Processes and Multitasking

COE718: Embedded Systems Design

http://www. ecb.torontomu.ca/~courses/coe718/

Dr. Gul N. Khan

http://www.ecb.torontomu.ca/~gnkhan

Electrical, Computer and Biomedical Engineering

Toronto Metropolitan University

Overview

- Processes and Tasks
- Concurrency
- Scheduling Priorities and Policies
- Multitasking Techniques
- CPU Scheduling

Chapters 9 and 10 of Text by D. W. Lewis, Chapter 6 of Text by M. Wolf and ARM/RTX Documents

Introduction to Processes

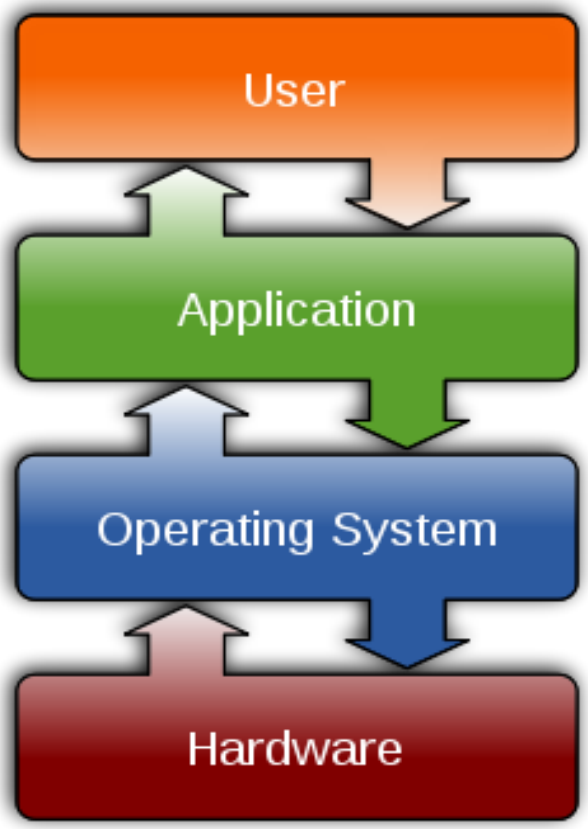
All multiprogramming operating systems are built around the concept of processes.

Operating System (OS) and Processes

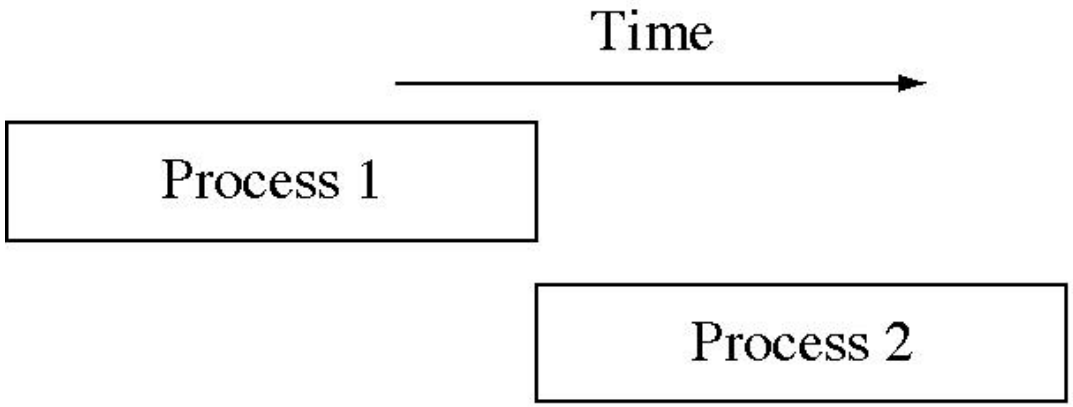
- OS must interleave the execution of several processes to maximize CPU usage.
- OS must allocate resources to processes.
- OS must also support:
  - IPC: Inter-process communication

Operating System (OS)

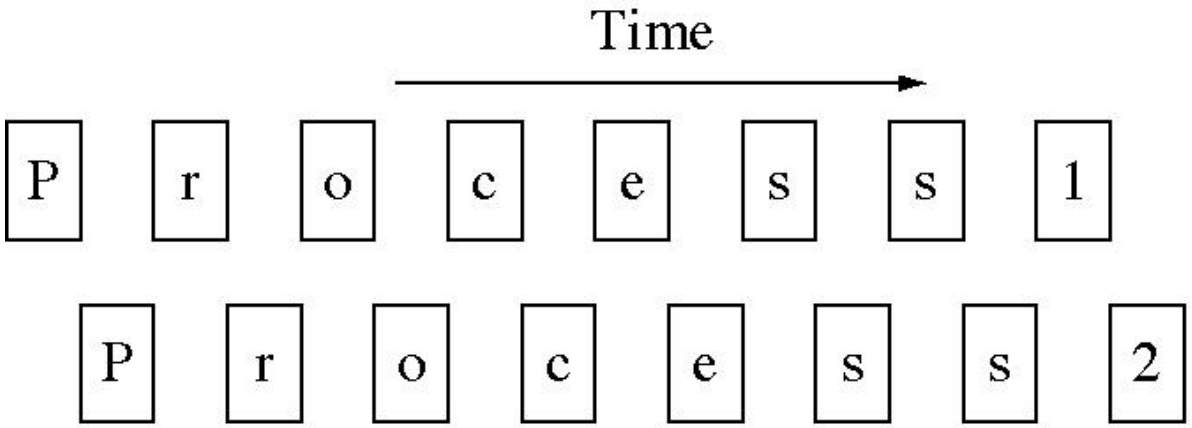
- OS is a Computer program that provides a software layer between the application software and the hardware.
- It provides three main functions:
  - Schedule task execution
  - Dispatch a task for execution.
  - Ensure communication and synchronization between tasks.



Task/Process Concept



Serial Execution of Two Processes



Interleaving the Execution of Process 1 and 2

Concurrency

- Only one thread runs at a time while others are *waiting*.
- Processor switches from one process to another so quickly that it appears all threads are running simultaneously. Processes run *concurrently*.
- Programmer assigns *priority* to each process and the s*cheduler* uses it to determine which process to run next.

Real-Time Kernel

- Processes call a library of run-time routines (known as the real-time *kernel*) manages resources.
- Kernel provides mechanisms to switch between processes, for coordination, synchronization, communications, and priority.

Multi-Tasking and Concurrency

- Most embedded systems have several inputs/outputs and multiple events occurring independently.
- Separating tasks simplifies programming, but requires somehow switching back and forth among multiple tasks (*multi-tasking*).
- *Concurrency* is the appearance of simultaneous execution of multiple tasks.

Concurrent Tasks for a Thermostat

```
/* Monitor Temperature */
do forever {
    measure temp ;
    if (temp < setting)
        start furnace ;
    else if (temp >
            setting + delta)
        stop furnace ;
}
```

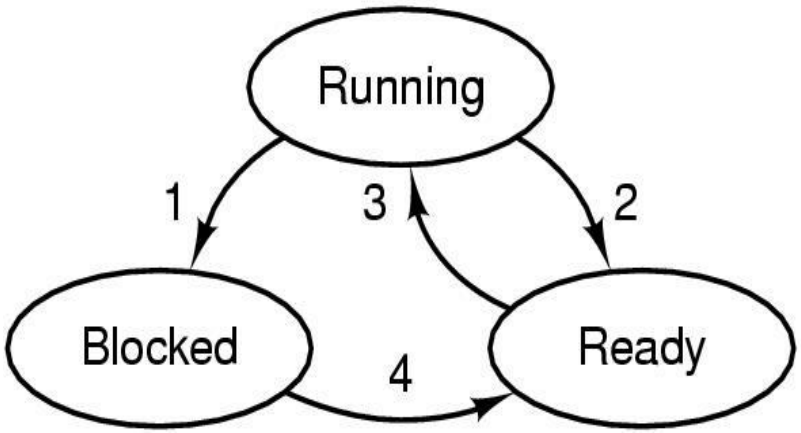
```
/* Monitor Time of Day */
do forever {
    measure time ;
    if (6:00am)
        setting = 72°F ;
    else if (11:00pm)
        setting = 60°F ;
}
```

```
/* Monitor Keypad */
do forever {
    check keypad ;
    if (raise temp)
        setting++ ;
    else if (lower temp)
        setting-- ;
}
```

