DEPARTMENT OF INFORMATION TECHNOLOGY

UITP307 REAL TIME OPERATING SYSTEMS LABORATORY

Laboratory Manual

G. H. Raisoni College of Engineering & Management, Pune **Third Year of Information Technology (2020 Pattern)** **UITP307 RTOS Laboratory**		
Teaching Scheme: **PR: 02 Hours/Week**	**Credit** **02**	**Examination Scheme:** **Int: 25 Marks** **Ext: -** **Total: 25 Marks**

Sr. No	**List of Laboratory Assignments.**
1.	Study of RTOS Simulator
2.	Study of RTLinux Features
3.	Study of VxWorks Features
4.	Implementation of Task Creation
5.	Implementation of Priority-Driven Scheduling
6.	Implementation of Fixed-Priority Scheduling
7.	Implementation of Preemptive scheduling
8.	Implementation of non-Preemptive scheduling
9.	Implementation of Clock Driven Scheduling
10.	Implementation of EDF Scheduling

Course Title: Real Time Operating Systems										
Semester	**VI**	**Teaching Scheme**				**Evaluation Scheme**				
						Theory			**Practical**	
Term	**VI**	**Th**	**Tu**	**Pr**	**Credits**	**TAE**	**CAE**	**ESE**	**INT**	**EXT**
Course Category	**C**	**2**		**2**	**3**	**10**	**15**	**50**	**25**	**--**

Course Code	**UITL307** **UITP307**									
Teaching Mode	**TH**	**40 Hrs**			**Total**	**75**			**25**	
Duration of ESE	**2 Hrs**					**100**				

Assignment 1: Study of RTOS Simulator.

Aim: To study RTOS Simulator.

Objective: To learn RTOS Simulator.

Theory:

Principle of Operation

Threads that run tasks

The Windows port layer creates a low priority Windows thread for each FreeRTOS task created by the FreeRTOS application. All the low priority Windows threads are then kept in the suspended state, other than the Windows thread that is running the FreeRTOS task selected by the FreeRTOS scheduler to be in the Running state. In this way, the FreeRTOS scheduler chooses which low priority Windows thread to run in accordance with its scheduling policy. All the other low priority windows threads cannot run because they are suspended.

FreeRTOS ports that run on microcontrollers have to perform complex context switching to save and restore the microcontroller context (registers, etc.) as tasks enter and leave the Running state. In contrast, the Windows simulator layer simply has to suspend and resume Windows threads as the tasks they represent enter and leave the Running state. The real context switching is left to Windows.

Simulating the tick interrupt

The tick interrupt generation is simulated by a high priority Windows thread that will periodically pre-empt the low priority threads that are running tasks. The tick rate achievable is limited by the Windows system clock, which in normal FreeRTOS terms is slow and has a very low precision. It is therefore not possible to obtain true real time behaviour.

Simulating interrupt processing

Simulated interrupt processing is performed by a second higher priority Windows thread that, because of its priority, can also pre-empt the low priority threads that are running FreeRTOS tasks. The thread that simulates interrupt processing waits until it is informed by another thread in the system that there is an interrupt pending. For example, the thread that simulates the generation of tick interrupts sets an interrupt pending bit, and then informs the Windows thread that simulates interrupts being processed that an interrupt is pending. The simulated interrupt processing thread will then execute and look at all the possible interrupt pending bits - performing any simulated interrupt processing and clearing interrupt pending bits as necessary.

Using the Eclipse and MingW (GCC) Demo

Obtaining the compiler

The MingW compilation tools are not included as part of the Eclipse distribution and must be downloaded separately.

Importing the FreeRTOS simulator project into an Eclipse workspace

To import the FreeRTOS simulator project into Eclipse:

1. Start the Eclipse IDE, and go to the Eclipse Workbench.
2. Select 'Import' from the Eclipse 'File' menu. A dialogue box will appear.
3. In the dialogue box, select 'General | Existing Projects Into Workspace'. Another dialogue box will appear that allows you to navigate to and select a root directory.
4. Select FreeRTOS/Demo/WIN32-MingW as the directory - this will reveal a project called RTOSDemo, which is the project that should be imported.

The Demo Application

Functionality

The constant mainCREATE_SIMPLE_BLINKY_DEMO_ONLY, which is #defined at the top of main.c, is used to switch between a simply Blinky style demo, and a more comprehensive test and demo application, as described in the next two sections.

Functionality when mainCREATE_SIMPLE_BLINKY_DEMO_ONLY is set to 1

If mainCREATE_SIMPLE_BLINKY_DEMO_ONLY is set to 1 then main() will call main_blinky(), which is implemented in main_blinky.c.

main_blinky() creates a very simple demo that includes two tasks and one queue. One task repeatedly sends the value 100 to the other task through the queue. The receiving task prints out a message each time it receives the value on the queue.

Functionality when mainCREATE_SIMPLE_BLINKY_DEMO_ONLY is set to 0

If mainCREATE_SIMPLE_BLINKY_DEMO_ONLY is set to 0 then main() will call main_full(), which is implemented in main_full.c.

The demo created by main_full() is very comprehensive. The tasks it creates consist mainly of the standard demo tasks - which don't perform any particular functionality other than testing the port and demonstrating how the FreeRTOS API can be used.

The 'check' task created by main_full()

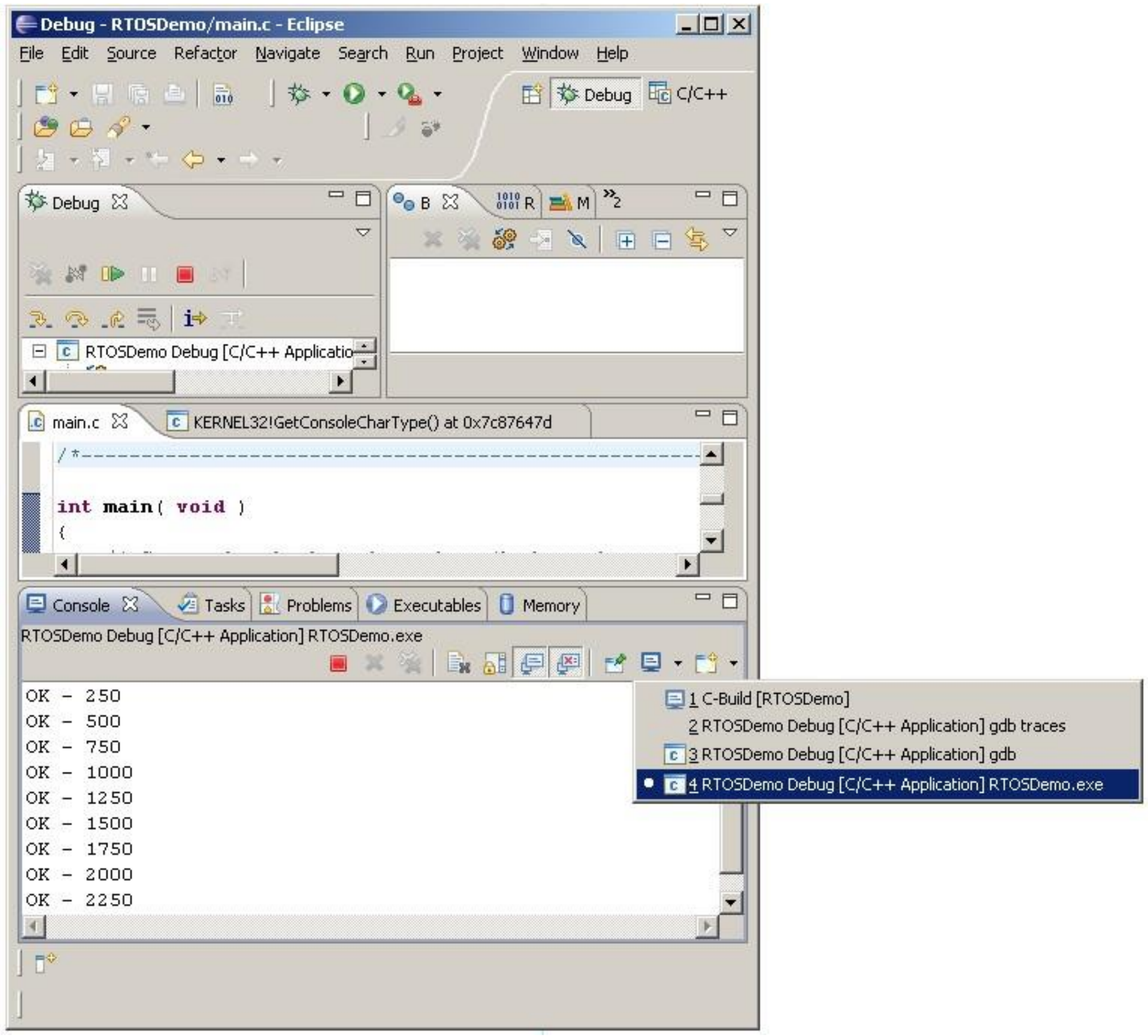
The full demo creates a 'check' task in addition to the standard demo tasks. This only executes every (simulated) five seconds, but has the highest priority to ensure it gets processing time. Its main function is to check that all the standard demo tasks are still operational.

The check task maintains a status string that is output to the console each time it executes. If all the standard demo tasks are running without error then the string will print out "OK" and the current tick count. If an error has been detected then the string will print out a message that indicates in which task the error was reported.

Viewing the console output

The eclipse project will output strings to an integrated console. To view these strings the "RTOSDemo.exe" console must be selected using the drop down list accessible from the little computer monitor icon speed button - as shown in the image below.

The Visual Studio console output will appear in a command prompt window.


Debug - RTOSDemo/main.c - Eclipse
File Edit Source Refactor Navigate Search Run Project Window Help
Debug
C/C++
Debug
RTOSDemo Debug [C/C++ Applicatio
B
R
M
main.c
KERNEL32!GetConsoleCharType() at 0x7c87647d
Console
Tasks
Problems
Executables
Memory
1 C-Build [RTOSDemo]
2 RTOSDemo Debug [C/C++ Application] gdb traces
3 RTOSDemo Debug [C/C++ Application] gdb
4 RTOSDemo Debug [C/C++ Application] RTOSDemo.exe

```
/*-----------------------------------------------------------

int main( void )
{
```

RTOSDemo Debug [C/C++ Application] RTOSDemo.exe

```
OK - 250
OK - 500
OK - 750
OK - 1000
OK - 1250
OK - 1500
OK - 1750
OK - 2000
OK - 2250
```

Selecting the "RTOSDemo.exe" console during an Eclipse debug session

---

Defining and Using Simulated Interrupt Service Routines

Defining a handler for a simulated interrupt service routine

Interrupt service routines must have the following prototype:

unsigned long ulInterruptName(void);

where 'ulInterruptName' can be any appropriate function name.

If executing the routine should result in a context switch then the interrupt function must return pdTRUE. Otherwise the interrupt function should return pdFALSE.

Installing a handler for a simulated interrupt service routine

Handlers for simulated interrupt service routines can be installed using the vPortSetInterruptHandler() function which is defined in the Win32 port layer. This has the prototype shown below:

```
void vPortSetInterruptHandler( unsigned long ulInterruptNumber, unsigned long (*pvHandler)( void ) );
```

ulInterruptNumber must be a value in the range 3 to 31 inclusive and be unique within the application (meaning a total of 29 simulated interrupts can be defined in any application). Numbers 0 to 2 inclusive are used by the simulator itself.

pvHandler should point to the handler function for the interrupt number being installed.

Triggering a simulated interrupt service routine

Interrupts can be set pending and, if appropriate, executed by calling the vPortGenerateSimulatedInterrupt() function, which is also defined as part of the Win32 port layer. It has the prototype shown below:

```
void vPortGenerateSimulatedInterrupt( unsigned long ulInterruptNumber );
```

ulInterruptNumber is the number of the interrupt that is to be set pending, and corresponds to the ulInterruptNumber parameter of vPortSetInterruptHandler().

An example of installing and triggering an interrupt

The simulator itself uses three interrupts, one for a task yield, one for the simulated tick, and one for terminating a Windows thread that was executing a FreeRTOS task that has since been deleted. As a simple example, shown below is the code for the yield interrupt.

The interrupt function does nothing other than request a context switch, so just returns pdTRUE. It is defined using the following code:

```
static unsigned long prvProcessYieldInterrupt( void )
{
   /* There is no processing to do here, this interrupt is just used to cause
   a context switch, so it simply returns pdTRUE. */
   return pdTRUE;
}
```

The simulated interrupt handler function is then installed using the following call, where portINTERRUPT_YIELD is defined as 2:

```
vPortSetInterruptHandler( portINTERRUPT_YIELD, prvProcessYieldInterrupt );
```

This is the interrupt that should execute whenever taskYIELD()/portYIELD() is called, so the Win32 port version of portYIELD() is defined as:

```
#define portYIELD() vPortGenerateSimulatedInterrupt( portINTERRUPT_YIELD )
```

Conclusion: Successfully studied RTOS Simulator.

Assignment 2: Study of RTLinux Features.

Aim: To Study RTLinux Features.

Objective: To learn RTLinux Features.

Theory:

RTLinux Overview

This section is intended to give users a top-level understanding of RTLinux. It is not designed as an in-depth technical discussion of the system's architecture. Readers interested in the topic can start with Michael Barabanov's Master's Thesis. (A postscript version is available for download at www.rtlinux.org/documents/papers/thesis.ps).

The basic premise underlying the design of RTLinux is that it is not feasible to identify and eliminate all aspects of kernel operation that lead to unpredictability. These sources of unpredictability include the Linux scheduling algorithm (which is optimized to maximize throughput), device drivers, uninterrruptible system calls, the use of interrupt disabling and virtual memory operations. The best way to avoid these problems is to construct a small, predictable kernel separate from the Linux kernel, and to make it simple enough that operations can be measured and shown to have predictable execution. This has been the course taken by the developers of RTLinux. This approach has the added benefit of maintainability - prior to the development of RTLinux, every time new device drivers or other enhancements to Linux were needed, a study would have to be performed to determine that the change would not introduce unpredictability.

Figure 1.1 shows the basic Linux kernel without hard realtime support. You will see that the Linux kernel separates the hardware from user-level tasks. The kernel has the ability to suspend any user-level task, once that task has outrun the ``slice of time'' allotted to it by the CPU. Assume, for example, that a user task controls a robotic arm. The standard Linux kernel could potentially preempt the task and give the CPU to one which is less critical (e.g. one that boots up Netscape). Consequently, the arm will not meet strict timing requirements. Thus, in trying to be ``fair'' to all tasks, the kernel can prevent critical events from occurring.

Figure 1.2 shows a Linux kernel modified to support hard realtime. An additional layer of abstraction - termed a ``virtual machine" in the literature - has been added between the standard Linux kernel and the computer hardware. As far as the standard Linux kernel is concedrned, this new layer appears to be actual hardware. More importantly, this new layer introduces its own fixed-priority scheduler. This scheduler assigns the lowest priority to the standard Linux kernel, which then runs as an independent task. Then it allows the user to both introduce and set priorities for any number of realtime tasks.

