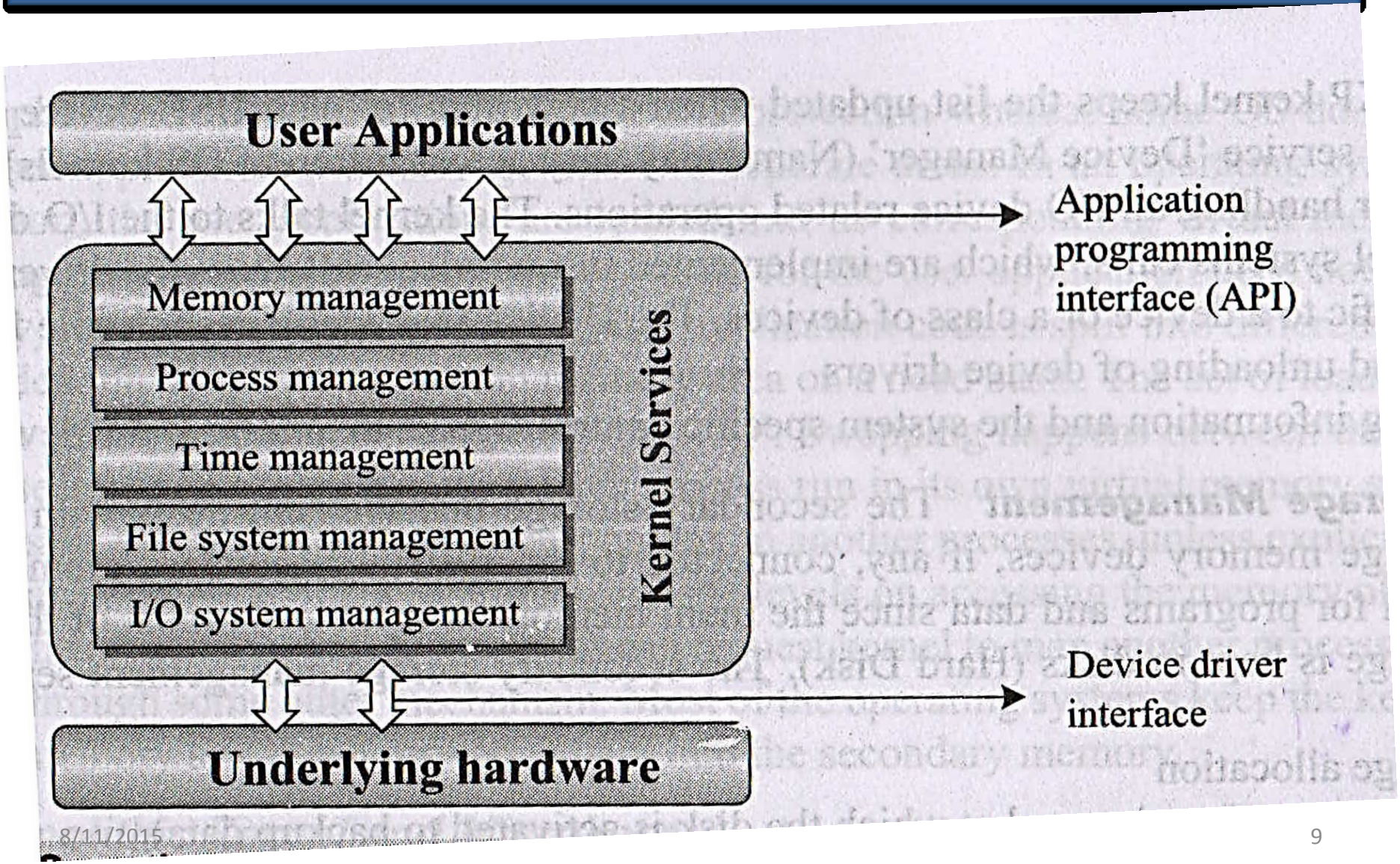
Processors	MIPS
Microcontrollers	~20
ARM7	100-133
ARM9	180-250
Strong ARM	206
Intel Xscale	400
Mips4Kcore	400
X86	

Operating System Basics contd…

The Kernel is:

- core of operating system
- responsible for managing the system resources and the communication among the hardware and other system services.
- act as the abstraction layer between system resources and user applications.
- contains a set of system libraries and services.

Operating System Basics contd…



Operating System Basics contd…

Process Management:

- deals with managing the processes/tasks.
- Includes setting up the memory space for the process
- Loading the process's code into the memory space
- Allocating system resources
- Scheduling and managing the execution of the process
- Setting up and managing the process control Block (PCB)
- Inter process communication and synchronization
- Process termination/deletion

Operating System Basics contd…

Primary Memory Management:

- Refers to the volatile memory (RAM) where processes are loaded and variables and shared data associated with each process are stored.
- Memory Management Unit (MMU) of the kernel is responsible for
  - Keeping track of which part of the memory area is currently used by which process
  - Allocating and De-allocating memory space on a need basis (DMA)

Operating System Basics contd…

File System Management: ***responsible for***

- **The creation, deletion and alteration of files.**
- **Creation, deletion and alteration of directories**
- **Saving of files in the secondary storage memory**
- **Providing automatic allocation of file space based on the amount of free space available**
- **Providing a flexible naming convention for the files.**

Operating System Basic contd…

I/O System (Device)Management

- loading and unloading of device drivers
- exchanging information and the system specific control signals to and from the device

Secondary storage management

- Disk storage allocation
- Disk scheduling (time interval at which the disk is activated to backup data)
- Free disk space management

Operating System Basic contd…

Protection systems (deals with implementing the security policies to restrict the access to both user and system resources by different application or processes or users)

Interrupt Handler (Kernel provides handler mechanism for all external/internal interrupts generated by the system)

Operating System Types contd…

General Purpose Operating System (GPOS)

Real - Time Operating System (RTOS)

- Implies deterministic timing behavior
- Means the OS services consumes only known and expected amounts of time regardless the number of services.
- Implements policies and rules concerning time critical allocation of a system's resources
- Applications should run in which order and how much time need to be allocated for each application.

- **A more complex software architecture is needed to handle multiple tasks, coordination, communication, and interrupt handling – an RTOS architecture**

- **Distinction:**
  - **Desktop OS – OS is in control at all times and runs applications, OS runs in different address space**
  - **RTOS – OS and embedded software are integrated, ES starts and activates the OS – both run in the same address space (RTOS is less protected)**
  - **RTOS includes only service routines needed by the ES application**
  - **RTOS vendors: VsWorks, VTRX, Nucleus, LynxOS, υC/OS**
  - **Most conform to POSIX (IEEE standard for OS interfaces)**
  - **Desirable RTOS properties: use less memory, application programming interface, debugging tools, support for variety of microprocessors, already-debugged network drivers**

Hard and Soft Real Time Systems

- **Hard Real Time System**
  - Failure to meet deadlines is fatal
  - example : Flight Control System
- **Soft Real Time System**
  - Late completion of jobs is undesirable but not fatal.
  - System performance degrades as more & more jobs miss deadlines
  - Online Databases
- **Qualitative Definition.**

Hard and Soft Real Time Systems
(Operational Definition)

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

Operating System Types contd…

The Real-Time Kernel: is highly specialized and it contains only the minimal set of services required for a running the user application/tasks. Basic functions are

- **Task/Process management**
- **Task/Process scheduling**
- **Task/Process synchronization**
- **Error/Exception handling**
- **Memory management**
- **Interrupt handling time management**

Tasks & Task State

Task are very simple to write: under most RTOSs a task is simply a subroutine.

1. Running— the microprocessor is executing the instructions that make up this task. one microprocessor, and hence only one task that is in the running state at any given time.

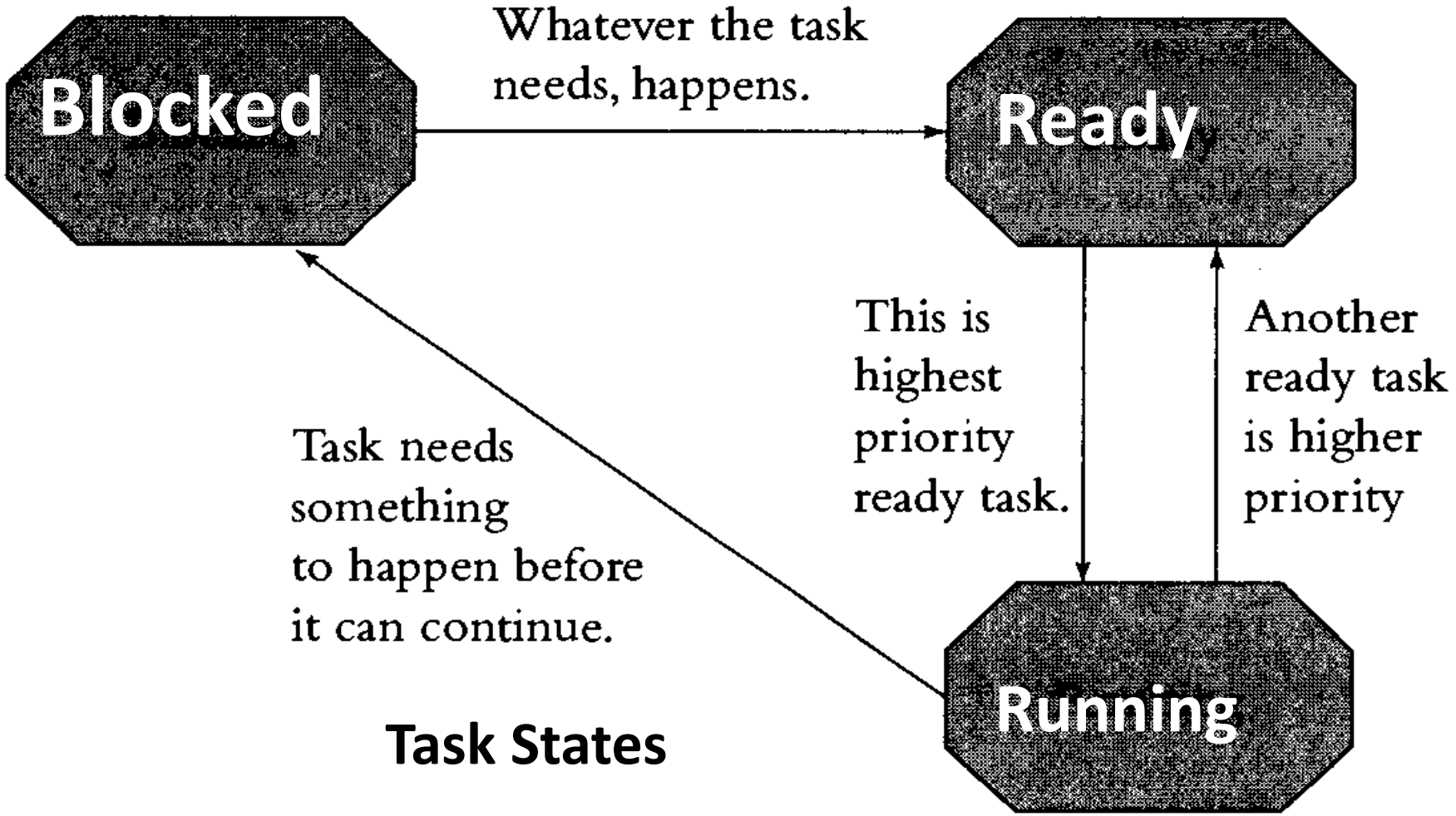
2. Ready— some other task is in the running state but that this task has things that it could do if the microprocessor becomes available. Any number of tasks can be in this state.

3. Blocked— this task hasn't got anything to do right now, even if the microprocessor becomes available. Tasks get into this state because they are waiting for some external event. For example, a task that handles data coming in from a network will have nothing to do when there is no data. A task that responds to the user when he presses a button has nothing to do until the user presses the button. Any number of tasks can be in this state as well.

- **ES application makes calls to the RTOS functions to start tasks, passing to the OS, start address, stack pointers, of the tasks**
- **Task States:**
  - **Running**
  - **Ready (possibly: suspended, pended)**
  - **Blocked (possibly: waiting, dormant, delayed)**
  - **[Exit]**

  - **Scheduler – schedules/shuffles tasks between Running and Ready states**
  - **Blocking is self-blocking by tasks, and moved to Running state via other tasks' interrupt signaling (when block-factor is removed/satisfied)**
  - **When a task is unblocked with a higher priority over the 'running' task, the scheduler 'switches' context immediately (for all pre-emptive RTOSs)**

Tasks



Task States

Tasks

Here are answers to some common questions about the scheduler and task states'.

***How does the scheduler know when a task has become blocked or unblocked*?**

What happens if all the tasks are blocked?

What if two tasks with the same priority are ready?

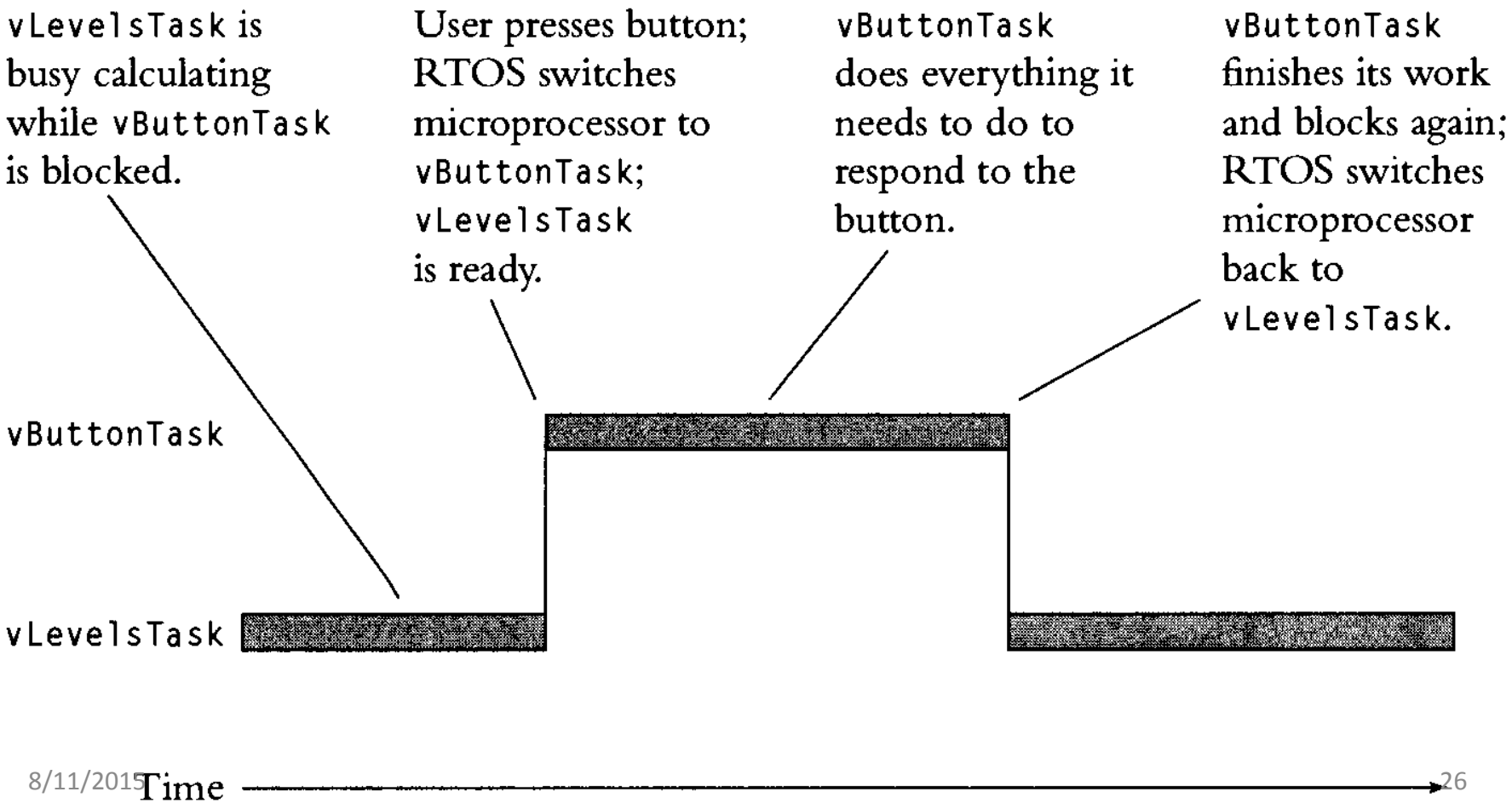
- Tasks – 1
  - Issue – Scheduler/Task signal exchange for block-unblock of tasks via function calls
  - Issue – All tasks are blocked and scheduler idles forever (not desirable!)
  - Issue – Two or more tasks with same priority levels in Ready state (time-slice, FIFO)
  - Example: scheduler switches from processor-hog vLevelsTask to vButtonTask (on user interruption by pressing a push-button), controlled by the main() which initializes the RTOS, sets priority levels, and starts the RTOS

