Real-Time Operating Systems

Important Notice:

This document does NOT contain full record of lecture material. In the classroom, these slides are accompanied by writings and drawings on the board, as well as verbal explanations. If you miss a class, it is your responsibility to consult with a colleague who attended the class to obtain as much information as possible about the presented material.

Material tested in a midterm and the final exam may include any material presented in class, regardless of whether it appears in these slides or not.

Digital Control Systems

Goals

- Identify what is different in the case of Digital control systems (DCS) and arrive at suitable mathematical models for analysis so that we can build on the existing knowledge of continuous control systems (CCS).
- If a stable CCS is transformed into a DCS, will it remain stable after conversion? If not, is there something similar to root locus for checking the stability of DCS?
- How can we convert CCS transfer functions into equivalent transfer functions for DCS?
- How can we convert differential equations to equivalent difference equations for implementation in software?

We shall revert to these topics later in the course

Review of Concepts

What is an operating system?

- Hardware cannot work without instructions
- How to get input from keyboard?
- How to control floppy drive?
- How to control hard disk?
- How to output something on screen?
- How to print a document?
- Write instructions (programs) for each function every time a user wants a job done, or
- All programs that are repeatedly used for such jobs can be put together and made available to

every user

- This collection of programs is operating system
- Is OS required?

Main roles of the operating system:

- Management or System Supervision (no direct user intervention)
  - Manages computer start-up
  - Manages resources: storage, main memory, cache, virtual memory
  - Manages program execution
  - Multitasking, multiprocessing, parallel processing, co-processing,

spooling

- File Management

- Services to Hardware
  - Drivers for various devices
- Services to Software
  - of file maintenance
  - of other software's interface with the hardware
  - of a user interface
- Security
  - Controls user access to directories and files (password etc)
  - Limits access of processes to allocated memory. A process

cannot access memory space of another process.

- Communication Services
  - Manages communication between different computers on a network
  - Inter-task communication
- User Support
  - Provides interface for the use of different services

Process or Task

1. An abstraction of a running program
2. The logical unit of of work

scheduled by operating system.

3. A program in execution.

Processes are independant, carry considerable state information, have separate address spaces and interact through system-provided inter-process communication mechanism.

A Thread is a light weight process that shares resources with other process or threads.

Context switching faster.

As a process executes, it changes its state and at any given time it may be in one (and only one) of the following

states:

Fig (3.5 textbook)