Basic Process States

There are three basic states of a process

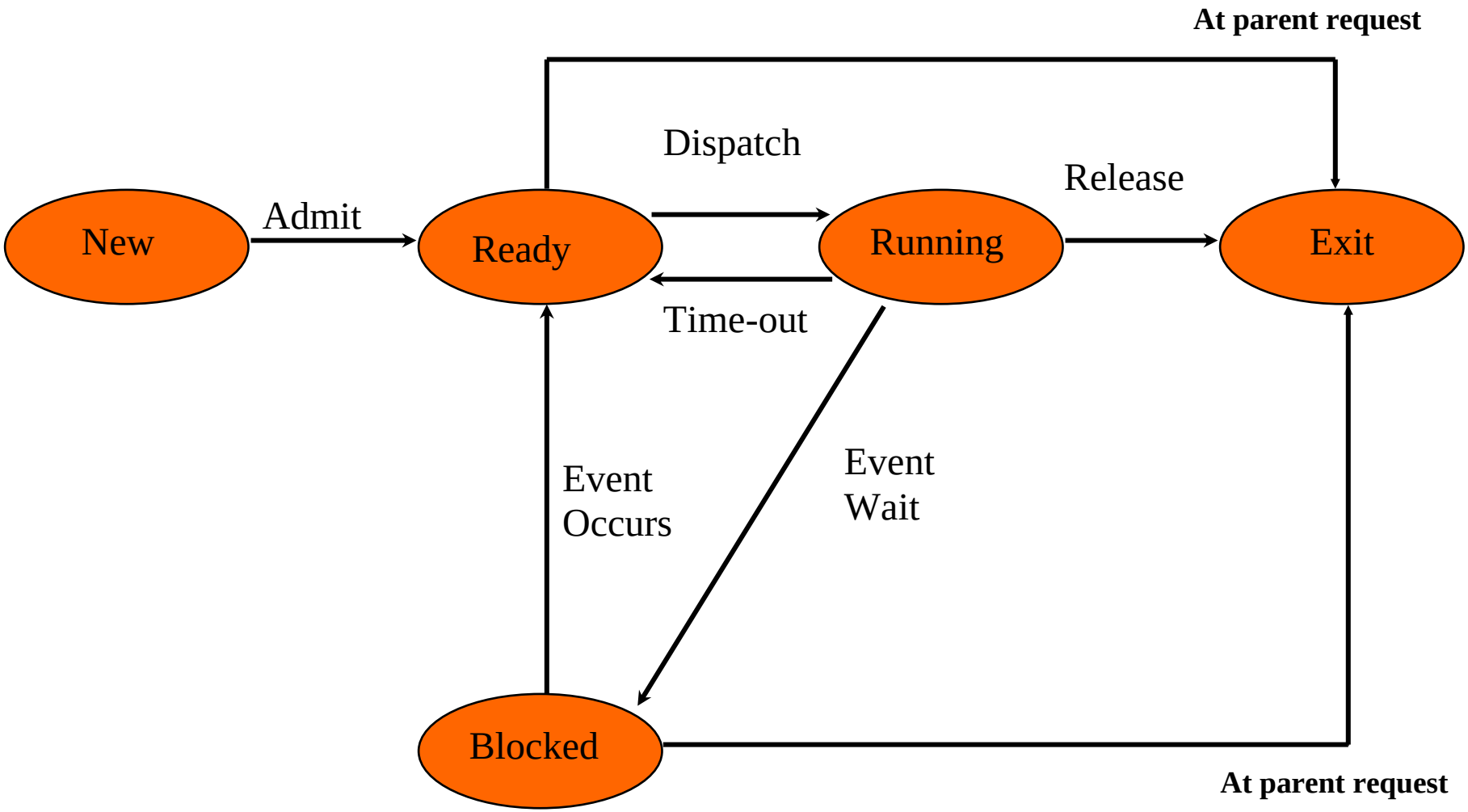
- The Running state
  - The process that gets executed.
- The Ready state
  - A process is ready to be executed.
- The Blocked state (Waiting)
  - When a process cannot execute until some event occurs.



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

More Process States

5-State Process Model



Modes of Execution

Most processors support at least two execution modes:

- Privileged mode
  - Manipulating control registers
  - Memory management ...
- User mode
  - Less-privileged mode
  - User programs execute in this mode

Therefore, CPU provides a (or a few) mode bit, which may only be set by an interrupt or trap or OS call

UNIX Processes

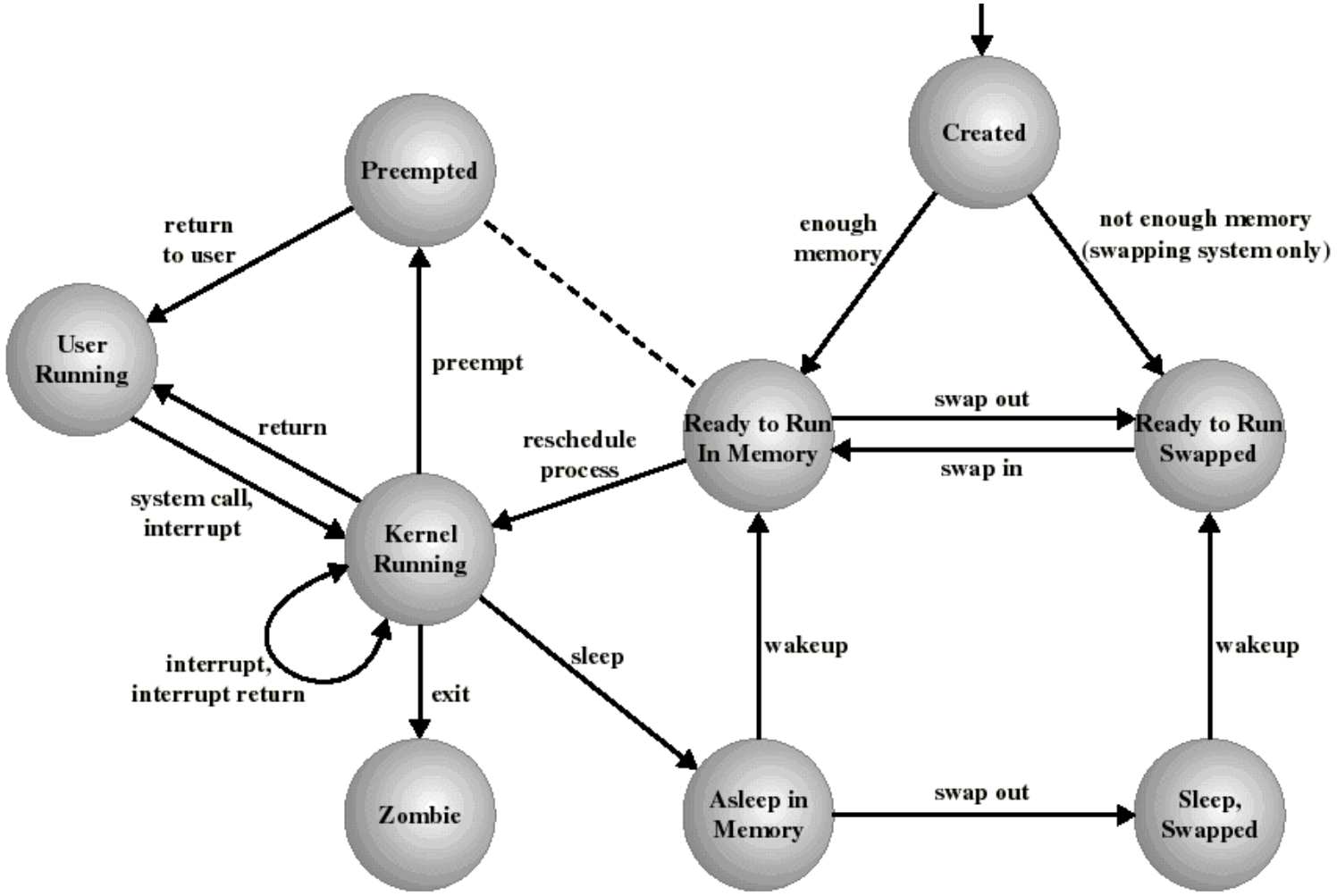
- 2 modes: User mode and Kernel mode.
- System processes run in Kernel mode.
- User processes run in user mode for user instructions and in kernel mode for OS/kernel instructions
- 9 states for processes

UNIX Process State

- Two running states for user or kernel modes.
- Pre-empted state is for processes returning from Kernel to user mode.

- A process running in Kernel mode cannot be pre-empted.

UNIX Process Transition Diagram



Two running states: User and Kernel

Preempted State: Kernel schedules another high priority process.

A Process running in Kernel mode cannot be preempted. That makes Unix/Linux unsuitable for real-time applications

UNIX Process Creation

Every process, except process 0, is created by the fork() system call.

- ***fork()*** allocates entry in process table and assigns a unique PID to the child process
- child gets a copy of process image of parent: both child and parent are executing the same code following fork().
- ***fork()*** returns the PID of the child to the parent process and returns 0 to the child process.

Process 0 is created at boot time and becomes the “swapper” after forking process 1 (the INIT process)

When a user logs in: process 1 creates a process for that user.