Tasks

```
/* "Button Task" */
void vButtonTask (void)   /* High priority */
{
   while (TRUE)
   {
      !! Block until user pushes a button
      !! Quick: respond to the user
   }
}

/* "Levels Task" */
void vLevelsTask (void)   /* Low priority */
{
   while (TRUE)
   {
      !! Read levels of floats in tank
      !! Calculate average float level
      !! Do some interminable calculation
      !! Do more interminable calculation
      !! Do yet more interminable calculation

      !! Figure out which tank to do next
   }
}
```

Tasks

Microprocessor Responds to a Button under an RTOS;



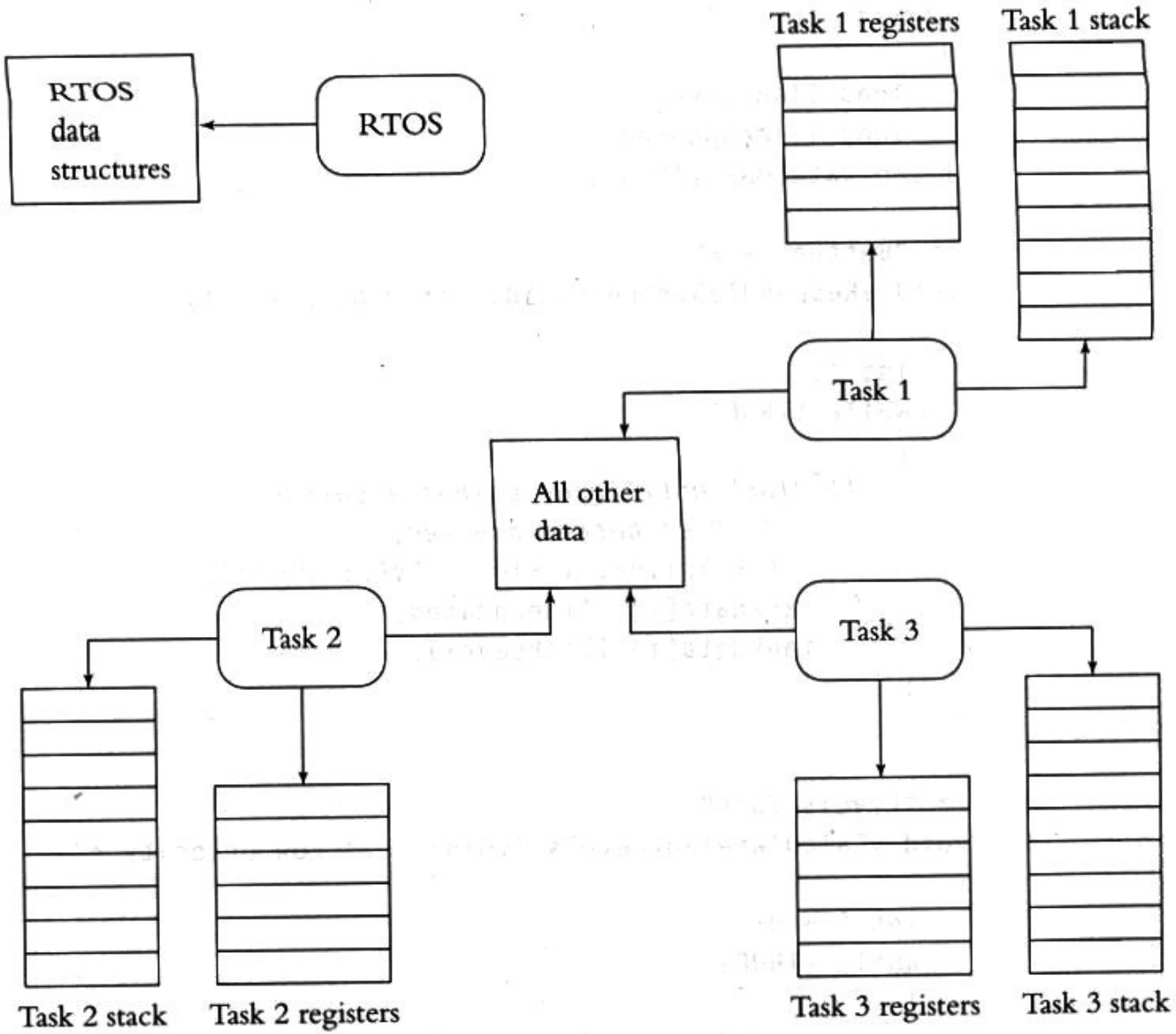
Tasks

RTOS Initialization Code

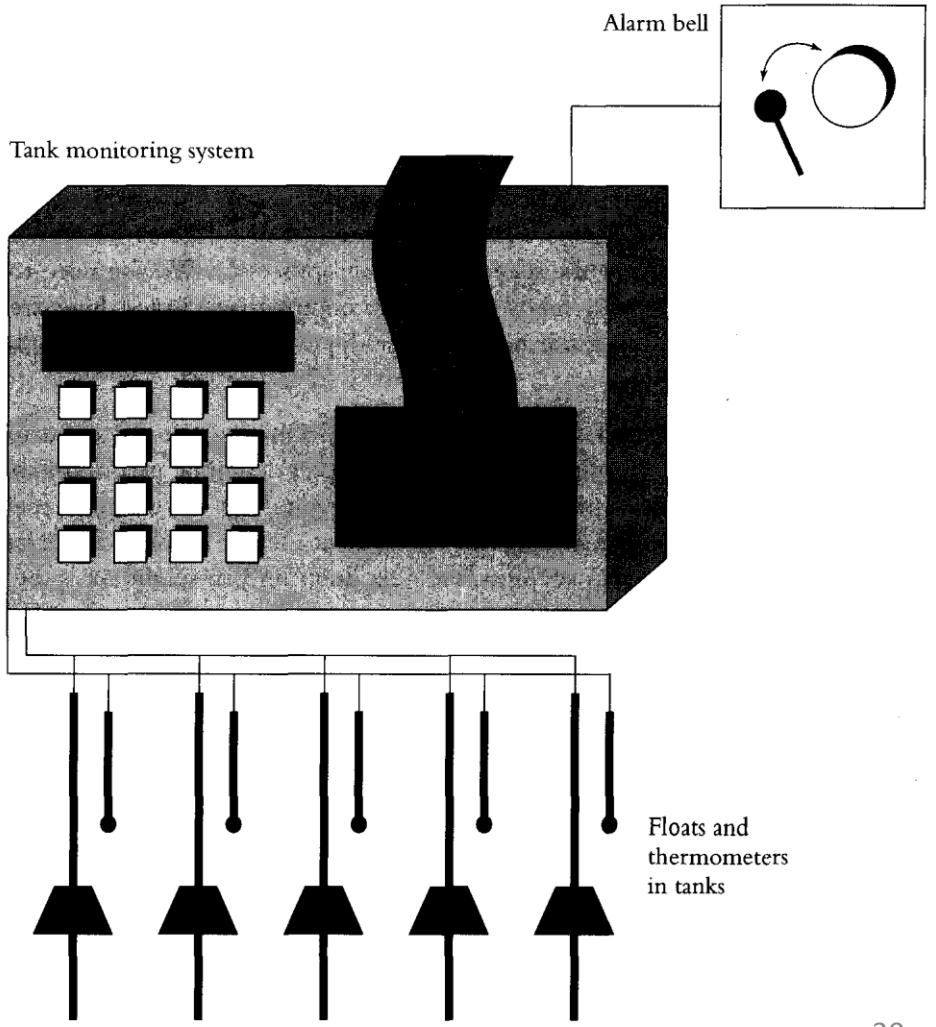
```
void main (void)
{
   /* Initialize (but do not start) the RTOS */
   InitRTOS ();

   /* Tell the RTOS about our tasks */
   StartTask (vRespondToButton, HIGH_PRIORITY);
   StartTask (vCalculateTankLevels, LOW_PRIORITY);

   /* Start the RTOS.  (This function never returns.) */
   StartRTOS ();
}
```

- Tasks and Data
  - Each tasks has its won context - not shared, private registers, stack, etc.
  - In addition, several tasks share common data (via global data declaration; use of ‘extern’ in one task to point to another task that declares the shared data
  - Shared data caused the ‘shared-data problem’ without solutions or use of ‘Reentrancy’ characterization of functions
  - (See Fig 6.5, Fig 6.6, Fig 6.7, and Fig 6.8)

Figure 6.5 Data in an RTOS-Based Real-Time System



Tank Monitoring System



Tasks in the Underground Tank System

Task	Priority	Reason for Creating This Task
Level calculation task	Low	Other processing is much higher priority than this calculation, and this calculation is a microprocessor hog.
Overflow detection task	High	This task determines whether there is an overflow; it is important that this task operate quickly.
Button handling task	High	This task controls the state machine that operates the user interface, relieving the button interrupt routine of that complication, but still responding quickly.
Display task	High	Since various other tasks use the display, this task makes sure that they do not fight over it.
Print formatting task	Medium	Print formatting might take long enough that it interferes with the required response to the buttons. Also, it may be simpler to handle the print queue in a separate task.

Tank Monitoring Design

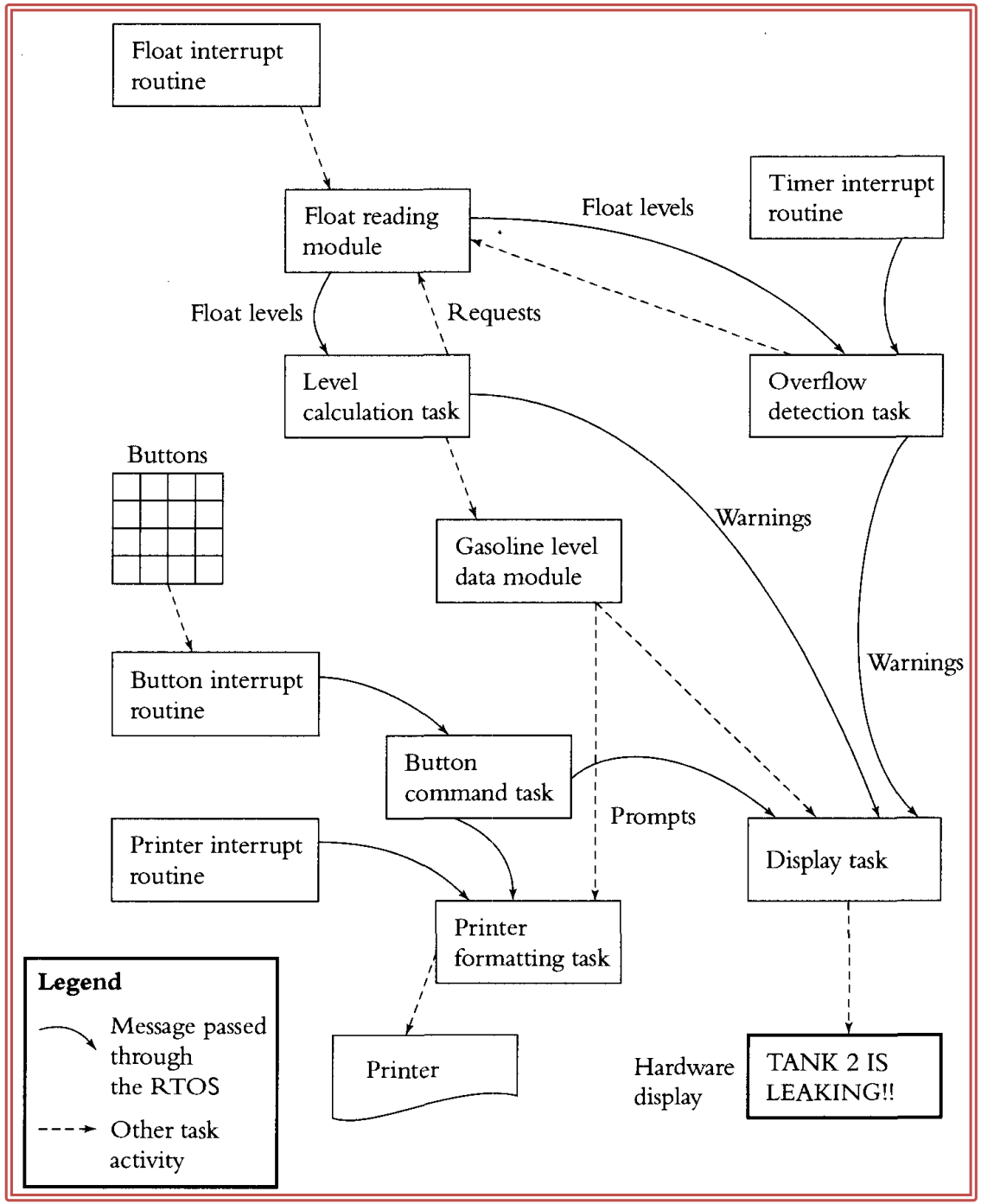


Figure 6.6 Sharing Data among RTOS Tasks

```
struct
{
   long lTankLevel;
   long lTimeUpdated;
} tankdata[MAX_TANKS];

/* "Button Task" */
void vRespondToButton (void)  /* High priority */
{
   int i;
   while (TRUE)
   {
      !! Block until user pushes a button
      i = !! ID of button pressed;
      printf ("\nTIME: %08ld   LEVEL: %08ld",
         tankdata[i].lTimeUpdated,
         tankdata[i].lTankLevel);
   }
}
```

(continued)

Figure 6.6 *(continued)*

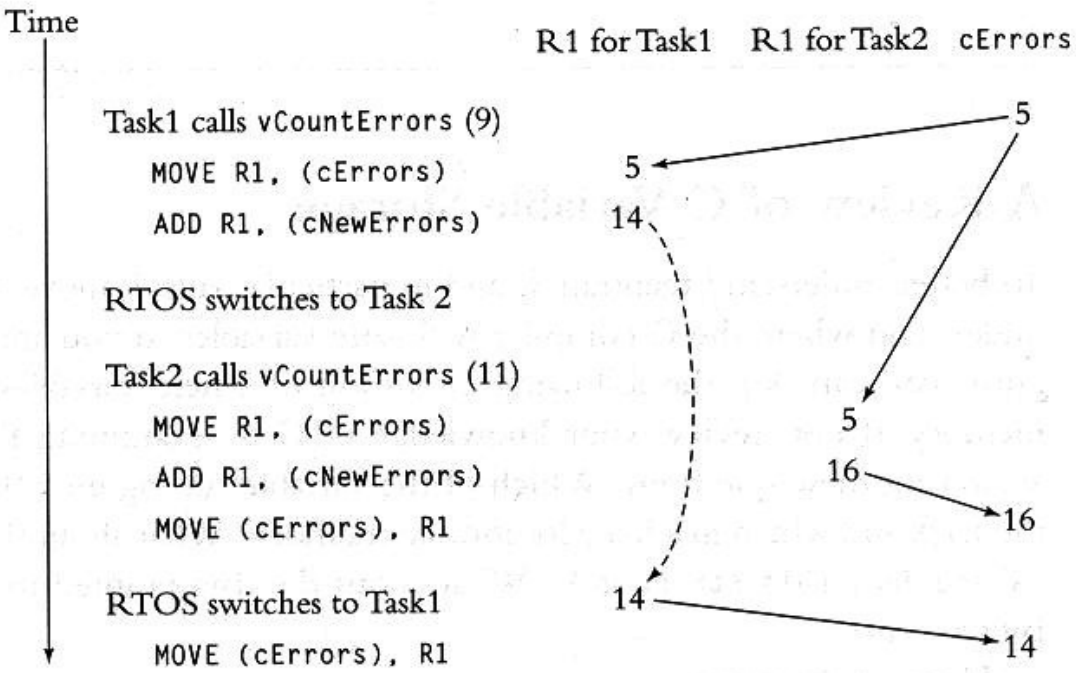
```
/* "Levels Task" */
void vCalculateTankLevels (void)   /* Low priority */
{
   int i = 0;
   while (TRUE)
   {
      !! Read levels of floats in tank i
      !! Do more interminable calculation
      !! Do yet more interminable calculation

      /* Store the result */
      tankdata[i].lTimeUpdated = !! Current time
         /* Between these two instructions is a
            bad place for a task switch */
      tankdata[i].lTankLevel = !! Result of calculation

      !! Figure out which tank to do next
      i = !! something new
   }
}
```

