Real-Time & Linux

Real-Time Systems

- ***Hard real-time systems***: A hard real-time system needs a guaranteed worst case response time. The entire system including OS, applications, HW, and so on must be designed to guarantee that response requirements are met. It doesn’t matter what the timings requirements are to be hard real-time (microseconds, milliseconds, etc.), just that they must be met every time. Failure to do so can lead to drastic consequences such as loss of life. Some examples of hard real-time systems include defense systems, flight and vehicle control systems, satellite systems, data acquisition systems, medical instrumentation, controlling space shuttles or nuclear reactors, gaming systems, and so on.

Real-Time Systems

- ***Soft real-time systems***: In soft real-time systems it is not necessary for system success that every time constraint be met. In the above DVD player example, if the decoder is not able to meet the timing requirement once in an hour, it’s ok. But frequent deadline misses by the decoder in a short period of time can leave an impression that the system has failed. Some examples are multimedia applications, VoIP, CE devices, audio or video streaming, and so on.

Real-Time Operating System

- POSIX 1003.1b defines real-time for operating systems as the ability of the operating system to provide a required level of service in a bounded response time.

- Multitasking/multithreading: An RTOS should support multitasking and multithreading.
- Priorities: The tasks should have priorities. Critical and time-bound functionalities should be processed by tasks having higher priorities.
- Priority inheritance: An RTOS should have a mechanism to support priority inheritance.

Real-Time Operating System

- Preemption: An RTOS should be preemptive; that is, when a task of higher priority is ready to run, it should preempt a lower-priority task.
- Interrupt latency: Interrupt latency is the time taken between a hardware interrupt being raised and the interrupt handler being called. An RTOS should have predictable interrupt latencies and preferably be as small as possible.
- Scheduler latency: This is the time difference when a task becomes runnable and actually starts running. An RTOS should have deterministic scheduler latencies.

Real-Time Operating System

- Interprocess communication and synchronization: The most popular form of communication between tasks in an embedded system is message passing. An RTOS should offer a constant time message-passing mechanism. Also it should provide semaphores and mutexes for synchronization purposes.
- Dynamic memory allocation: An RTOS should provide fixed-time memory allocation routines for applications.

Linux and Real-Time

- Linux evolved as a general-purpose operating system. The main reasons stated for the non–real-time nature of Linux were:
  - High interrupt latency
  - High scheduler latency due to nonpreemptive nature of the kernel
  - Various OS services such as IPC mechanisms, memory allocation, and the like do not have deterministic timing behavior.
  - Other features such as virtual memory and system calls also make Linux undeterministic in its response.

Linux and Real-Time

- The key difference between any general-purpose operating system like Linux and a hard real-time OS is the deterministic timing behavior of all the OS services in an RTOS. By deterministic timing we mean that any latency involved or time taken by any OS service should be well bounded.

- ***kernel response time*** is the amount of time that elapses from when the interrupt is raised to when the task that was waiting for I/O to complete runs. As you can see from the example there are four components to the kernel response time.
  - Interrupt latency: Interrupt latency is the time difference between a device raising an interrupt and the corresponding handler being called.
  - ISR duration: the time needed by an interrupt handler to execute.
  - Scheduler latency: Scheduler latency is the amount of time that elapses between the interrupt service routine completing and the scheduling function being run.
  - Scheduler duration: This is the time taken by the scheduler function to select the next task to run and context switch to it.

Interrupt Latency

- interrupt latency is one of the major factors contributing to nondeterministic system response times. Some of the common causes for high-interrupt latency.

ISR Duration

- ISR duration is the time taken by an interrupt handler to execute and it is under the control of the ISR writer. However nondeterminism could arise if an ISR has a softirq component also. In order to have less interrupt latency, an interrupt handler needs to do minimal work (such as copying some IO buffers to the system RAM) and the rest of the work (such as processing of the IO data, waking up tasks) should be done outside the interrupt handler. So an interrupt handler has been split into two portions: the top half that does the minimal job and the softirq that does the rest of the processing. The latency involved in softirq processing is unbounded.

ISR Duration

- The following latencies are involved during softirq processing.
  - A softirq runs with interrupts enabled and can be interrupted by a hard IRQ (except at some critical sections).
  - A softirq can also be executed in the context of a kernel daemon ksoftirqd, which is a non–real-time thread.
- Thus you should make sure that the ISR of your real-time device does not have any softirq component and all the work should be performed in the top half only.