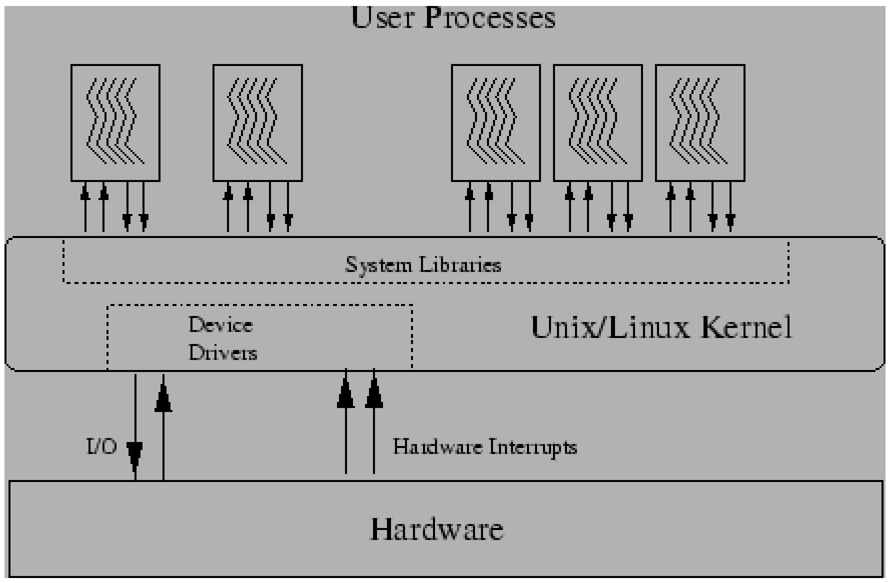


Figure 1.1: Detail of the bare Linux kernel

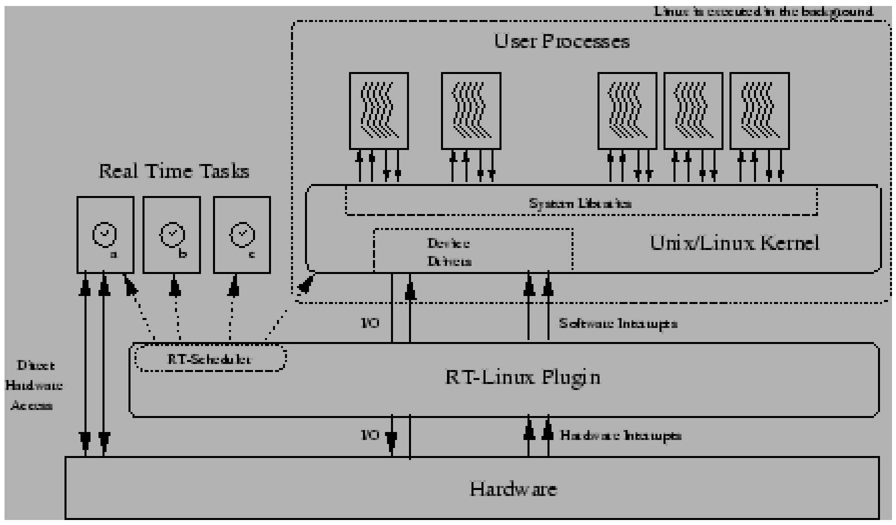


Figure 1.2: Detail of the RTLinux kernel

The abstraction layer introduced by RTLinux works by intercepting all hardware interrupts. Hardware interrupts not related to realtime activities are held and then passed to the Linux kernel as software interrupts when the RTLinux kernel is idle and the standard Linux kernel runs. Otherwise, the appropriate realtime interrupt service routine (ISR) is run. The RTLinux executive is itself nonpreemptible. Unpredictable delays within the RTLinux executive are eliminated by its small size and limited operations. Realtime tasks have two special attributes: they are ``privileged'' (that is, they have direct access to hardware), and they do not use virtual memory. Realtime tasks are written as special Linux modules that can be dynamically loaded into memory. They are are not expected to execute Linux system calls. The initialization code for a realtime tasks initializes the realtime task structure and informs RTLinux of its deadline, period, and release-time constraints. Non-periodic tasks are supported through the use of interrupts.

In contrast with some other approaches to realtime, RTLinux leaves the Linux kernel essentially untouched. Via a set of relatively simple modifications, it manages to convert the existing Linux kernel into a hard realtime environment without hindering future Linux development.

The Basic API: Writing RTLinux Modules

This chapter Introduces critical concepts that must be grasped in order to successfully write RTLinux modules. It also presents the basic Application Programming Interface (API) used in all RTLinux programs. Then it steps the user through the creation of a basic ``Hello World'' programming example, which is intended to help the user in developing their very first RTLinux program.

Understanding an RTLinux Program

In the latest versions of RTLinux, programs are not created as standalone applications. Rather, they are modelled as modules which are loaded into the Linux kernel space. A Linux module is nothing but an object file, usually created with the -c flag argument to gcc. The module itself is created by compiling an ordinary C language file in which the main () function is replaced by a pair of init/cleanup functions:

```
int init_module();
void cleanup_module();
```

As its name implies, the init_module () function is called when the module is first loaded into the kernel. It should return 0 on success and a negative value on failure. Similarly, the cleanup_module is called when the module is unloaded.

For example, if we assume that a user has created a C file named my_module.c, the code can be converted into a module by typing the following:

```
gcc -c {SOME-FLAGS} my_module.c
```

This command creates a module file named my_module.o, which can now be inserted into the kernel. To insert the module into the kernel, we use the insmod command. To remove it, the rmmod command is used.

Documentation for both of these commands can be accessed by typing:

man 8 insmod, and man 8 rmmod.

Here, the ``8'' forces the man command to look for the manual pages associated with system administration. From now on, we will refer to commands by their name and manual category. Using this format, these two commands would be referred to as insmod (8) and rmmod (8).

For further information on running RTLinux programs, refer to Appendix A.

The Basic API

Now that we understand the general structure of modules, and how to load and unload them, we are ready to look at the RTLinux API.

Creating RTLinux POSIX Threads

A realtime application is usually composed of several ``threads'' of execution. Threads are light-weight processes which share a common address space. Conceptually, Linux kernel control threads are also RTLinux threads (with one for each CPU in the system). In RTLinux, all threads share the Linux kernel address space.

To create a new realtime thread, we use the pthread_create(3) function. This function must only be called from the Linux kernel thread (i.e., using init_module()):

```
#include <pthread.h>
int pthread_create(pthread_t * thread,
                   pthread_attr_t * attr,
                   void *(*start_routine)(void *),
                   void * arg);
```

The thread is created using the attributes specified in the ``attr'' thread attributes object. If attr is NULL, default attributes are used. For more detailed information, refer to the POSIX functions:

- pthread_attr_init(3),
- pthread_attr_setschedparam(3), and
- pthread_attr_getschedparam(3)

as well as these RTL-specific functions:

- pthread_attr_getcpu_np(3) , and
- pthread_attr_setcpu_np(3)

which are used to get and set general attributes for the scheduling parameters and the CPUs in which the thread is intended to run.

The ID of the newly created thread is stored in the location pointed to by ``thread''. The function pointed to by start_routine is taken to be the thread code. It is passed the ``arg'' argument.

To cancel a thread, use the POSIX function:

```
pthread_cancel(pthread thread);
```

You should join the thread in cleanup_module with pthread_join() for its resources to be deallocated.

You must make sure the thread is cancelled before calling pthread_join() from cleanup_module(). Otherwise, Linux will hang waiting for the thread to finish. If unsure, use pthread_delete_np(3) instead of pthread_cancel()/pthread_join().

Time Facilities

RTLinux provides several clocks that can be used for timing functionality, such as as referencing for thread scheduling and obtaining timestamps. Here is the general timing API:

```
#include <rtl_time.h>

int clock_gettime(clockid_t clock_id, struct timespec *ts);
hrtime_t clock_gethrtime(clockid_t clock);

struct timespec {
   time_t tv_sec; /* seconds */
   long tv_nsec; /* nanoseconds */
};
```

To obtain the current clock reading, use the clock_gettime(3) function where clock_id is the clock to be read and ts is a structure which stores the value obtained.

The hrtime_t value is expressed as a single 64-bit number of nanoseconds. Thus, clock_gethrtime(3) is the same as clock_gettime, but returns the time as an hrtime_t rather than as a timespec structure.

Conversion Routines

Several routines exist for converting from one form of time reporting to the other:

```
#include <rtl_time.h>

hrtime_t timespec_to_ns(const struct timespec *ts);
struct timespec timespec_from_ns(hrtime_t t)
const struct timespec * hrt2ts(hrtime_tvalue);
```

These are especially useful macros for passing time values into nanosleep, pthread_cond_timedwait and the like.

Currently supported clocks are:

- CLOCK_MONOTONIC: This POSIX clock runs at a steady rate, and is never adjusted or reset.
- CLOCK_REALTIME: This is the standard POSIX realtime clock. Currently, it is the same as CLOCK_MONOTONIC. It is planned that in future versions of RTLinux this clock will give the world time.
- CLOCK_RTL_SCHED: The clock that the scheduler uses for task scheduling.

The following clocks are architecture-dependent. They are not normally found in user programs.

- CLOCK_8254: Used on non-SMP x86 machines for scheduling.
- CLOCK_APIC: Used on SMP x86 machines.
- CLOCK_APIC: corresponds to the local APIC clock of the processor that executes clock_gettime. You cannot read or set the APIC clock of other processors.

Scheduling Threads

RTLinux provides scheduling, which allows thread code to run at specific times. RTLinux uses a pure priority-driven scheduler, in which the highest priority (ready) thread is always chosen to run. If two threads have the same priority, which one is chosen is undefined. RTLinux uses the following scheduling API:

```
int pthread_setschedparam(pthread_t thread,
                          int policy,
                          const struct sched_param *param);
int pthread_make_periodic_np(pthread_t thread,
                             const struct itimerspec *its);
int pthread_wait_np(void);
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);

struct itimerspec {
  struct timespec it_interval; /* timer period */
  struct timespec it_value;    /* timer expiration */
};
```

Thread priority can be modified at thread creation time by using:

pthread_attr_setschedparam(3)

or afterwards by using

pthread_setschedparam(3) .

The policy argument is currently not used in RTLinux, but should be specified as SCHED_FIFO for compatibility with future versions. The structure sched_param contains the sched_priority member. Higher values correspond to higher priorities. Use:

- sched_get_priority_max(3) , and
- sched_get_priority_min(3)

to determine possible values of sched_priority.

To make a realtime thread execute periodically, users may use the *non-portable*[2.1] function:

pthread_make_periodic_np(3)

which marks the thread as periodic. Timing is specified by the itimer structure its. The it_value member of the passed struct itimerspec specifies the time of the first invocation; the it_interval is the thread period. Note that when setting up the period for task **T**, the period specified in the itimer structure can be 0. This means that task **T** will execute only once.

The actual execution timing is performed by use of the function:

pthread_wait_np(3)

This function suspends the execution of the calling thread until the time specified by:

pthread_make_periodic_np(3)

In the next section we'll put the API to practical use.

A Simpl ``Hello World" RTLinux program

We'll now write a small program that uses all of the API that we've learned thus far. This program will execute two times per second, and during each iteration it will print the message:

```
I'm here, my arg is 0
```

Code Listing

Save the following code under the filename `hello.c`:

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>

pthread_t thread;
void * start_routine(void *arg) {
   struct sched_param p;
   p . sched_priority = 1;
   pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
   pthread_make_periodic_np (pthread_self(), gethrtime(),
                           500000000);

   while (1) {
     pthread_wait_np();
     rtl_printf("I'm here; my arg is %x\n", (unsigned) arg);
   }
   return 0;
}

int init_module(void) {
   return pthread_create (&thread, NULL, start_routine, 0);
}

void cleanup_module(void) {
   pthread_cancel (thread);
   pthread_join (thread, NULL);
}
```

This program can be found in examples/hello.

Now, let's analyze the code.

Dissecting ``Hello World"

In our program, the

init_module()

function begins the entire process by creating our execution thread - embodied in the function start_routine() - with an argument of 0 passed to start_routine().

start_routine has three components: initialization, run-time and termination - best understood as the blocks before, during and after the while() loop, respectively.

Upon the first call to the newly-created thread start_routine(), the initialization section tells the scheduler to assign this thread a scheduling priority of 1 (one) with the call to p.sched_priority. Next, the thread sets the scheduler's behavior to be SCHED_FIFO for all subsequent executions with the call to pthread_setschedparam. Finally, by calling the function:

pthread_make_periodic_np()

the thread tells the scheduler to periodically execute this thread at a frequency of 2 Hz (500 microseconds). This marks the end of the initialization section for the thread.

The while() loop begins with a call to the function:

pthread_wait_np()

which blocks all further execution of the thread until the scheduler calls it again. Once the thread is called again, it executes the rest of the contents inside the while loop, until it encounters another call to:

pthread_wait_np()

Because we haven't included any way to exit the loop, this thread will continue to execute forever at a rate of 2Hz. The only way to stop the program is by removing it from the kernel with the rmmod(8) command.

Compiling and Executing ``Hello World''

In order to execute our program, we must first do the following:

1. *Compile the source code and create a module*. We can normally accomplish this by using the Linux GCC compiler directly from the command line. To simplify things, however, we'll create a Makefile. Then we'll only need to type ``make'' to compile our code.
2. *Locate and copy the rtl.mk file*. The rtl.mk file is an include file which contains all the flags needed to compile our code. For simplicity, we'll copy it from the RTLinux source tree and place it alongside of our hello.c file.

3. *Insert the module into the running RTLinux kernel*. The resulting object binary must be ``plugged in'' to the kernel, where it will be executed by RTLinux.

Let's look at these steps in some detail.

We begin by creating the Makefile that will be used to compile our hello.c program. Type the following into a file called Makefile and put it in the same directory as your hello.c program:

```
hello.o: hello.c
    gcc $(CFLAGS) hello.c
```

If you haven't already done so, locate the file rtl.mk and copy it into the same directory as your hello.c and Makefile files. The rtl.mk file can usually be found at /usr/include/rtlinux/rtl.mk.

```
cp /usr/include/rtlinux/rtl.mk .
```

(Note the trailing dot (.).)

Now, type the following:

```
make -f rtl.mk hello.o
```

This compiles the hello.c program and produces an object file named hello.o.

We now need to load the RTLinux modules. There are several ways to do this. The easiest is to use the rtlinux(1) command (as root):

```
rtlinux start hello
```

You can check the status of your modules by typing the command:

```
rtlinux status hello
```

For more information about the usage of the rtlinux(1) command, refer to its man page, or type:

```
rtlinux help
```

You should now be able to see your hello.o program printing its message twice per second. Depending on the configuration of your machine, you should either be able to see it directly in your console, or by typing:

```
dmesg
```

To stop the program, we need to remove it from the kernel. To do so, type:

```
rtlinux stop hello
```

For other ways on running RTLinux programs, refer to Appendix A.

Congratulations, you have now successfully created and run your very first RTLinux program!

The Advanced API: Getting More Out of Your RTLinux Modules

RTLinux has a rich assortment of functions which can be used to solve most realtime application problems. This chapter describes some of the more advanced concepts.

Using Floating Point Operations in RTLinux POSIX Threads

The use of floating-point operations in RTL POSIX threads is prohibited by default. The RTL-specific function pthread_setfp_np(3) is used to change the status of floating-point operations.