Figure 6.7 Tasks Can Share Code

```
void Task1 (void)
{
   .
   .
   .
   vCountErrors (9);
   .
   .
   .
}

void Task2 (void)
{
   .
   .
   .
   vCountErrors (11);
   .
   .
   .
}

static int cErrors;

void vCountErrors (int cNewErrors)
{
   cErrors += cNewErrors;
}
```

Figure 6.8 Why the Code in Figure 6.7 Fails

```
; Assembly code for vCountErrors
;
; void vCountErrors (int cNewErrors)
;{
;       cErrors += cNewErrors;
        MOVE R1, (cErrors)
        ADD R1, (cNewErrors)
        Move (cErrors), R1
        RETURN
;}
```



- Tasks – 2
- Reentrancy – A function that works correctly regardless of the number of tasks that call it between interrupts
- Characteristics of reentrant functions –
  - Only access shared variable in an atomic-way, or when variable is on callee's stack
  - A reentrant function calls only reentrant functions
  - A reentrant function uses system hardware (shared resource) atomically

- Inspecting code to determine Reentrancy:
  - See Fig 6.9 – Where are data stored in C? Shared, non-shared, or stacked?
  - See Fig 6.10 – Is it reentrant? What about variable *fError*? Is ***printf*** reentrant?
  - If shared variables are not protected, could they be accessed using single assembly instructions (guaranteeing non-atomicity)?

Figure 6.9 Variable Storage

```
static int static_int;
int public_int;
int initialized = 4;
char *string = "Where does this string go?";
void *vPointer;

void function (int parm, int *parm_ptr)
{
    static int static_local;
    int local;
    .
    .
    .
}
```

Figure 6.10 Another Reentrancy Example

```
BOOL fError;    /* Someone else sets this */

void display (int j)
{
   if (!fError)
   {
      printf ("\nValue: %d", j);
      j = 0;
      fError = TRUE;
   }
   else
   {
      printf ("\nCould not display value");
      fError = FALSE;
   }
}
```

- Semaphores and Shared Data – A new tool for atomicity
  - Semaphore – a system resource for implementing mutual exclusion in shared resource access or restricting the access to the shared resources (to avoid shared-data problems in RTOS)
  - Protection at the start is via primitive function, called ***take***, indexed by the semaphore
  - Protection at the end is via a primitive function, called ***release***, also indexed similarly
  - Simple semaphores – Binary semaphores are often adequate for shared data problems in RTOS

Figure 6.12 Semaphores Protect Data

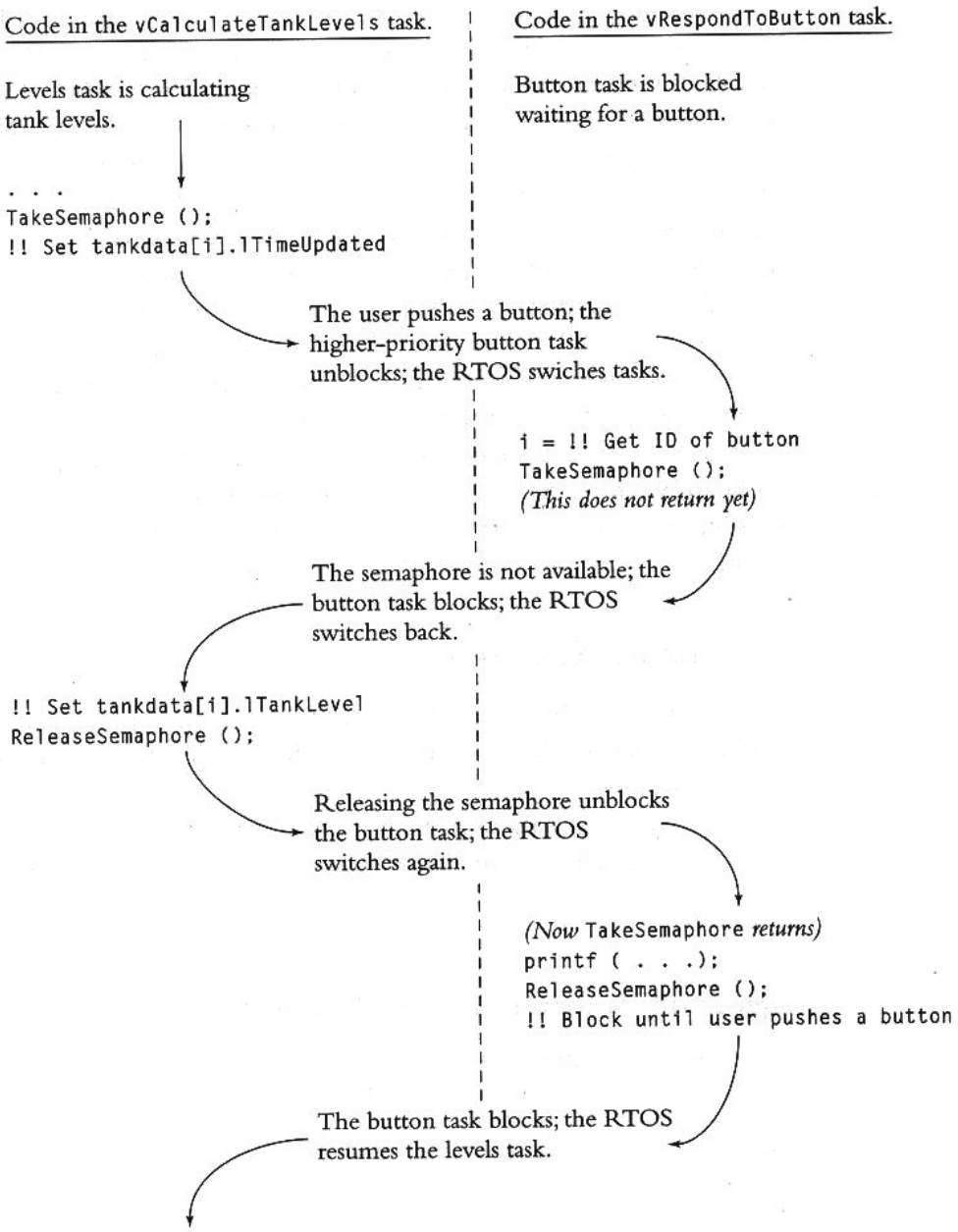
```
struct
{
   long lTankLevel;
   long lTimeUpdated;
} tankdata[MAX_TANKS];

/* "Button Task" */
void vRespondToButton (void)  /* High priority */
{
   int i;
   while (TRUE)
   {
      !! Block until user pushes a button
      i = !! Get ID of button pressed
      TakeSemaphore ();
      printf ("\nTIME: %08ld   LEVEL: %08ld",
         tankdata[i].lTimeUpdated,
         tankdata[i].lTankLevel);
      ReleaseSemaphore ();
   }
}
```

(continued)

Figure 6.12 *(continued)*

```
/* "Levels Task" */
void vCalculateTankLevels (void)   /* Low priority */
{
   int i = 0;
   while (TRUE)
   {
      .
      .
      .
      TakeSemaphore ();
      !! Set tankdata[i].lTimeUpdated
      !! Set tankdata[i].lTankLevel
      ReleaseSemaphore ();
      .
      .
      .
   }
}
```

Figure 6.13 Execution Flow with Semaphores



- Semaphores and Shared Data – 1
  - RTOS Semaphores & Initializing Semaphores
  - Using binary semaphores to solve the ‘tank monitoring’ problem
  - (See Fig 6.12 and Fig 6.13)
  - The nuclear reactor system: The issue of initializing the semaphore variable in a dedicated task (not in a ‘competing’ task) before initializing the OS – timing of tasks and priority overrides, which can undermine the effect of the semaphores
  - Solution: Call OSSemInit() before OSInit()
  - (See Fig 6.14)

Figure 6.14 Semaphores Protect Data in the Nuclear Reactor

```
#define TASK_PRIORITY_READ 11
#define TASK_PRIORITY_CONTROL 12
#define STK_SIZE 1024
static unsigned int ReadStk [STK_SIZE];
static unsigned int ControlStk [STK_SIZE];

static int iTemperatures[2];
OS_EVENT *p_semTemp;
```

(continued)

Figure 6.14 *(continued)*

```
void main (void)
{
   /* Initialize (but do not start) the RTOS */
   OSInit ();

   /* Tell the RTOS about our tasks */
   OSTaskCreate (vReadTemperatureTask,  NULLP,
      (void *)&ReadStk[STK_SIZE], TASK_PRIORITY_READ);
   OSTaskCreate (vControlTask,  NULLP,
      (void *)&ControlStk[STK_SIZE], TASK_PRIORITY_CONTROL);

   /* Start the RTOS.  (This function never returns.) */
   OSStart ();
}
```

(continued)

Figure 6.14 *(continued)*

```
void vReadTemperatureTask (void)
{
   while (TRUE)
   {
      OSTimeDly (5); /* Delay about 1/4 second */

      OSSemPend (p_semTemp, WAIT_FOREVER);
      !! read in iTemperatures[0];
      !! read in iTemperatures[1];
      OSSemPost (p_semTemp);
   }
}

void vControlTask (void)
{
   p_semTemp = OSSemInit (1);
   while (TRUE)
   {
      OSSemPend (p_semTemp, WAIT_FOREVER);
      if (iTemperatures[0] != iTemperatures[1])
         !! Set off howling alarm;
      OSSemPost (p_semTemp);

      !! Do other useful work
   }
}
```

- Semaphores and Shared Data – 2
  - Reentrancy, Semaphores, Multiple Semaphores, Device Signaling,
  - Fig 6.15 – a reentrant function, protecting a shared data, cErrors, in critical section
  - Each shared data (resource/device) requires a separate semaphore for individual protection, allowing multiple tasks and data/resources/devices to be shared exclusively, while allowing efficient implementation and response time
  - Fig 6.16 – example of a printer device signaled by a report-buffering task, via semaphore signaling, on each print of lines constituting the formatted and buffered report

Figure 6.15 Semaphores Make a Function Reentrant

```
void Task1 (void)
{
   .
   .
   .
   vCountErrors (9);
   .
   .
   .
}

void Task2 (void)
{
   .
   .
   .
   vCountErrors (11);
   .
   .
   .
}

static int cErrors;
static NU_SEMAPHORE semErrors;

void vCountErrors (int cNewErrors)
{
   NU_Obtain_Semaphore (&semErrors, NU_SUSPEND);
   cErrors += cNewErrors;
   NU_Release_Semaphore (&semErrors);
}
```

Figure 6.16 Using a Semaphore as a Signaling Device

```
/* Place to construct report. */
static char a_chPrint[10][21];

/* Count of lines in report. */
static int iLinesTotal;

/* Count of lines printed so far. */
static int iLinesPrinted;

/* Semaphore to wait for report to finish. */
static OS_EVENT *semPrinter;
```

(continued)

Figure 6.16 *(continued)*

```
void vPrinterTask(void)
{
   BYTE byError;   /* Place for an error return. */
   Int wMsg;

   /* Initialize the semaphore as already taken. */
   semPrinter = OSSemInit(0);

   while (TRUE)
   {
      /* Wait for a message telling what report to format. */
      wMsg = (int) OSQPend (QPrinterTask, WAIT_FOREVER, &byError);

      !! Format the report into a_chPrint
      iLinesTotal = !! count of lines in the report

      /* Print the first line of the report */
      iLinesPrinted = 0;
      vHardwarePrinterOutputLine (a_chPrint[iLinesPrinted++]);

      /* Wait for print job to finish. */
      OSSemPend (semPrinter, WAIT_FOREVER, &byError);
   }
}
```

(continued)

Figure 6.16 *(continued)*

```
void  vPrinterInterrupt (void)
{
   if (iLinesPrinted == iLinesTotal)
      /* The report is done.  Release the semaphore. */
      OSSemPost (semPrinter);

   else
      /* Print the next line. */
      vHardwarePrinterOutputLine (a_chPrint[iLinesPrinted++]);
}
```

- Semaphores and Shared Data – 3
  - Semaphore Problems – ‘Messing up’ with semaphores
    - The initial values of semaphores – when not set properly or at the wrong place
    - The ‘symmetry’ of takes and releases – must match or correspond – each ‘take’ must have a corresponding ‘release’ somewhere in the ES application
    - ‘Taking’ the wrong semaphore unintentionally (issue with multiple semaphores)
    - Holding a semaphore for too long can cause ‘waiting’ tasks’ deadline to be missed
    - Priorities could be ‘inverted’ and usually solved by ‘priority inheritance/promotion’
    - (See Fig 6.17)
    - Causing the deadly embrace problem (cycles)
    - (See Fig 6.18)