Dormant or sleeping Waiting for service request

Ready

Executing

Suspended or Blocked

Terminated

Kernel

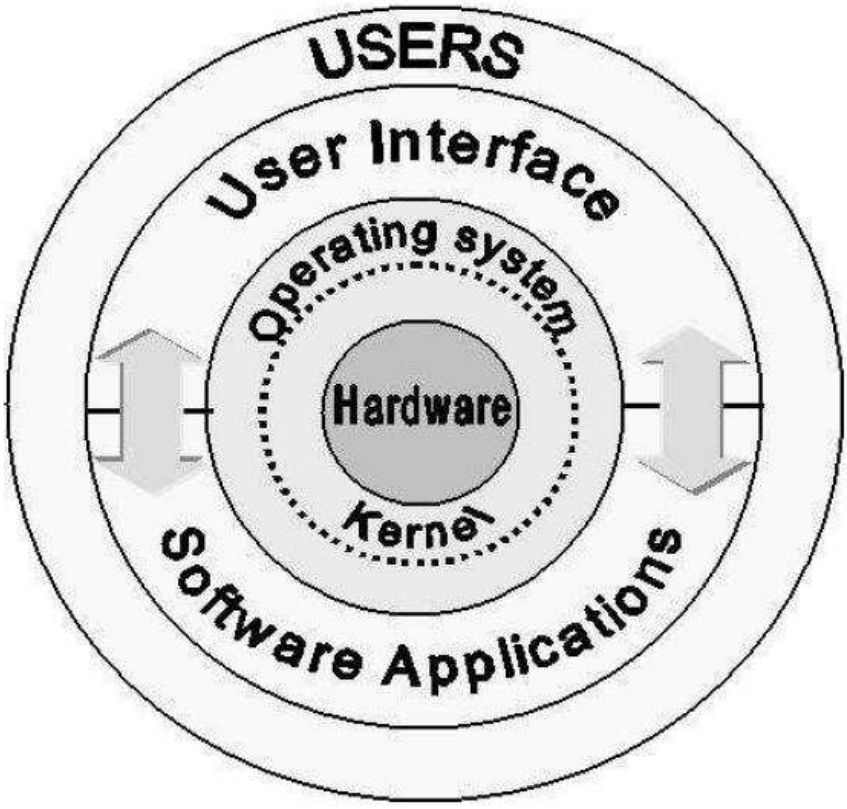

USERS
User Interface
Operating system
Hardware
Kernel
Software Applications

Figure 1:

Abstraction Layer for hardware resources
Programmer / user does not need to know hardware details
Mandatory part of OS,
Part of software closest to hardware
Basic purpose: Manage resources, allow other programs to run and use these resources. Usually provides features for:

- process creation and destruction
- scheduling of processes
- dispatching
- process suspension and resumption

- interrupt handling

  An interrupt is a hardware/software signal that indicates an event. On receipt of interrupt the processor:

  - completes the instruction being executed
  - saves the program counter so as to return to the same point
  - loads the program counter with the location of the interrupt handler code
  - executes the interrupt handler

    Real time systems can handle several interrupts in priority fashion

- Most of the PCs use Intel 82C59A-2 as Programmable Interrupt Controller
- interrupts can be enabled or disabled
- highest priority interrupt is served first

Timer Interrupts:

- Timer interrupt usually has two modes: periodic timer and one-shot timer.
- Periodic timer interrupt is a timer interrupt with a fixed timer interval.
- Interrupt generated at the end of

each tic

  - One shot timer more precise but does not repeat itself, needs to be restarted.

- context switch
  - contents of general registers
  - contents of program counter
  - contents of co-processor registers, if present
  - Memory page register
  - images of memory-mapped I/O locations
- manipulation of task control blocks
- inter-process communication

- process synchronization

A kernel can be divided into further layers such as micro-kernel (for task scheduling) and nano-kernel (for thread scheduling)

How does OS deliver all these services to a user process?

System Call

- User processes request a service from OS by making a **System Call**

- There is a library procedure corresponding to each system call that a user process can call.
- This procedure puts parameters of the system call in suitable registers and then issues a **TRAP** instruction.
- The control is passed on to OS, it checks the validity of parameters, performs the requested service
- When finished a code is put in a register telling if the operation was carried out successfully or it failed.
- A Return from TRAP instruction is then executed and the control is

passed back to the user process.

User and Supervisor modes

- The set of instructions that can be executed by a CPU are most often (except some embedded systems) divided into two classes.
- Those that can be executed by a user process and those that can only be executed by the Kernel
- This results in two modes of operation: a restricted user mode and a supervisor mode in which Kernel can execute any instruction

(including those belonging to user mode).

Process Scheduling

1. More than one process is runable
2. OS must decide which one to run first and in what order the remaining processes should run.
3. Scheduler and the scheduling algorithm.

A good scheduling algorithm for non-realtime systems, has the following goals:

1. Fairness: make sure that each process gets its fair share of CPU.
2. Efficiency: keep the CPU busy 100 percent of the time.
3. Response Time: minimize response time for interactive users.
4. Turnaround: minimize the time batch users must wait for the output.
5. Throughput: maximize the number of tasks processed per hour.

Preemptive/Nonpreemptive Scheduling

A nonpreemptive scheduler will not interrupt an executing task until it completes execution or decides on its own to release the allocated resources. In preemptive scheduling an executing task can be interrupted, if a more urgent task requests service (or its time slice expires).

How to run a real time task?

Timing Constraints

Goal of real-time scheduling is to meet the deadline of every task by ensuring that each task can complete execution by its specified deadline derived by environmental constraints imposed by the application.