Scheduler Latency

- Among all the latencies discussed, scheduler latency is the major contributor to the increased kernel response time. Some of the reasons for large scheduler latencies in the earlier Linux 2.4 kernel are as follows.
  - *Nonpreemptive nature of the kernel*: Scheduling decisions are made by the kernel in the places such as return from interrupt or return from system call, and so on. However, if the current process is running in kernel mode (i.e., executing a system call), the decision is postponed until the process comes back to user mode. This means that a high-priority process cannot preempt a low-priority process if the latter is executing a system call. Thus, because of the nonpreemptive nature of kernel mode execution, scheduling latencies may vary from tens to hundreds of milliseconds depending on the duration of a system call.

Scheduler Latency

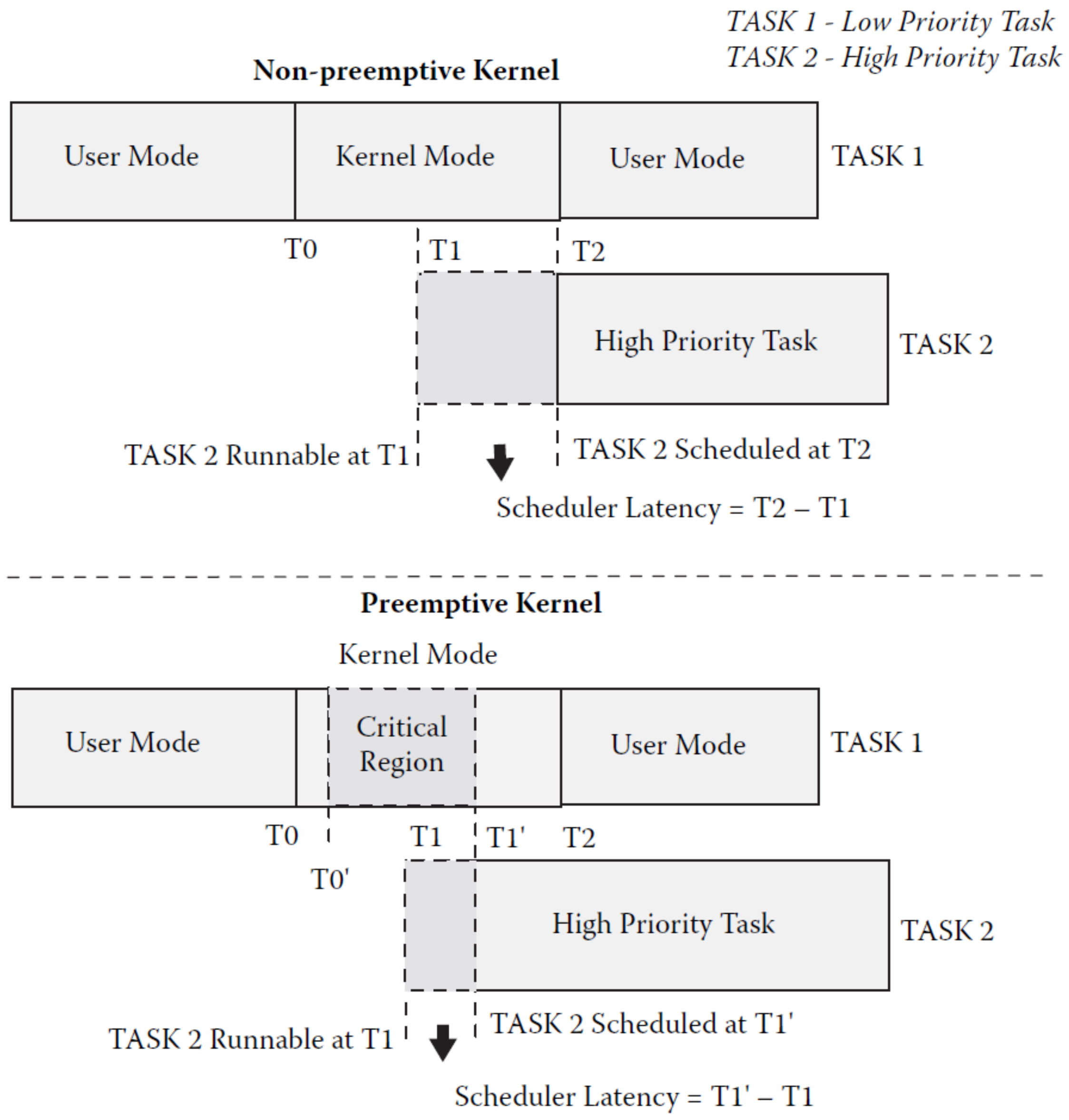
- *Interrupt disable times*: A scheduling decision is made as early as the return from the next timer interrupt. If the global interrupts are disabled for a long time, the timer interrupt is delayed thus increasing scheduling latency.