UNIX-style Process Creation

int fork()

- Creates an exact copy of the calling process.

int execve(char *progName, char *argv[ ])

- Runs a new program in the calling process
- Destroying the old program

int exit(int retCode)

- Exits the calling process

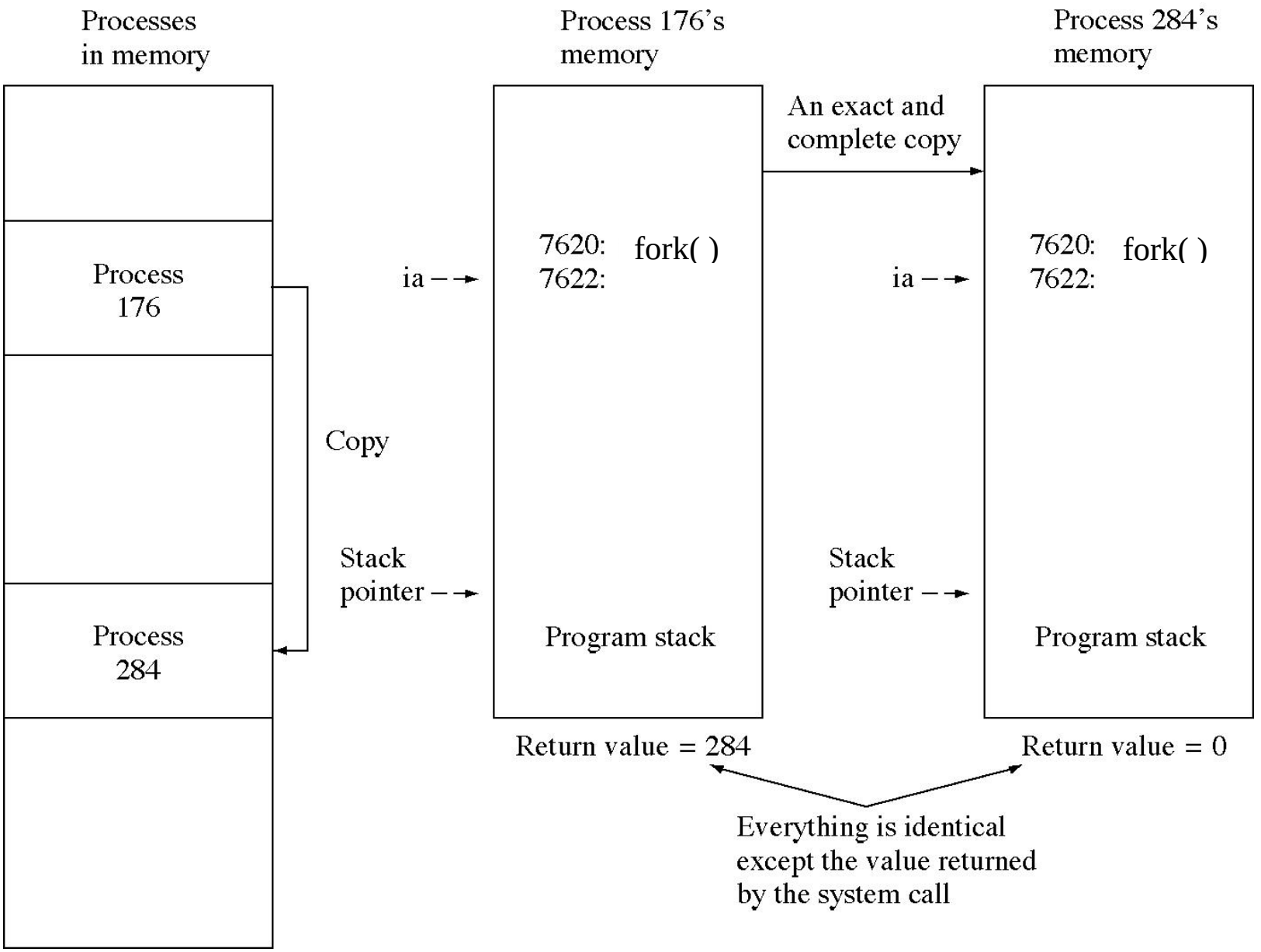
int wait(int *retCode)

- Waits for any exited child, returns its pid

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate **Blocks itself**
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

UNIX Fork

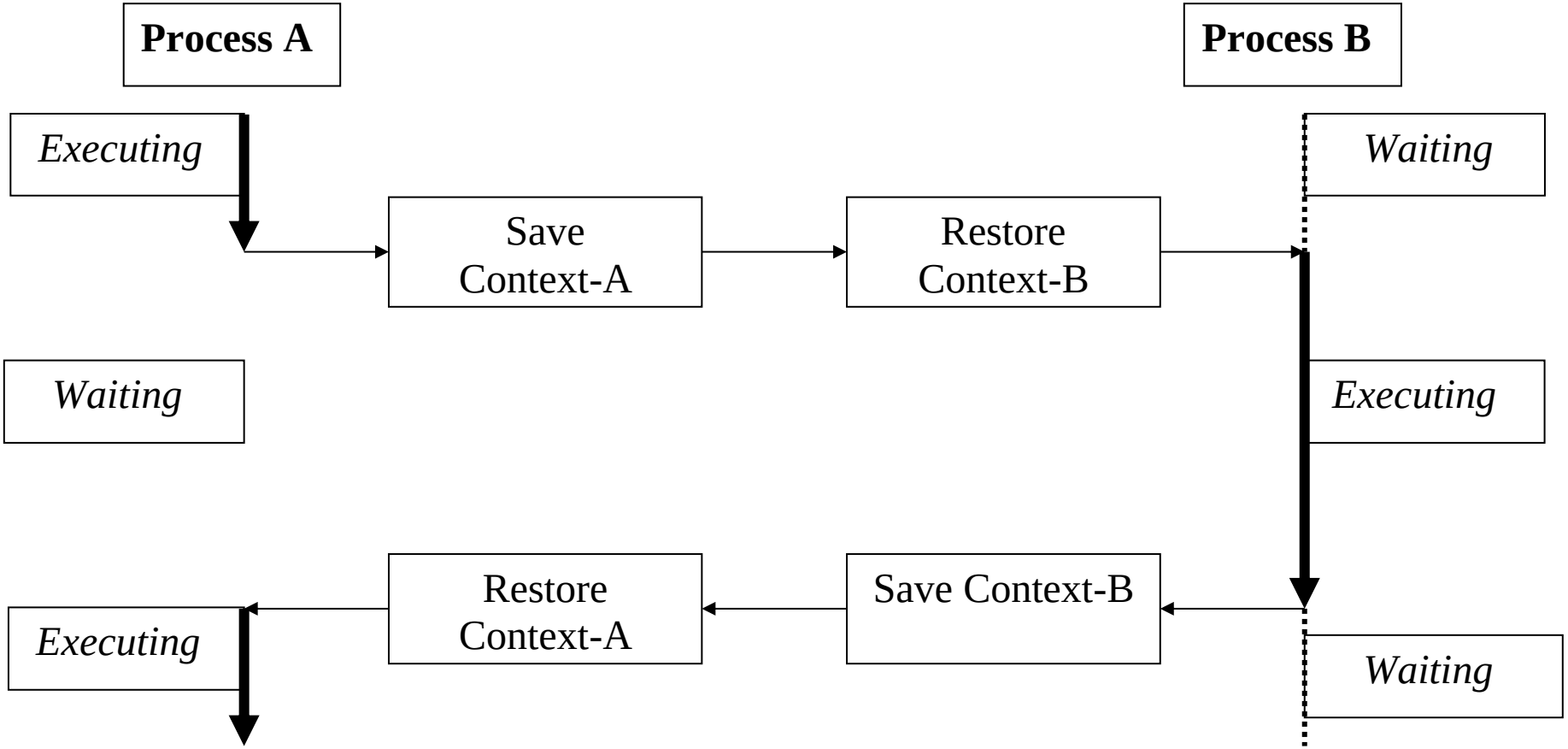

Processes
in memory
Process 176's
memory
Process 284's
memory
An exact and
complete copy
Process
176
Process
284
Copy
ia
7620:
7622:
fork( )
Stack
pointer
Program stack
Return value = 284
Return value = 0
Everything is identical
except the value returned
by the system call