Polled loop Systems

- A single repetitive instruction tests a flag that indicates whether or not an event has occurred.

```
Consider a system that handles
packets of data that arrive at
```

```
the rate of 1 per sec
On arrival of a packet a flag
packet-here is set to 1
for(; ;) {
        if (packet-here)
        {  process-data();
           packet-here = 0;
        }
       }
```

- No inter-task communication or scheduling needed as only a single task exists
- Excellent for handling high speed data channels when events occur at

widely spaced intervals and a process is dedicated to handling the data channel.

Pros

- Simple to write and debug
- Response time easy to determine

Cones

- Can fail due to burst of events
- generally not sufficient to handle complex systems
- waste of CPU time particularly when

events polled occur infrequently

Often used inside other RT systems
e. g. poll a suite of sensors for data or check for user input.

Synchronized Polled Loop

A variation to take care of switch bounce

```
for(; ;) {
          if(flag)
          {
             pause(20);
```

```
            process-event();
            flage = 0;
        }
    }
```

Cyclic Executives (Clock driven approach)

Decisions on what job executes at what time are made at specific time instants, chosen in advance before the system begins execution. Typically for such schedulers, all parameters of hard real time jobs are fixed and known. A schedule of jobs is computed off-line and is stored for use

at run time. For example:

```
for(; ;)
          {
             process-1();
             process-2();
             process-3();
             ............
             process-N();
          }
```

Interrupt driven systems

- Pre-emptive priority Systems

Higher priority job interrupts a lower priority job

Either fixed priority or dynamic

- Hybrid systems

  Interrupts occur both at fixed rate and sporadically (critical errors)

  Combination of round-robin and preemptive system

- Foreground-Background Systems

  A set of interrupt driven or RT processes run in the foreground

  a collection of non-interrupt driven jobs run in background

  A background job can be interrupted by a foreground job any

time.

What is Real-Time Operating System (RTOS)?

- A class of operating systems that are meant for real time applications.
- What is a real time application? Not only logically correct but also meets timing deadlines
- An RTOS has facilities to guarantee that deadlines are met
- It provides scheduling algorithms that enable deterministic behaviour

of systems

- Predictability is more valuable than throughput in RTOS
- AN RTOS is modular and extensible (embedded systems have small ROM/RAM space)
- Some of the tasks that may delay things are: IO, memory management, IPC, interrupt handling, context switching
  An RTOS has facilities to cut back on overheads for such tasks.
- Examples of commercial RTOSs: QNX, VxWorks, LynxOS etc.

LINUX as RTOS

- Is LINUX a RTOS?
- uses coarse-grained synchronization - a kernel task may have exclusive access to some data for a long time delaying a RT task
- does not preempt the execution of any task during system calls
- Linux makes high-priority tasks wait for low-priority tasks to release resources.
- Linux reorders requests from multiple processes to use the hardware more efficiently

- Linux will batch operations to use the hardware more efficiently.
- Real-time and general-purpose operating systems have contradictory design requirements.
- Linux provides a few basic features to support RT applications
- Varients of LINUX support RT applications by using a RT Kernel that interacts with the main kernel
- Treat LINUX OS as the lowest priority running task.
- Linux only executes when there are no real time tasks to run and the real time kernel is idle.

- A Linux task can neither block interrupts nor prevent itself from preemption.
- Architecture of RTAI