```
int pthread_setfp_np (pthread_tthread, int flag);
```

To enable FP operations in the thread, set the flag to 1. To disable FP operations, pass 0.

The examples/fp directory contains several examples of tasks which use floating point and the math library.

RTLinux Inter-Process Communication (IPC)

The general philosophy of RTLinux requires the realtime component of an application to be lightweight, small and simple. Applications should be split in such a way that, as long as timing restrictions are met, most of the work is done in user space. This approach makes for easier debugging and better understanding of the realtime part of the system. Consequently, communication mechanisms are necessary to interface RTLinux tasks and Linux.

RTLinux provides several mechanisms which allow communication between realtime threads and user space Linux processes. The most important are realtime FIFOs and shared memory.

Using Real-Time FIFOs

Realtime FIFOs are First-In-First-Out queues that can be read from and written to by Linux processes and RTLinux threads. FIFOs are uni-directional - you can use a pair of FIFOs for bi-directional data exchange. To use the FIFOs, the system/rtl_posixio.o and fifos/rtl_fifo.o Linux modules must be loaded in the kernel.

RT-FIFOs are Linux character devices with the major number of 150. Device entries in /dev are created during system installation. The device file names are /dev/rtf0, /dev/rtf1, etc., through /dev/rtf63 (the maximum number of RT-FIFOs in the system is configurable during system compilation).

Before a realtime FIFO can be used, it must be initialized:

```
#include <rtl_fifo.h>
int rtf_create(unsigned int fifo, int size);
int rtf_destroy(unsigned int fifo);
```

rtf_create allocates the buffer of the specified size for the fifo buffer. The fifo argument corresponds to the minor number of the device. rtf_destroy deallocates the FIFO.

These functions must only be called from the Linux kernel thread (i.e., from init_module()).

After the FIFO is created, the following calls can be used to access it from RTLinux threads: open(2) , read(2) , write(2) and close(2) . Support for other STDIO functions is planned for future releases.

Current implementation requires the FIFOs to be opened in non-blocking mode (O_NONBLOCK) by RTL threads.

You can also use the RTLinux-specific functions rtf_put (3) and rtf_get (3) .

Linux processes can use UNIX file IO functions without restriction. See the examples/measurement/rt_process.c example program for a practical application of RT-FIFOs.

Using Shared Memory

For shared memory, you can use the excellent mbuff driver by Tomasz Motylewski (motyl@chemie.unibas.ch. It is included with the RTLinux distribution and is installed in the drivers/mbuff directory. A manual is included with the package. Here, we'll just briefly describe the basic mode of operation.

First, the mbuff.o module must be loaded in the kernel. Two functions are used to allocate blocks of shared memory, connect to them and eventually deallocate them.

#include <mbuff.h>
void * mbuff_alloc(const char *name, int size);
void mbuff_free(const char *name, void * mbuf);

The first time mbuff_alloc is called with a given name, a shared memory block of the specified size is allocated. The reference count for this block is set to 1. On success, the pointer to the newly allocated block is returned. NULL is returned on failure. If the block with the specified name already exists, this function returns a pointer that can be used to access this block and increases the reference count.

mbuff_free deassociates mbuff from the specified buffer. The reference count is decreased by 1. When it reaches 0, the buffer is deallocated.

These functions are available for use in both Linux processes and the Linux kernel threads.

mbuff_alloc and mbuff_free cannot be used from realtime threads. You should call them from init_module and cleanup_module only.

Waking and Suspending RTLinux Threads

Interrupt-driven RTLinux threads can be created using the thread wakeup and suspend functions:

int pthread_wakeup_np(pthread_t thread);
int pthread_suspend_np(void);

The general idea is that a threaded task can be either awakened or suspended from within an interrupt service routine.

An interrupt-driven thread calls pthread_suspend_np(pthread_self()) and blocks. Later, the interrupt handler calls pthread_wakeup_np(3) for this thread. The thread will run until the next call to pthread_suspend_np(3) . An example can be found in examples/sound/irqthread.c.

Another way to implement interrupt-driven threads is to use semaphores. See examples/measurements/irqsema.c for examples of this method.

Mutual Exclusion

Mutual exclusion refers to the concept of allowing only one task at a time (out of many) to read from or write to a shared resource. Without mutual exclusion, the integrity of the data found in that shared resource could become compromised. Refer to the appendix for further information on mutual exclusion.

RTLinux supports the POSIX pthread_mutex_ family of functions (include/rtl_mutex.h). Currently the following functions are available:

- pthread_mutexattr_getpshared(3)
- pthread_mutexattr_setpshared(3)
- pthread_mutexattr_init(3)
- pthread_mutexattr_destroy(3)
- pthread_mutexattr_settype(3)

- pthread_mutexattr_gettype(3)
- pthread_mutex_init(3)
- pthread_mutex_destroy(3)
- pthread_mutex_lock(3)
- pthread_mutex_trylock(3)
- pthread_mutex_unlock(3)

The supported mutex types include:

- PTHREAD_MUTEX_NORMAL (default POSIX mutexes) and
- PTHREAD_MUTEX_SPINLOCK (spinlocks)

See examples/mutex for a test program. POSIX semaphores are also supported. An example using POSIX semaphores can be found in examples/mutex/sema_test.c.

Accessing Physical Memory and I/O Ports from RTLinux Threads

These capabilities are essential for programming hardware devices in the computer. RTLinux, just like ordinary Linux, supports the /dev/mem device (**man 4 mem**) for accessing physical memory from RTLinux threads. The rtl_posixio.o module must be loaded. The program opens /dev/mem, *mmaps* it, and then proceeds to read and write the mapped area. See examples/mmap for an example.

In a module, you can call mmap from Linux mode only (i.e., from init_module()). Calling mmap from RT-threads will fail.

Another way to access physical memory is via Linux's **ioremap** call:

```
char *ptr = ioremap(PHYS_AREA_ADDRESS, PHYS_AREA_LENGTH);
...
ptr[i] = x;
```

IO port access functions (specifically for x86 architecture) are as follows:

- Output a byte to a port:
- #include <asm/io.h>
- void rtl_outb(char value, short port)
- void rtl_outb_p(char value, short port)
- Output a word to a port:
- #include <asm/io.h>
- void outw(unsigned int value, unsigned short port)
- void outw_p(unsigned int value, unsigned short port)
- Read a byte from a port:
- #include <asm/io.h>
- char rtl_inb(unsigned short port)
- char rtl_inb_p(unsigned short port)
- Read a word from a port:
- #include <asm/io.h>

- short inw(unsigned short port)
- short inw_p(unsigned short port)

Note, the order of arguments is value, port in the output functions. This may cause confusion when porting old code from other systems.

Functions with the ``_p'' suffix (e.g., outb_p) provide a small delay after reading or writing to the port. This delay is needed for some slow ISA devices on fast machines. (See also the Linux I/O port programming mini-HOWTO).

Check out examples/sound to see how some of these functions are used to program the PC realtime clock and the speaker.

Soft and Hard Interrupts

There are two types of interrupts in RTLinux: hard and soft.

Soft interrupts are normal Linux kernel interrupts. They have the advantage that some Linux kernel functions can be called from them safely. However, for many tasks they do not provide hard realtime performance; they may be delayed for considerable periods of time.

Hard interrupts (or realtime interrupts), on the other hand, have much lower latency. However, just as with realtime threads, only a very limited set of kernel functions may be called from the hard interrupt handlers.

Hard Interrupts

The two functions:

- rtl_request_irq(3) and
- rtl_free_irq(3)

are used for installing and uninstalling hard interrupt handlers for specific interrupts. The manual pages describe their operation in detail.

```
#include <rtl_core.h>
int rtl_request_irq(unsigned int irq,
                unsigned int (*handler) (unsigned int,
                struct pt_regs *));
int rtl_free_irq(unsigned int irq);
```

Soft interrupts

```
int rtl_get_soft_irq(
        void (*handler)(int, void *, struct pt_regs *),
        const char * devname);
void rtl_global_pend_irq(int ix);
void rtl_free_soft_irq(unsigned int irq);
```

The rtl_get_soft_irq(3) function allocates a virtual irq number and installs the handler function for it. This virtual interrupt can later be triggered using rtl_global_pend_irq(3) . rtl_global_pend_irq is safe to use from realtime threads and realtime interrupts. rtl_free_soft_irq(3) frees the allocated virtual interrupt.

Note that soft interrupts are used in the RTLinux FIFO implementation (fifos/rtl_fifo.c).

Special Topics

You may never find yourself needing to know any of the following. Then again, you might.

Symmetric Multi-Processing Considerations

From the point of view of thread scheduling, RTLinux implements a separate UNIX process for each active CPU in the system. In general, thread control functions can only be used for threads running on the local CPU. Notable exceptions are:

- int pthread_wakeup_np(pthread_t thread) : wake up suspended thread
- int pthread_cancel (pthread_t thread) : cancel thread
- int pthread_join(pthread_t thread) : wait for thread to finish
- int pthread_delete_np (pthread_t thread) : kill the thread

By default, a thread is created to run on the current CPU. To assign a thread to a particular CPU, use the pthread_attr_setcpu_np(3) function to set the CPU pthread attribute. See examples/mutex/mutex.c.

RTLinux Serial Driver (rt_com)

rt_com(3) is a driver for 8250 and 16550 families of UARTs commonly used in PCs (COM1, COM2, etc.). The available API is as follows:

```
#include <rt_com.h>
#include <rt_comP.h>
void rt_com_write(unsigned int com, char *pointer, int cnt);
int rt_com_read(unsigned int com, char *pointer, int cnt);
int rt_com_setup(unsigned int com, unsigned int baud,
            unsigned int parity, unsigned int stopbits,
            unsigned int wordlength);


#define RT_COM_CNT n
struct rt_com_struct
{
   int magic;              // unused
   int baud-base;            // base-rate; 11520
                     // (BASE_BAUD in rt_comP.h;
                     // for standard ports.
   int port;               // port number
   int irq;               // interrupt number (IRQ)
                     //     for the port
   int flag;               // flags set for this port
   void (*isr)(void)          // address of the interrupt
                     //     service routine
   int type;               //
   int ier;               // a copy of the IER register
   struct rt_buf_struct ibuf; // address of the port input
                     //     buffer
   struct rt_buf_struct obuf; // address of the port output
                     //     buffer
} rt_com_table [RT_COM_CNT];
```

where

- rt_com_write(3) - writes cnt characters from buffer ptr to the realtime serial port com.
- rt_com_read(3) - attempts to read cnt characters to buffer ptr from the realtime serial port com.
- rt_com_setup(3) - is used to dynamically change the parameters of each realtime serial port.

rt_com is a Linux module. The user must specify relevant serial port information via entries in rt_com_setup. In addition, the user must specify - via entries in the rt_com_table (located in rt_com.h) - the following:

- Number of serial ports available (n)
- Serial ports and relevant parameters for each, and
- An ISR to be executed when the port irq fires.

When rt_com (3) is installed with either insmod(8), modprobe(8) or rtlinux(1) , its init_module() function (in rt_com.c) requests the port device memory, registers the ISR and sets various default values for each port entry in rt_com_table.

Interfacing RTLinux Components to Linux

RTLinux threads, sharing a common address space with the Linux kernel, can in principle call Linux kernel functions. This is usually *not* a safe thing to do, however, because RTLinux threads may run even while Linux has interrupts disabled. Only functions that do not modify Linux kernel data structures (e.g., vsprintf) should be called from RTLinux threads.

RTLinux provides two delayed execution mechanisms to overcome this limitation: soft interrupts and task queues.

The RTLinux white paper discusses this topic in more detail.

Writing RTLinux Schedulers

Most users will never be required to write a scheduler. Future versions of RTLinux are expected to have a fully customizable scheduler, but in the meantime, here are some points to help the rest of you along:

- The scheduler is implemented in the scheduler/rtl_sched.c file
- The scheduler's architecture-dependent files are located in include/arch-i386 and scheduler/i386
- The scheduling decision is taken in the rtl_schedule() function. Thus, by modifying this function, it is possible to change the scheduling policy.

Further questions in this area may be addressed directly to the FSM Labs Crew.

Running RTLinux Programs

Your RTLinux distribution comes complete with several examples in the examples/ sub-directory. These examples are useful, not only for testing your brand new RTLinux distribution, but for helping get you started writing your own RTLinux programs.

General

Before you will be able to run any RTLinux programs, you must first insert the RTLinux scheduler and support modules in the modules into the Linux kernel. Use any of the following:

- **rtlinux(1)** script, the preferred method,
- **insmod(8)**,
- **modprobe(8)**, or
- the insrtl script file that has been supplied for you in the scripts directory.

For more information on Linux modules and how to manipulate them, see the Linux Kernel-HOWTO .

The following sections describe each of these methods in more detail.

Examples

Using **rtlinux**

Beginning with RTLinux 3.0-pre9, users can load and remove user modules by using the rtlinux(1) command. To insert, remove, and obtain status information about RTLinux modules, use the following commands:

```
rtlinux start my_program
rtlinux stop my_program
rtlinux staus my_program
```

For further information on the the rtlinux(1) script, type either:

```
man 1 rtlinux
```

or

```
rtlinux help.
```

Using **modprobe**

all the RTLinux modules, type the following:

```
modprobe -a rtl rtl_time rtl_sched rtl_posixio rtl_fifo
```

Using modprobe requires that modules be installed in /lib/modules/kernel_version.

Using insmod and rmmod

Suppose we have the appropriately named my_program.o. Assuming that all the appropriate RTLinux modules have already been loaded, all that's left to do is to load this module into the kernel:

```
insmod my_program.o
```

To stop the program, all we need do is type:

```
rmmod my_program
```

The RTLinux API at a Glance

Some paths to be aware of:

- RTLinux is installed in the directory /usr/rtlinux-xxx, where xxx is the version number. To simplify future development, a symbolic link has been created as /usr/rtlinux which points to /usr/rtlinux-xxx. Users are encouraged to specify their paths via this symbolic link to maintain future compatibility with new RTLinux versions.
- /usr/rtlinux/include contains all the include files necessary for development projects.
- /usr/rtlinux/examples contains the RTLinux example programs, which illustrate the use of much of the API.
- /usr/doc/rtlinux/man contains the manual pages for RTLinux.
- /usr/rtlinux/modules contains the core RTLinux modules.
- /usr/rtlinux/bin contains RTLinux scripts and utilities.

The following sections provide a listing of the various utilities and APIs available in RTLinux.

Getting Around

There are several manual pages which give overviews on the technology and the APIs.

- rtl_v1 (3) : RTLinux facilities for RTLinux v1.x.

The RTLinux V1 API is presented exclusively for backwards compatibility. It is no longer recommended for new projects. Users are strongly discouraged from starting any new projects with this API.

- rtf (4) : realtime fifo devices
- rtl_index (4) : A comprehensive list of RTLinux functions.
- rtlinux (4) : A general roadmap and description to RTLinux

Scripts and Utilities

The following utilities are designed to make your programming job easier.

- rtl-config (1) : script used to get information about the installed version of RTLinux, cflags, include paths, and documentation paths.
- rtlinux (1) : SysV compatible script used to start RTLinux and load the user's RTLinux modules

Core RTLinux API

Here is the main RTLinux API. You are encouraged to use this API for all new projects.

- clock_gethrtime (3) : get high resolution time using the specified clock
- clock_gettime : clock and timer functions
- clock_settime : clock and timer functions
- gethrtime (3) : get high resolution time
- nanosleep : high resolution sleep
- pthread_attr_getcpu_np (3) : examine and change the CPU pthread attribute
- pthread_attr_getschedparam : dynamic thread scheduling parameters access
- pthread_attr_getdetachstate : get detachstate attributes
- pthread_attr_getstacksize : get stacksize attribute
- pthread_attr_init : initialize threads attribute object
- pthread_attr_setcpu_np (3) : examine and change the CPU pthread attribute
- pthread_attr_setdetachstate : set detachstate attributes
- pthread_attr_setfp_np (3) : set and get floating point enable attribute
- pthread_attr_setschedparam : dynamic thread scheduling parameters access
- pthread_attr_setstacksize : set stacksize attribute
- pthread_cancel (3) : stop and cancel a thread (not recommended)
- pthread_create (3) : create a thread
- pthread_condattr_destroy : destroy condition variable attributes object
- pthread_condattr_getpshared : get the process-shared condition variable attributes
- pthread_condattr_init : initialize condition variable attributes object
- pthread_condattr_setpshared : set the process-shared condition variable attributes
- pthread_cond_broadcast : broadcast a condition
- pthread_cond_destroy : destroy condition variable
- pthread_cond_init : initialize condition variable
- pthread_cond_signal : signal a condition
- pthread_cond_timedwait : wait on a condition variable
- pthread_cond_wait : wait on a condition variable
- pthread_delete_np (3) : delete a realtime thread

- pthread_exit : thread termination
- pthread_join (3) : terminate a thread
- pthread_kill (3) : send a signal to a thread
- pthread_linux (3) : get the thread identifier of the Linux thread
- pthread_make_periodic_np (3) : mark a realtime thread as periodic
- pthread_mutexattr_destroy(3) : Destroys a mutex attribute object.
- pthread_mutexattr_getprioceiling : get priority ceiling attribute of mutex attribute object.
- pthread_mutexattr_getpshared : obtains the process-shared setting of a mutex attribute object.
- pthread_mutexattr_gettype : get the mutex type
- pthread_mutexattr_init : initializes a mutex attribute object.
- pthread_mutexattr_setprioceiling : set priority ceiling attribute of mutex attribute object.
- pthread_mutexattr_setpshared : sets the process-shared attribute of a mutex attribute object
- pthread_mutexattr_settype : set the mutex type
- pthread_mutex_destroy : destroys a mutex
- pthread_mutex_init(3) : initializes a mutex with the attributes specified in the specified mutex attribute object.
- pthread_mutex_lock : locks an unlocked mutex. If the mutex is already locked, the calling thread blocks until the thread that currently holds the mutex releases it.
- pthread_mutex_trylock : tries to lock a mutex. If the mutex is already locked, the calling thread returns without wating for the mutex to be freed.
- pthread_mutex_unlock : unlocks a mutex.
- pthread_getschedparam : get schedparam attribute
- pthread_self : get calling thread's ID
- pthread_setcancelstate : set cancelability state
- pthread_setschedparam : set schedparam attribute
- pthread_setfp_np (3) : allow use of floating-point operations in a thread.
- pthread_suspend_np (3) : suspend execution of a realtime thread.
- pthread_wait_np (3) : suspend the current thread until the next period
- pthread_wakeup_np (3) : wake up a realtime thread.
- rt_com (3) : serial port driver for RTLinux
- rt_com_read (3) : read data in realtime from a serial por
- rt_com_setup (3) : dynamically change the parameters of each realtime serial port.
- rt_com_table (3) : an array of descriptions, one per serial port.
- rt_com_write (3) : write data in realtime to a serial port
- rtf_create (3) : create a realtime fifo
- rtf_create_handler (3) : install a handler for realtime fifo data
- rtf_create_rt_handler (3) : install a handler for realtime fifo data
- rtf_destroy (3) : remove a realtime fifo created with rtf_create(3)
- rtf_flush (3) : empty a realtime FIFO
- rtf_get (3) : read data from a realtime fifo

- rtf_link_user_ioctl (3) : install an ioctl (3) handler for a realtime FIFO.
- rtf_put (3) : write data to a realtime fifo
- rtf_make_user_pair (3) : make a pair of RT-FIFOs act like a bidirectional FIFO
- rtl_allow_interrupts (3) : control the CPU interrupt state
- rtl_free_irq (3) : install and remove realtime interrupt handlers
- rtl_free_soft_irq (3) : install and remove software interrupt handlers
- rtl_get_soft_irq (3) : install and remove software interrupt handlers
- rtl_getcpuid (3) : get the current processor id
- rtl_getschedclock (3) : get the current scheduler clock
- rtl_global_pend_irq (3) : schedule a Linux interrupt
- rtl_hard_disable_irq (3) : interrupt control
- rtl_hard_enable_irq (3) : interrupt control
- rtl_no_interrupts (3) : control the CPU interrupt state
- rtl_printf (3) : print formatted output
- rtl_request_irq (3) : install and remove realtime interrupt handlers
- rtl_restore_interrupts (3) : control the CPU interrupt state
- rtl_setclockmode (3) : set the RTLinux clock mode
- rtl_stop_interrupts (3) : control the CPU interrupt state
- rtlinux_sigaction (3) : RTLinux v3 User-Level signal handling functions.
- rtlinux_signal (3) : list of available RTLinux User-Level signals
- rtlinux_sigprocmask (3) : RTLinux v3 User-Level signal handling functions.
- rtlinux_sigsetops (3) : RTLinux User-Level signal set operations
- sched_get_priority_max : get priority limits for the scheduling policy
- sched_get_priority_min : get priority limits for the scheduling policy
- sem_init : initialize POSIX semaphore
- sem_destroy : destroy an unnamed POSIX semaphore
- sem_getvalue : get the value of a sempahore
- sem_post : unlock a semaphore
- sem_trywait : lock a semaphore
- sem_wait : lock a semaphore
- sigaction (2) : RTLinux POSIX signal handling functions
- sysconf : get configurable system variables
- time : clock and timer functions
- uname : get name of current system
- usleep : suspend execution for an interval

Version 1.x API: Not for New Projects

The v1 API is exclusively for older RTLinux projects. It is NOT recommended for use with new projects. This listing is for backward compatibility only:

- free_RTirq (3) : uninstall an interrupt handler
- request_RTirq (3) : install an interrupt handler
- rt_get_time (3) : get time in ticks
- rt_task_delete (3) : delete a realtime task
- rt_task_init (3) : create a realtime task

- rt_task_make_periodic (3) : mark a realtime task for execution.
- rt_task_suspend (3) : suspend execution of a realtime task.
- rt_task_wait (3) : suspend execution for the current period until the next period
- rt_task_wakeup (3) : allow a previously suspended realtime task to run.
- rt_use_fp (3) : set/remove permission for task to use floating point unit

Conclusion: Successfully studied RTLinux Features.

Assignment 3: Study of VxWorks Features.

Aim: To study VxWorks Features.

Objective: To learn VxWorks Features.

Theory:

VxWorks Operating System

VxWorks operating system provides unrivalled deterministic high performance. It establishes the benchmark for a scalable, safe, secure, and dependable operating environment for mission-critical computing systems that require the highest requirements. Leading global innovators have used VxWorks for more than 40 years to produce award-winning, creative solutions for aerospace, military, rail, vehicles, medical devices, manufacturing facilities, and communications networks.

What is VxWorks Operating System?

VxWorks Operating System

VxWorks is created as proprietary software by Wind River Systems, a completely owned subsidiary of Aptiv. It was firstly launched in 1987. It is mainly intended for embedded systems that require real-time and deterministic performance. In many cases, it needs safety and security certification in aerospace and robotics, medical devices, industrial equipment, energy, transportation, defence, automotive, network infrastructure, and consumer electronics.

It supports the AMD/Intel architecture, the ARM architecture, the POWER architecture, and the RISC-V architecture. On 32 and 64-bit processors, the real-time operating system may be utilized in multicore mixed modes, symmetric multiprocessing, multi-OS architectures, and asymmetric multiprocessing.

The VxWorks development environment contains the kernel, board support packages, the Wind River Workbench development suite, and third-party software and hardware technologies. The real-time operating system in VxWorks 7 version has been redesigned for modularity and upgradeability, with the operating system kernel separated from middleware, applications, and other packages. Scalability, security, safety, connection, and graphics have all been enhanced to meet the demands of the Internet of Things (IoT).

History of VxWorks Operating System

VxWorks started in the 1980s as a set of upgrades to VRTX, a crude RTOS sold by Ready Systems in 1995. Wind River obtained the distribution rights to VRTX and significantly upgraded it by integrating a file system and an integrated development environment, among other things. Wind River designed and developed its kernel to replace VRTX within VxWorks in 1987, anticipating the loss of its reseller contract by Ready Systems.

The architecture of the VxWorks operating system

The wind microkernel is at the core of the VxWorks real-time OS. The kernel is a link between the shell and the hardware, which is a software component. The kernel should execute the LabVIEW program while providing secure access to the machine's hardware.

Main Capabilities of VxWorks Operating System

There are various capabilities of the VxWorks OS. Some capabilities of the VxWorks OS are as follows:

1. Reliability and Performance

It is the first real-time operating system on Earth and Mars where reliability is required. It delivers the highest levels of performance when it's needed the most.

2. Safety

VxWorks was developed with security in mind. It has undergone extensive testing and certification to meet specified requirements.

3. Security

It provides a set of capabilities designed to protect devices, data, and intellectual property in the linked world. VxWorks Security Services meet strict security standards across industries when combined with the development processes.

Functions of the VxWorks Operating System

There are various functions of the VxWorks OS. Some functions of the VxWorks OS are as follows:

1. Task Management

Task management is an example of software that is being run. The task comprises many components, such as a memory address, identifier, programme counter, and context data. The task consists of carrying out their directions.

There are mainly two types of operating system tasks: single-tasking and multitasking. The single-task approach only works with one process at a time. The multitasking method allows numerous processes to run at the same time. Because the VxWorks kernel supports multitasking, we can run numerous jobs simultaneously.

2. Scheduling

The scheduling system is the backbone of the RTOS and is used to keep the processor's workload consistent and balanced. As a result, each process is completed within a certain amount of time. Priority and round round-robin scheduling are two key techniques in Vxworks OS.

3. Memory Management

Memory management is a key part of the OS, which handles the computer's memory. Physical and virtual memory are the two types of memory modules found in a CPU. The hard disk is defined as physical memory, while virtual memory is defined as RAM. An OS manages the RAM address spaces, and the virtual memory address is assigned after the real memory address.

All application jobs in the VxWorks embedded RTOS share the same address space, implying that faulty apps could mistakenly access system resources and compromise the overall system's stability. The VxWorks system includes one optional tool called VxVMI, which can be used to give each task its own address space. VxWorks does not provide privilege protection. VxWorks privilege level is always 0.

4. Interrupts

VxWorks OS interrupt service routines execute in a distinct context outside of any process context to give the quickest possible response to external interrupts. There are no process context switches involved. The interrupt vector table stores the ISR address, which is invoked directly from the hardware. The ISR first does some work (for example, saving registers and setting up the stack) before calling the C functions that the user has connected.

5. Round-Robin Scheduling

This scheduling algorithm is utilized by the processor while executing the process. It is specially intended for time-sharing systems. Round-robin scheduling allots a specific amount of time to each process. Once a process has been completed in a certain time period, other processes are permitted to complete in the same time period. In a real-time operating system, round-robin scheduling performs better.

6. Priority Scheduling

Priority scheduling assigns a priority to each process (thread). The highest priority thread would be executed first. Priority processes are implemented on a first-come, first-served basis. Priority can be determined based on time, memory, or any other resource demand.

Hardware Support of VxWorks Operating System

VxWorks has been adapted to a variety of platforms and can currently run on almost any latest CPU used in the embedded industry. It includes the Intel x86 processor family (including the Intel Quark SoC), MIPS, PowerPC (including BAE RAD), Intel i960, SPARC, Fujitsu FR-V, Freescale ColdFire, SH-4, and the ARM, StrongARM, and xScale CPU families. It provides an interface between all supported hardware and the operating system via a standard board support package (BSP). Its developer kit offers a standardized API and a dependable environment for designing RTOS. Popular SSL/TLS libraries, such as wolfSSL, support VxWorks.

Platforms of VxWorks Operating System

VxWorks operating system is a collection of runtime components and development tools. The run time components are an OS (UP and SMP), which are the software for app support and hardware support. VxWorks' primary development tools include compilers such as Diab, GNU, and Intel C++ Compiler (ICC) and build and setup tools. Additionally, the system provides productivity tools, including development support tools, Workbench development suite, and Intel tools for asset tracking and host support.

The VxWorks OS platform is a modular, vendor-agnostic, open system that may run on various third-party applications and hardware. The OS kernel is isolated from middleware, programs, and other packages, making problem fixes and testing new features easier. A layered source development system solution enables the simultaneous installation of several versions of any stack, allowing developers to choose which version of any feature set must be included in the VxWorks kernel libraries.

Uses of Vxwork Operating System

VxWorks is utilized in many market segments, including aerospace and industrial, consumer electronics, defence, automotive, medical, and networking. VxWorks is also utilized as the onboard operating system in several well-known goods.

Features of the VxWorks Operating System

There are various features of the VxWorks OS. Some features of the VxWorks OS are as follows:

1. Memory protection strategies insulate user-mode applications from other user-mode apps and the kernel.
2. It provides memory protection.
3. It provides a real-time processor.
4. It contains several file systems, including Disk Operating System Filing System, High-Reliability File System, and Network File System.
5. It is an error-handling framework.
6. It offers an Internet Protocol Version 6 (IPV 6) networking stack.
7. It has a multitasking kernel with preemptive, round-robin scheduling and quick interrupt response.
8. It has a 64-bit operating system.
9. It contains a dual-mode IPv6 networking stack with IPv6 Ready Logo certification.
10. It offers support for symmetric multiprocessing and asymmetric multiprocessing.

Conclusion: Successfully studied VxWorks Features.

Assignment 4: Implementation of Task Creation.

Aim: To implement task creation in RTOS Simulator.

Objective: To learn to implement task creation in RTOS Simulator.

Theory:

- Suggestion to read
- FreeRTOS LPC2148 Tutorial – Task Creation
- Introduction
- API Used
  - xTaskCreate
  - vTaskDelay
  - vTaskStartScheduler
- Code
- Output

Suggestion to read

- RTOS Basics – Part 1
- RTOS Basics – PART 2
- FreeRTOS Porting for LPC2148
- LPC2148 UART Tutorial

FreeRTOS LPC2148 Tutorial – Task Creation

Introduction

FreeRTOS contains many APIs. But the very basic is Creating a Task. Because tasks are concurrently running when the system boots up. So today we will look at simple task creation.

API Used

- xTaskCreate
- vTaskDelay
- vTaskStartScheduler

xTaskCreate

This FreeRTOS API is used to create a task. Using this API we can create more tasks.

```
portBASE_TYPE xTaskCreate  (      pdTASK_CODE pvTaskCode,

                        const signed portCHAR * const pcName,

                        unsigned portSHORT usStackDepth,

                        void *pvParameters,
                        unsigned portBASE_TYPE uxPriority,

                        xTaskHandle *pxCreatedTask );
```

- **pvTaskCode**: a pointer to the function where the task is implemented. (Address of the function)
- **pcName**: given name to the task. This is useless to FreeRTOS but is intended for debugging purposes only.
- **usStackDepth**: length of the stack for this task in words. The actual size of the stack depends on the microcontroller.
- **pvParameters**: a pointer to arguments given to the task.
- **uxPriority**: priority given to the task, a number between 0 and MAX_PRIORITIES – 1.
- **pxCreatedTask**: a pointer to an identifier that allows handling the task. If the task does not have to be handled in the future, this can be left NULL.

vTaskDelay

We are using this API for delay purposes.

void vTaskDelay(TickType_t xTicksToDelay);

- **xTicksToDelay** : The number of tick interrupts that the calling task will remain in the Blocked state before being transitioned back into the Ready state. For example, if a task called vTaskDelay(100) when the tick count was 10,000, then it would immediately enter the Blocked state and remain in the Blocked state until the tick count reached 10,100. Any time that remains between vTaskDelay() being called, and the next tick interrupt occurring, counts as one complete tick period. Therefore, the highest time resolution that can be achieved when specifying a delay period is, in the worst case, equal to one complete tick interrupt period. The macro pdMS_TO_TICKS() can be used to convert milliseconds into ticks.

vTaskStartScheduler

Starts the FreeRTOS scheduler running.
Typically, before the scheduler has been started, main() (or a function called by main()) will be executing. After the scheduler has been started, only tasks and interrupts will ever execute. Starting the scheduler causes the highest priority task that was created while the scheduler was in the Initialization state to enter the Running state.

void vTaskStartScheduler(void);

Code

In this code, I'm creating two tasks.

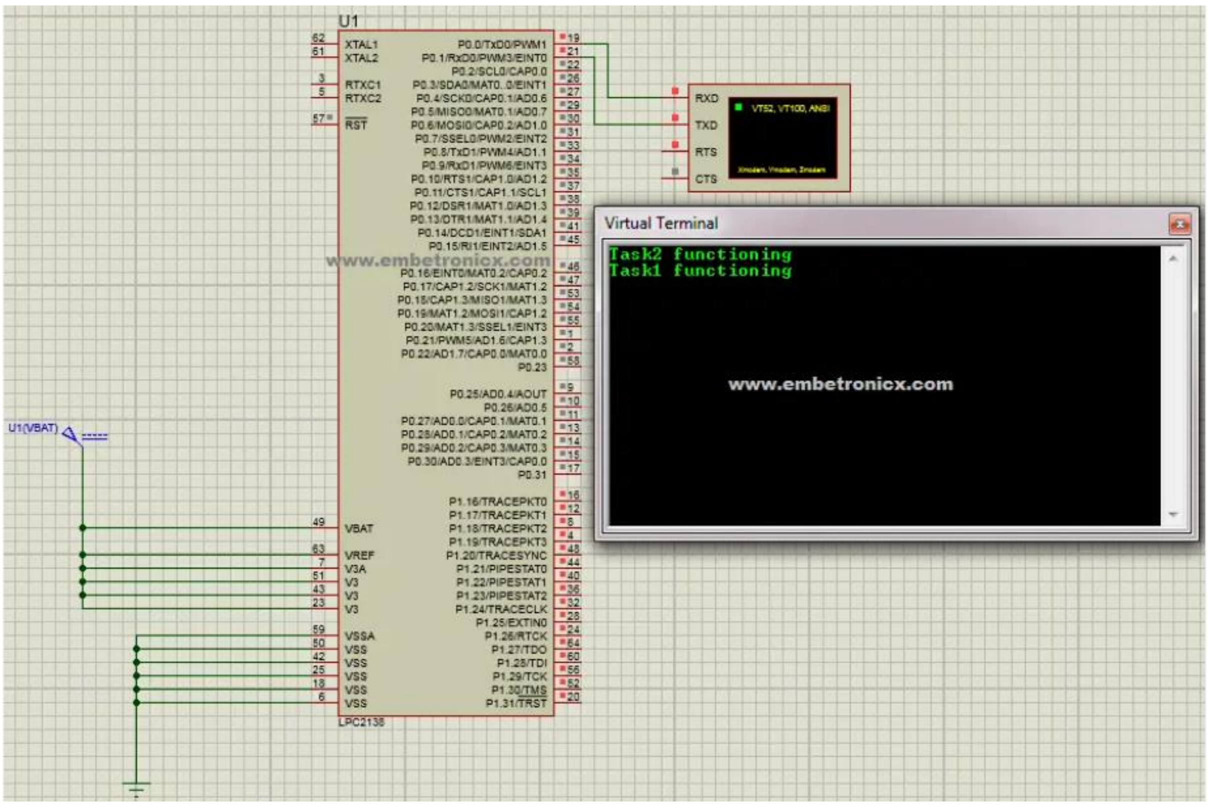
- Task 1
- Task 2

Task 1 prints "Task1 functioning" in the serial port. Task 2 prints "Task2 functioning" in the serial port. Let's go through the code. Here I'm adding only the main file. If you want to download the full project, please visit here (GitHub).