Figure 6.17 Priority Inversion

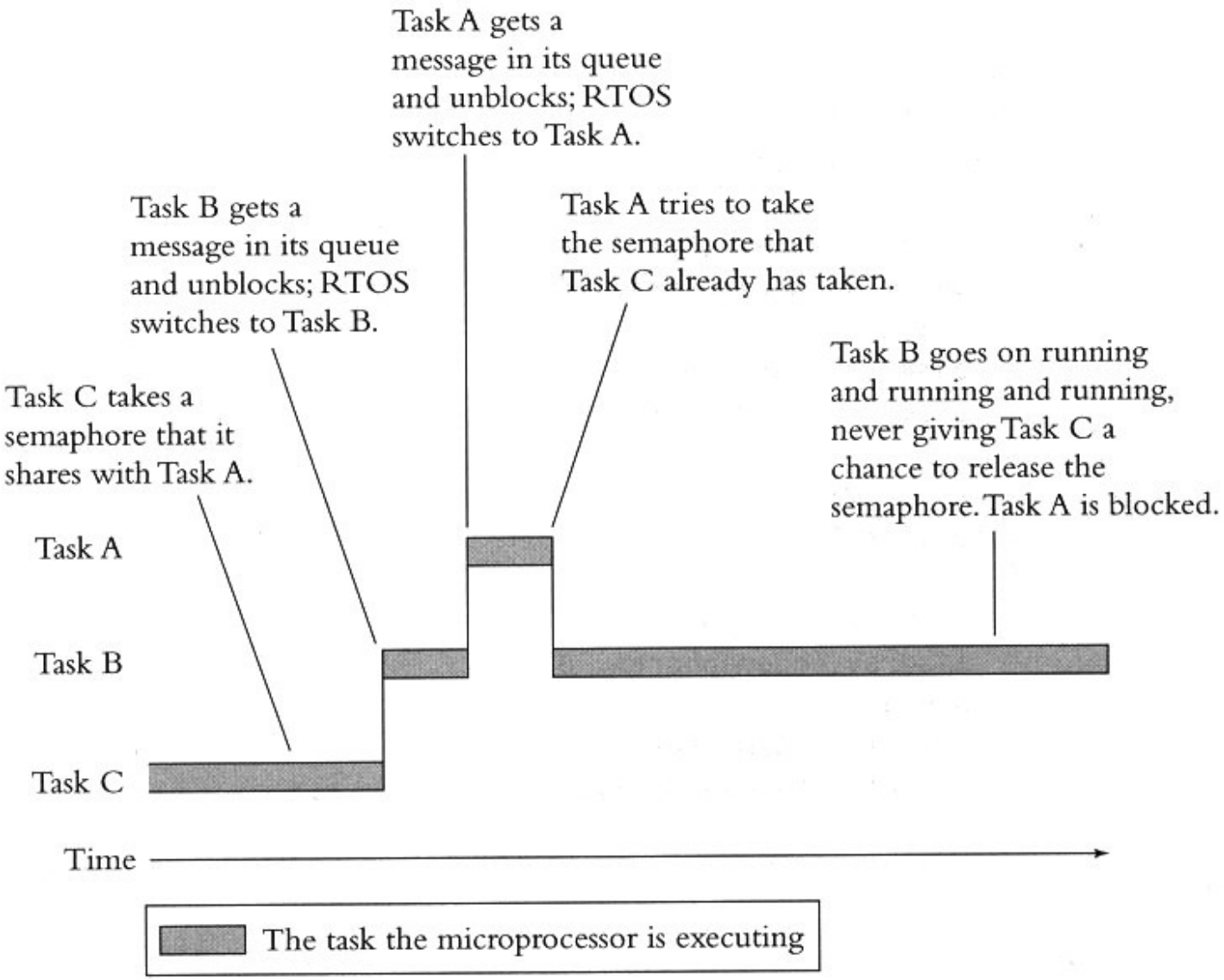


Figure 6.18 Deadly-Embrace Example

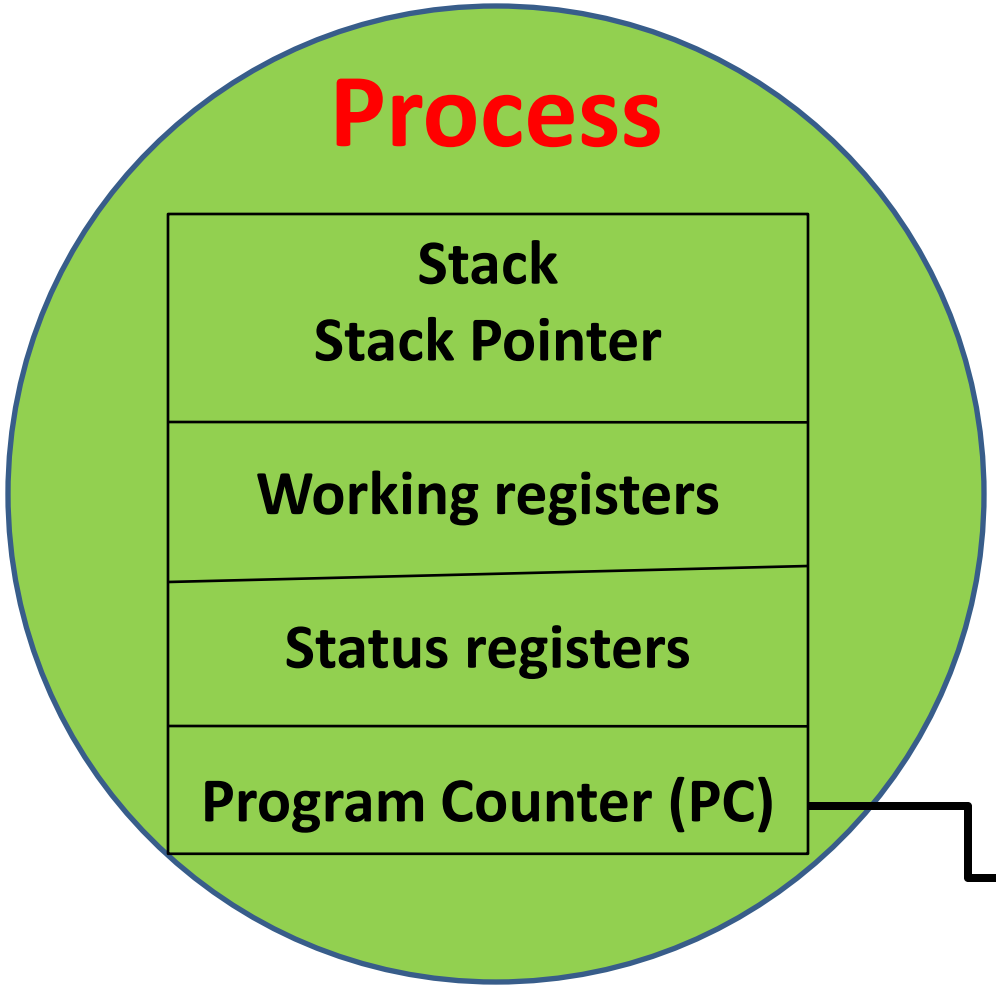
```
int a;
int b;
AMXID hSemaphoreA;
AMXID hSemaphoreB;
void vTask1 (void)
{
   ajsmrsv (hSemaphoreA, 0, 0);
   ajsmrsv (hSemaphoreB, 0, 0);
   a = b;
   ajsmrls (hSemaphoreB);
   ajsmrls (hSemaphoreA);
}

void vTask2 (void)
{
   ajsmrsv (hSemaphoreB, 0, 0);
   ajsmrsv (hSemaphoreA, 0, 0);
   b = a;
   ajsmrls (hSemaphoreA);
   ajsmrls (hSemaphoreB);
}
```

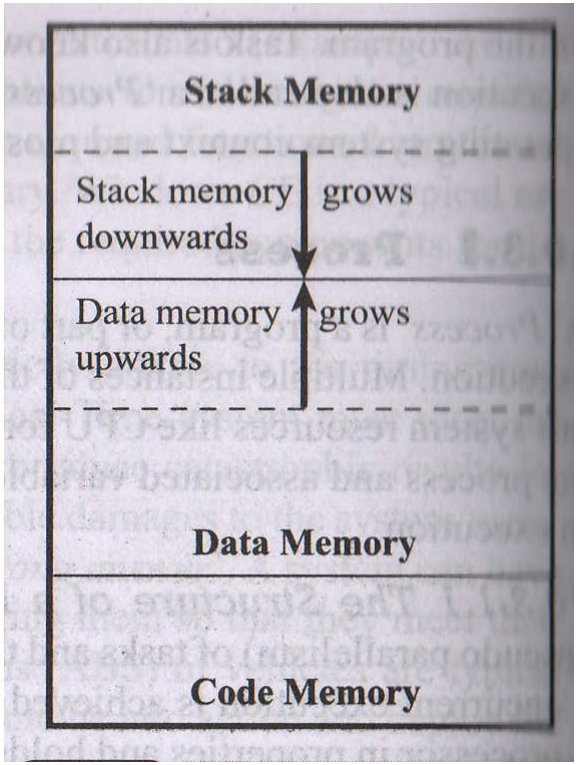
- **Semaphores and Shared Data – 4**
  - **Variants:**
    - **Binary semaphores – single resource, one-at-a time, alternating in use (also for resources)**
    - **Counting semaphores – multiple instances of resources, increase/decrease of integer semaphore variable**
    - **Mutex – protects data shared while dealing with priority inversion problem**
  - **Summary – Protecting shared data in RTOS**
    - **Disabling/Enabling interrupts (for task code and interrupt routines), faster**
    - **Taking/Releasing semaphores (can't use them in interrupt routines), slower, affecting response times of those tasks that need the semaphore**
    - **Disabling task switches (no effect on interrupt routines), holds all other tasks' response**

Process:

- is a program, or part of it execution.
- an instance of a program in execution; multiple instances of the same program can execute simultaneously.
- Requires various system resources like CPU for executing the process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange.
- is sequential in execution.

Process Structure:



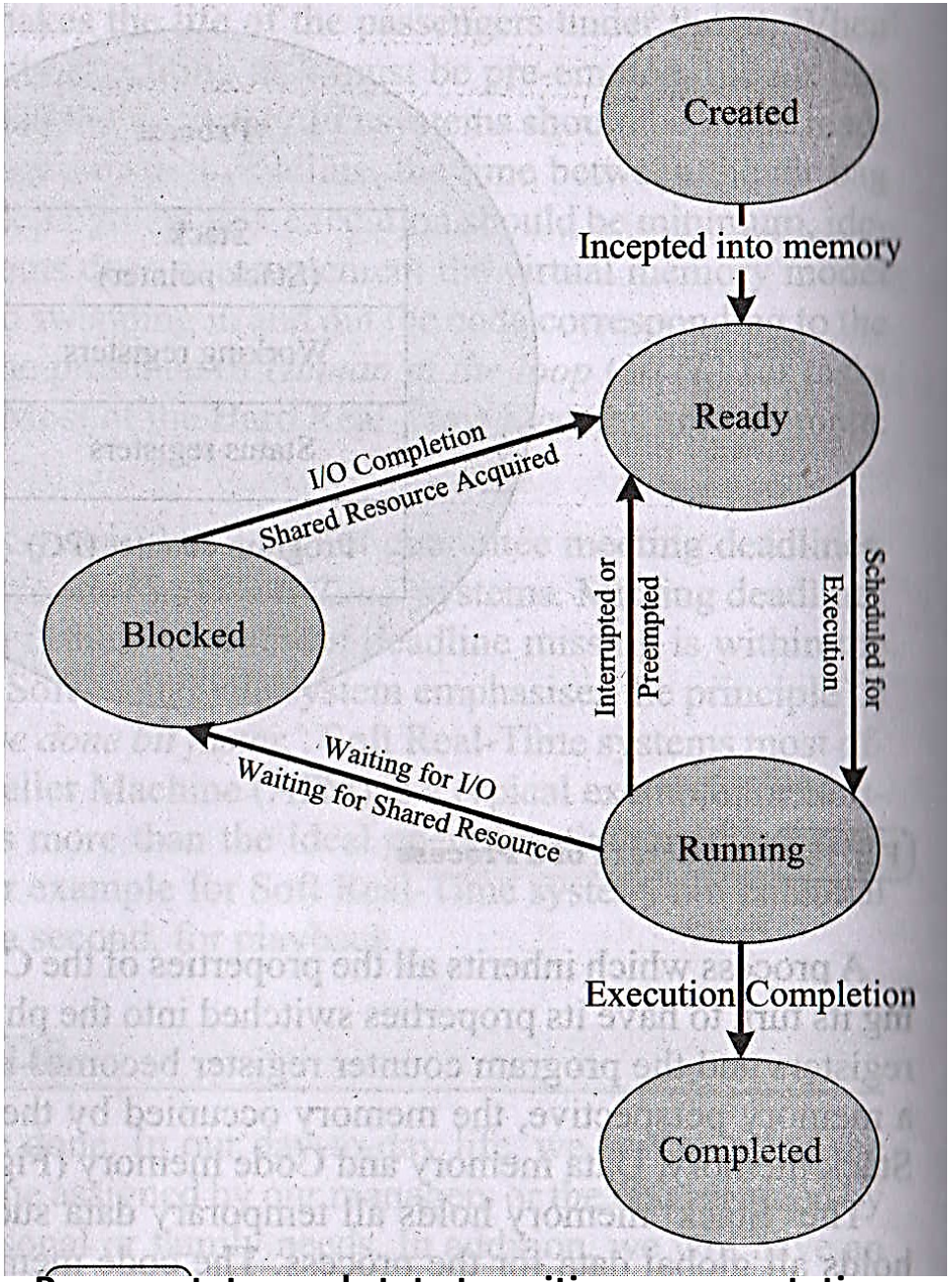


Code memory corresponding to the *Process*

Process Life Cycle – process changes its state from *newly created* to *execution completed*

Created state – a process is being created is referred. OS recognizes a process but no resources are allocated to the process.

Ready State – the state, where a process is incepted into the memory and awaiting the processor time for execution.

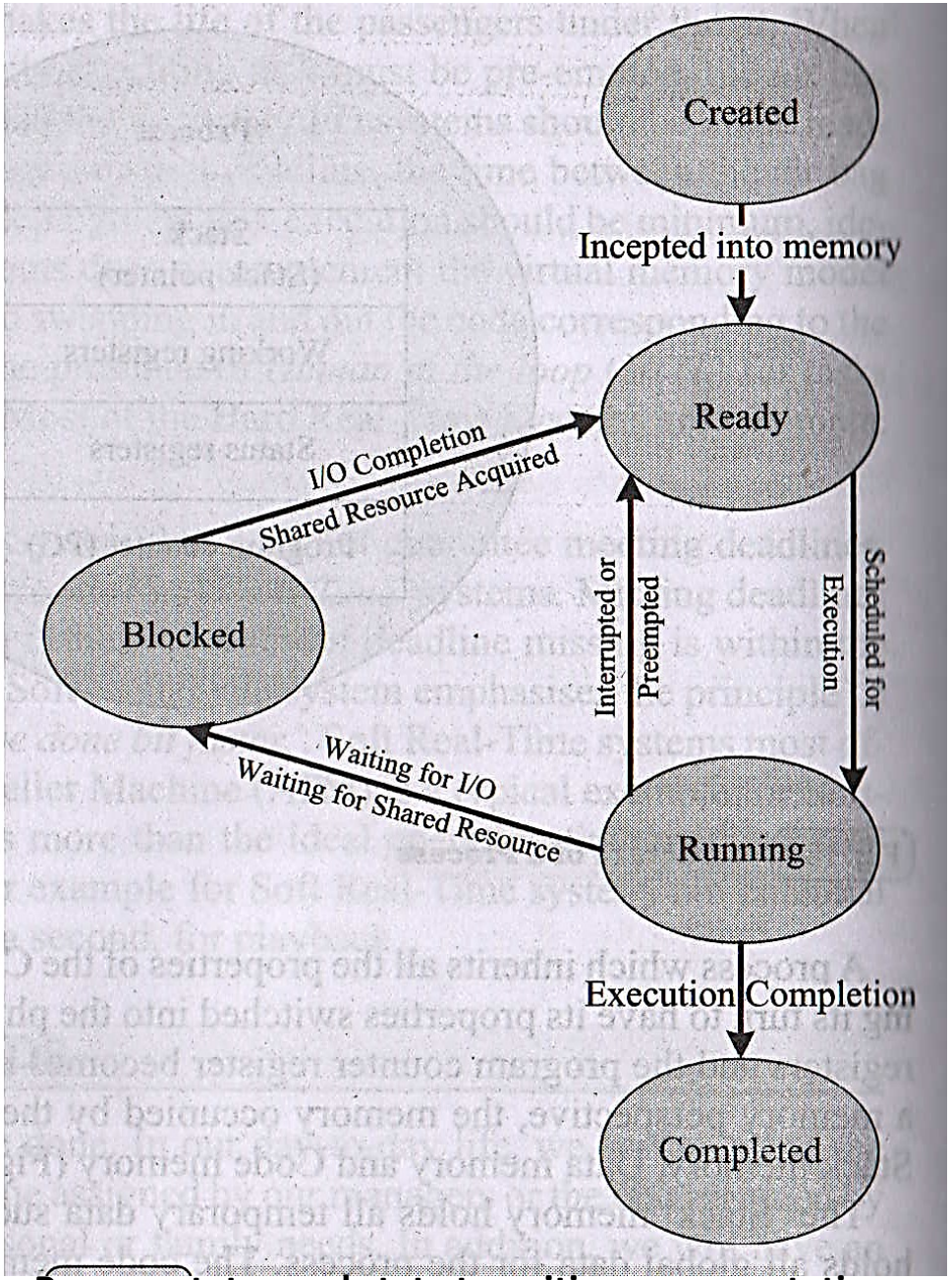


Process states and state transition representation

Ready List – queue maintained by the OS.

Running State – the state where in the source code instructions corresponding to the process is being executed.

Blocked State/Wait state – refers to a state where a running process is temporarily suspended from execution and does not have immediate access to resources..