Unix Fork Example

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
  pid_t pid;
  pid = getpid();   /* Parent process created, get its ID */
  pid = fork();     /* Create a child process */
  if (pid == 0)
  {    /* only the child process code should get here */
   while(1) {
       fprintf(stderr, "I am child process \n");
       usleep(10000000); /* wait for 10 seconds */
   }
  }
  /* Only parent should get here  */
  fprintf(stderr," I am PARENT: I wait for 20 seconds\n");
  usleep(20000000);
  fprintf(stderr,"I am PARENT: Kill child: %u\n",pid);
  kill(pid,9);
  return(0);
}
```

Process Context and Switching

- Each process has its own stack and *context*.
- A *context switch* from process ‘A’ to process ‘B’ first saves registers in context A, and then reloads all CPU registers from context B.

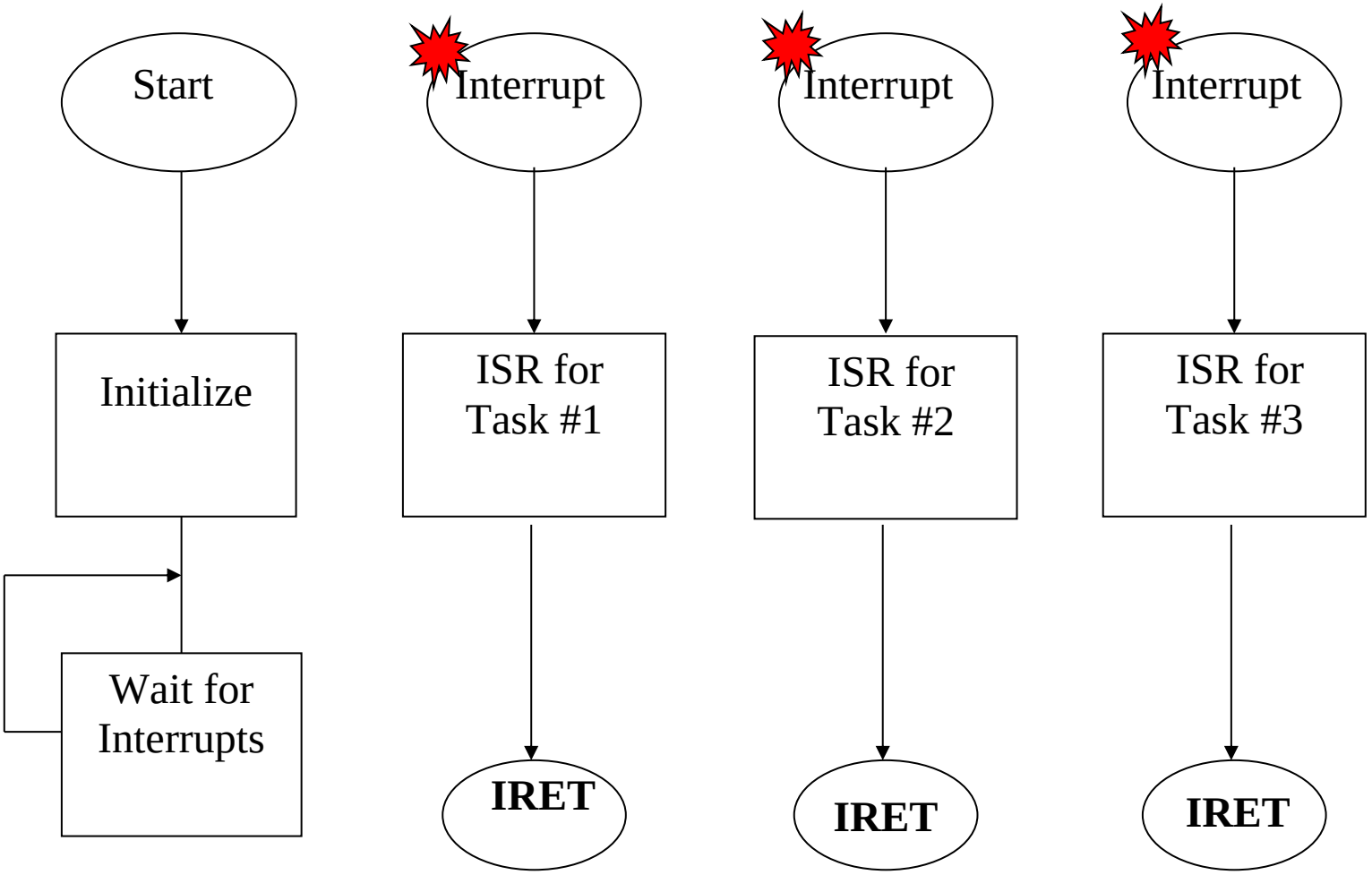


Process/Task Switching

How to change a process state

- Save context of processor including PC and other registers
- Update the PCB/TCB (process/task control block) with the new state and other associated information.
- Move PCB to appropriate queue.
- Select another process for execution.
- Update the process (task) control block of the process (task) selected.
- Update memory-management data structures
- Restore context of the selected process by reloading previous PC and registers.

Foreground/Background Multitasking System



Foreground/Background System

- Most of the actual work is performed in the "foreground" ISRs, with each ISR processing a particular hardware event.
- Main program performs initialization and then enters a "background" loop that waits for interrupts to occur.
- System responds to external events with a predictable amount of latency.

Moving to Background

- Move non-time-critical work (such as updating a display) into background task.
- Foreground ISR writes data to queue, then background removes and processes it.
- An alternative to ignoring one or more interrupts as the result of input overrun.

Limitations of the Foreground/Background Multitasking

- Best possible performance requires moving as much as possible into the background.
- Background becomes collection of queues and associated routines to process the data.
- Optimizes latency of the individual ISRs, but background requires a managed allocation of processor time.

Co-operative Multitasking

- Hides context switching mechanism;
- Still relies on processes to give up CPU.
- Each process allows a context switch at cswitch() call.
- Separate scheduler chooses which process runs next.

Context switching

Who controls when the context is switched?
How is the context switched?

Problems with co-operative multitasking

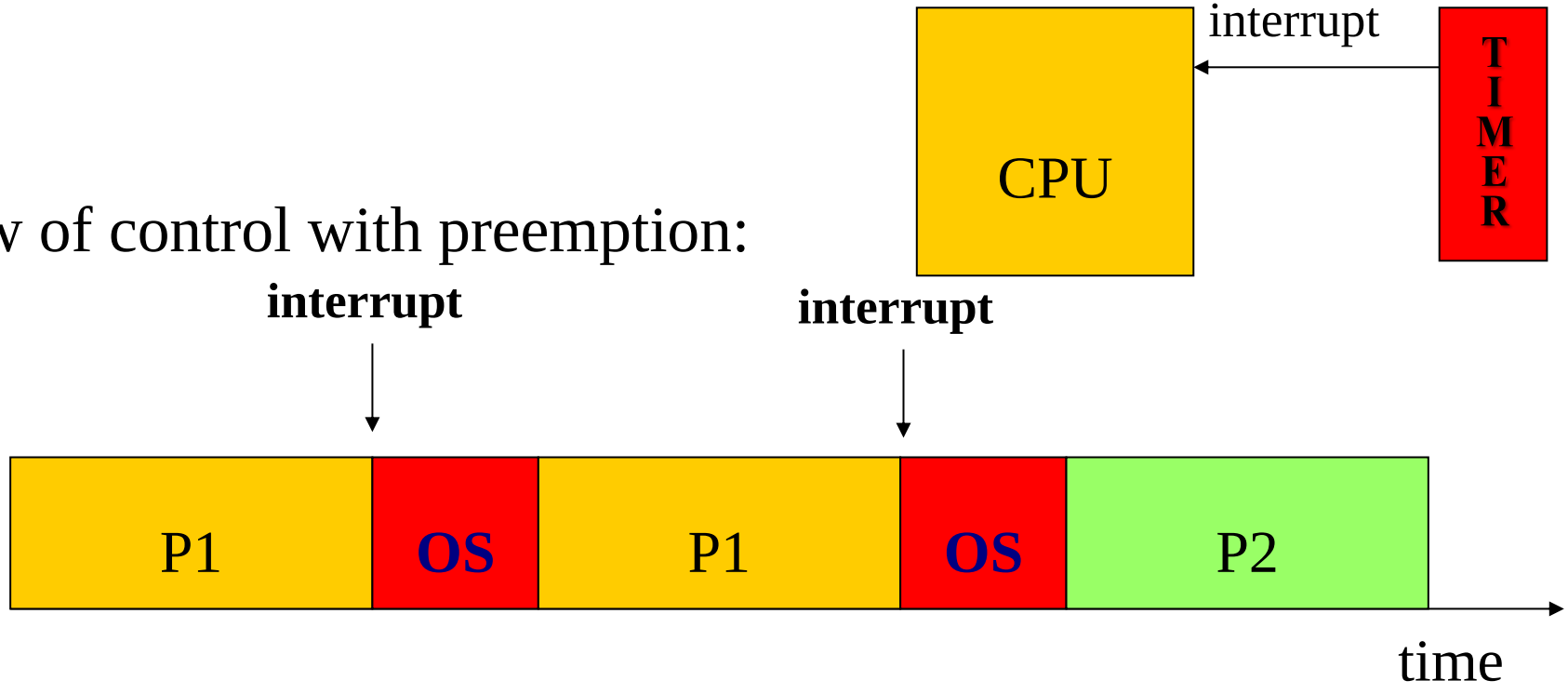
Programming errors can keep other processes out:

- Process never gives up CPU;
- Process waits too long to switch, missing input.

Preemptive Multitasking

- Most powerful form of multitasking
- OS controls when contexts switches
- OS determines what process runs next
- Use timer to call OS, switch contexts:

Flow of control with preemption:



VxWorks Multitasking

Modern real-time systems are based on the complementary concepts of multitasking and inter-task communications.

In VxWorks, tasks have immediate, shared access to most system resources, while also maintaining separate context to maintain individual task control.

A multitasking environment allows a real-time application to be constructed as a set of independent tasks, each with its own thread of execution and set of system resources.

It is often essential to organize the real-time applications into independent but cooperating, programs known *tasks*.

VxWorks Task Context

A task's context includes:

- a thread of execution; that is, the task's program counter
- the CPU registers and (optionally) floating-point registers
- a stack for dynamic variables and function calls
- I/O assignments for standard input, output, and error
- a delay timer
- a time-slice timer
- kernel control structures
- signal handlers
- debugging and performance monitoring values

In VxWorks, one important resource that is *not* part of a task's context is memory address space.
All code executes in a single common address space.

RTX - RTOS Kernel

The RTX kernel is a real time operating system (RTOS)

RTX: **R**eal **T**ime e**X**ecutive for µcontrollers based on ARM CPU cores

It works with the microcontrollers:

- ARM7™TDMI,
- ARM9™,
- or Cortex™-M3 CPU core

Basic functionality -- to start and stop concurrent tasks (processes).
It also has functions for Inter Process Communication (IPC) to:

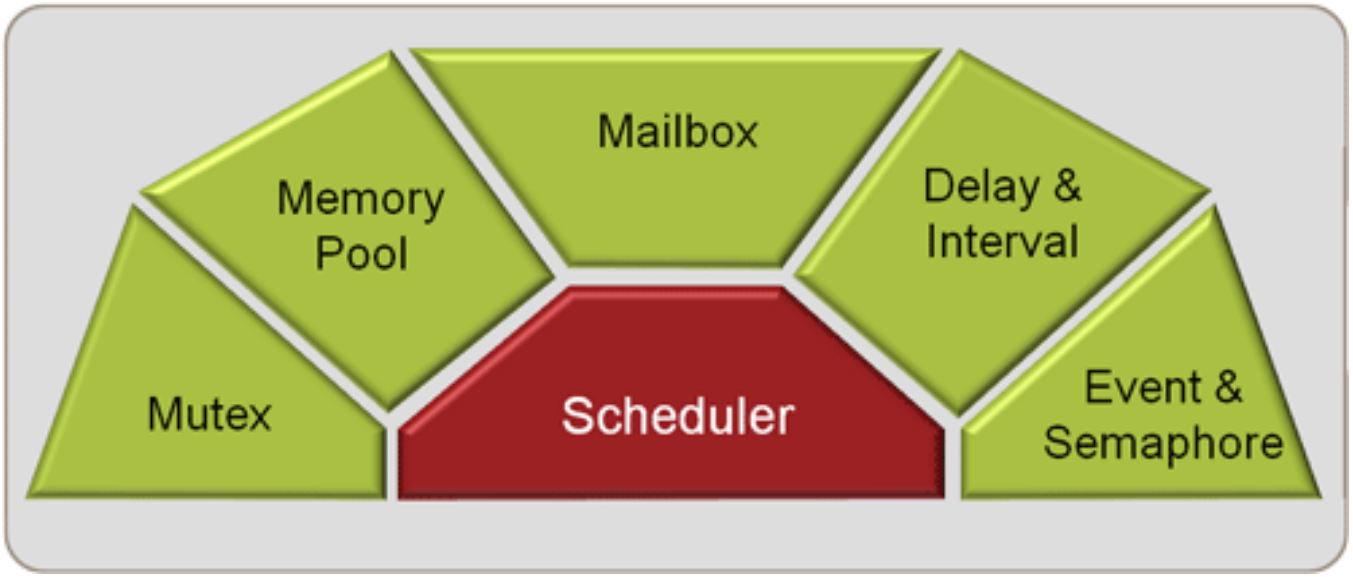
- synchronize different tasks,
- manage common resources (peripherals or memory regions),
- and pass complete messages between tasks.

RTX/RTOS Advantages

The application is split up into several smaller tasks that run concurrently. There are many advantages of RTX/RTOS kernel:

- Real world processes may consist of several concurrent activities. This pattern can be represented in software by using the RTX kernel.
- Different activities occur at different times, for example, just at the moment when they are needed. This is possible because each activity is packed into a separate task, which can be executed on its own.
- Tasks can be prioritized.
- It is easier to understand/manage small pieces of code than one large software.
- Splitting up the application software into independent parts reduces the system complexity, errors, and may facilitates testing.
- The RTX kernel is scalable. Additional tasks can be added easily at a later time.
- The RTX kernel offers services needed in many real-time applications, for example, interrupt handling, periodical activation of tasks, and time-limits on wait functions.

RTX - Deterministic RTOS



- Royalty-free, deterministic RTOS
- Flexible Scheduling: round-robin, pre-emptive, and collaborative
- High-Speed real-time operation with low interrupt latency
- Small footprint for resource constrained systems
- Unlimited number of tasks each with 254 priority levels
- Unlimited number of mailboxes, semaphores, mutex, and timers
- Support for multithreading

RTX Kernal – RTOS

Keil's **R**eal **T**ime e**X**ecutive for ARM CPUs

<RTL.h> file defines the RTX functions and macros.
We need to declare tasks and access all RTOS features:

- Offers interrupt handling, multitasking, periodic task activations, scalable task creation.
- Use RTX_Config_CM.c to specify parameters and configuration in the RTOS/RTX kernel
  - Ports the kernel to your CPU
  - Includes cmsis_os.h
- Include cmsis_os.h so that your application (.c) may access the CMSIS RTOS API
  - Explicitly used in (lab3a and 3b) for thread management

RTX Multitasking

RTOS enables us to create applications that simultaneously perform multiple functions or **tasks**.

Flexible **Scheduling** of system resources like CPU and memory, and offers ways/supports to communicate between tasks.

Description	ARM7™/ARM9™	Cortex™-M
Defined Tasks	Unlimited	Unlimited
Active Tasks	250 max	250 max
Mailboxes	Unlimited	Unlimited
Semaphores	Unlimited	Unlimited
Mutexes	Unlimited	Unlimited
Signals / Events	16 per task	16 per task
User Timers	Unlimited	Unlimited
Code Space	<4.2 Kbytes	<4.0 Kbytes
RAM Space for Kernel	300 bytes + 80 bytes User Stack	300 bytes + 128 bytes Main Stack
RAM Space for a Task	TaskStackSize + 52 bytes	TaskStackSize + 52 bytes
RAM Space for a Mailbox	MaxMessages * 4 + 16 bytes	MaxMessages * 4 + 16 bytes
RAM Space for a Semaphore	8 bytes	8 bytes
RAM Space for a Mutex	12 bytes	12 bytes
RAM Space for a User Timer	8 bytes	8 bytes
Hardware Requirements	One on-chip timer	SysTick timer
User task priorities	1 - 254	1 - 254
Context switch time	<5.3 µsec @ 60 MHz	<2.6 µsec @ 72 MHz
Interrupt lockout time	<2.7 µsec @ 60 MHz	Not disabled by RTX

Timing Specifications

Function	ARM7™/ARM9™ (cycles)	Cortex™-M (cycles)
Initialize system, (*os_sys_init*), start task	1721	1147
Create task (no task switch)	679	403
Create task (switch task)	787	461
Delete task (*os_tsk_delete*)	402	218
Task switch (by *os_tsk_delete_self*)	458	230
Task switch (by *os_tsk_pass*)	321	192
Set event (no task switch)	128	89
Set event (switch task)	363	215
Send semaphore (no task switch)	106	72
Send semaphore (switch task)	364	217
Send message (no task switch)	218	117
Send message (switch task)	404	241
Get own task identifier (*os_tsk_self*)	23	65
Interrupt lockout	<160	0

- The table for RTX Kernel library is measured on (ARM7, Cortex-M3), code execution from internal flash with zero-cycle latency.
- The RTX Kernel for the test is configured for 10 tasks, 10 user timers and stack checking disabled.

Task Creation and Execution

Create the Init Task that will create the other two application tasks 1 & 2 with **ROUND ROBIN Scheduling**