Scheduler Latency

- Kernel Preemption
  - *It was observed that it’s safe to preempt a process executing in the kernel mode if it is not in any critical section protected using spinlock. This property was exploited by embedded Linux vendor MontaVista and they introduced the kernel preemption patch. The patch was incorporated in the mainstream kernel during the 2.5 kernel development and is now maintained by Robert Love.*

Scheduler Latency

- Kernel Preemption

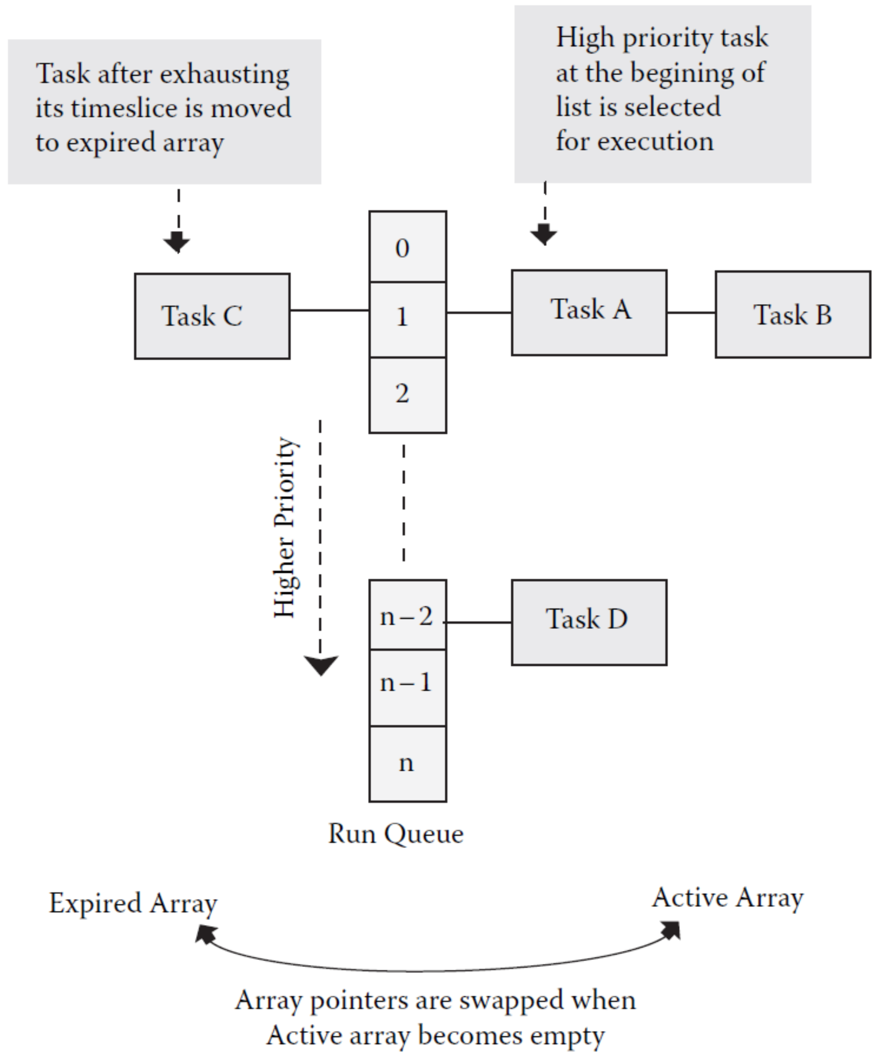


Scheduler Duration

- the scheduler duration is the time taken by the scheduler to select the next task for execution and context switch to it. The Linux scheduler like the rest of the system was written originally for the desktop and it remained almost unchanged except for the addition of the POSIX realtime capabilities. The major drawback of the scheduler was its nondeterministic behavior: The scheduler duration increased linearly with the number of tasks in the system, the reason being that all the tasks including real-time tasks are maintained in a single run queue and every time the scheduler was called it went through the entire run queue to find the highest-priority task.