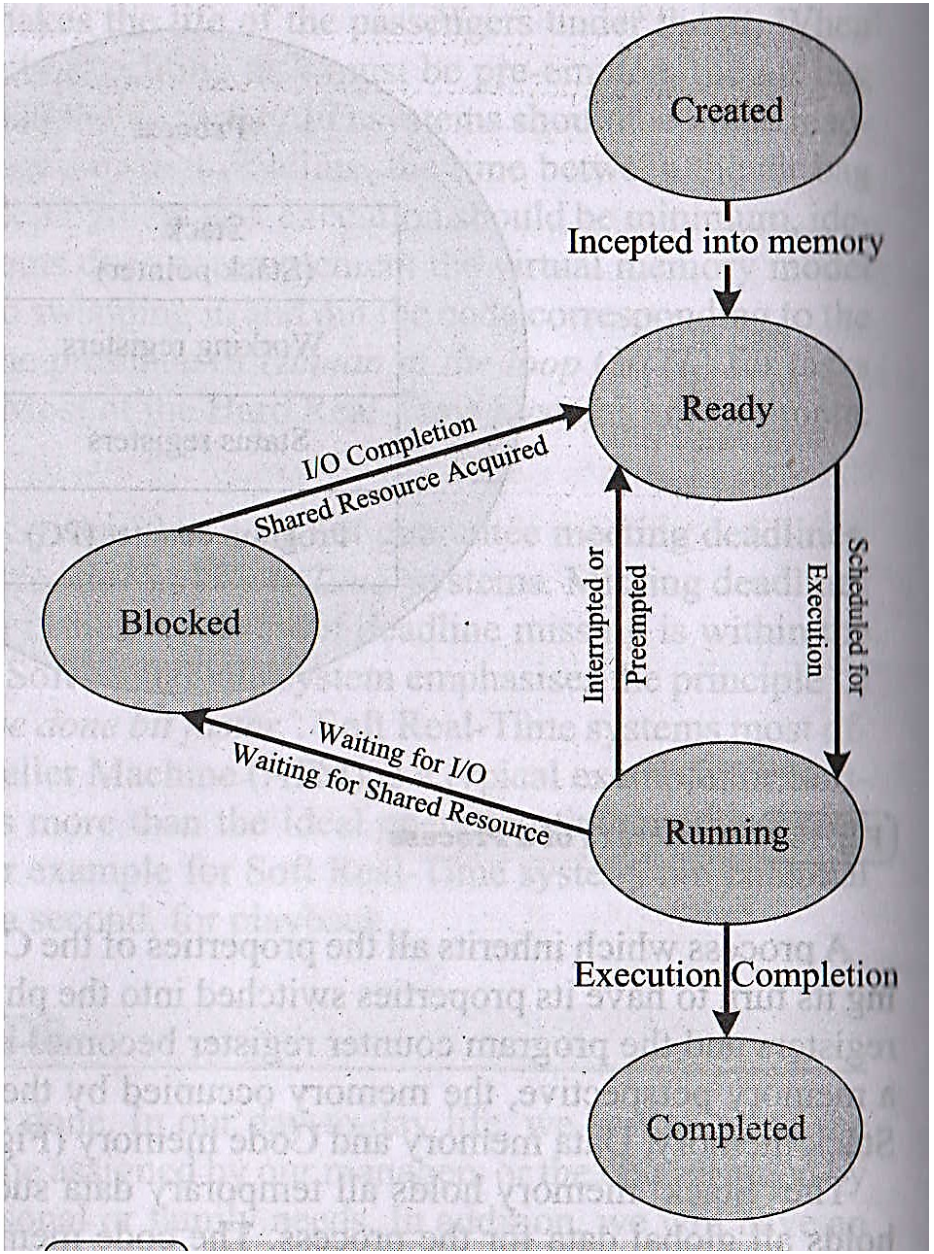


Process states and state transition representation

Completed State – a state where the process completes its execution

State transition – the transition of a process from one state to another

Process Management – deals with the creation of a process, setting up the memory space for the process, loading the process's code into the memory space, allocating system resources, setting up a Process Control Block (PCB) for the process and process termination / deletion.



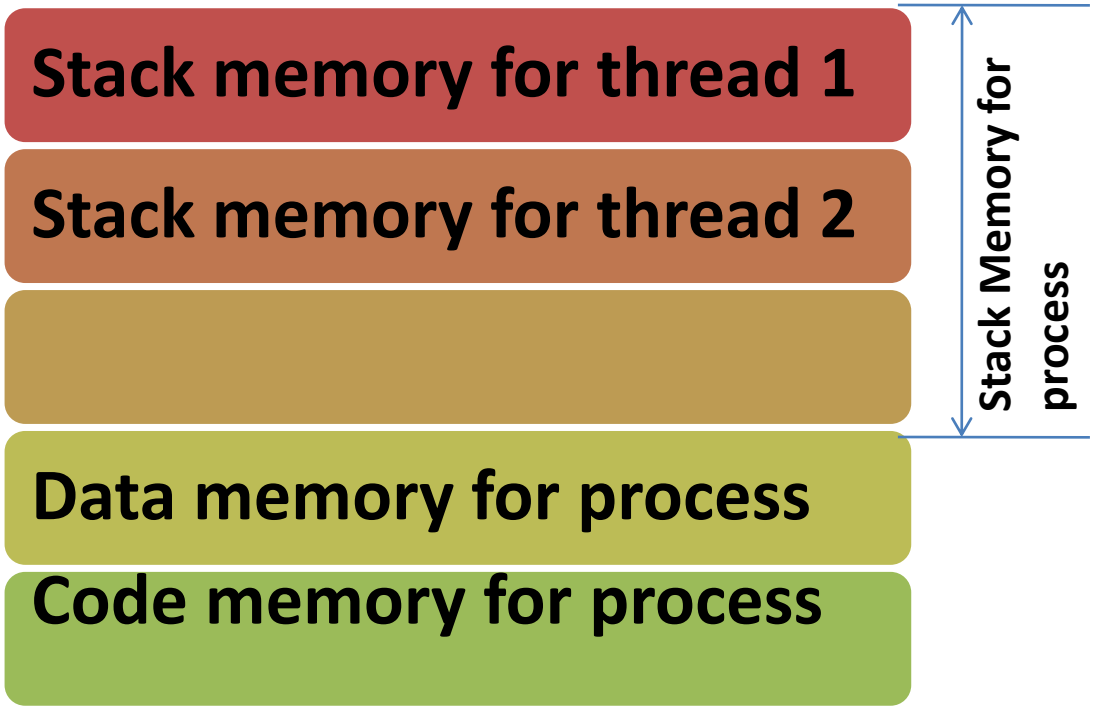
Process states and state transition representation

Process Management

- Deals with the creation of a process
- Setting up the memory space for the process
- Loading the process's code into the memory space
- Allocating system resources
- Setting up a Process Control Block (PCB) for the process termination / deletion

Threads:

- Is the primitive that can execute code
- Is a single sequential flow of control within a process
- Also known as light weight process



Memory organization of a process and its associated Threads

- A process can have many threads of execution

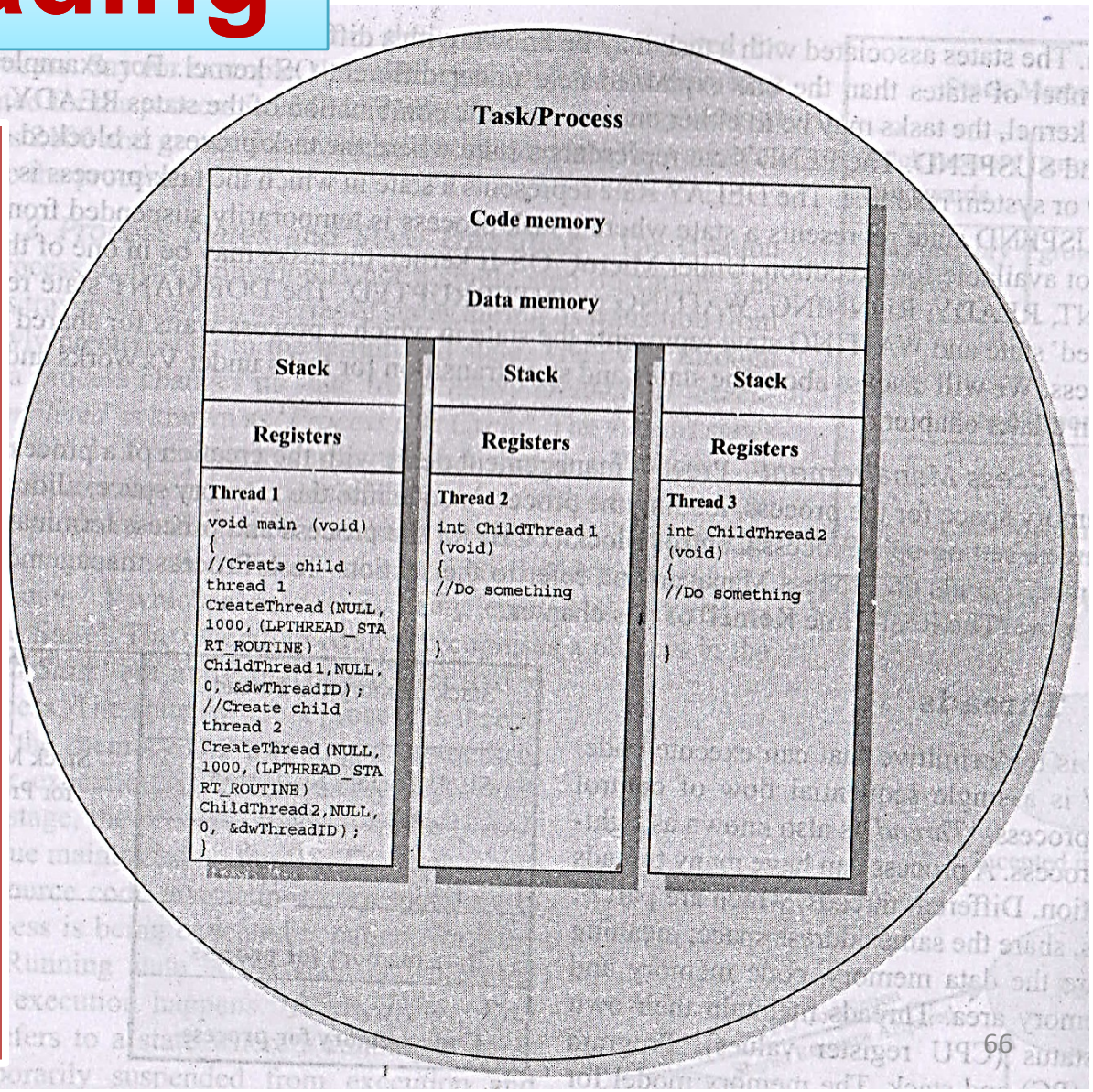
Threads: contd. …

- Different threads, which are part of a process, share the same address space; meaning they share the data memory, code memory and the heap memory area.

- Threads maintain their own thread status (CPU register values), Program Counter (PC) and stack.

Multithreading

- Application may complex and lengthy
- Various sub - operations like getting input from I/O devices connected to the processor
- Performing some internal calculations / operations
- Updating some I/O devices



Multithreading … …

all the sub-functions of a task are executed in sequence (?) – the CPU utilization may not be efficient

Advantages of multiple threads to execute:

- **Better memory utilization** ***(same process share the address space of the same memory & reduces complexity of inter threads comm.)***
- **Speed up execution of the process** ***(splitting into different threads, when one thread enters a wait state, the CPU can be utilized by the other threads of the process that do not require the event, which other thread is waiting, for processing)***
- **Efficient CPU utilization. CPU – engaged all time.**

Multithreading … …

Thread Standards: deals with different standards available for thread creation and management; utilized by OS

Thread Class libraries are:

- **POSIX Threads (Portable Operating System Interface)**
- **Win 32 Threads**
- **Java Threads**

Multithreading … …

- **POSIX Threads (Portable Operating System Interface)**

POSIX.4 standard deals with the Real-Time extensions
POSIX.4a standard deals with thread extensions

"Pthreads" library defines the set of POSIX thread creation and management functions in C language

Write a multithreaded application to print "Hello I'm in main thread" from the main thread and "Hello I'm in new thread" 5 times each, using the *pthread_create()* and *pthread_join()* POSIX primitives.

```
//Assumes the application is running on an OS where POSIX library is
//available
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
//*************************************************************
//New thread function for printing "Hello I'm in new thread"
void *new_thread( void *thread_args )
{
  int i, j;
  for( j= 0; j < 5; j++ )
   {
    printf("Hello I'm in new thread\n" );
    //Wait for some time. Do nothing
    //The following line of code can be replaced with
    //OS supported delay function like sleep(), delay () etc…
    for( i= 0; i < 10000; i++ );
   }
  return NULL;
}
```

```
//*************************************************************
//Start of main thread
int main( void )
{
  int i, j;
  pthread_t tcb;
//Create the new thread for executing new_thread function
  if (pthread_create( &tcb, NULL, new_thread, NULL ))
    {
      //New thread creation failed
      printf("Error in creating new thread\n" );
      return -1;
    }
  for( j= 0; j < 5; j++ )
   {
    printf("Hello I'm in main thread\n" );
    //Wait for some time. Do nothing
    //The following line of code can be replaced with
    //OS supported delay function like sleep(), delay etc…
    for( i= 0; i < 10000; i++ );
   }
  if (pthread_join(tcb, NULL ))
    {
      //Thread join failed
      printf("Error in Thread join\n" );
      return -1;
    }
  return 1;
}
```

Win 32 Threads:

- are the threads supported by various flavors of windows OS.
- Win 32 Application Programming Interface (Win 32 API) libraries provide the standard set of Win 32 thread creation and management functions.
- Win 32 threads are created with the API

HANDLE CreateYThread (LPSECURITY_ATTRIBUTES lpThreadAttributes, DWORD dwStackSize, LPTHREAD_START_ROUTINE lpStartAddresss, LPVOID lpParameter, DWORD dwCreationFlags, LPWORD lpThreadId) ;

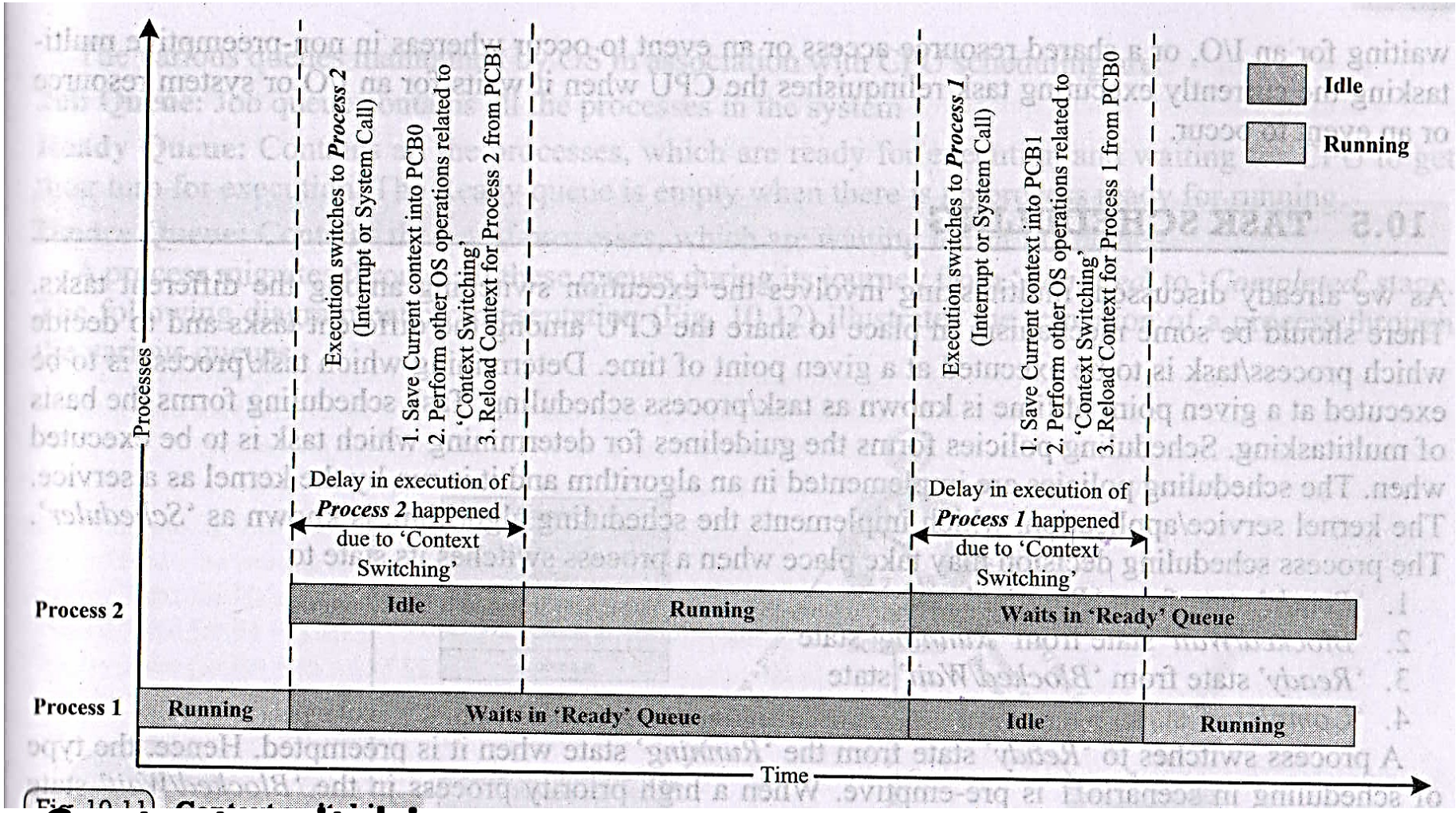
Thread

- is a single unit of execution and is part of process
- Does not have its own data memory and heap memory. Shares these memory with other threads of the same process
- Cannot live independently; it lives within the process
- Can be multiple threads in a process; the first thread (main thread) calls the main function and occupies the start of stack memory of the process
- Are very inexpensive to create
- Context switching is inexpensive and fast
- If a thread expires, its stack is reclaimed by the process