```
#include <RTL.h> /* RTX header file for RTX system calls */
#include <LPC17xx.H> /* LPC17xx Cortex M3 board definitions */
long global_c1 = 0, global_c2 = 0;

__task void task1(void){
    for(;;){
        global_c1 += 3; }
}
__task void task2(void){
    for(;;){
        global_c2 += 2; }
}

int main (void) {
   SystemInit(); /* initialize the Coretx-M3 processor */
   os_tsk_create (task1, 1); /* Creates task1 priority 1 */
   os_tsk_create (task2, 1); /* Creates task2 priority 1 */
   os_tsk_delete_self(); /* Kill itself & task1 starts */

   os_sys_init(task1); /* init RTX and start task1 */
}
```

RTX: Task States and Management

Each RTX task is always in exactly one **state**, which tells the disposition of the task.

RUNNING: The task that is currently running is in the **RUNNING** state. Only one task at a time can be in this state.

READY: Tasks which are ready to run are in the **READY** state. Once the running task has completed processing, RTX selects the next ready task with the **highest priority** and starts it.

WAIT_DLY Tasks which are waiting for a delay to expire are in the WAIT_DLY State. Once the delay has expired, the task is switched to the **READY** state.

RTX: Task States and Management

WAIT_ITV: Tasks which are waiting for an interval to expire are in the WAIT_ITV State. Once the interval delay has expired, the task is switched back to the READY state.
os_itv_wait() function is used to place a task in the WAIT_ITV State.

WAIT_SEM: Tasks which are waiting for a semaphore are in the WAIT_SEM state. When the token is obtained from the semaphore, the task is switched to the READY state.
os_sem_wait() function is used to place a task in the WAIT_SEM state.

WAIT_MUT: Tasks which are waiting for a free mutex are in the WAIT_MUT state. When a mutex is released, the task acquire the mutex and switch to the READY state.
os_mut_wait() function is used to place a task in the WAIT_MUT state.

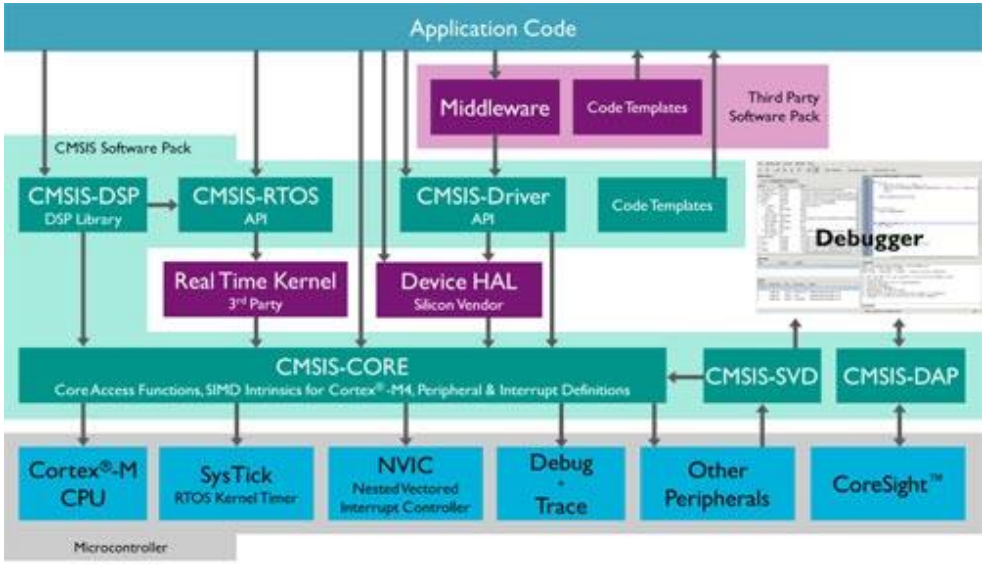
INACTIVE: Tasks which have not been started or tasks which have been deleted are in the INACTIVE state. **os_tsk_delete()** function places a task that has been started [with **os_tsk_create()**] into the INACTIVE state.

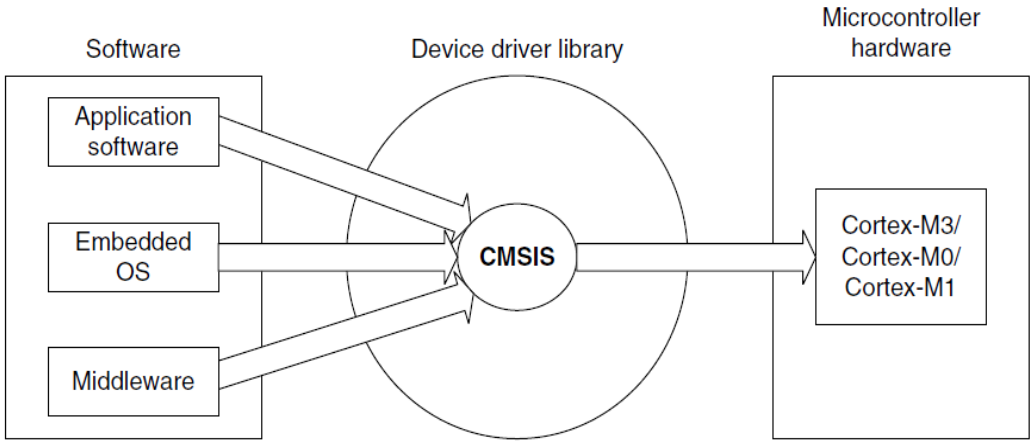
RTX- Task Management Routines

os_sys_init Initializes and starts RL-RTX.
os_sys_init_prio Initializes and starts RL-RTX assigning a priority to the starting task.
os_sys_init_user Initializes and starts RL-RTX assigning a priority and custom stack to the starting task.
os_tsk_create Creates and starts a new task.
os_tsk_create_ex Creates, starts, and passes an argument pointer to a new task.
os_tsk_create_user Creates, starts, and assigns a custom stack to a new task.
os_tsk_create_user_ex Creates, starts, assigns a custom stack, and passes an argument pointer to a new task.
os_tsk_delete Stops and deletes a task.
os_tsk_delete_self Stops/deletes the currently running task.
os_tsk_pass Passes control to the next task of the same priority.
os_tsk_prio Changes a task's priority.
os_tsk_prio_self Changes the currently running task's priority.
os_tsk_self Obtains the task ID of the currently running task.
isr_tsk_get Obtains the task ID of the interrupted task.

CMSIS

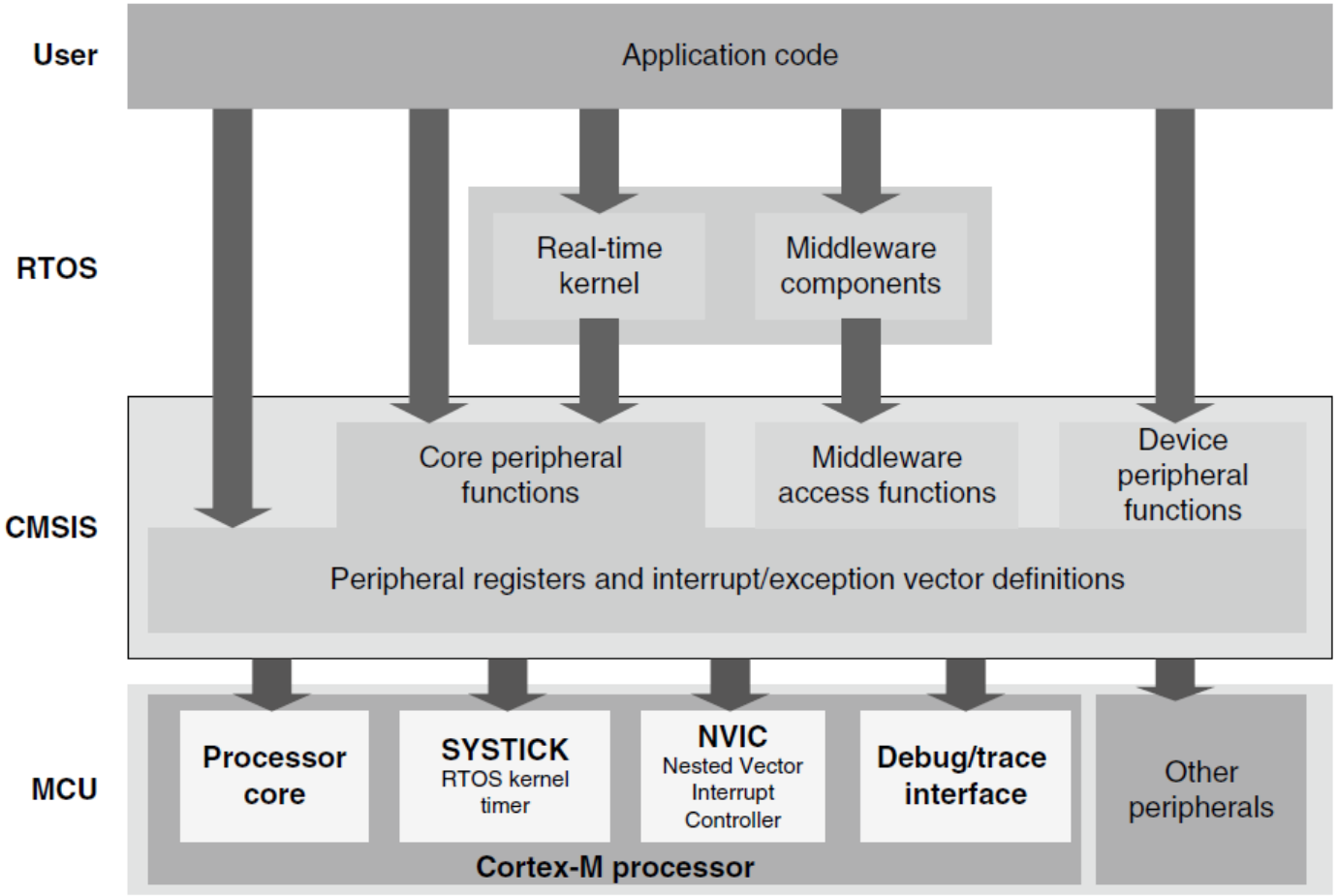
CMSIS: Cortex Microcontroller Software Interface Standard



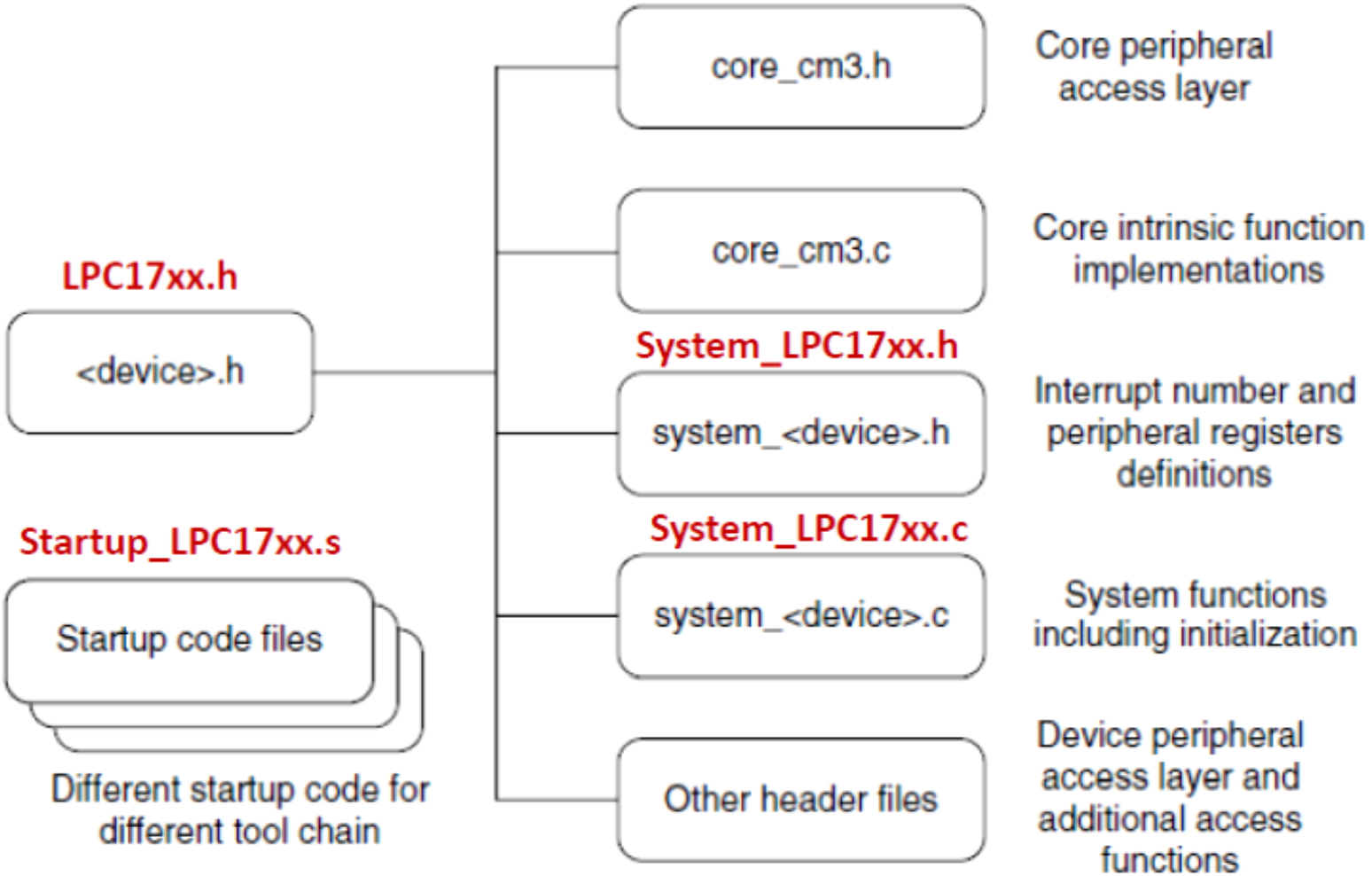


CMSIS

CMSIS is a Device Driver Library providing an independent HW abstraction layer for interfacing applications to the µController.



CMSIS Files


core_cm3.h
Core peripheral access layer
core_cm3.c
Core intrinsic function implementations
LPC17xx.h
<device>.h
System_LPC17xx.h
system_<device>.h
Interrupt number and peripheral registers definitions
System_LPC17xx.c
Startup_LPC17xx.s
system_<device>.c
System functions including initialization
Startup code files
Different startup code for different tool chain
Other header files
Device peripheral access layer and additional access functions

RTX- CMSIS Thread Management

osKernelInitialize Initialize the RTOS kernel.
osKernelStart Start the RTOS kernel.
osKernelGetState Get the current RTOS Kernel state.
osKernelGetSysTimerCount Get RTOS kernel system timer count.
osKernelSuspend Suspend the RTOS kernel scheduler.
osKernelResume Resume the RTOS kernel scheduler.
osDelay Wait for Timeout (Time Delay).

osThreadCreate Creates and starts a new thread.
and passes an argument pointer to a new task.
osThreadTerminate Stops and deletes a thread.
osThreadYield Passes control to the next thread.
osThreadSetPriority Changes a thread's priority.
osThreadGetPriority Get the currently running thread's priority.
osThreadGetId Obtains the thread ID of the currently running thread.

osSemaphoreCreate Define and initialize a semaphore.
osSemaphoreWait Obtain semaphore token or Wait until it becomes available.
osSemaphoreRelease Release a semaphore token.
osSemaphoreDelete Delete a semaphore.

Thread Creation and Execution

ROUND ROBIN Scheduling