Scheduler Duration

- Making the Scheduler Real-Time: The O(1) Scheduler


Task after exhausting its timeslice is moved to expired array
High priority task at the begining of list is selected for execution
0
1
2
n – 2
n – 1
n
Task C
Task A
Task B
Task D
Higher Priority
Run Queue
Expired Array
Active Array
Array pointers are swapped when Active array becomes empty

User-Space Real-Time

- The IEEE 1003.1b (or POSIX.1b) standard defines interfaces to support portability of applications with real-time requirements.
  - Fixed-priority scheduling with real-time scheduling classes
  - Memory locking
  - POSIX message queues
  - POSIX shared memory
  - Real-time signals
  - POSIX semaphores
  - POSIX clocks and timers
  - Asynchronous I/O (AIO)

Process Scheduling

- The scheduler for the 2.6 kernel. There are three basic parameters to define a real-time task on Linux:
  - Scheduling class
  - Process priority
  - Timeslice

Process Scheduling

- Scheduling Class
  - SCHED_FIFO: First-in first-out real-time scheduling policy. The scheduling algorithm does not use any timeslicing. A SCHED_FIFO process runs to completion unless it is blocked by an I/O request, preempted by a higherpriority process, or it voluntarily relinquishes the CPU. The following points should be noted.
    - A SCHED_FIFO process that has been preempted by another process of higher priority stays at the head of the list for its priority and will resume execution as soon as all processes of higher priority are blocked again.
    - When a SCHED_FIFO process is ready to run (e.g., after waking from a blocking operation), it will be inserted at the end of the list of its priority.
    - A call to sched_setscheduler or sched_setparam will put the SCHED_FIFO process at the start of the list. As a consequence, it may preempt the currently running process if its priority is the same as that of the running process.Timeslice

Process Scheduling

- Scheduling Class

  - SCHED_RR: Round-robin real-time scheduling policy. It’s similar to SCHED_FIFO with the only difference being that the SCHED_RR process is allowed to run for a maximum time quantum. If a SCHED_RR process exhausts its time quantum, it is put at the end of the list of its priority. A SCHED_RR process that has been preempted by a higher-priority process will complete the unexpired portion of its time quantum after resuming

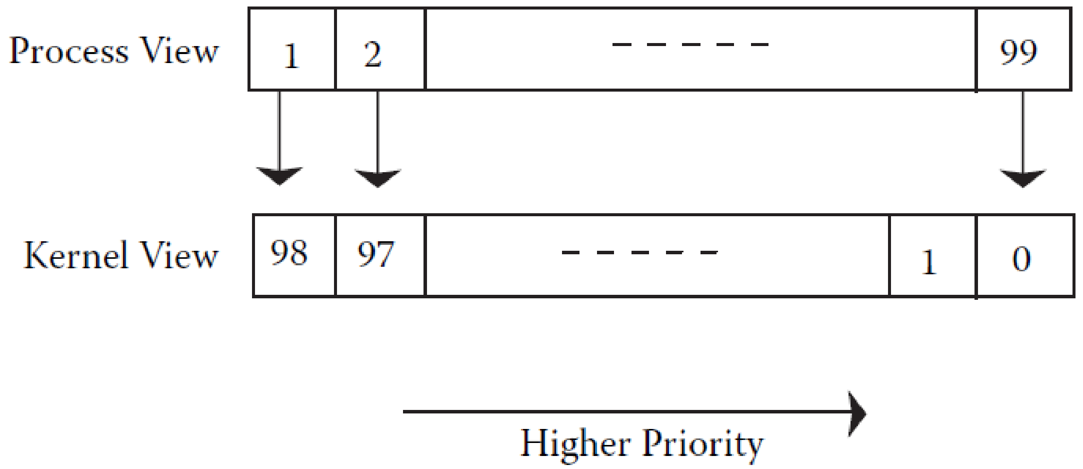
  - SCHED_OTHER: Standard Linux time-sharing scheduler for non–real-time processes.