Process

- Is a program in execution & contains 1 or more threads
- Has its own code memory, data memory & stack memory
- Contains at least one thread
- Threads within a process share the code, data & heap memory. Each thread holds separate memory area for stack (shares the total stack memory of the process)
- Are very expensive to create. Involves many OS overhead
- Context switching is complex and involves lot of OS overhead & is comparatively slower
- If process dies, the resources allocated to it are reclaimed by OS & all the associated threads of the process also dies

Multiprocessing & Multitasking



Context switching

Real-Time Kernels

- A process is an abstraction of a running program and is the logical unit of work scheduled by OS
- Threads are light-weighted processes sharing resources of the parent process
- RTOS task management functions: scheduling, dispatching, intercommunication and synchronization

- The kernel of the OS is the smallest portion that provides for task management functions
- A scheduler determines which task will run next
- A dispatcher provides a necessary bookkeeping to start the next task
- Intertask communication and synchronization assures that the tasks cooperate

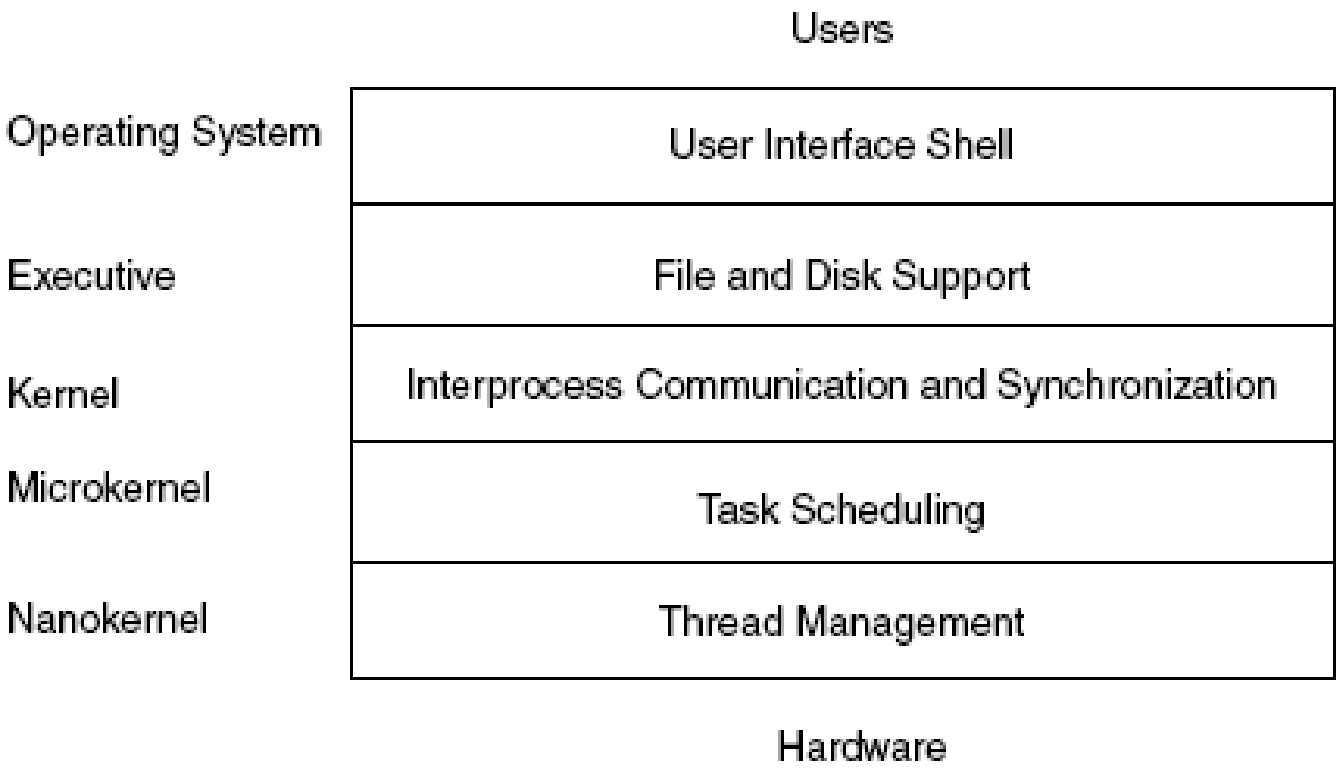


Figure 3.1 The role of the kernel in operating systems. Moving up the taxonomy from the low-level nanokernel to the full-featured operating system shows the additional functionality provided and also indicates the relative closeness to hardware versus human users.

Pseudo-kernels

- Polled Loop

```
For(;;){/*do forever*/
 if (packet_here){/*check flag*/
  process_data();/*process data*/  packet_here=0;/*reset flag*/
 }
}
```

- Synchronized polled loop

```
For(;;){/*loop forever*/
 if (flag){  pause(20); /* wait 20 ms to avoid switch-bounce*/
  process_event(); flag=0;
 }
}
```

Cyclic Executives

```
For(;;){/* do forever in round-robin fashion*/
 Process1();
 Process2();
 ..
 ProcessN();
}
```

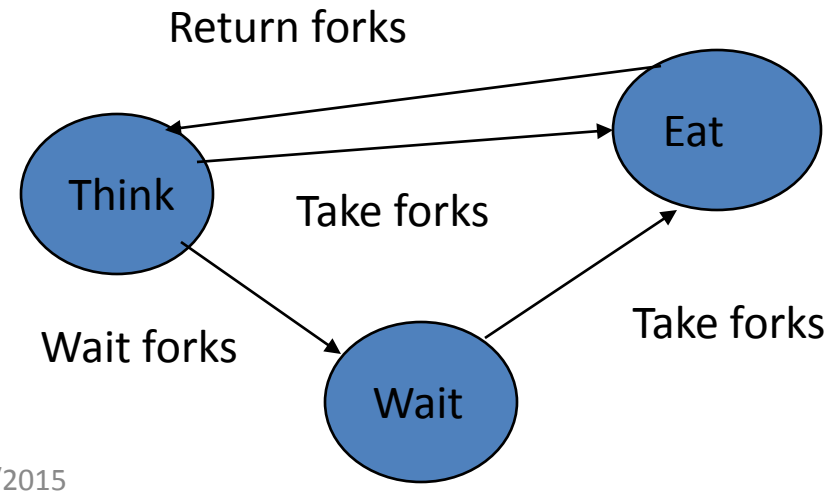
Different rates example:

```
For(;;){/* do forever in round-robin fashion*/
 Process1();
 Process2();
 Process3();/*process 3 executes 50% of the time*/
 Process3();
}
```

State-Driven Code

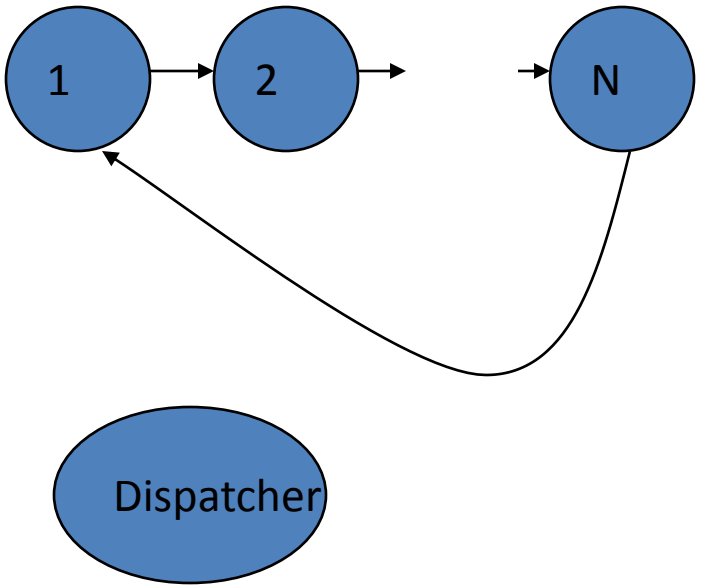
It uses if-then, case statements or finite state automata to break up processing of functions into code segments

```
For(;;){/*dining philosophers*/
 switch (state)
  case Think: pause(random()); state=Wait; break;
  case Wait: if (forks_available()) state=Eat;
  case Eat: pause(random()); return_forks(); state=Think;
 }
}
```



Coroutines

```
Void process_i(){//code of the i-th process
 switch (state_i){// it is a state variable of the i-th process
 case 1: phase1_i(); break;
 case 2: phase2_i(); break;
 ..
 case N: phaseN_i();break;
 }
}
Dispatcher(){
For(;;){ /*do forever*/
 process_1();
 ..
 process_M();
}
```



Interrupt-Driven Systems

Interrupt Service Routine (ISR) takes action in response to the interrupt

Reentrant code can be used by multiple processes. Reentrant ISR can serve multiple interrupts. Access to critical resources in mutually exclusive mode is obtained by disabling interrupts

On context switching save/restore:

- General registers
- PC, PSW
- Coprocessor registers
- Memory page register
- Images of memory-mapped I/O locations

The stack model is used mostly in embedded systems

Pseudocode for Interrupt Driven System

```
Main(){//initialize system, load interrupt handlers
 init();
 while(TRUE);// infinite loop
}
Intr_handler_i(){// i-th interrupt handler
 save_context();// save registers to the stack
 task_i(); // launch i-th task
 restore_context();// restore context from the stack
}
```

Work with a stack:

Push x: SP-=2; *SP=x;

Pop x: x=*SP; SP+=2;

Preemptive Priority System

A higher-priority task is said to preempt a lower-priority task if it interrupts the lower-priority task

The priorities assigned to each interrupt are based on the urgency of the task associated with the interrupt

Prioritized interrupts can be either priority or dynamic priority

Low-priority tasks can face starvation due to a lack of resources occupied by high-priority tasks

In rate-monotonic systems higher priority have tasks with higher frequency (rate)

Hybrid systems

Foreground-background systems (FBS)– polling loop is used for some job (background task – self-testing, watchdog timers, etc)

Foreground tasks run in round-robin, preemptive priority or hybrid mode

FBS can be extended to a full-featured real-time OS

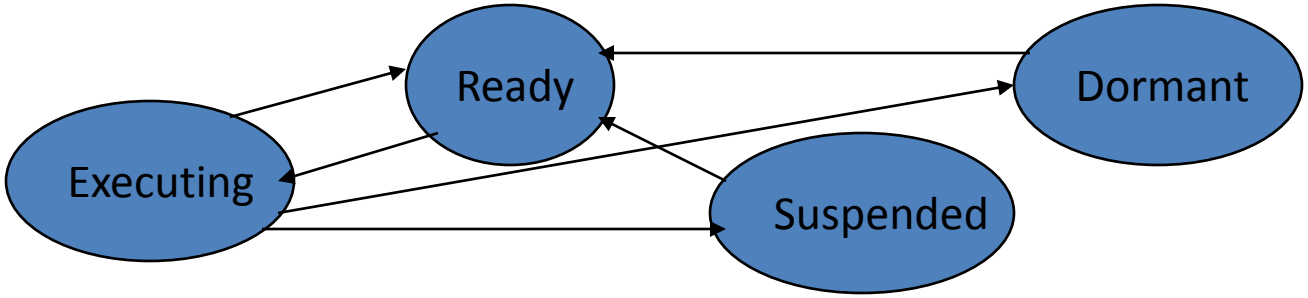
The Task Control Model of Real-Time Operating System

Each task is associated with a structure called Task Control Block (TCB). TCB keeps process' context: PSW, PC, registers, id, status, etc

TCBs may be stored as a linked list

A task typically can be in one of the four following states:

1) Executing; 2) Ready; 3) Suspended (blocked); 4) Dormant (sleeping)



RTOS maintains a list of the ready tasks' TCBs and another list for the suspended tasks

When a resource becomes available to a suspended task, it is activated

Process Scheduling

Pre-run time and run-time scheduling. The aim is to meet time restrictions

Each task is characterized typically by the following temporal parameters:

1) Precedence constraints; 2) Release or Arrival time $r_{i,j}$ of j-th instance of task i; 3) Phase φ_i ; 4) Response time; 5) Absolute deadline d_i

6) Relative deadline D_i

7) Laxity type – notion of urgency or margin in a task's execution

8) Period p_i

9) Execution time e_i

$$\varphi_i = r_{i,1} \qquad r_{i,k} = \varphi_i + (k-1)p_i$$

$$d_{i,k} = \varphi_i + (k-1)p_i + D_i$$

Assume for simplicity: all tasks are periodic and independent, relative deadline is a period/frame, tasks are pre-emptible, preemption time is neglected

Round-Robin Scheduling

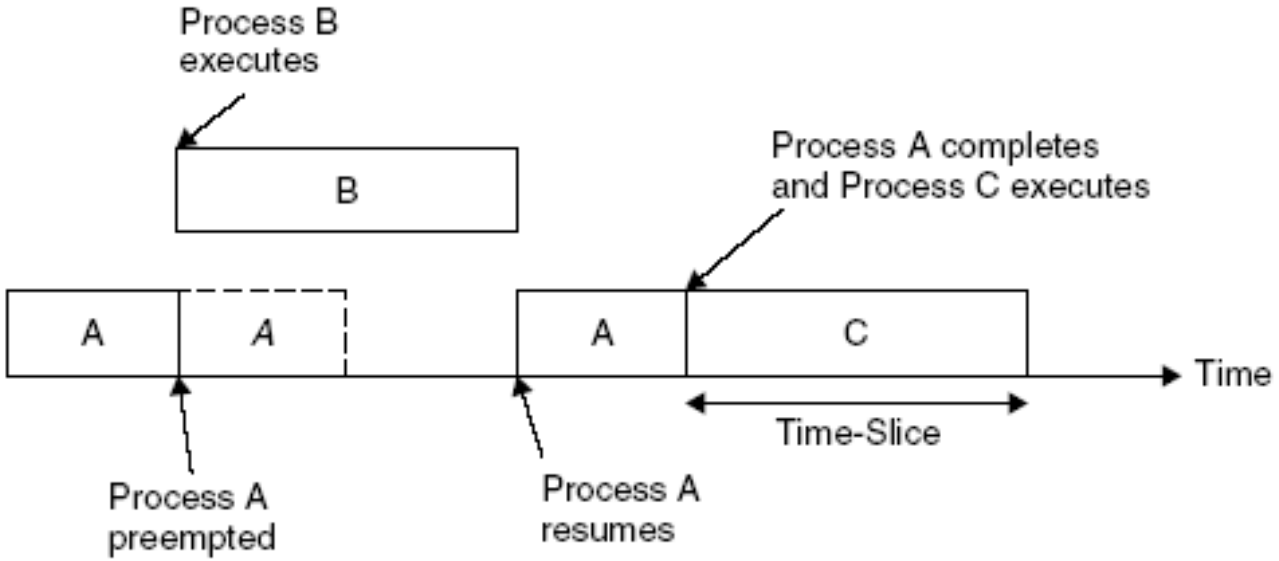


Figure 3.6 Mixed scheduling of three tasks.