```
/* CMSIS-RTOS 'main' function template */

#define osObjectsPublic                 // define objects in main module
#include "osObjects.h"                  // RTOS object definitions

#include "cmsis_os.h"                   // CMSIS RTOS header file
unsigned int global_c1=0;
unsigned int global_c2=0;

extern int Init_Thread (void);

/* main: initialize and start the system */

int main (void) {
  osKernelInitialize ();                // initialize CMSIS-RTOS
      Init_Thread ();

  osKernelStart ();                     // start thread execution
      osDelay(osWaitForever);
}
```

/* Thread.c */

```
void Thread1 (void const *argument); // thread function
void Thread2 (void const *argument); // thread function

osThreadId tid_Thread; // thread id
osThreadDef (Thread1, osPriorityNormal, 1, 0);          // thread object

osThreadId tid2_Thread; // thread id
osThreadDef (Thread2, osPriorityNormal, 1, 0);          // thread object

int Init_Thread (void) {
  tid_Thread = osThreadCreate (osThread(Thread1), NULL);
       tid2_Thread = osThreadCreate (osThread(Thread2), NULL);
  if(!tid_Thread) return(-1); //  Failed to create the thread
  if(!tid2_Thread) return(-1); //  Failed to create the thread

  return(0);
}
```

Thread Creation and Execution

/* Thread.c */

```
void Thread2 (void const *argument) {
        for(;;) {
   global_c2 += 2;

  }
}

void Thread1 (void const *argument) {
       for(;;) {
  global_c1 += 3;

  }
}
```

RTX: Cooperative Multitasking

We can design and implement tasks so that they execute/work **cooperatively**.

Specifically, we must call the system wait function such as **os_dly_wait()** function or the **os_tsk_pass()** function somewhere in each task. These functions signal the RTX kernel to switch.

An example for Cooperative Multitasking.

- The RTX kernel starts executing task1 that creates task2.
- After counter1 is incremented, the kernel switches to task2.
- After counter2 is incremented, the kernel switches back to task1. This process repeats indefinitely.

Cooperative Multitasking- Example

```
#include <rtl.h>

int counter1;
int counter2;

__task void task1 (void);
__task void task2 (void);


__task void task1 (void) {
  os_tsk_create (task2, 0);  /* Create task 2 and
                                mark it as ready */
  for (;;) {                 /* loop forever */
    counter1++;              /* update the counter */
    os_tsk_pass ();          /* switch to 'task2' */
  }
}
```

Cooperative Multitasking- Example (cont.)

```
__task void task2 (void) {
  for (;;) {                    /* loop forever */
    counter2++;                 /* update the counter */
    os_tsk_pass ();             /* switch to 'task1' */
  }
}

void main (void) {
  os_sys_init(task1);           /* Initialize RTX Kernel
                                   and start task 1 */
  for (;;);
}
```

The System wait function allows your task to wait for an event, while **os_tsk_pass ()** switches to another ready task immediately.

If the next ready task has a lower priority than the currently running task, then calling **os_tsk_pass** does not cause a task switch.

Threads and Cooperative Multitasking

```
/* CMSIS-RTOS 'main' function template */

#define osObjectsPublic                 // define objects in main module
#include "osObjects.h"                  // RTOS object definitions

#include "cmsis_os.h"                   // CMSIS RTOS header file
unsigned int counter1, counter2;

extern int Init_Thread (void);

/* main: initialize and start the system */

int main (void) {
  osKernelInitialize ();                // initialize CMSIS-RTOS
      Init_Thread ();

  osKernelStart ();                     // start thread execution
      osDelay(osWaitForever);
}
```

Threads and Cooperative Multitasking

```
/* Thread.c */

void Thread1 (void const *argument); // thread function
void Thread2 (void const *argument); // thread function

osThreadId tid_Thread; // thread id
osThreadDef (Thread1, osPriorityNormal, 1, 0);              // thread1 object

osThreadId tid2_Thread; // thread id
osThreadDef (Thread2, osPriorityNormal, 1, 0);              // thread2 object

int Init_Thread (void) {
  tid_Thread = osThreadCreate (osThread(Thread1), NULL);
  tid2_Thread = osThreadCreate (osThread(Thread2), NULL);
    if(!tid_Thread) return(-1); //  Failed to create the thread
    if(!tid2_Thread) return(-1); //  Failed to create the thread

  return(0);
}
```

Threads and Cooperative Multitasking

```
/* Thread.c */

void Thread2 (void const *argument) {
 osStatus   status;                    // status of the executed function
      while(1) {
            counter2++;
            status =  osThreadYield();
            if (status != osOK) {  // thread switch not occurred
            }
      }
}
void Thread1 (void const *argument) {
 osStatus   status;                    // status of the executed function
      while(1) {
            counter1++;
            status =  osThreadYield();
            if (status != osOK) {  // thread switch not occurred
            }
      }
}
```

IPC: Interprocess Communication

OS provides mechanisms so that processes can pass data.
Two types of semantics:

- blocking: sending process waits for response;
- non-blocking: sending process continues.

IPC styles
Shared memory:

- processes have some memory in common;
- must cooperate to avoid destroying and/or missing any messages.

Message passing:

- processes send messages along a communication channel---no common address space.

Critical Regions

Critical region: section of code that cannot be interrupted by another process. For example:

- writing shared memory;
- accessing I/O device.

Semaphores and Mutex

Semaphore: OS primitive for controlling access to critical regions.

- Get access to semaphore with P().

  Perform critical region operations.

- Release semaphore with V().

