Processes and Multitasking

EE8205: Embedded Computer Systems
http://www.ee.ryerson.ca/~courses/ee8205/

Dr. Gul N. Khan
http://www.ee.ryerson.ca/~gnkhan
Electrical and Computer Engineering
Ryerson University

Overview

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

Chapter 6 of the Text by M. Wolf, Chapters 9 and 10 of Text by D. W. Lewis
Introduction to Processes

All multiprogramming operating systems are built around the concept of processes.
Process is also called a task.

OS and Processes

- OS must interleave the execution of several processes to maximize CPU usage.
  Keeping reasonable/minimum response time
- OS must allocate resources to processes.
  By avoiding deadlock
- OS must also support:
  - IPC: Inter-process communication
  - Creation of processes by other processes
Task/Process Concept

Serial Execution of Two Processes

Interleaving the Execution of Process 1 and 2
Processes - Managing Timing Complexity

Multiple rates
- multimedia
- automotive

Asynchronous Input
- user interfaces
- communication systems

Engine Control Tasks
- spark control
- crankshaft sensing
- fuel/air mixture
- oxygen sensor
- Kalman filter
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 scheduler 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.
Process Context

- 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.
Basic Process States

There are three basic states of a process

- The Running state
  - The process that gets executed. *(Max of one for one CPU)*
- The Ready state
  - A process is ready to be executed.
- The Blocked state (Waiting)
  - When a process cannot execute until some event occurs.
    *(e.g. completion of an I/O)*

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

- New: Commitment to execution when ready (in terms of # of processes & memory)
- Ready: Admit
- Running: Dispatch, Scheduling-allowed
- Blocked: Event Occurs
- Exit: Release, Not eligible for Execution but Info preserved temporarily

Event Wait: File request I/O IPC

At parent request
Process Transitions

• **Ready ⇒ Running**
  ▪ Dispatcher selects a new process to run.
    When the turn comes.

• **Running ⇒ Ready**
  ▪ Running process has expired its time slot.
  ▪ A higher priority process is in the ready state.

• **Running ⇒ Blocked** (waiting)
  ▪ When a process requests something for which it must wait.
    - A service that the OS is not ready to perform.
    - An access to a resource not yet available.
    - Initiates I/O and must wait for the result.
    - Waiting for a process to provide input (IPC).

• **Blocked ⇒ Ready**
  ▪ When the event, for which process is waiting, occurs.
Process Modes of Execution

Most processors support at least two execution modes:

• Privileged mode
  System mode, kernel mode, supervisor 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
Process Hierarchies

- Forms a hierarchy
  UNIX calls this a "process group"
- Windows has no concept of process hierarchy
  All processes are created equally.

- Process A created two child processes, B and C.
- Process B created three child processes, D, E & F.
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.
  
  **Kernel schedules another higher-priority process.**

- A process running in Kernel mode cannot be pre-empted.
  
  This makes UNIX unsuitable for real-time. More later
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

<table>
<thead>
<tr>
<th>Process management</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Call</strong></td>
</tr>
<tr>
<td>pid = fork()</td>
</tr>
<tr>
<td>pid = waitpid(pid, &amp;statloc, options)</td>
</tr>
<tr>
<td>s = execve(name, argv, environp)</td>
</tr>
<tr>
<td>exit(status)</td>
</tr>
</tbody>
</table>
UNIX Fork

Processes in memory

Process 176

Process pointer →

Copy

Stack pointer →

Process 284

Program stack

An exact and complete copy

7620: fork()
7622:

Return value = 284

Everything is identical except the value returned by the system call

Return value = 0

Process 176’s memory

Process 284’s memory
# Unix Fork Example

```c
#include <sys/types.h>
#include <stdio.h>     // Include stdio.h for fprintf function
#include <unistd.h>    // Include unistd.h for getpid and fork functions

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 Switching

A process switch may occur whenever the OS gain control of the CPU.

• Supervisor Call
  ♦ Transfer control to a piece of OS code (e.g. file open).
  ♦ Process may be switched to a blocked state.

• Trap
  ♦ An error resulted from the last instruction.
  ♦ Process moves to Exit state.

• Interrupt by an external independent event.
  ♦ Clock Interrupt: process has executed for the maximum allowable time slice. Switch to Blocked state.
  ♦ I/O Interrupt: OS moves waiting processes to READY
  ♦ Memory Fault: Memory address block is not in virtual memory so it must be brought into main memory.

Move process to blocked state. (Waiting for the I/O to complete)
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. e.g. accounting
• Move PCB to appropriate queue. Ready, blocked, suspend.
• Select another process for execution. Scheduling decision
• 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

Start

Initialize

Wait for Interrupts

Interrupt

ISR for Task #1

IRET

Interrupt

ISR for Task #2

IRET

Interrupt

ISR for Task #3

IRET

Wait for Interrupts

Start

Initialize

Wait for Interrupts

Interrupt

ISR for Task #1

IRET

Interrupt

ISR for Task #2

IRET

Interrupt

ISR for Task #3

IRET
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.
Context Switching

Must copy all registers to activation record, keeping proper return value for PC.
Must copy new activation record into CPU state.
How does the program that copies the context keep its own context?

Context switching in ARM

Start new process:

ADR r0,NEXTPROC
LDR r13,[r0]
LDMDB r13,{r0,r14}
MSR SPSR,r0
LDMIA r13,{r0-r14}^ 
MOVS pc,r14

Save old process:

STMIA r13,{r0-r14}^ 
MRS r0,SPSR
STMDB r13,{r0,r15}
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:
Preemptive Context Switching

- Timer-interrupt gives control to OS, which saves interrupted process’s state in an activation record.
- OS chooses next process to run.
- OS installs desired activation record as current CPU state.

Why not use interrupts

We could change the interrupt vector at every period, but:
- We would need management code anyway;
- We would have to know the next period’s process at the start of the current process.
Non-Preemptive Context Switch

State of Process A

Running  Interrupted  Running  Ready

Interrupt Routine

Run-Time Kernel

State of Process B

Blocked  Ready  Running

Context Switch
Preemptive Context Switch

State of Process A
- Running
- Interrupted
- Ready

State of Process B
- Blocked
- Ready
- Interrupted
- Running

Interrupt Routine
Run-Time Kernel

Context Switch
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 Multitasking and Interrupts

Another key facility in real-time systems is hardware interrupt handling.

- Interrupts are the usual mechanism to inform a system of external events.
- It is important to have the fastest possible response to external interrupts.

In VxWorks, interrupt service routines (ISRs) run in a special context of their own, outside any task’s context.
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.
VxWorks Task States

**READY:**
The state of a task that is not waiting for any resource other than the CPU.

**PEND:**
The state of a task that is blocked due to the unavailability of some resource.

**DELAY:** The state of a task that is asleep for some duration.

**SUSPEND:**
The state of a task that is unavailable for execution. This state is used primarily for debugging. Suspension does not inhibit state transition, only task execution. Thus, *pered-suspended* tasks can still unblock and *delayed-suspended* tasks can still be awaken.

**DELAY + S:** The state of a task that is both delayed and suspended.

**PEND + S:** The state of a task that is both pended and suspended.

**PEND + T:** The state of a task that is pended with a timeout value.

**PEND + S + T:** The state of a task that is both pended with a timeout value and suspended.

**state + I:** The state of task specified by *state*, plus an inherited priority.
Task-State Transitions

The highest-priority ready task is executing.

ready \(\rightarrow\) pended \(\rightarrow\) delayed \(\rightarrow\) suspended
ready \(\rightarrow\) suspended
pended \(\rightarrow\) ready \(\rightarrow\) suspended
pended \(\rightarrow\) suspended
delayed \(\rightarrow\) suspended
delayed \(\rightarrow\) suspended
suspended \(\rightarrow\) ready
suspended \(\rightarrow\) pended
suspended \(\rightarrow\) delayed

\(\text{semTake()} / \text{msgQReceive()}\)
\(\text{taskDelay()}\)
\(\text{taskSuspend()}\)
\(\text{semGive()} / \text{msgQSend()}\)
\(\text{taskSuspend()}\)
expired delay
\(\text{taskSuspend()}\)
\(\text{taskResume()} / \text{taskActivate()}\)
\(\text{taskResume()}\)
\(\text{taskResume()}\)
Wind (VxWorks) Task Scheduling

- The default algorithm in wind is priority-based preemptive scheduling.
- You can also select to use round-robin scheduling for your applications.

Both algorithms rely on the task’s priority. The wind kernel has 256 priority levels, numbered 0 through 255. Priority 0 is the highest and priority 255 is the lowest.