Cyclic Executives

Scheduling decisions are made periodically, rather than at arbitrary times

Time intervals during scheduling decision points are referred to as frames or minor cycles, and every frame has a length, *f*, called the frame size

The major cycle is the minimum time required to execute tasks allocated to the processor, ensuring that the deadlines and periods of all processes are met

The major cycle or the hyperperiod is equal to the least common multiple (lcm) of the periods, that is, *lcm*(p_1,..,p_n)

Scheduling decisions are made at the beginning of every frame. The phase of each task is a non-negative integer multiple of the frame size.

Frames must be long enough to accommodate each task:

$$C_1 : f \geq \max_{1 \leq i \leq n} e_i$$

Cyclic Executives

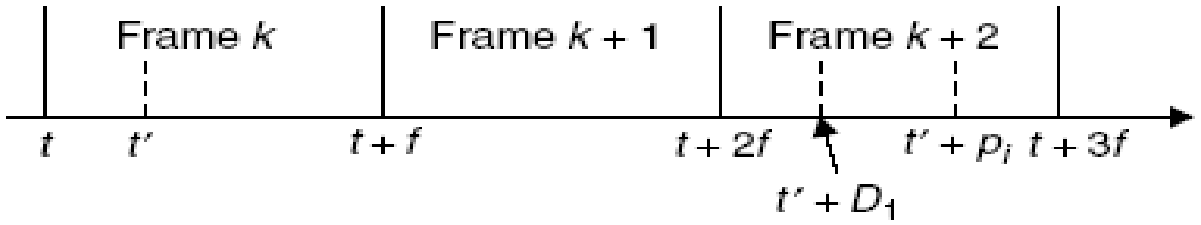


Figure 3.7 Constraints on the value of frame size.

Hyper period should be a multiple of the frame size:

$$C_2 : \lfloor p_i / f \rfloor - p_i / f = 0$$

To insure that every task completes by its deadline, frames must be small so that between the release time and deadline of every task, there is at least one frame.

Cyclic Executives

The following relation is derived for a worst-case scenario, which occurs when the period of a process starts just after the beginning of a frame, and, consequently, the process cannot be released until the next frame:

$$C_3 : 2f - \gcd(p_i, f) \leq D_i$$

$t' > t$:

$$t + 2f \leq t' + D_i$$

$$2f - (t' - t) \leq D_i$$

$$t' - t = lp_i - kf \geq lp_i - kf \geq \gcd(p_i, f)$$

$$f \leq 2f - \gcd(p_i, f) \leq D_i$$

Cyclic Executives

To illustrate the calculation of the framesize, consider the set of tasks shown in Table 3.1. The hyperperiod is equal to 660, since the least common multiple of 15, 20, and 22 is 660. The three conditions, C_1, C_2 and C_3 are evaluated as follows:

$$C_1 : \forall i f \geq e_i \Rightarrow f \geq 3$$

$$C_2 : \lfloor p_i/f \rfloor - p_i/f = 0 \Rightarrow f = 2, 3, 4, 5, 10, \ldots$$

$$C_3 : 2f - gcd(p_i, f) \leq D_i \Rightarrow f = 2, 3, 4, 5$$

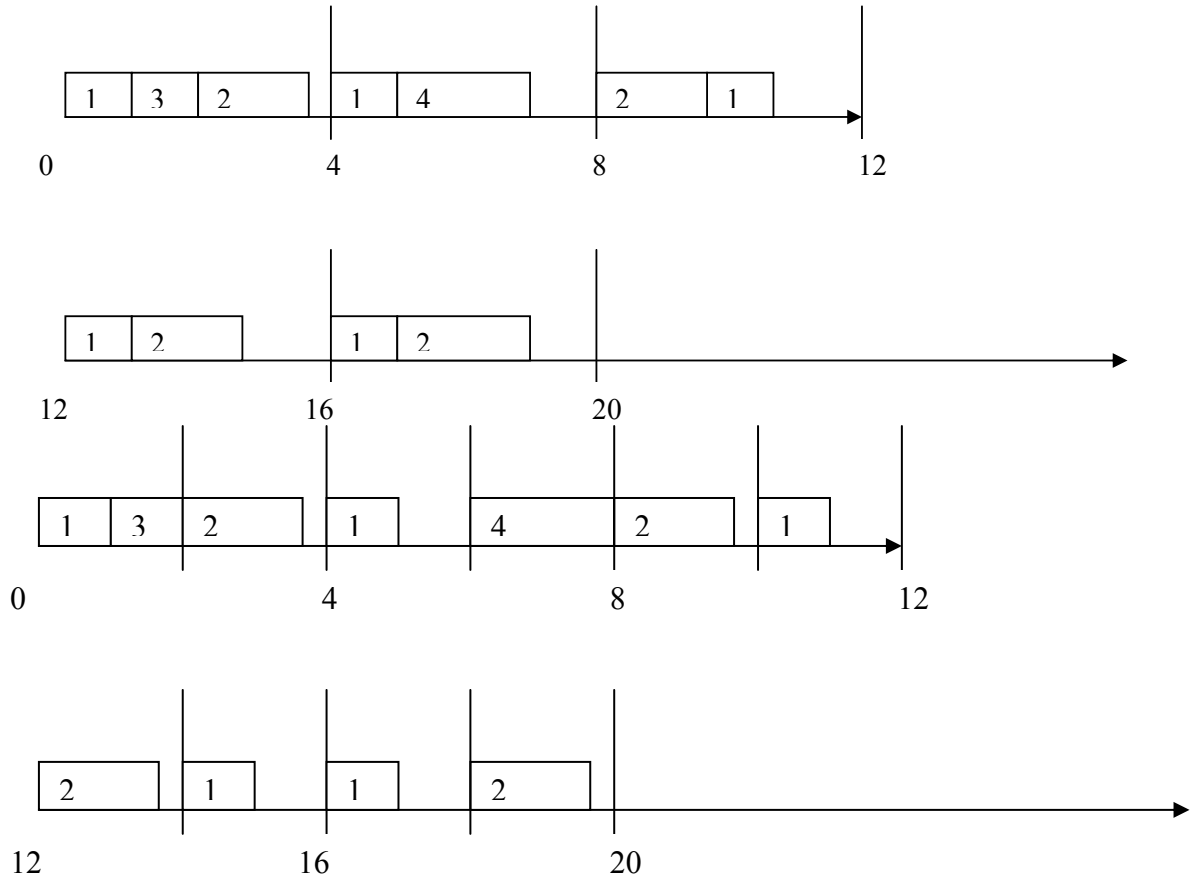
From these three conditions, it can be inferred that a possible value for f could be any one of the values of 3, 4, or 5.

Table 3.1 Example task set for framesize calculation

τ_i	p_i	e_i	D_i
τ_2	15	1	14
τ_3	20	2	26
τ_4	22	3	22

Cyclic Executives

For example, for tasks T1(4,1), T2(5,1.8), T3(20,1), T4(20,2), hyper-period is 20 (without and with frames – *f=2*)



Fixed Priority Scheduling – Rate-Monotonic Approach (RMA)

Table 3.2 Sample task set for utilization calculation

τ_i	e_i	p_i	$u_i = e_i / p_i$
τ_1	1	4	0.25
τ_2	2	5	0.4
τ_3	5	20	0.25

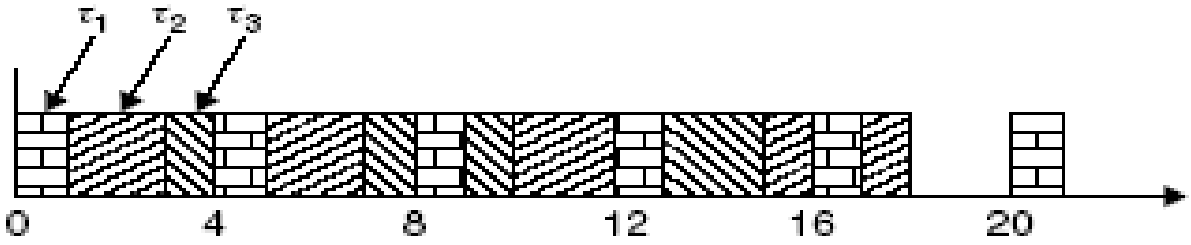


Figure 3.8 Rate-monotonic task schedule.

Rate-Monotonic Scheduling

Theorem (RMA Bound). Any set of n periodic tasks is RM schedulable if the processor utilization

$$U = \sum_{i=1}^{n} \frac{e_i}{p_i} \leq n(2^{1/n} - 1)$$

Table 3.3 Upper bound on utilization *U* for *n* tasks scheduled using the rate-monotonic discipline

n	1	2	3	4	5	6	. . .	∞
RMA bound	1.0	0.83	0.78	0.76	0.74	0.73	. . .	0.69

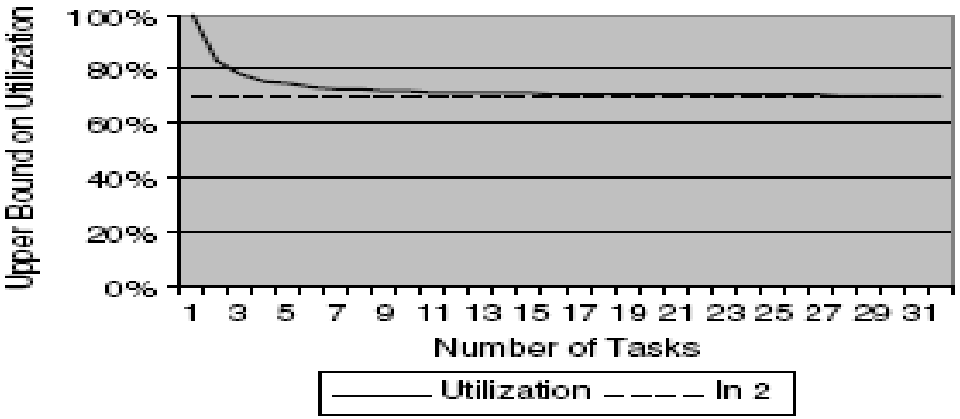


Figure 3.9 Upper bound on utilization in a rate-monotonic system as a function of the number of tasks. Notice how it rapidly converges to 0.69.

Dynamic-Priority Scheduling – Earliest-Deadline-First Approach

Theorem (EDF Bound). A set of n periodic tasks, each of whose relative deadline equals its period, can be feasibly scheduled by EDF if and only if

$$U \leq 1$$

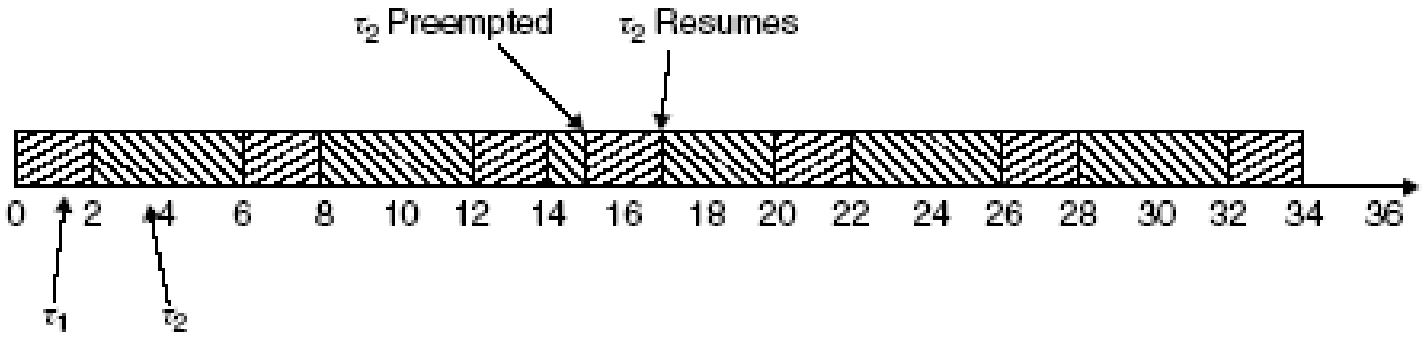


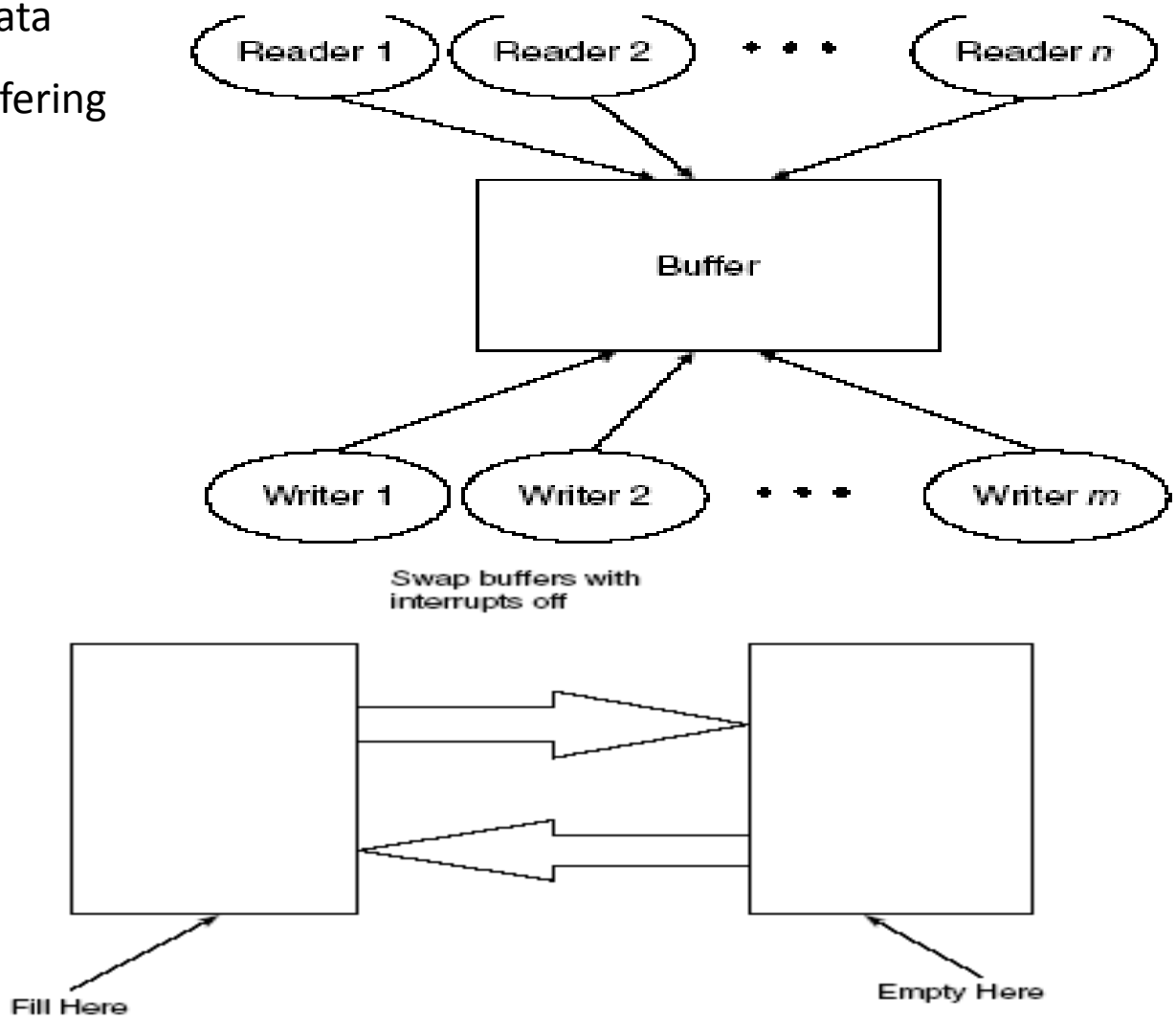
Figure 3.10 EDF task schedule for task set in Table 3.4.