```
#include <lpc214x.h>

#include <stdlib.h>

#include "FreeRTOS.h"

#include "task.h"

#include "uart0.h"

void task1(void *q);

void task2(void *a);

void initpll(void);

int main(void)

{

  initpll();

  initserial();

  xTaskCreate(task1,"task1",128,NULL,1,NULL);

  xTaskCreate(task2,"task2",128,NULL,2,NULL);

  vTaskStartScheduler();

}

void task1(void *q)

{

  while(1) {

    sendsserial("Task1 functioning");

    sendsserial("\r\n");

    vTaskDelay(1000);

  }

}

void task2(void *a)
```

```
{
  while(1) {
    sendsserial("Task2 functioning");
    sendsserial("\r\n");
    vTaskDelay(1000);
  }
}
void initpll(void)
{
  PLL0CON=0x01;
  PLL0CFG=0x24;
  PLL0FEED=0xAA;
  PLL0FEED=0x55;
  while(!(PLL0STAT&1<<10));
  PLL0CON=0x03;
  PLL0FEED=0xAA;
  PLL0FEED=0x55;
  VPBDIV=0x01;
}
```

Output:



```
Task2 functioning
Task1 functioning
```

Conclusion: Successfully implemented a task using RTOS Simulator.

Assignment 5: Implementation of Priority-Driven Scheduling.

Aim: To implement Priority-Driven Scheduling.

Objective: To learn to implement Priority-Driven Scheduling.

Theory:

Priority Scheduling Algorithm

The algorithms schedule processes that need to be executed on the basis of priority. A Process with higher priority will be given the CPU first and this shall keep until the processes are completed. Job Queue and Ready Queue are required to perform the algorithm where the processes are placed according to their prioritised order in the ready queue and selected to be placed in the job queue for execution.

The Priority Scheduling Algorithm is responsible for moving the process from the ready queue into the job queue on the basis of the process having a higher priority.

How is the prioritization decided?

The priority of a process can be decided by keeping certain factors in mind such as the burst time of the process, lesser memory requirements of the process etc.

A few of the important terminologies to know as a part of process scheduling:-

- **Arrival Time:-** The time of the arrival of the process in the ready queue
- **Waiting Time:-** The interval for which the process needs to wait in the queue for its execution
- **Burst Time:-** The time required by the process for completion.
- **Turnaround Time:-** The summation of waiting time in the queue and burst time.

Types of Priority Scheduling Algorithm

Priority Scheduling Algorithm can be bifurcated into two halves, mentioned as-

1. Non-preemptive Priority Scheduling Algorithm
2. Preemptive Priority Scheduling Algorithm

Non-preemptive Priority Scheduling Algorithm

During the execution of a process with the highest order, if and if a process with a higher priority arrives in the queue, the execution of the ongoing process is not stopped and the arriving process, with higher priority, has to wait until the ongoing process concludes.

Preemptive Priority Scheduling Algorithm

During the execution of a process with the highest order, if and if a process with a higher priority arrives in the queue, the execution stops with the CPU provided to the process with a higher priority, thus enabling preemption as a result.

Note:- On the occasion of a special case, when two processes have the same priority then the tie-breaker for the earliest process to be executed is decided on the basis of a first come first serve (FCFS) which is a scheduling algorithm in itself.

Dry Run Example of Priority Scheduling Algorithm

Let us take an example with three processes A, B and C with mentioned priority, arrival time and burst time in the table below. We trace one by one which process takes place using a non-preemptive priority scheduling algorithm.

Process	**Burst Time**	**Arrival Time**	**Priority**
A	5	0	3
B	3	3	1
C	2	5	2

As mentioned, Process A will get the CPU being the only process at the point in time. Being non-preemptive, the process will finish its execution before another process gets the CPU.

Process	**Burst Time**	**Arrival Time**	**Priority**
C	2	5	2
B	3	3	1

On the 5th second, Process B and Process C are in waiting state with B already elapsed time, but C will get the CPU due to the reason that it has a higher priority.

Process	**Burst Time**	**Arrival Time**	**Priority**
B	3	3	1

On the 7th second, B will finish execution and CPU will be passed to the only process left i.e., C to complete execution.

The algorithm will bind up performing all the processes at the 10th second.

Algorithm of Priority Scheduling Program in C

Now that we are done discussing the theoretical concept and working example of Priority Scheduling Algorithm, we have the hint to come up with an algorithm for the priority scheduling program in c and implement the priority scheduling program in c.

- Input the total count of processes to be executed
- Input the burst time and priority for each process to be executed
- Sort the process depending on the execution by higher priority
- Calculate the Average Waiting and Average Turnaround Time
- Print the ordered execution of process and the average waiting and turnaround time

Program Code of Priority Scheduling in C

```
#include <stdio.h>

void swap(int *a,int *b)

{

int temp=*a;

*a=*b;

*b=temp;

}

int main()

{

int n;

printf("Enter Number of Processes: ");

scanf("%d",&n);

int burst[n],priority[n],index[n];
```

```
for(int i=0;i<n;i++)

{

printf("Enter Burst Time and Priority Value for Process %d: ",i+1);

scanf("%d %d",&burst[i],&priority[i]);

index[i]=i+1;

}

for(int i=0;i<n;i++)

{

int temp=priority[i],m=i;

for(int j=i;j<n;j++)

{

if(priority[j] > temp)

{

temp=priority[j];

m=j;

}

}

swap(&priority[i], &priority[m]);

swap(&burst[i], &burst[m]);

swap(&index[i],&index[m]);

}

int t=0;

printf("Order of process Execution is\n");

for(int i=0;i<n;i++)

{

printf("P%d is executed from %d to %d\n",index[i],t,t+burst[i]);

t+=burst[i];
}

printf("\n");

printf("Process Id\tBurst Time\tWait Time\n");

int wait_time=0;

int total_wait_time = 0;

for(int i=0;i<n;i++)

{

printf("P%d\t\t%d\t\t%d\n",index[i],burst[i],wait_time);

total_wait_time += wait_time;

wait_time += burst[i];

}

float avg_wait_time = (float) total_wait_time / n;

printf("Average waiting time is %f\n", avg_wait_time);

int total_Turn_Around = 0;

for(int i=0; i < n; i++){

total_Turn_Around += burst[i];

}

float avg_Turn_Around = (float) total_Turn_Around / n;

printf("Average TurnAround Time is %f",avg_Turn_Around);

return 0;

}
```

Explanation: input for a number of processes is taken from the user followed by the input for the priority and burst time for each process. Selection Sort is used to sort the process depending on their priority values with the swap function used to swap their position in the existing arrays for burst time, priority values and ordered execution. In further code, total waiting time and total turnaround time is calculated, only to be divided by the total number of processes to find the average waiting time and average turnaround time.

Output

```
Enter Number of Processes: 2
Enter Burst Time and Priority Value for Process 1: 5 3

Enter Burst Time and Priority Value for Process 2: 4 2



Order of process Execution is

P1 is executed from 0 to 5

P2 is executed from 5 to 9



Process Id  Burst Time         Wait Time

P1                  5                    0

P2                  4                    5



Average waiting time is 2.500000

Average TurnAround Time is 4.500000
```

Analysis of Code – Priority Scheduling Program in C

As sorting has been performed in the priority scheduling program in C, the time complexity will rise to O(N^2) in the worst case.

The space complexity is going to be O(1) as no extra auxiliary space will be required for operations given that we are already providing all the necessary values like burst time and priority beforehand.

Conclusion: Successfully implemented Priority Scheduling using C.

Assignment 6: Implementation of Fixed-Priority Scheduling.

Aim: To implement Fixed-Priority Scheduling.

Objective: To learn to implement Fixed-Priority Scheduling.

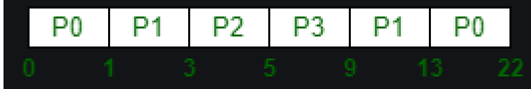
Theory:

Fixed priority pre-emptive scheduling algorithm is mostly used in real time systems.
In this scheduling algorithm the processor makes sure that the highest priority task is to be performed first ignoring the other task to be executed.
The process having highest priority is served first.

Decision Mode:
Pre-emptive: When a process arrives, its priority is compared with the current process's priority. If the new job has higher priority than the current process, the current process is suspended and new process is started.
Implementation:
Sorted FIFO queue is used for this strategy. As the new process is identified it is placed in the queue according to its priority. Hence the process having higher priority is considered first as it is placed at higher position.
Example:
Let us take the following example having 4 set of processes along with its arrival time and time taken to complete the process. Also the priority of all the process are mentioned. Consider all time values in millisecond and small value of priority means higher priority of process.

Process	Arrival Time(T0)	Time Required for Completion(T`)	Priority
P0	0	10	5
P1	1	6	4
P2	3	2	2
P3	5	4	0

Gantt chart:



Initially only P0 is present and it is allowed to run. But when P1 comes, it has higher priority. So, P0 is pre-empted and P1 is allowed to run. This process repeated till all processes complete their execution.

Statistic:

Process	Arrival Time(T0)	Completion time(T`)	Finish Time(T1)	TurnAround time(TAT=T1-T0)	Waiting time(TAT-T`)
P0	0	10	22	22	12
P1	1	6	13	12	6

P2	3	2	5	2	0
P3	5	4	9	4	0

Average Turnaround time:

= (22+12+2+4) / 4

= 40 / 4

= 10 ms

Average Waiting time:

= (12+6+0+0) / 4

= 18 / 4

= 4.5 ms

Advantages:
Priority is considered.Critical process can get even better response.
Disadvantage:
Starvation is possible for low priority process.It can be overcome by using technique called “Aging“. Aging gradually increases the priority of the process that wait in the system for long time. Context switch overhead is there.

Implementation using C:

```
#include<stdio.h>

#include<conio.h>

void main()

 {

   int x,n,p[10],pp[10],pt[10],w[10],t[10],awt,atat,i;

   printf("Enter the number of process : ");

   scanf("%d",&n);

   printf("\n Enter process : time priorities \n");

   for(i=0;i<n;i++)

    {

      printf("\nProcess no %d : ",i+1);

      scanf("%d  %d",&pt[i],&pp[i]);
      p[i]=i+1;
    }
  for(i=0;i<n-1;i++)
   {
     for(int j=i+1;j<n;j++)
     {
       if(pp[i]<pp[j])
       {
         x=pp[i];
         pp[i]=pp[j];
         pp[j]=x;
         x=pt[i];
         pt[i]=pt[j];
         pt[j]=x;
         x=p[i];
         p[i]=p[j];
         p[j]=x;
      }
   }
}
w[0]=0;
awt=0;
t[0]=pt[0];
atat=t[0];
for(i=1;i<n;i++)
 {
   w[i]=t[i-1];
   awt+=w[i];
   t[i]=w[i]+pt[i];
   atat+=t[i];
 }
printf("\n\n Job \t Burst Time \t Wait Time \t Turn Around Time   Priority \n");
for(i=0;i<n;i++)
  printf("\n %d \t\t %d  \t\t %d \t\t %d \t\t %d \n",p[i],pt[i],w[i],t[i],pp[i]);
awt/=n;
atat/=n;
printf("\n Average Wait Time : %d \n",awt);
printf("\n Average Turn Around Time : %d \n",atat);
getch();
}
```

Output:

```
Enter the number of process : 4


 Enter process : time priorities


Process no 1 : 3

1


Process no 2 : 4

2


Process no 3 : 5

3
```

```
Process no 4 : 6

4



 Job    Burst Time    Wait Time     Turn Around Time   Priority


 4          6           0           6           4


 3          5           6           11           3


 2          4           11           15           2


 1          3           15           18           1


 Average Wait Time : 8


 Average Turn Around Time : 12
```

Conclusion: Successfully implemented Fixed-Priority Scheduling.

Assignment 7: Implementation of Preemptive scheduling.

Aim: To implement Preemptive Scheduling.

Objective: To learn to implement Preemptive Scheduling.

Theory:

Preemptive Priority CPU Scheduling Algorithm is a pre-emptive method of CPU scheduling algorithm that works **based on the priority** of a process. In this algorithm, the scheduler schedules the tasks to work as per the priority, which means that a higher priority process should be executed first. In case of any conflict, i.e., when there is more than one process with equal priorities, then the pre-emptive priority CPU scheduling algorithm works on the basis of FCFS (First Come First Serve) algorithm.

How does Preemptive Priority CPU Scheduling Algorithm decide the Priority of a Process?

Preemptive Priority CPU Scheduling Algorithm uses a rank-based system to define a rank for each process, where lower rank processes have higher priority and higher rank processes have lower priority. For instance, if there are 10 processes to be executed using this Preemptive Algorithm, then process with rank 1 will have the highest priority, the process with rank 2 will have comparatively lesser priority, and process with rank 10 will have least priority.

How does Preemptive Priority CPU Scheduling Algorithm work?

- **Step-1:** Select the first process whose arrival time will be 0, we need to select that process because that process is only executing at time t=0.
- **Step-2:** Check the priority of the next available process. Here we need to check for 3 conditions.
    - if $\textbf{priority}_{(current\_process)} > \textbf{priority}_{(prior\_process)}$:- then execute the current process.
    - if $\textbf{priority}_{(current\_process)} < \textbf{priority}_{(prior\_process)}$:- then execute the prior process.
    - if $\textbf{priority}_{(current\_process)} = \textbf{priority}_{(prior\_process)}$:- then execute the process which arrives first i.e., arrival time should be first.
- **Step-3:** Repeat **Step-2** until it reaches the final process.
- **Step-4:** When it reaches the final process, choose the process which is having the highest priority & execute it. Repeat the same step until all processes complete their execution.

Program for Preemptive Priority CPU Scheduling

The preemptive Priority algorithm can be implemented using Min Heap data structure. For the detailed implementation of the Preemptive priority scheduling algorithm, please refer: Program for Preemptive Priority CPU Scheduling.

Examples to show the working of Preemptive Priority CPU Scheduling Algorithm

Example-1: Consider the following table of arrival time, Priority, and burst time for five processes **P1, P2, P3, P4,** and **P5**.

Process	Arrival Time	Priority	Burst Time
P1	0 ms	3	3 ms

Process	Arrival Time	Priority	Burst Time
P2	1 ms	2	4 ms
P3	2 ms	4	6 ms
P4	3 ms	6	4 ms
P5	5 ms	10	2 ms

The Preemptive Priority CPU Scheduling Algorithm will work on the basis of the steps mentioned below:

- **At time t = 0,**
  - Process **P1** is the only process available in the ready queue, as its arrival time is 0ms.
  - Hence Process **P1** is executed first for 1ms, from 0ms to 1ms, irrespective of its priority.
  - Remaining Burst time (B.T) for P1 = 3-1 = 2 ms.

Time Instance	Process	Arrival Time	Priority	Execution Time	Initial Burst Time	Final Burst Time
0 ms – 1 ms	P1	0 ms	3	1ms	3 ms	2 ms

- **At time t = 1 ms,**
  - There are 2 processes available in the ready queue: P1 and P2.
  - Since the priority of process P2 is higher than the priority of process P1, therefore Process P2 will get executed first.
  - Hence Process **P2** is executed for 1ms, from 1ms to 2ms.
  - Remaining Burst time (B.T) for P2 = 4-1 = 3 ms.

Time Instance	Process	Arrival Time	Priority	Execution Time	Initial Burst Time	Final Burst Time
1 ms – 2 ms	P1	0 ms	3	0	2 ms	2 ms
	P2	1 ms	2	1	4 ms	3 ms

- **At time t = 2 ms,**
  - There are 3 processes available in the ready queue: P1, P2, and P3.
  - Since the priority of process P2 is higher than the priority of process P1 and P3, therefore Process P2 will get executed first.
  - Hence Process **P2** is executed for 1ms, from 2ms to 3ms.
  - Remaining Burst time (B.T) for P2 = 3-1 = 2 ms.

Time Instance	Process	Arrival Time	Priority	Execution Time	Initial Burst Time	Final Burst Time
2 ms – 3 ms	P1	0 ms	3	0	2 ms	2 ms
	P2	1 ms	2	1	3 ms	2 ms
	P3	2 ms	4	0	6 ms	6 ms

- **At time t = 3 ms,**
  - There are 4 processes available in the ready queue: P1, P2, P3, and P4.
  - Since the priority of process P2 is highest among the priority of processes P1, P2, P3, and P4, therefore Process P2 will get executed first.
  - Hence Process **P2** is executed for 1ms, from 3ms to 4ms.
  - Remaining Burst time (B.T) for P2 = 1-1 = 0 ms.

Time Instance	**Process**	**Arrival Time**	**Priority**	**Execution Time**	**Initial Burst Time**	**Final Burst Time**
	P1	0 ms	3	0	2 ms	2 ms
	P2	1 ms	2	1	2 ms	1 ms
	P3	2 ms	4	0	6 ms	6 ms
3 ms – 4 ms	P4	3 ms	6	0	4 ms	4 ms

- **At time t = 4 ms,**
  - There are 5 processes available in the ready queue: P1, P2, P3, P4, and P5.
  - Since the priority of process P2 is highest among the priority of processes P1, P2, P3, P4, and P5, therefore Process P2 will get executed first.
  - Hence Process **P2** is executed for 1ms, from 4ms to 5ms.
  - Remaining Burst time (B.T) for P2 = 1-1 = 0 ms.
  - Since Process P2's burst time has become 0, therefore it is complete and will be removed from the process queue.

Time Instance	**Process**	**Arrival Time**	**Priority**	**Execution Time**	**Initial Burst Time**	**Final Burst Time**
	P1	0 ms	3	0	2 ms	2 ms
	~~P2~~	~~1 ms~~	~~2~~	~~1~~	~~1 ms~~	~~0 ms~~
	P3	2 ms	4	0	6 ms	6 ms
4 ms – 5 ms	P4	3 ms	6	0	4 ms	4 ms

Time Instance	Process	Arrival Time	Priority	Execution Time	Initial Burst Time	Final Burst Time
	P5	5 ms	10	0	2 ms	2 ms

- **At time t = 5 ms,**
  - There are 4 processes available in the ready queue: P1, P3, P4, and P5.
  - Since the priority of process P1 is the highest among the priority of processes P1, P3, P4 and P5, therefore Process P1 will get executed first.
  - Hence Process **P1** is executed for 2ms, from 5ms to 7ms.
  - Remaining Burst time (B.T) for P1 = 2-2 = 0 ms.
  - Since Process P1's burst time has become 0, therefore it is complete and will be removed from the process queue.

Time Instance	Process	Arrival Time	Priority	Execution Time	Initial Burst Time	Final Burst Time
5 ms – 7 ms	~~P1~~	~~0 ms~~	~~3~~	~~2~~	~~2 ms~~	~~0 ms~~
	P3	2 ms	4	0	6 ms	6 ms
	P4	3 ms	6	0	4 ms	4 ms
	P5	5 ms	10	0	2 ms	2 ms

- **At time t = 7 ms,**
  - There are 3 processes available in the ready queue: P3, P4, and P5.
  - Since the priority of process P3 is the highest among the priority of processes P3, P4, and P5, therefore Process P3 will get executed first.
  - Hence Process **P3** is executed for 6ms, from 7ms to 13ms.
  - Remaining Burst time (B.T) for P3 = 6-6 = 0 ms.
  - Since Process P3's burst time has become 0, therefore it is complete and will be removed from the process queue.

Time Instance	Process	Arrival Time	Priority	Execution Time	Initial Burst Time	Final Burst Time
7 ms – 13 ms	~~P3~~	~~2 ms~~	~~4~~	~~6~~	~~6 ms~~	~~0 ms~~
	P4	3 ms	6	0	4 ms	4 ms
	P5	5 ms	10	0	2 ms	2 ms

- **At time t = 13 ms,**
  - There are 2 processes available in the ready queue: P4 and P5.
  - Since the priority of process P4 is highest among the priority of process P4 and P5, therefore Process P4 will get executed first.
  - Hence Process **P4** is executed for 4ms, from 13ms to 17ms.
  - Remaining Burst time (B.T) for P4 = 4-4 = 0 ms.
  - Since Process P4's burst time has become 0, therefore it is complete and will be removed from the process queue.

Time Instance	Process	Arrival Time	Priority	Execution Time	Initial Burst Time	Final Burst Time
13 ms – 17 ms	~~P4~~	~~3 ms~~	~~6~~	4	~~4 ms~~	~~0 ms~~
	P5	5 ms	10	0	2 ms	2 ms

- **At time t = 17 ms,**
  - There is only 1 process available in the ready queue: P5.
  - Hence Process **P5** is executed for 2ms, from 17ms to 19ms.
  - Remaining Burst time (B.T) for P5 = 2-2 = 0 ms.
  - Since Process P5's burst time has become 0, therefore it is complete and will be removed from the process queue.

Time Instance	**Process**	**Arrival Time**	**Priority**	**Execution Time**	**Initial Burst Time**	**Final Burst Time**
17 ms – 19 ms	P5	5 ms	10	2	2 ms	0 ms

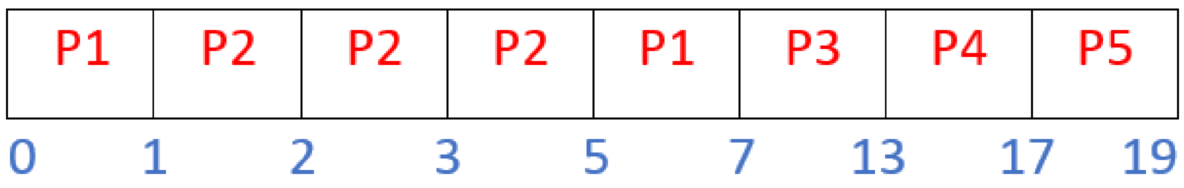
- **At time t = 19 ms,**
    - There is no more process available in the ready queue. Hence the processing will now stop.
    - The overall execution of the processes will be as shown below:

Time Instance	**Process**	**Arrival Time**	**Priority**	**Execution Time**	**Initial Burst Time**	**Final Burst Time**
0 ms – 1 ms	**P1**	**0 ms**	**3**	**1**	**3 ms**	**2 ms**
1 ms – 2 ms	P1	0 ms	3	0	2 ms	2 ms
	P2	1 ms	2	1	4 ms	3 ms
2 ms – 3 ms	P1	0 ms	3	0	2 ms	2 ms
	P2	**1 ms**	**2**	**1**	**3 ms**	**2 ms**
	P3	2 ms	4	0	6 ms	6 ms
3 ms – 4 ms	P1	0 ms	3	0	2 ms	2 ms
	P2	**1 ms**	**2**	**1**	**2 ms**	**1 ms**

Time Instance	**Process**	**Arrival Time**	**Priority**	**Execution Time**	**Initial Burst Time**	**Final Burst Time**
	P3	2 ms	4	0	6 ms	6 ms
	P4	3 ms	6	0	4 ms	4 ms
4 ms – 5 ms	P1	0 ms	3	0	2 ms	2 ms
	~~**P2**~~	~~**1 ms**~~	~~**2**~~	~~**1**~~	~~**1 ms**~~	~~**0 ms**~~
	P3	2 ms	4	0	6 ms	6 ms
	P4	3 ms	6	0	4 ms	4 ms
	P5	5 ms	10	0	2 ms	2 ms
5 ms – 7 ms	~~**P1**~~	~~**0 ms**~~	~~**3**~~	~~**2**~~	~~**2 ms**~~	~~**0 ms**~~
	P3	2 ms	4	0	6 ms	6 ms
	P4	3 ms	6	0	4 ms	4 ms
	P5	5 ms	10	0	2 ms	2 ms
7 ms – 13 ms	~~**P3**~~	~~**2 ms**~~	~~**4**~~	~~**6**~~	~~**6 ms**~~	~~**0 ms**~~
	P4	3 ms	6	0	4 ms	4 ms

Time Instance	**Process**	**Arrival Time**	**Priority**	**Execution Time**	**Initial Burst Time**	**Final Burst Time**
	P5	5 ms	10	0	2 ms	2 ms
13 ms – 17 ms	~~**P4**~~	~~**3 ms**~~	~~**6**~~	**4**	~~**4 ms**~~	~~**0 ms**~~
	P5	5 ms	10	0	2 ms	2 ms
17 ms – 19 ms	**P5**	**5 ms**	**10**	**2**	**2 ms**	**0 ms**

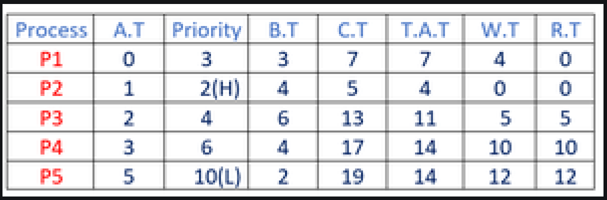
Gantt chart for above execution:



Now we need to calculate the completion time (C.T) & First Arrival Time (F.A.T) of each process through Gantt chart, as per the below relations:

) = (First Arrival Time) – (Arrival Time)

After calculating the above fields, the final table looks like



Process	A.T	Priority	B.T	C.T	T.A.T	W.T	R.T
P1	0	3	3	7	7	4	0
P2	1	2(H)	4	5	4	0	0
P3	2	4	6	13	11	5	5
P4	3	6	4	17	14	10	10
P5	5	10(L)	2	19	14	12	12

The final table

Here, ***H – Highest Priority,*** *and* ***L – Least Priority***
Some other important relations for this example include:

- **Total Turn Around Time** = 7 + 4 + 11 + 14 + 14 = 50 ms
- **Average Turn Around Time** = (Total Turn Around Time)/(no. of processes) = 50/5 = 10.00 ms

- **Total Waiting Time** = 4 + 0 + 5 + 10 + 12 = 31 ms
- **Average Waiting Time** = (Total Waiting Time)/(no. of processes) = 31/5 = 6.20 ms
- **Total Response Time** = 0 + 0 + 5 + 10 + 12 = 27 ms
- **Average Response Time** = (Total Response Time)/(no. of processes) = 27/5 = 5.40 ms

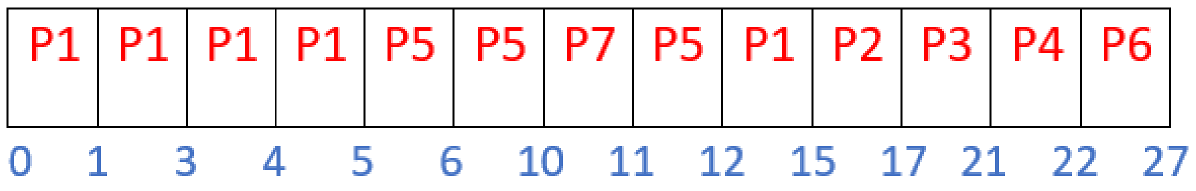
Example 2:
Consider the following table of arrival time, Priority and burst time for seven processes P1, P2, P3, P4, P5, P6 and P7

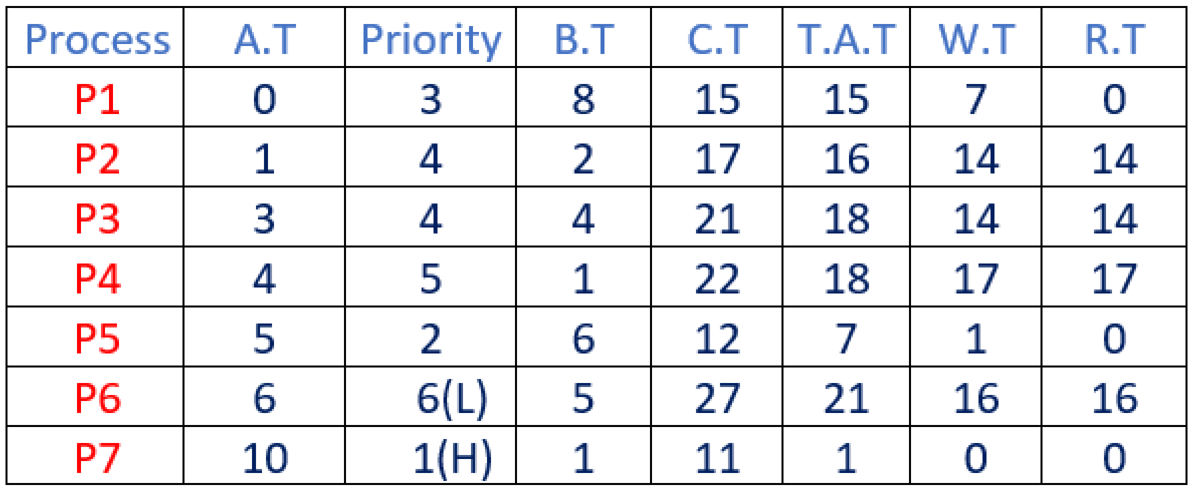
Process	Arrival Time	Priority	Burst Time
P1	0 ms	3	8 ms
P2	1 ms	4	2 ms
P3	3 ms	4	4 ms
P4	4 ms	5	1 ms
P5	5 ms	2	6 ms
P6	6 ms	6	5 ms
P7	10 ms	1	1 ms

- At time **t = 0**,
  - Process P1 is available in the ready queue, executing P1 for 1 ms
  - Remaining B.T for P1 = 8-1 = 7 ms.
- At time **t = 1**,
  - The priority of P1 is greater than P2, so we execute P1 for 2 ms, from 1 ms to 3 ms.
  - Remaining B.T for P1 = 7-2 = 5 ms.
- At time **t = 3**,
  - The priority of P1 is greater than P3, so we execute P1 for 1 ms.
  - Remaining B.T for P1 = 5-1 = 4 ms.
- At time **t = 4**,

    - The priority of P1 is greater than P4, so we execute P1 for 1 ms.
    - Remaining B.T for P1 = 4-1 = 3 ms.
- At time **t = 5**,
    - The priority of P5 is greater than P1, so we execute P5 for 1 ms.
    - Remaining B.T for P5 = 6-1 = 5 ms.
- At time **t = 6**,
    - The priority of P5 is greater than P6, so we execute P5 for 4 ms.
    - Remaining B.T for P5 = 5-4 = 1 ms.
- At time **t = 10**,
    - The priority of P7 is greater than P5, so we execute P7 for 1 ms.
    - Remaining B.T for P7 = 1-1 = 0 ms.
    - Here Process P7 completes its execution.
- At time **t = 11**,
    - Now we take the process which is having the highest priority.
    - Here we find P5 is having the highest priority & execute P5 completely
    - Remaining B.T of P5 = 1-1 = 0 ms.
    - Here Process P5 completes its execution.
- At time **t = 12**,
    - Now we take the process which is having the highest priority.
    - Here we find P1 is having the highest priority & execute P1 completely
    - Remaining B.T of P1 = 3-3 = 0 ms.
    - Here Process P1 completes its execution.
- At time **t = 15**,
    - Now we take the process which is having the highest priority.
    - Here we find P2 is having the highest priority & execute P2 completely
    - Remaining B.T of P2 = 2-2 = 0 ms.
    - Here Process P2 completes its execution.
- At time **t = 17**,
    - Now we take the process which is having the highest priority.
    - Here we find P3 is having the highest priority & execute P3 completely
    - Remaining B.T of P3 = 4-4 = 0 ms.
    - Here Process P3 completes its execution.
- At time **t = 21**,
    - Now we take the process which is having the highest priority.
    - Here we find P4 is having the highest priority & execute P4 completely
    - Remaining B.T of P4 = 1-1 = 0 ms.
    - Here Process P4 completes its execution.
- At time **t = 22**,
    - Now we take the process which is having the highest priority.
    - Here we find P6 is having the highest priority & execute P6 completely
    - Remaining B.T of P6 = 5-5 = 0 ms.
    - Here Process P6 completes its execution.

Gantt chart:





Process	A.T	Priority	B.T	C.T	T.A.T	W.T	R.T
P1	0	3	8	15	15	7	0
P2	1	4	2	17	16	14	14
P3	3	4	4	21	18	14	14
P4	4	5	1	22	18	17	17
P5	5	2	6	12	7	1	0
P6	6	6(L)	5	27	21	16	16
P7	10	1(H)	1	11	1	0	0

Here, ***H – Higher Priority, L – Least Priority***

Drawbacks of Preemptive Priority Scheduling Algorithm:

One of the most common drawbacks of the Preemptive priority CPU scheduling algorithm is the **Starvation Problem**. This is the problem in which a process has to wait for a longer amount of time to get scheduled into the CPU. This condition is called the starvation problem.

Example: In Example 2, we can see that process P1 is having Priority 3 and we have pre-empted the process P1 and allocated the CPU to P5. Here we are only having 7 processes.

Now, if suppose we have many processes whose priority is higher than P1, then P1 needs to wait for a longer time for the other process to be pre-empted and scheduled by the CPU. This condition is called a starvation problem.

Solution: The solution to this Starvation problem is ***Ageing***. This can be done by decrementing the priority number by a certain number of a particular process which is waiting for a longer period of time after a certain interval.

After every 3 units of time, all the processes which are in waiting for the state, the priority of those processes will be decreased by 2, So, if there is a process P1 which is having priority 5, after waiting for 3 units of time its priority will be decreased from 5 to 3 so that if there is any process P2 which is having priority as 4 then that process P2 will wait and P1 will be scheduled and executed.

Implementation using C++:

```
#include<iostream>
#include<algorithm>
using namespace std;

struct node{
    char pname;
    int btime;
    int atime;
    int priority;
    int restime=0;
    int ctime=0;
    int wtime=0;
}a[1000],b[1000],c[1000];

void insert(int n){
    int i;
    for(i=0;i<n;i++){
        cin>>a[i].pname;
        cin>>a[i].priority;
        cin>>a[i].atime;
        cin>>a[i].btime;
        a[i].wtime=-a[i].atime+1;
    }
}

bool btimeSort(node a,node b){
    return a.btime < b.btime;
}

bool atimeSort(node a,node b){
    return a.atime < b.atime;
}
bool prioritySort(node a,node b){
    return a.priority < b.priority;
}
int k=0,f=0,r=0;
void disp(int nop,int qt){
    int n=nop,q;
    sort(a,a+n,atimeSort);
    int ttime=0,i;
    int j,tArray[n];
    int alltime=0;
    bool moveLast=false;
    for(i=0;i<n;i++){
        alltime+=a[i].btime;
    }
```

```
alltime+=a[0].atime;
for(i=0;ttime<=alltime;){
    j=i;
    while(a[j].atime<=ttime&&j!=n){
        b[r]=a[j];
        j++;
        r++;
    }
    if(r==f){
        c[k].pname='i';
        c[k].btime=a[j].atime-ttime;
        c[k].atime=ttime;
        ttime+=c[k].btime;
        k++;
        continue;
    }
    i=j;
    if(moveLast==true){
        sort(b+f,b+r,prioritySort);
        // b[r]=b[f];
        // f++;
        // r++;
    }

    j=f;
    if(b[j].btime>qt){
        c[k]=b[j];
        c[k].btime=qt;
        k++;
        b[j].btime=b[j].btime-qt;
        ttime+=qt;
        moveLast=true;
        for(q=0;q<n;q++){
            if(b[j].pname!=a[q].pname){
                a[q].wtime+=qt;
            }
        }
    }
    else{
        c[k]=b[j];
        k++;
        f++;
        ttime+=b[j].btime;
        moveLast=false;
        for(q=0;q<n;q++){
            if(b[j].pname!=a[q].pname){
                a[q].wtime+=b[j].btime;
            }
        }
    }
    if(f==r&&i>=n)
    break;
}
tArray[i]=ttime;
ttime+=a[i].btime;
for(i=0;i<k-1;i++){
    if(c[i].pname==c[i+1].pname){
        c[i].btime+=c[i+1].btime;
        for(j=i+1;j<k-1;j++)
            c[j]=c[j+1];
        k--;
        i--;
    }
}

int rtime=0;
for(j=0;j<n;j++){
    rtime=0;
    for(i=0;i<k;i++){
        if(c[i].pname==a[j].pname){
            a[j].restime=rtime;
            break;
        }
        rtime+=c[i].btime;
    }
}

float averageWaitingTime=0;
float averageResponseTime=0;
float averageTAT=0;

cout<<"\nGantt Chart\n";
rtime=0;
for (i=0; i<k; i++){
    if(i!=k)
        cout<<"|  "<<'P'<< c[i].pname << "    ";
    rtime+=c[i].btime;
    for(j=0;j<n;j++){
        if(a[j].pname==c[i].pname)
            a[j].ctime=rtime;
    }
}
cout<<"\n";
rtime=0;
for (i=0; i<k+1; i++){
    cout << rtime << "\t";
```

```
        tArray[i]=rtime;
        rtime+=c[i].btime;
    }

    cout<<"\n";
    cout<<"\n";
    cout<<"P.Name Priority AT\tBT\tCT\tTAT\tWT\tRT\n";
    for (i=0; i<nop&&a[i].pname!='i'; i++){
        if(a[i].pname=='\0')
            break;
        cout <<'P'<< a[i].pname << "\t";
        cout << a[i].priority << "\t";
        cout << a[i].atime << "\t";
        cout << a[i].btime << "\t";
        cout << a[i].ctime << "\t";
        cout << a[i].wtime+a[i].ctime-rtime+a[i].btime << "\t";
        averageTAT+=a[i].wtime+a[i].ctime-rtime+a[i].btime;
        cout << a[i].wtime+a[i].ctime-rtime << "\t";
        averageWaitingTime+=a[i].wtime+a[i].ctime-rtime;
        cout << a[i].restime-a[i].atime << "\t";
        averageResponseTime+=a[i].restime-a[i].atime;
        cout <<"\n";
    }

    cout<<"Average Response time: "<<(float)averageResponseTime/(float)n<<endl;
    cout<<"Average Waiting time: "<<(float)averageWaitingTime/(float)n<<endl;
    cout<<"Average TA time: "<<(float)averageTAT/(float)n<<endl;

}

int main(){
    int nop,choice,i,qt;
    cout<<"Enter number of processes\n";
    cin>>nop;
    cout<<"Enter process, priority, AT, BT\n";
    insert(nop);
    disp(nop,1);
    return 0;
}
```

Output:

```
Enter number of processes
5
Enter process, priority, AT, BT
1 6 0 5
```

```
2 4 1 2

3 3 2 4

4 1 3 1

5 2 4 7

Gantt Chart

| P1 | P2 | P3 | P4 | P3 | P5 | P3 | P2 | P1

0 1 2 3 4 5 12 14 15 19

P.Name Priority AT BT CT TAT WT RT

P1 6 0 5 19 19 14 0

P2 4 1 2 15 14 12 0

P3 3 2 4 14 12 8 0

P4 1 3 1 4 1 0 0

P5 2 4 7 12 8 1 1

Average Response time: 0.2

Average Waiting time: 7

Average TA time: 10.8
```

Conclusion: Successfully implemented Preemptive Scheduling using C++.

Assignment 8: Implementation of non-Preemptive scheduling.

Aim: To implement non-Preemptive scheduling.

Objective: To learn to implement non-Preemptive scheduling.

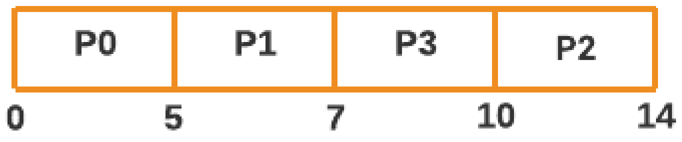
Theory:

If a resource is allocated to a process under non-preemptive scheduling, that resource will not be released until the process is completed. Other tasks in the ready queue will have to wait their time, and it will not get the CPU forcefully. After a process has been assigned to it, it will hold the CPU until it has completed its execution or until an I/O action is required. When a non-preemptive process with a long CPU burst time runs, the other process must wait for an extended period, increasing the average waiting time in the ready queue. Non-preemptive scheduling, on the other hand, contains no overhead when switching processes from the ready queue to the CPU. The execution process isn't even interrupted by a higher-priority task, implying that the scheduling is rigorous.
Example:
Let's look at the above preemptive scheduling problem we have solved and see how we can handle it in a non-preemptive method. In contrast, the same four processes are P0, P1, P2, and P3, with the same arrival time and CPU burst time.
So, by using a Non-preemptive scheduler, the Gantt chart would look like-

Process	Arrival Time	CPU Burst time
P0	**0**	**5**

Process	Arrival Time	CPU Burst time
P1	1	2
P2	3	4
P3	4	3

Gantt chart:



- The processor is allocated to process P0 since it arrives at 0 and takes five milliseconds to complete.
- Meanwhile, all of the processes, P1, P2, and P3, arrive in the ready queue. All operations, however, must wait until Process P0 finishes their CPU burst period.
- Process P0 will complete its execution at time = 5ms. The burst times are compared for the rest of the processes in the ready queue. Process P1 is executed because it has a shorter burst duration than other processes.
- And similarly, following P1, P3, and then P2 will complete its execution.

So, let's calculate the average waiting time for the non-preemptive method.

Process	Arrival Time	CPU Burst time	Completion Time	Turn Around Time	Waiting Time
P0	0	5	5	5	0
P1	1	2	7	6	4
P2	3	4	14	11	7
P3	4	3	10	6	3

Average waiting time = [WT(P0) + WT(P1) + WT(P2) + WT(P3)] / 4
= [0 + 4 + 7 + 3] / 4
= 3.5 ms.
So, the average waiting time for the preemptive method is less than the non-preemptive method which can further conclude that the preemptive method is more efficient in terms of time efficiency for the CPU.

Advantages of Non-Preemptive Scheduling

- Low overhead in terms of scheduling is being offered.
- It has a tendency to have a high throughput.
- The approach is relatively easier to implement than other scheduling methods.
- Very few computational resources are required in Non-preemptive scheduling.

Disadvantages of Non-Preemptive Scheduling

- The response time is relatively much slower and inefficient.
- It can make a priority and real-time scheduling challenging.
- It can result in starvation, particularly for those real-time tasks.
- The Bugs due to this scheduling can cause a system to become unresponsive.

Implementation using C++:

```
#include<iostream>
#include<algorithm>
using namespace std;

struct node{
   char pname[50];
   int btime;
   int atime;
}a[50];

void insert(int n){
   int i;
   for(i=0;i<n;i++){
      cin>>a[i].pname;
      cin>>a[i].atime;
      cin>>a[i].btime;
   }
}

bool btimeSort(node a,node b){
   return a.btime < b.btime;
}

bool atimeSort(node a,node b){
   return a.atime < b.atime;
}

void disp(int n){
   sort(a,a+n,btimeSort);
   sort(a,a+n,atimeSort);
   int ttime=0,i;
   int j,tArray[n];
   for(i=0;i<n;i++){
      j=i;
      while(a[j].atime<=ttime&&j!=n){
         j++;
      }
      sort(a+i,a+j,btimeSort);
      tArray[i]=ttime;
```

```
        ttime+=a[i].btime;
    }
    tArray[i] = ttime;

    float averageWaitingTime=0;
    float averageResponseTime=0;
    float averageTAT=0;
    cout<<"\n";
    cout<<"P.Name  AT\tBT\tCT\tTAT\tWT\tRT\n";
    for (i=0; i<n; i++){
        cout << a[i].pname << "\t";
        cout << a[i].atime << "\t";
        cout << a[i].btime << "\t";
        cout << tArray[i+1] << "\t";
        cout << tArray[i]-a[i].atime+a[i].btime << "\t";
        averageTAT+=tArray[i]-a[i].atime+a[i].btime;
        cout << tArray[i]-a[i].atime << "\t";
        averageWaitingTime+=tArray[i]-a[i].atime;
        cout << tArray[i]-a[i].atime << "\t";
        averageResponseTime+=tArray[i]-a[i].atime;
        cout <<"\n";
    }
    cout<<"\n";
    cout<<"\nGantt Chart\n";
    for (i=0; i<n; i++){
        cout <<"|   "<< a[i].pname << "   ";
    }
    cout<<"\n";
    for (i=0; i<n+1; i++){
        cout << tArray[i] << "\t";
    }
    cout<<"\n";
    cout<<"Average Response time: "<<(float)averageResponseTime/(float)n<<endl;
    cout<<"Average Waiting time: "<<(float)averageWaitingTime/(float)n<<endl;
    cout<<"Average TA time: "<<(float)averageTAT/(float)n<<endl;
}

int main(){
    int nop, choice, i;
    cout<<"Enter number of processes\n";
    cin>>nop;
    insert(nop);
    disp(nop);
    return 0;
}
```

Output:

```
Enter number of processes
5
1 0 5
2 1 2
3 2 4
4 3 1
5 4 7
P.Name AT BT CT TAT WT RT
1 0 5 5 5 0 0
4 3 1 6 3 2 2
2 1 2 8 7 5 5
3 2 4 12 10 6 6
5 4 7 19 15 8 8
Gantt Chart
| 1 | 4 | 2 | 3 | 5
0 5 6 8 12 19
Average Response time: 4.2
Average Waiting time: 4.2
Average TA time: 8
```

Conclusion: Successfully implemented non-Preemptive scheduling using C++.

Assignment 9: Implementation of Clock Driven Scheduling.

Aim: To implement Clock Driven Scheduling.

Objective: To learn to implement Clock Driven Scheduling.

Theory:

Clock - Driven Scheduling:

The scheduling in which the scheduling points are determined by the interrupts received from a clock, It's known as Clock-driven Scheduling.

Clock-driven scheduling handles which task is to be processed next is dependent at clock interrupt point. In a clock-driven scheduler, the scheduling points are defined at the time instants marked by interrupts generated by a periodic timer.

enter image description here

When workload is mostly periodic and the schedule is cyclic, timing constraints can be checked and enforced at each frame boundary

That is, the scheduler pre-determines which task will run when. Therefore, these schedulers incur very little run Time overhead.

However, a prominent shortcoming of this class of schedulers is that they can not satisfactorily handle aperiodic and sporadic tasks since the exact Time of occurrence of these tasks can not be predicted.

Advantages:

The important advantage is conceptual simplicity.

Time triggered System Based on clock driven scheduling is easy to validate, test and certify.

Clock driven scheduling paradigm are time triggered. in this systems interrupts to external events are queued and polled periodically.

Disadvantages:

The system based on clock driven approach is Brittle (soft).

The release time of all jobs must be fixed.

The pure clock driven approach is not suitable for many systems that contain both hard and soft real time applications.

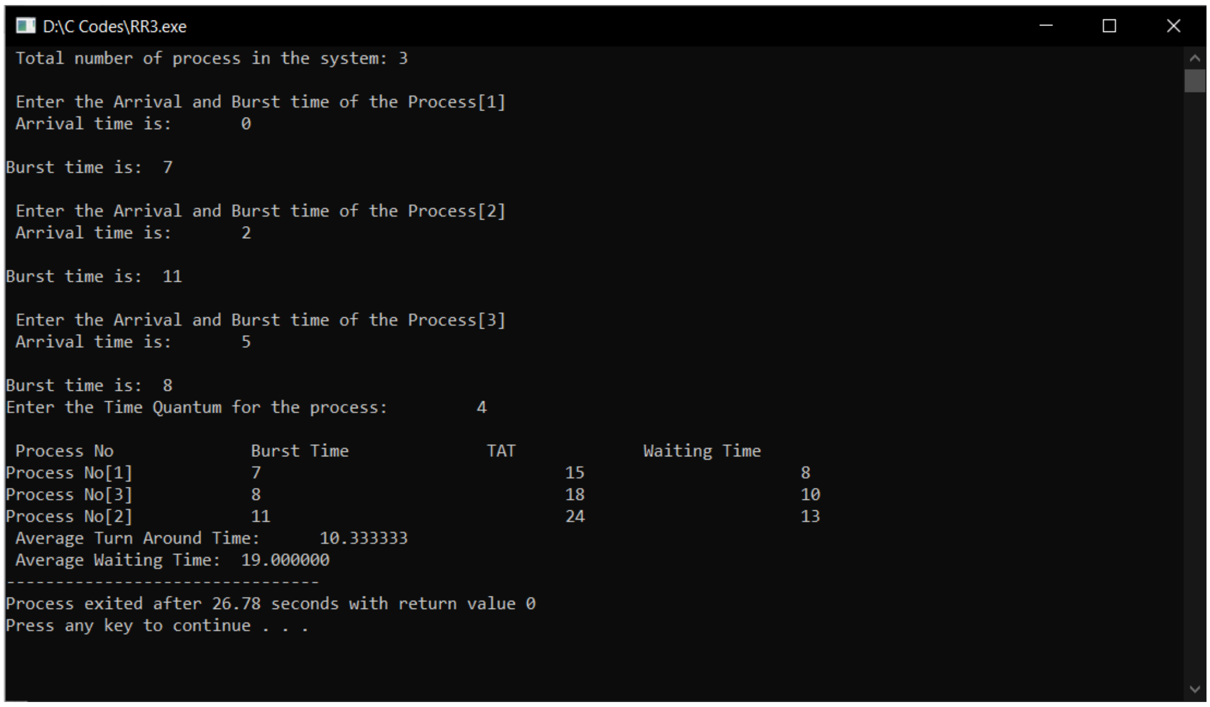
In this system all combinations of periodic task that might execute at same time.

Implementation:

```
#include<stdio.h>
#include<conio.h>

int main()
{
    int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];
    float avg_wt, avg_tat;
    printf(" Total number of process in the system: ");
    scanf("%d", &NOP);
    y = NOP;

for(i=0; i<NOP; i++)
{
printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);
printf(" Arrival time is: \t");
scanf("%d", &at[i]);
printf(" \nBurst time is: \t");
scanf("%d", &bt[i]);
temp[i] = bt[i];
}

printf("Enter the Time Quantum for the process: \t");
scanf("%d", &quant);
printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");
for(sum=0, i = 0; y!=0; )
{
```

```
if(temp[i] <= quant && temp[i] > 0)
{
    sum = sum + temp[i];
    temp[i] = 0;
    count=1;
    }
    else if(temp[i] > 0)
    {
        temp[i] = temp[i] - quant;
        sum = sum + quant;
    }
    if(temp[i]==0 && count==1)
    {
        y--;
        printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);
        wt = wt+sum-at[i]-bt[i];
        tat = tat+sum-at[i];
        count =0;
    }
    if(i==NOP-1)
    {
        i=0;
    }
    else if(at[i+1]<=sum)
    {
        i++;
    }
    else
    {
        i=0;
    }
}

avg_wt = wt * 1.0/NOP;
avg_tat = tat * 1.0/NOP;
printf("\n Average Turn Around Time: \t%f", avg_wt);
printf("\n Average Waiting Time: \t%f", avg_tat);
return 0;
}
```

Output:



```
 Total number of process in the system: 3

 Enter the Arrival and Burst time of the Process[1]
 Arrival time is:       0

Burst time is:  7

 Enter the Arrival and Burst time of the Process[2]
 Arrival time is:       2

Burst time is:  11

 Enter the Arrival and Burst time of the Process[3]
 Arrival time is:       5

Burst time is:  8
Enter the Time Quantum for the process:         4

 Process No             Burst Time              TAT             Waiting Time
Process No[1]           7                               15                      8
Process No[3]           8                               18                      10
Process No[2]           11                              24                      13
 Average Turn Around Time:      10.333333
 Average Waiting Time:  19.000000
--------------------------------
Process exited after 26.78 seconds with return value 0
Press any key to continue . . .
```

Conclusion: Successfully implemented Clock Driven Scheduling.

Assignment 10: Implementation of EDF Scheduling.

Aim: To implement EDF Scheduling.

Objective: To learn to implement EDF Scheduling.

Theory:

Earliest deadline first (EDF) is dynamic priority scheduling algorithm for real time embedded systems. Earliest deadline first selects a task according to its deadline such that a task with earliest deadline has higher priority than others. It means priority of a task is inversely proportional to its absolute deadline. Since absolute deadline of a task depends on the current instant of time so every instant is a scheduling event in EDF as deadline of task changes with time. A task which has a higher priority due to earliest deadline at one instant it may have low priority at next instant due to early deadline of another task. EDF typically executes in pre-emptive mode i.e., currently executing task is pre-empted whenever another task with earliest deadline becomes active.

EARLIEST DEADLINE FIRST (EDF) SCHEDULING ALGORITHM

EDF is an optimal algorithm which means if a task set is feasible then it is surely scheduled by EDF. Another thing is that EDF does not specifically take any assumption on periodicity of tasks so it is independent of Period of task and therefore can be used to schedule aperiodic tasks as well. If two tasks have same absolute deadline choose one of them randomly. you may also like to read

- **Scheduling Algorithms in RTOS**
- **Rate monotonic scheduling Algorithm**

Example of EARLIEST DEADLINE FIRST (EDF) SCHEDULING ALGORITHM

An example of EDF is given below for task set of table-2.

Task	**Release time(ri)**	**Execution Time(Ci)**	**Deadline (Di)**	**Time Period(Ti)**
T1	0	1	4	4
T2	0	2	6	6
T3	0	3	8	8

Table 2. Task set

U= 1/4 +2/6 +3/8 = 0.25 + 0.333 +0.375 = 0.95 = 95%
As processor utilization is less than 1 or 100% so task set is surely schedulable by EDF.

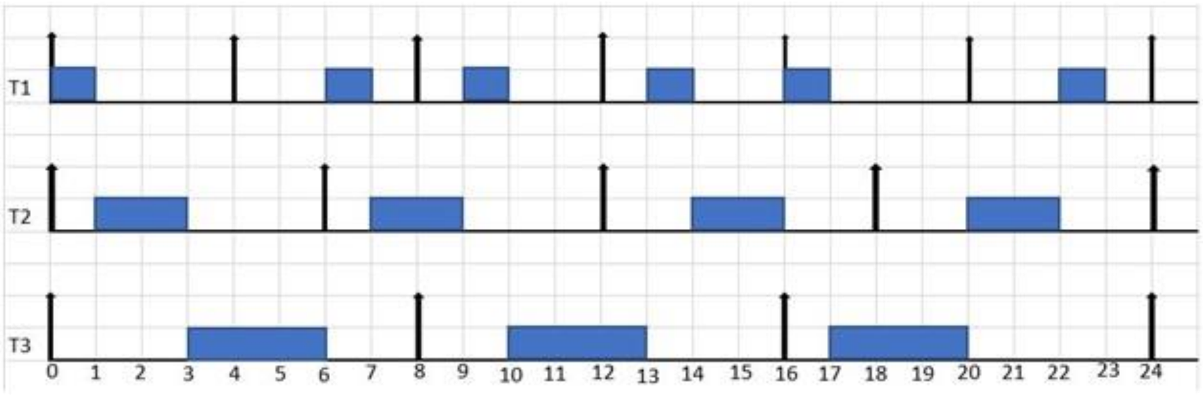


Figure 2. Earliest deadline first scheduling of task set in Table -2

1. At t=0 all the tasks are released, but priorities are decided according to their absolute deadlines so T1 has higher priority as its deadline is 4 earlier than T2 whose deadline is 6 and T3 whose deadline is 8, that's why it executes first.
2. At t=1 again absolute deadlines are compared and T2 has shorter deadline so it executes and after that T3 starts execution but at t=4 T1 comes in the system and deadlines are compared, at this instant both T1 and T3 has same deadlines so ties are broken randomly so we continue to execute T3.
3. At t=6 T2 is released, now deadline of T1 is earliest than T2 so it starts execution and after that T2 begins to execute. At t=8 again T1 and T2 have same deadlines i.e. t=16, so ties are broken randomly an T2 continues its execution and then T1 completes. Now at t=12 T1 and T2 come in the system simultaneously so by comparing absolute deadlines, T1 and T2 has same deadlines therefore ties broken randomly and we continue to execute T3.
4. At t=13 T1 begins it execution and ends at t=14. Now T2 is the only task in the system so it completes it execution.
5. At t=16 T1 and T2 are released together, priorities are decided according to absolute deadlines so T1 execute first as its deadline is t=20 and T3's deadline is t=24.After T1 completion T3 starts and reaches at t=17 where T2 comes in the system now by deadline comparison both have same deadline t=24 so ties broken randomly ant we T continue to execute T3.
6. At t=20 both T1 and T2 are in the system and both have same deadline t=24 so again ties broken randomly and T2 executes. After that T1 completes it execution. In the same way system continue to run without any problem by following EDF algorithm.

Transient Over Load Condition & Domino Effect in Earliest deadline first

Transient over load is a short time over load on the processor. Transient overload condition occurs when the computation time demand of a task set at an instant exceeds the processor timing capacity available at that instant. Due to transient over load tasks miss their deadline. This transient over load may occur due many reasons such as changes in the environment, simultaneous arrival of asynchronous jobs, system exception. In real time operating systems under EDF, whenever a task in Transient overload condition miss its deadline and as result each of other tasks start missing their deadlines one after the other in sequence, such an effect is called domino effect. It jeopardizes the behavior of the whole system. An example of such condition is given below.

Task	**Release time(ri)**	**Execution Time(Ci)**	**Deadline (Di)**	**Period(Ti)**
T1	0	2	5	5
T2	0	2	6	6
T3	0	2	7	7
T4	0	2	8	8

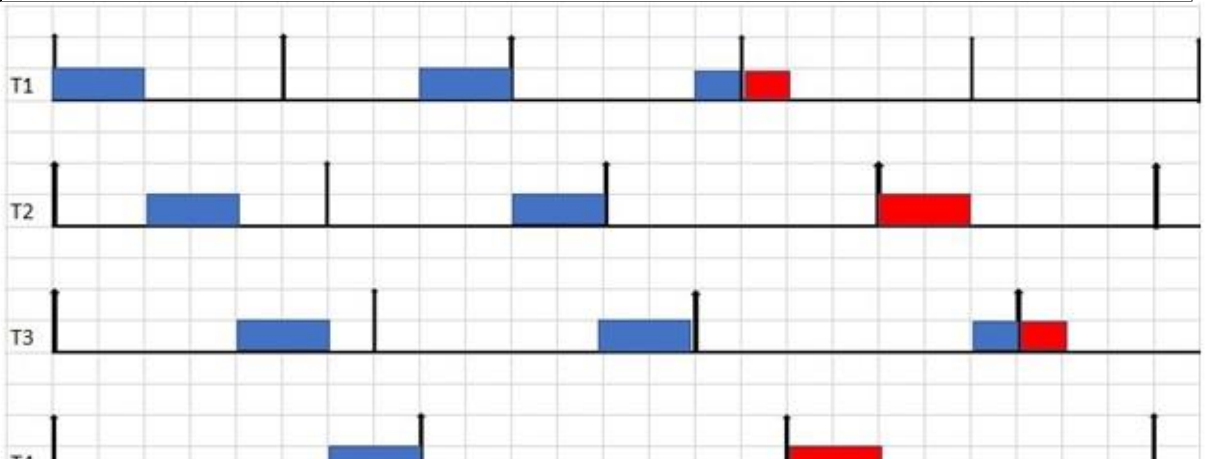


Figure 3. Domino effect under Earliest deadline first

As in the above figure at t=15 T1 misses it deadline and after that at t=16 T4 is missing its deadline then T2 and finally T3 so the whole system is collapsed. It is clearly proved that EDF has a shortcoming due to domino effect and as a result critical tasks may miss their deadlines. The solution of this problem is another scheduling algorithm that is least laxity first (LLF). It is an optimal scheduling algorithm. Demand bound function ad Demand bound analysis are also used for schedualability analysis of given set of tasks.

Advantages of EDF over rate monotonic

- No need to define priorities offline
- It has less context switching than rate monotonic
- It utilize the processor maximum up to 100% utilization factor as compared to rate monotonic

Disadvantages of EDF over rate monotonic

- It is less predictable. Because response time of tasks are variable and response time of tasks are constant in case of rate monotonic or fixed priority algorithm.
- EDF provided less control over the execution
- It has high overheads

Implementation in C:

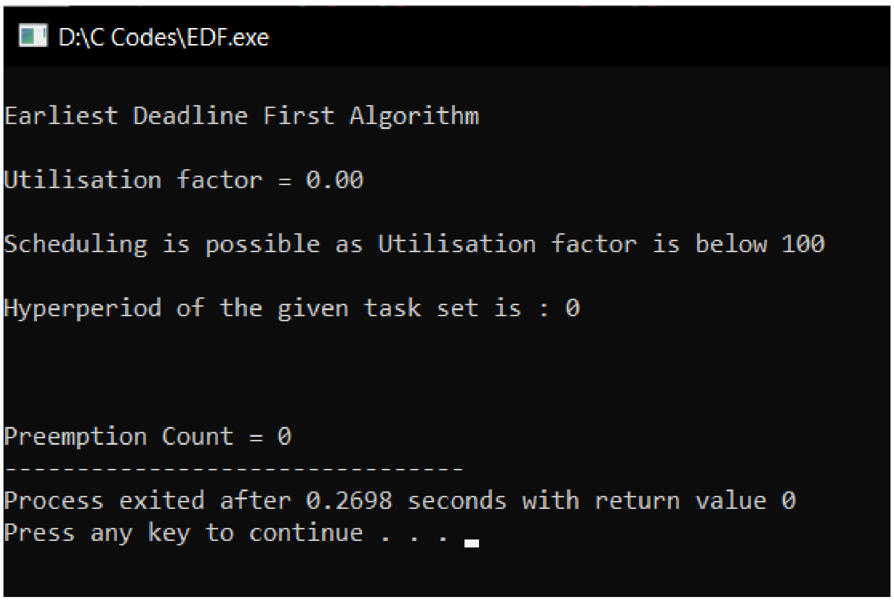
```
//Earliest Deadline First (EDF) Scheduling in C

#include<stdio.h>

#include<string.h>

int gcd(int a,int b){

if(b==0)

return a;

else

gcd(b,a%b);}

int lcm(int a,int b){

return((a*b)/gcd(a,b));}

int hyperperiod(float period[],int n){

int k=period[0];

n--;

while(n>=1){

k=lcm(k,period[n--]);}

return k;}

int edf(float *period,int n,int t,float *deadline){

int i,small=10000.0f,smallindex=0;
```

```
for(int i=0;i<n;i++){

if(period[i]<small&&(period[i]-t)<=deadline[i]){

small=period[i];

smallindex=i;}}

if(small==10000.0f)

return -1;

return smallindex;}

int main()

{

int i,n,c,d,k,j,nexttime=0,time=0,task,preemption_count;

float
exec[20],period[20],individual_util[20],flag[20],release[20],deadline[20],instance[
20],ex[20],responsemax[20],responsemin[20],tempmax;

float util=0;

printf("\nEarliest Deadline First Algorithm\n");

FILE *read;

read=fopen("Sampledata.docx","r");          // Sampledata

fscanf(read,"%d ",&n);

for(i=0;i<n;i++)

{

fscanf(read,"%f ",&release[i]);

fscanf(read,"%f ",&period[i]);

fscanf(read,"%f ",&exec[i]);

fscanf(read,"%f ",&deadline[i]);

}

fclose(read);

for(i=0;i<n;i++)
```

```
{
individual_util[i]=exec[i]/period[i];
util+=individual_util[i];
responsemax[i]=exec[i];
deadline[i]=period[i];
instance[i]=0.0f;
}
util=util*100;
if(util>100)
printf("\n Utilisation factor = %0.2f \n\nScheduling is not possible as Utilisation factor is above 100 \n",util);
else
{
printf("\nUtilisation factor = %0.2f \n\nScheduling is possible as Utilisation factor is below 100 \n ",util);
printf("\nHyperperiod of the given task set is : %d\n\n",k=hyperperiod(period , n));
c=0;
while(time<k)
{
nexttime=time+1;
task = edf(period,n,time,deadline);
if(task==-1)
{
printf("-");
time++;
continue;
}
```

```
instance[task]++;
printf("T%d ",task);
ex[c++]=task;
if(instance[task]==exec[task])
{
tempmax=nexttime-(period[task]-deadline[task]);
if(instance[task]<tempmax)
{
responsemax[task]=tempmax;
}
else
{
responsemin[task]=instance[task];
}
if(deadline[task]==k)
{
responsemin[task]=responsemax[task];
}
period[task]+=deadline[task];
instance[task]=0.0f;
}
time++;
}
for(i=0;i<n;i++)
{
printf("\n\nMaximum Response time of Task %d = %f",i,responsemax[i]);
printf("\n\nMinimum Response time of Task %d = %f",i,responsemin[i]);
}
preemption_count=0;
for(i=0;i<k;i=j)
{
flag[i]=1;
d=ex[i];
for(j=i+1;d==ex[j];j++)
flag[d]++;
if(flag[d]==exec[d])
flag[d]=1;
else
{
flag[d]++;
preemption_count++;
}
}
printf("\n\nPreemption Count = %d",preemption_count);
}
return 0;
}
```

Output:



```
D:\C Codes\EDF.exe

Earliest Deadline First Algorithm

Utilisation factor = 0.00

Scheduling is possible as Utilisation factor is below 100

Hyperperiod of the given task set is : 0



Preemption Count = 0
--------------------------------
Process exited after 0.2698 seconds with return value 0
Press any key to continue . . .
```

Conclusion: Successfully implemented EDF Scheduling.