Tasks are assigned a priority when created. You can also change a task’s priority level while it is executing by calling taskPrioritySet().

The ability to change task priorities dynamically allows applications to track precedence changes in the real world.
VxWorks Task Control

VxWorks library taskLib provide routines for task creation and control, as well as for retrieving information about tasks.

Task Creation and Activation
- taskSpawn() Spawns (creates and activates) a new task.
- taskInit() Initializes a new task.
- taskActivate() Activates an initialized task.

Task Name and ID Routines
- taskName() Gets the task name associated with a task ID.
- taskNameToId() Looks up the task ID associated with a task.
- taskIdSelf() Gets the calling task’s ID.
- taskIdVerify() Verifies the existence of a specified task.

Task Information Routines
- taskIdListGet() Fills an array with the IDs of all active tasks.
- taskInfoGet() Gets information about a task.
- taskPriorityGet() Examines the priority of a task.
- taskRegsGet() Examines a task’s registers (cannot be used for current task).
VxWorks Task Control

Task-Deletion Routines

- **exit()** Terminates the calling task and frees memory (task stacks and task control blocks only).
- **taskDelete()** Terminates a specified task and frees memory (task stacks and task control blocks only).
- **taskSafe()** Protects the calling task from deletion.
- **taskUnsafe()** Undoes a **taskSafe()** (makes the calling task available for deletion).

Task Control Routines

- **taskSuspend()** Suspends a task.
- **taskResume()** Resumes a task.
- **taskRestart()** Restarts a task.
- **taskDelay()** Delays a task; delay units and resolution in ticks.
- **nanosleep()** Delays a task; delay units are nanoseconds.
Task Scheduler Control

VxWorks provide routines for task scheduler control.

- **taskPrioritySet( )** Changes the priority of a task.

- **kernelTimeSlice( )** Controls round-robin scheduling.
  Round-robin scheduling is enabled by calling `kernelTimeSlice( )`, which takes a parameter for a time slice, or interval. This interval is the amount of time each task is allowed to run before relinquishing the processor to another equal-priority task.

- **taskLock( )** Disables task rescheduling.
- **taskUnlock( )** Enables task rescheduling.

  - The *wind* scheduler can be explicitly disabled and enabled on a per-task basis with the routines `taskLock( )` and `taskUnlock( )`.
  - When a task disables the scheduler by calling `taskLock( )`, no priority-based preemption can take place while that task is running.
  - Note that preemption locks prevent task context switching, but do not lock out interrupt handling.
  - Preemption locks can be used to achieve mutual exclusion; however, keep the duration of preemption locking to a minimum.
RTX - RTOS Kernel

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

**RTX:** Real Time eXecutive 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 Performance

<table>
<thead>
<tr>
<th>Task Specifications</th>
<th>Performance (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt Latency</td>
<td>n/a</td>
</tr>
<tr>
<td>Initialize system, start task</td>
<td>1,147</td>
</tr>
<tr>
<td>Create task (no task switch)</td>
<td>403</td>
</tr>
<tr>
<td>Create task (with task switch)</td>
<td>461</td>
</tr>
<tr>
<td>Delete Task</td>
<td>218</td>
</tr>
<tr>
<td>Task Switch</td>
<td>192</td>
</tr>
<tr>
<td>Set event</td>
<td>89</td>
</tr>
<tr>
<td>Send semaphore</td>
<td>72</td>
</tr>
<tr>
<td>Send message</td>
<td>117</td>
</tr>
</tbody>
</table>

The RTX Kernel library for Cortex-M does not disable interrupts. Interrupt response time for ISRs is the same as without the RTX kernel.
## RTX Memory Requirements

### Task Specifications

<table>
<thead>
<tr>
<th>CODE Size</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 4.0 KBytes</td>
<td></td>
</tr>
</tbody>
</table>

| RAM Space for Kernel | < 300 Bytes + 128 Bytes User Stack |

<table>
<thead>
<tr>
<th>RAM Space for a Task</th>
<th>TaskStackSize + 52 Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM Space for a Mailbox</td>
<td>MaxMessages*4 + 16 Bytes</td>
</tr>
<tr>
<td>RAM Space for a Semaphore</td>
<td>8 Bytes</td>
</tr>
<tr>
<td>RAM Space for a Mutex</td>
<td>12 Bytes</td>
</tr>
<tr>
<td>RAM Space for a User Timer</td>
<td>8 Bytes</td>
</tr>
</tbody>
</table>

### Hardware Requirements

<table>
<thead>
<tr>
<th>Hardware Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>SysTick timer</td>
</tr>
</tbody>
</table>

- The RTX Kernel for this test was configured for 10 tasks, 10 user timers, and stack checking disabled.
- RAM requirements depend on the number of concurrently running tasks.
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.

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

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

- 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

Add header files for RTX and Cortex-M3 board
#include <RTL.h> /* RTX header file */
#include <LPC17xx.H> /* LPC17xx Cortex M3 definitions */

__task void init (void) {
    os_tsk_create (task1, 1); /* Creates task1 with default priority 1 */
    os_tsk_create (task2, 1); /* Creates task2 */
    os_tsk_delete_self(); /* Kill itself & task1 starts */
}

Create the main() function:
int main (void) {
    SystemInit(); /* initialize the Coretx-M3 processor */
    os_sys_init(init); /* Start the init task */
}
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. The `os_tsk_self()` returns the Task ID (TID) of the currently executing task.