Table 3.4 Task set for example of EDF scheduling

τ_i	e_i	p_i
τ_1	2	5
τ_2	4	7

Intertask Communication and Synchronization

- Buffering data
- Double-buffering



Intertask Communication and Synchronization

Ring Buffers

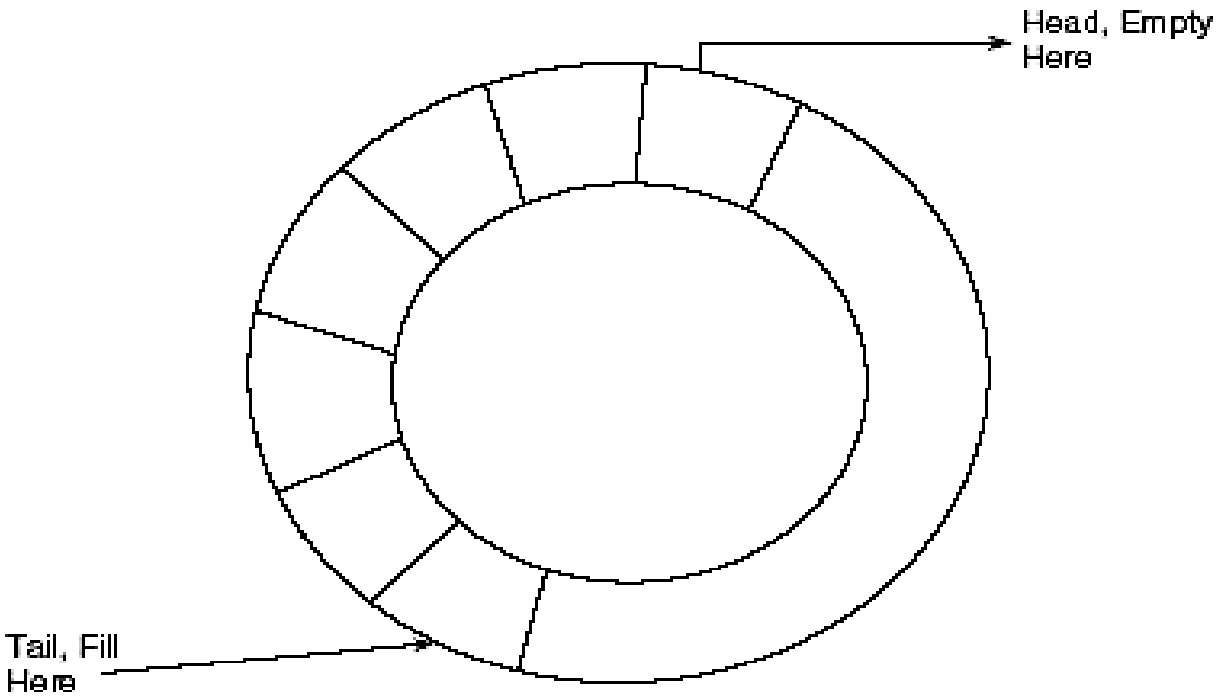


Figure 3.13 A ring buffer. Processes write to the buffer at the tail index and read data from the head index. Data access is synchronized with a counting semaphore set to size of ring buffer, to be discussed later.

Intertask Communication and Synchronization

```
typedef struct ring_buffer
{
    int contents[N];
    int head;
    int tail;
}
```

It is further assumed that the head and tail indices have been initialized to 0, that is, the start of the buffer.

An implementation of the read(data,S) and write(data,S) operations, which reads from and writes to ring buffer S, respectively, are given below in C code.

```
void read (int data, ring_buffer *s)
{
    if (s->head==s->tail)
     data=NULL;                    /* underflow */
    else
     {
       data=s->contents +head;     /* retrieve data from buffer */
       s->head=(s->head+1) % N;    /* decrement head index */
     }
}
void write (int data, ring_buffer *s)
{
    if ((s->tail+1) %N==head)
     error();                      /* overflow, invoke error handler */
    else
    {
       s->contents+tail=data;
       tail=(tail+1) % N;          /*take care of wrap-around */
    }
}
```

Intertask Communication and Synchronization

Mailbox: void pend (int data, s); void post (int data, s);

Access to mailbox is mutually exclusive; tasks wait access granting

Table 3.6 Task resource request table

Task ID #	Resource	Status
100	Printer	Has it
102	Mailbox 1	Has it
104	Mailbox 1	Pending

Table 3.7 Resource table used in conjunction with task resource request table

Resource	Status	Owner
Printer 1	Busy	100
Mailbox 1	Busy	102
Mailbox 2	Empty	None

Intertask Communication and Synchronization

- Queues – can be implemented with ring buffers
- Critical regions – sections of code to be used in the mutually exclusive mode
- Semaphores – can be used to provide critical regions

```
void P(int S)
{
     while (S == TRUE);
       S=TRUE;
}

void V(int S)
{
       S=FALSE;
}
```

```
Process_1
.
.
.
P(S)
critical region
V(S)
.
.
.
```

Intertask Communication and Synchronization

Mailboxes and Semaphores

```
void P(int S)
{
  int KEY=0;

  pend(KEY,S);
}
```

The accompanying signal operation utilizes the mailbox `post` operation.

```
void V(int S)
{
   int KEY=0;

   post(KEY,S);
}
```

Intertask Communication and Synchronization

Semaphores and mailboxes

```
Sema mutex=0/*open*/, proc_sem=1;/*closed*/
Bool full_slots=0, empty_slots=1;
Void post( int mailbox, int message){
  while (1){ wait(mutex);
            if (empty_slots){
                 insert(mailbox, message); update(); signal(mutex);
                   signal(proc_sem); break;
            }
            else{ signal(mutex); wait(proc_sem);
            }
   }
}
```

Intertask Communication and Synchronization

Semaphores and mailboxes

```
Void pend( int mailbox, int *message){
  while (1){ wait(mutex);
              if (full_slots){
                   extract(mailbox, message); update(); signal(mutex);
                     signal(proc_sem); break;
              }
              else{ signal(mutex); wait(proc_sem);
              }
   }
}
```

Intertask Communication and Synchronization

```
Driver{ while(1){
        if(data_for_I/O){
         prepare(command);
         V(busy); P(done);}
 }}
Controller{while(1){
  P(busy); exec(command);
  V(done);
}}
```

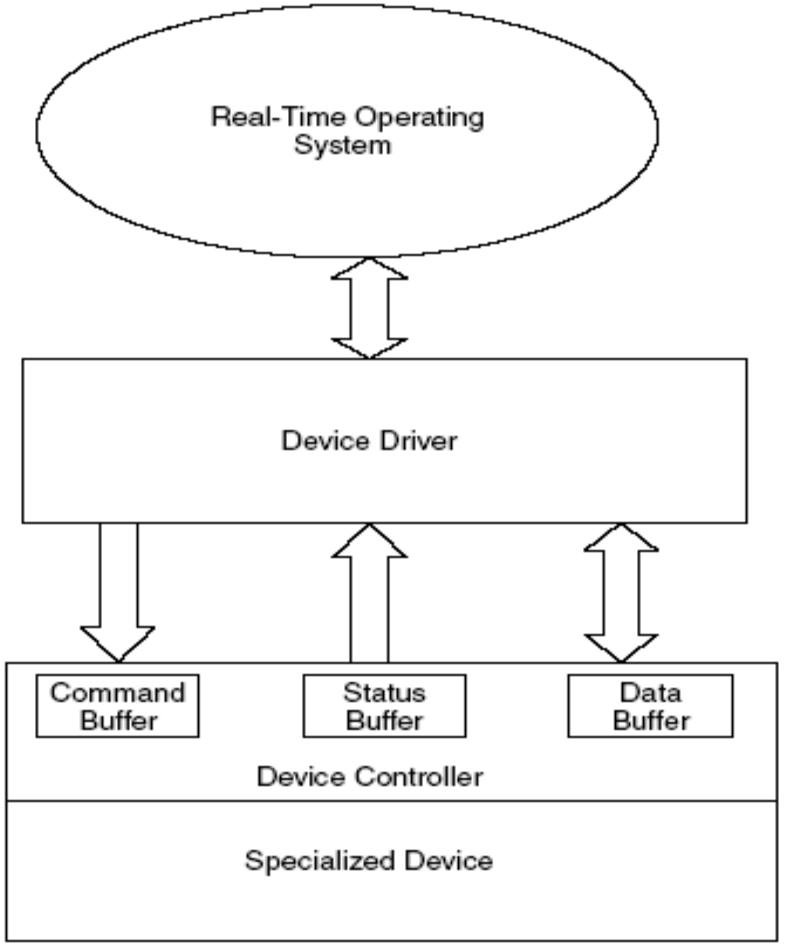


Figure 3.14 The device–controller interface.

Intertask Communication and Synchronization

Counting Semaphores:

Wait: void MP(int &S){

S=S-1; while(S<0);

}

Signal: void MV(int &S){

S=S+1

}

Intertask Communication and Synchronization

Some real-time kernels provide only binary semaphores, but the counting semaphore can be simulated with binary semaphores in the following way. Suppose S and T are binary semaphores and P(S) and V(S) are the `wait` and `signal` operations, respectively, on binary semaphores. The counting semaphore operations MP(R) and MV(R) can be created on multiple semaphores R using global binary semaphores S and T and integer R as follows:

```
void MP (int R)              /* multiple wait */
{
    P(S);                    /* lock counter       */
    R=R-1;                   /* request a resource */
    if(R < 0)                /* none available?    */
    {
       V(S);                 /* release counter    */
       P(T);                 /* wait for free resource   */
    };
    V(S);                    /* release counter    */
}

void MV(int R)               /* multiple signal    */
{
    P(S);                    /* lock counter       */
    R=R+1;                   /* free resource      */
    if (R <= 0)              /* give that task the go ahead */
      V(T);
    else
       V(S);                 /* release counter}
}
```

Intertask Communication and Synchronization

Problems with semaphores:

Wait: void P(int &S){

```
        while(S==TRUE);
        S=TRUE;
}
```

```
        LOAD    R1,S        ; address of S in R1
        LOAD    R2,1        ; 1 in R2
@1      TEST    R1,I,R2     ; compare (R1)=*S with R2=1
        JEQ     @1          ; repeat if *S=1
        STORE   R2,S,I      ; store 1 in *S
```

Interruption between JEQ and STORE, passing control to a next process, can cause that several processes will see *S=FALSE

Intertask Communication and Synchronization

The Test-and-Set Instruction

```
Void P(int &S){
  while(test_and_set(S)==TRUE);//wait
}
Void V(int &S){
 S=FALSE;
}
```

The instruction fetches a word from memory and tests the high-order (or other) bit . If the bit is 0, it is set to 1 and stored again, and a condition code of 0 is returned. If the bit is 1, a condition code of 1 is returned and no store is performed. The fetch, test and store are indivisible.

Intertask Communication and Synchronization

Dijkstra's implementation of semaphore operation (if test-and-set instruction is not available):

```
Void P(int &S){
  int temp=TRUE;
  while(temp){
    disable();          //disable interrupts
    temp=S;
    S=TRUE;
    enable();           //enable interrupts
  }
}
```

Intertask Communication and Synchronization

Other Synchronization Mechanisms:

•Monitors (generalize critical sections – only one process can execute monitor at a time. Provide public interface for serial use of resources

•Events – similar to semaphores, but usually all waiting processes are released when the event is signaled. Tasks waiting for event are called blocked

Deadlocks

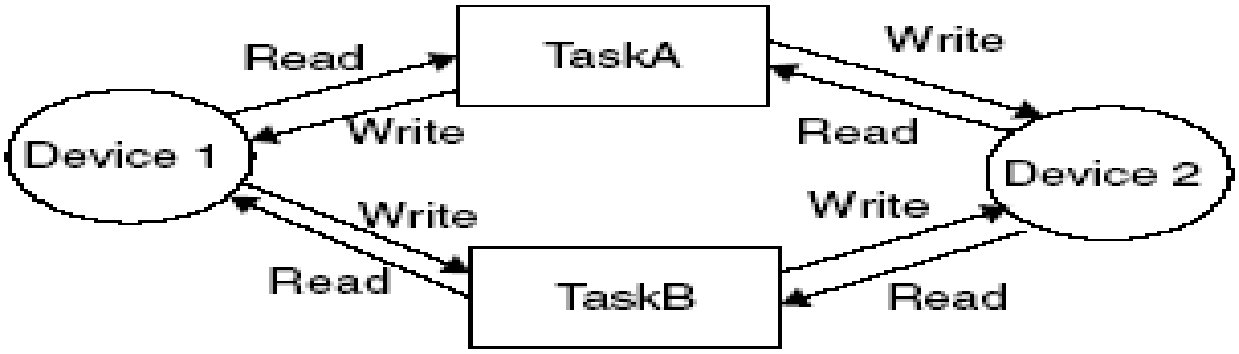


Figure 3.15 Deadlock realization in a resource diagram.

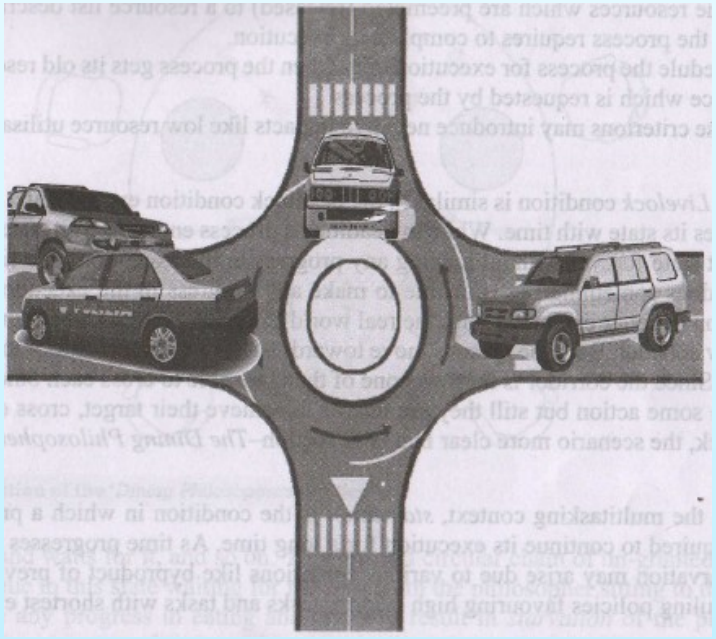
Intertask Communication and Synchronization

Deadllocks:

```
Task_A
    .
    .
    .
  P(S)
  use resource 1
    .
    .
    .
   P(R)
   stuck here
   use resource 2

   V(R)
   V(S)
    .
    .
    .
```

```
Task_B
    .
    .
    .
   P(R)
   use resource 2
    .
    .
    .

   P(S)
   stuck here
   use resource 1
   V(S)
   V(R)

    .
    .
    .
```

Deadlocks



Four conditions are necessary for deadlock:

- Mutual exclusion
- Circular wait
- Hold and wait
- No preemption

Eliminating any one of the four necessary conditions will prevent deadlock from occurring

One way to eliminate circular wait is to number resources and give all the resources with the numbers greater or equal than minimal required to processes. For example: Disk – 1, Printer – 2, Motor control – 3, Monitor – 4. If a process wishes to use printer, it will be assigned printer, motor control and monitor. If another process requires monitor, it will have wait until the monitor will be released. This may lead to starvation.

Deadlock avoidance

To avoid deadlocks, it is recommended :

- Minimize the number of critical regions as well as minimizing their size
- All processes must release any lock before returning to the calling function
- Do not suspend any task while it controls a critical region
- All critical regions must be error-free
- Do not lock devices in interrupt handlers
- Always perform validity checks on pointers used within critical regions.

It is difficult to follow these recommendations

A Separate Task Helps Control Shared Hardware