Process Scheduling

- Priority

Table 7.1 User-Space Priority Range

Scheduling Class	*Priority Range*
`SCHED_OTHER`	0
`SCHED_FIFO`	1–99
`SCHED_RR`	1–99



Real-time task priority mapping.

Process Scheduling

- Timeslice
  - Timeslice is valid only for SCHED_RR processes. SCHED_FIFO processes can be thought of as having an infinite timeslice. Linux sets a minimum timeslice for a process to 10 msec, default timeslice to 100 msec, and maximum timeslice to 200 msec. Timeslices get refilled after they expire.
  - All SCHED_RR processes run at the default timeslice of 100 msec as they normally have nice 0.
  - A nice –20 SCHED_RR process will get a timeslice of 200 msec and a nice +19 SCHED_RR process will get a timeslice of 10 msec. Thus the nice value can be used to control timeslice allocation for SCHED_RR processes.
  - The lower the nice value (i.e., higher priority), the higher the timeslice is.

Process Scheduling

- POSIX.1b Scheduling Functions

Method	*Description*
`sched_getscheduler`	Get the scheduling class of a process.
`sched_setscheduler`	Set the scheduling class of a process.
`sched_getparam`	Get the priority of a process.
`sched_setparam`	Set the priority of a process.
`sched_get_priority_max`	Get the max allowed priority for a scheduling class.
`sched_get_priority_min`	Get the min allowed priority for a scheduling class.
`sched_rr_get_interval`	Get the current timeslice of the SCHED_RR process.
`sched_yield`	Yield execution to another process.

sched.c(1)

```
#include <sched.h>

int main(){
		struct sched_param param, new_param;

		/*
		 * A process starts with the default policy SCHED_OTHER unless
		 * it's spawned by a SCHED_RR or SCHED_FIFO process.
		 */
		printf("start policy = %d\n", sched_getscheduler(0));
		/* output -> start policy = 0 */

		/* 0 <- SCHED_OTHER, 1 <- SCHED_FIFO, 2 <- SCHED_RR */
```

sched.c(2)

```
		/*
		 * Create a SCHED_FIFO process running with average priority
		 */
		param.sched_priority = (sched_get_priority_min(SCHED_FIFO) +
sched_get_priority_max(SCHED_FIFO))/2;
		printf("max priority = %d, min priority = %d, my priority = %d\n",
				sched_get_priority_max(SCHED_FIFO),
sched_get_priority_min(SCHED_FIFO), param.sched_priority);
		/* output -> max priority = 99, min priority = 1, my priority = 50 */

		if (sched_setscheduler(0, SCHED_FIFO, &param) != 0){
				perror("sched_setscheduler failed\n");
				return;
		}

		/*
		 * perform time critical operation
		 */
```

sched.c(3)

```
         * Lets give some other RT thread / process a chance to run. Note that call to sched_yield will
         * put the current process at the end of its priority queue. If there are no other process in the queue
         * then the call will have no effect
         */
        sched_yield();

        /*
         * You can also change the priority at run time
         */
        param.sched_priority = sched_get_priority_max(SCHED_FIFO);
        if (sched_setparam(0, &param) != 0){
                perror("sched_setparam failed\n");
                return;
        }
        sched_getparam(0, &new_param);
        printf("I am running at priority %d\n", new_param.sched_priority);
        /* output -> I am running at priority 99 */

        return ;
}
```

sched_rr.c(1)

```
#include <sched.h>

int main(){
		struct sched_param param;
		struct timespec ts;
		param.sched_priority = sched_get_priority_max(SCHED_RR);

		/*
		 * need maximum timeslice
		 */
		nice(-20);
		sched_setscheduler(0, SCHED_RR, &param);
		sched_rr_get_interval(0, &ts);
		printf ("max timeslice = %d msec\n", ts.tv_nsec/1000000);
		/* output -> max timeslice = 199 msec */
```

sched_rr.c(2)

```
        /*
         * Need minimum timeslice. Also note the argument to nice represents 'increment' and not
         * absolute value. Thus we are doing nice(39) to make it running at nice priority +19
         */
        nice(39);
        sched_setscheduler(0, SCHED_RR, &param);
        sched_rr_get_interval(0, &ts);
        printf ("min timeslice = %d", ts.tv_nsec/1000000);
        /* output -> min timeslice = 9 msec */

        return ;
}
```