```
wait(flag)
....
critical section (instructions exe)
....
signal(flag)
```

Embedded *vs.* General-Purpose Scheduling

Workstations try to avoid starving processes of CPU access.

- Fairness = access to CPU.

Embedded systems must meet deadlines.

- Low-priority processes may not run for a long time.

Priority-driven Scheduling

- Each process has a priority
- CPU goes to highest-priority process that is ready
- Priorities determine the scheduling policy:

RTOS: Real Time Operating System

RTOS is designed to serve real-time application processes and threads with deterministic delays

- Often just consists of a OS kernel (nothing fancy, no user interface, etc.)
- Provides: task scheduling, task dispatching, and inter-task communication
- Timing behavior must be predictable - short and deterministic times, predictable memory accesses, etc.

  Late answer = wrong answer
- Must manage timing and scheduling of task - must be aware of task deadlines, and provide precise time services.

CPU Scheduling

CPU scheduling determines which process is going to execute next.

- CPU scheduler is also known as the dispatcher
- It is invoked on an event that may lead to choose another process for execution:
  - Clock interrupts
  - I/O interrupts
  - Operating system calls and traps
  - Signals

Short-term scheduling

Scheduling Policies

The selection function: It determines which process in the ready queue is selected next for execution.

The decision mode: It specifies the instants in time at which the selection function is exercised

Non-preemptive

- Once a process is in the running state, it will continue until it terminates or blocks itself for I/O.

Preemptive

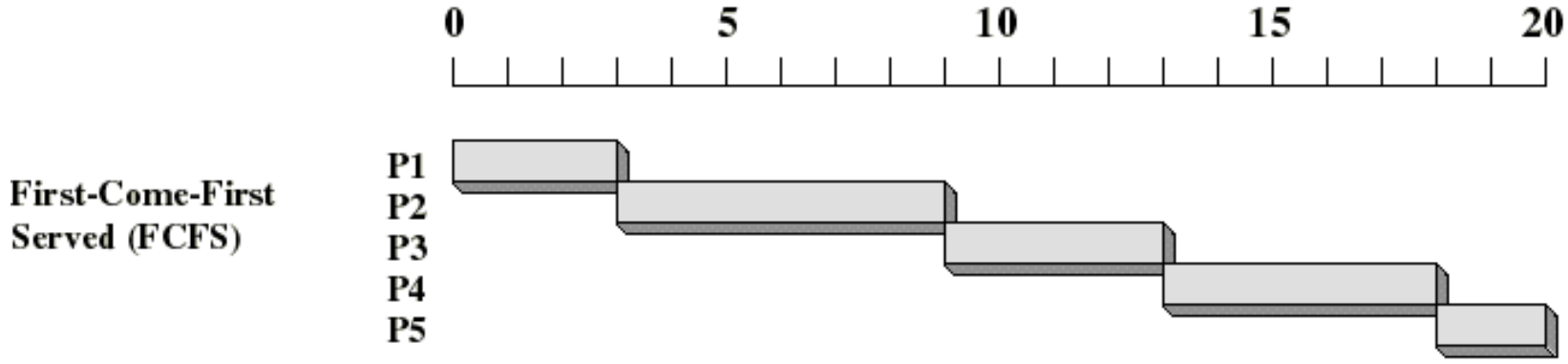
- Currently running process may be interrupted and moved to the Ready state by the OS.
- Allows for better service since any one process cannot monopolize the processor for very long.

FCFS Scheduling

Service time =
Total processor time needed in a (CPU-I/O) cycle

Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

FCFS: First Come First Served



FCFS: First Come First Served

- Selection function: The process that has been waiting the longest in the ready queue
- Decision mode: Non-preemptive

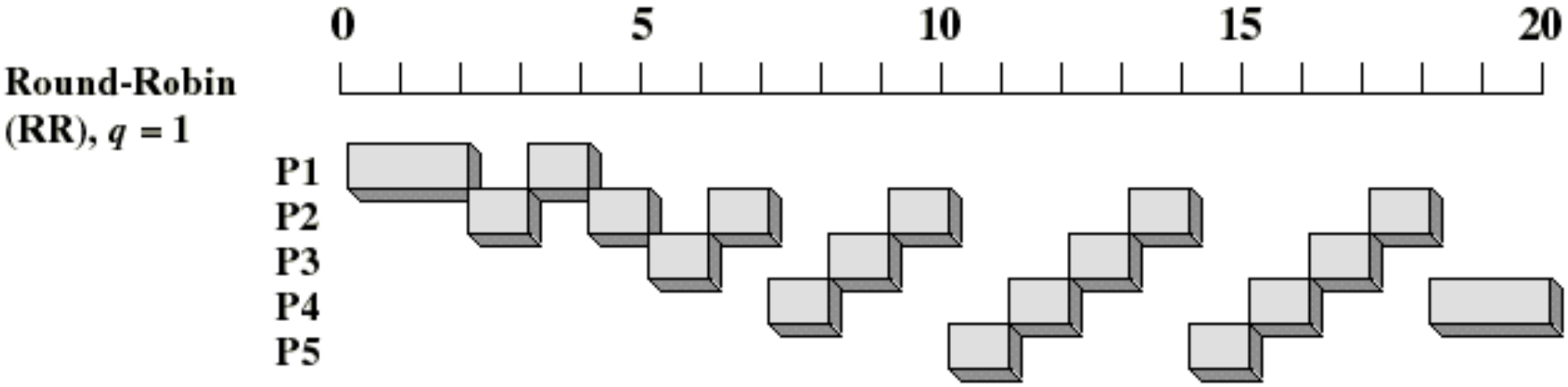
FCFS Drawbacks

- Process that does not perform any I/O will monopolize the processor.
- Favors CPU-bound processes:
  - I/O-bound processes have to wait until CPU-bound process completes.
  - I/O-bound processes have to wait even when their I/O is completed.
  - We could have kept the I/O devices busy by giving a bit more priority to I/O bound processes.

Time-Sliced Scheduling

- Known as Round Robin
- Each process runs for a fixed amount of time.
- Processes are run in a round-robin sequence.
- Appropriate for regular multi-programming environments.
- Poor response time performance.
- Need better strategy for real-time system applications.

Round Robin (RR) Scheduling



- Selection function: FCFS
- Decision mode: Preemptive
  - A process is allowed to run until the time slice period has expired
  - Then a clock interrupt occurs, and the running process is put on the ready queue.

Round Robin Scheduling

Time quantum must be substantially larger than the time required to handle the clock interrupt and dispatching.
Round Robin favors CPU-bound processes

- I/O bound process uses the CPU for a time less than the time quantum and it is blocked waiting for I/O.
- A CPU-bound process run for full time slice and put back into the ready queue.

Solution: Use Virtual Round Robin

- When an I/O completes, the blocked process is moved to an auxiliary queue that gets preference over the main ready queue.

Problem: Consider the following processes are to be scheduled using FCFS and Round Robin

Process		A	B	C	D
Arrival Time	T_a	0	1	2	3
Service Time	T_s	1	9	1	9

Perform the analysis for each scheduling algorithm.

FCFS	A	B	B	B	B	B	B	B	B	B	C	D	D	D	D	D	D	D	D	D
RR, q = 1	A	B	C	B	D	B	D	B	D	B	D	B	D	B	D	B	D	B	D	D

		A	B	C	D	
FCFS	T_f	1.00	10.00	11.00	20.00	
	T_r	1.00	9.00	9.00	17.00	9.00
	T_r/T_s	1.00	1.00	9.00	1.89	3.22
RR $q = 1$	T_f	1.00	18.00	3.00	20.00	
	T_r	1.00	17.00	1.00	17.00	9.00
	T_r/T_s	1.00	1.89	1.00	1.89	1.44

Problem. Consider the following processes, A, B, C, D and E that are to be scheduled using, FCFS and Round Robin scheduling techniques with time quantum 1 and 4.

	A	B	C	D	E
T_a	0	1	3	9	12
T_s	3	5	2	5	5

Where Ta = Process Arrival Time
Ts = Process Service Time

Show a complete schedule for both cases.