**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. The `os_dly_wait()` function is used to place a task in the **WAIT_DLY** 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_OR: Tasks which are waiting for at least one event flag are in the WAIT_OR state. When the event occurs, task will switch to READY state. 

`os_evt_wait_or()` function is used to place a task in the WAIT_OR state.

WAIT_AND: Tasks which are waiting for all the set events to occur are in the WAIT_AND state. When all event flags are set, the task is switched to the READY state.

`os_evt_wait_and()` function will place a task in the WAIT_AND 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.
RTX: Task States and Management

**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. The **os_mut_wait()** function is used to place a task in the WAIT_MUT state.

**WAIT_MBX:** Tasks which are waiting for a mailbox message are in the **WAIT_MBX** state. Once the message has arrived, the task is switched to the READY state. The **os_mbx_wait()** function is used to place a task in the WAIT_MBX state. Tasks waiting to send a message when the mailbox is full are also put into the WAIT_MBX state. When the message is read out from the mailbox, the task is switched to the READY state. The **os_mbx_send()** function is used to place a task in the WAIT_MBX state.

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

RTX configuration settings are defined RTX_Config.c file in ......\ARM\RTX\Config directory that:

• Specify the number of concurrent running tasks
• Specify the number of tasks with user-provided stack
• Specify the stack size for each task
• Enable or disable the stack checking
• Enable or disable running tasks in privileged mode
• Specify the CPU timer number used as the system tick timer
• Specify the input clock frequency for the selected timer
• Specify the timer tick interval
• Enable or disable the round-robin task switching
• Specify the time slice for the round-robin task switching
• Define idle task operations
• Specify the number of user timers
• Specify code for the user timer callback function
• Specify the FIFO Queue size
• Specify code for the runtime error function

There is no default configuration in the RL-RTX library.
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.
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 task 1 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++;
    counter2++;
    os_tsk_pass (); /* switch to 'task2' */
  }
}
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.
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.

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

**Semaphores**

**Semaphore**: OS primitive for controlling access to critical regions.

- Get access to semaphore with P(). Perform critical region operations.
- Release semaphore with V().
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:
    Fixed priority
    Time-varying priorities
CPU Scheduling

CPU scheduling determines which process is going to execute next.

**Relevant to Real-time Systems**
- 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

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Service Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>

FCFS: First Come First Served

When the current process ceases to execute, the oldest process in the Ready queue is selected
FCFS: First Come First Served

• Selection function: The process that has been waiting the longest in the ready queue
• Decision mode: Non-preemptive
  Process run until it blocks itself

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 (poor device utilization).
  ▪ 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 (RR), $q = 1$

0 5 10 15 20
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.
- A process dispatched from the auxiliary queue runs no longer than the basic time quantum minus the time spent running since it was selected from the ready queue.
**Problem:** Consider the following processes are to be scheduled using FCFS and Round Robin

<table>
<thead>
<tr>
<th>Process</th>
<th>Ta</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival Time</td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Service Time</td>
<td></td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>

Perform the analysis for each scheduling algorithm.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FCFS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T&lt;sub&gt;f&lt;/sub&gt;</td>
<td>1.00</td>
<td>10.00</td>
<td>11.00</td>
<td>20.00</td>
</tr>
<tr>
<td>T&lt;sub&gt;r&lt;/sub&gt;</td>
<td>1.00</td>
<td>9.00</td>
<td>9.00</td>
<td>17.00</td>
</tr>
<tr>
<td>T&lt;sub&gt;r&lt;/sub&gt;/T&lt;sub&gt;S&lt;/sub&gt;</td>
<td>1.00</td>
<td>1.00</td>
<td>9.00</td>
<td>1.89</td>
</tr>
<tr>
<td><strong>RR, q = 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T&lt;sub&gt;f&lt;/sub&gt;</td>
<td>1.00</td>
<td>18.00</td>
<td>3.00</td>
<td>20.00</td>
</tr>
<tr>
<td>T&lt;sub&gt;r&lt;/sub&gt;</td>
<td>1.00</td>
<td>17.00</td>
<td>1.00</td>
<td>17.00</td>
</tr>
<tr>
<td>T&lt;sub&gt;r&lt;/sub&gt;/T&lt;sub&gt;S&lt;/sub&gt;</td>
<td>1.00</td>
<td>1.89</td>
<td>1.00</td>
<td>1.89</td>
</tr>
</tbody>
</table>
**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.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ta</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>Ts</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Where Ta = Process Arrival Time
Ts = Process Service Time

Show a complete schedule for both cases.