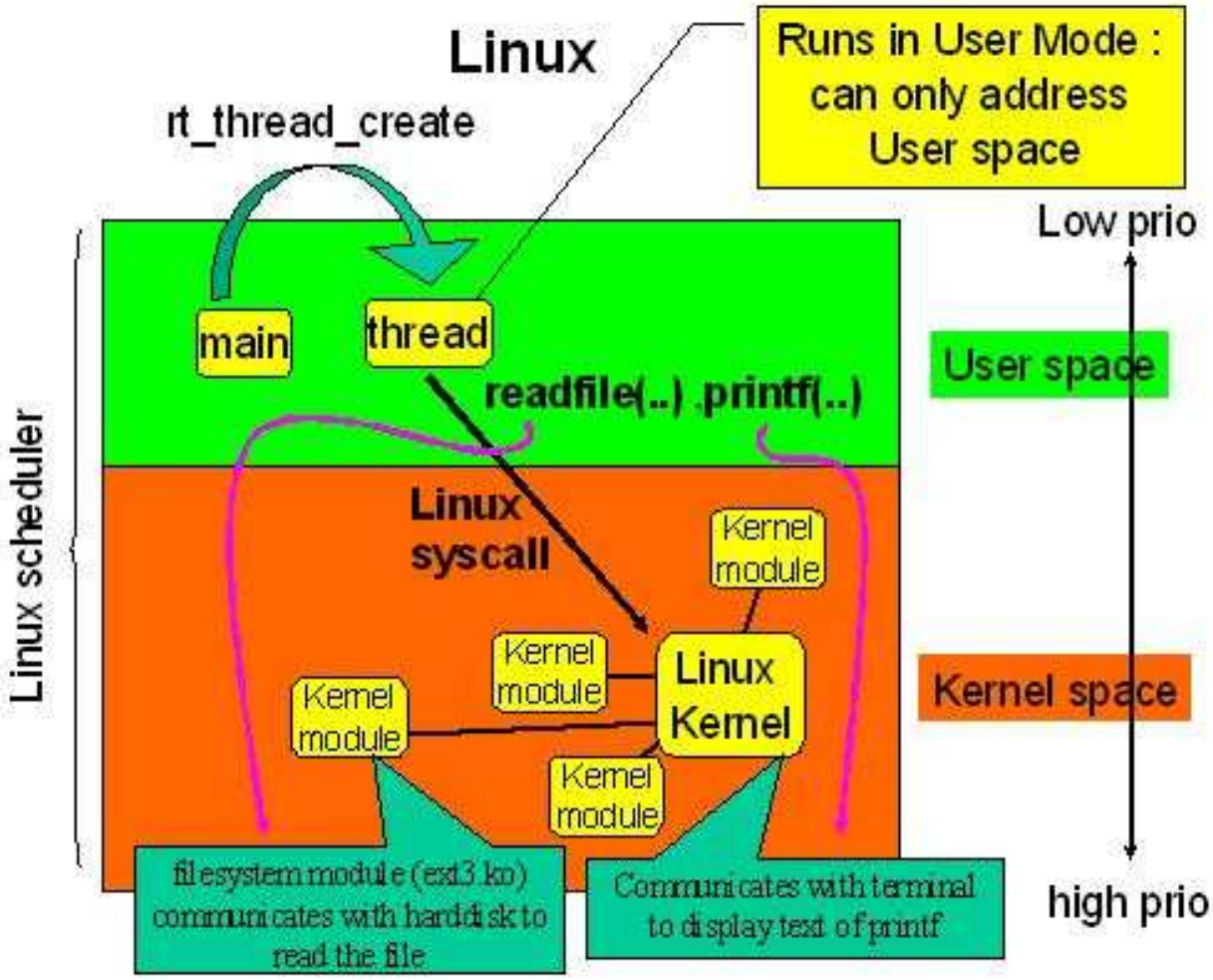

Linux
rt_thread_create
Runs in User Mode : can only address User space
Low prio
main
thread
readfile(..) printf(..)
User space
Linux scheduler
Linux syscall
Kernel module
Kernel module
Linux Kernel
Kernel module
Kernel module
Kernel space
filesystem module (ext3.ko) communicates with harddisk to read the file
Communicates with terminal to display text of printf
high prio

Figure 2:

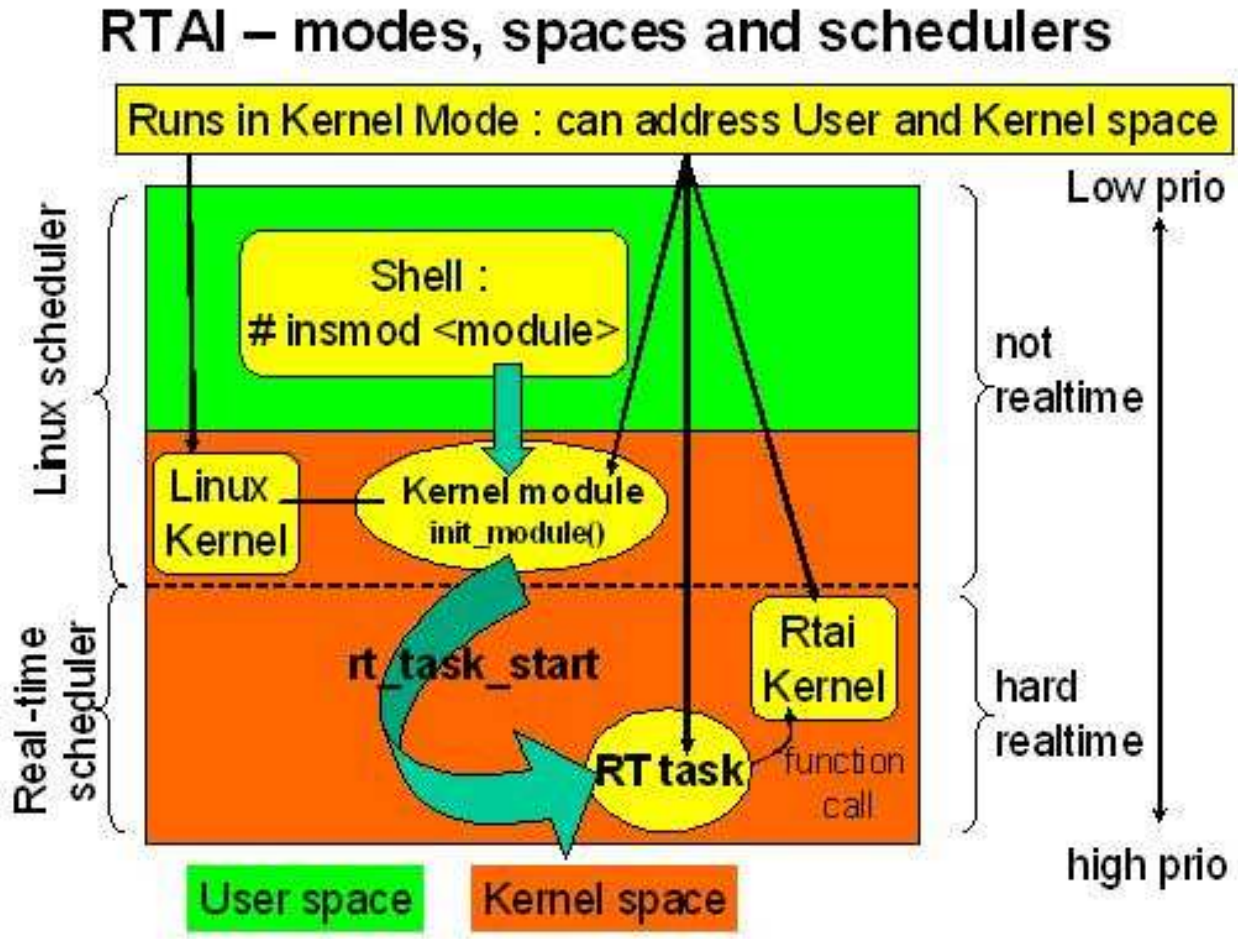

RTAI – modes, spaces and schedulers
Runs in Kernel Mode : can address User and Kernel space
Low prio
Linux scheduler
Shell :
insmod <module>
not
realtime
Linux
Kernel
Kernel module
init_module()
Real-time
scheduler
rt_task_start
Rtai
Kernel
hard
realtime
RT task
function
call
high prio
User space
Kernel space

Figure 3:

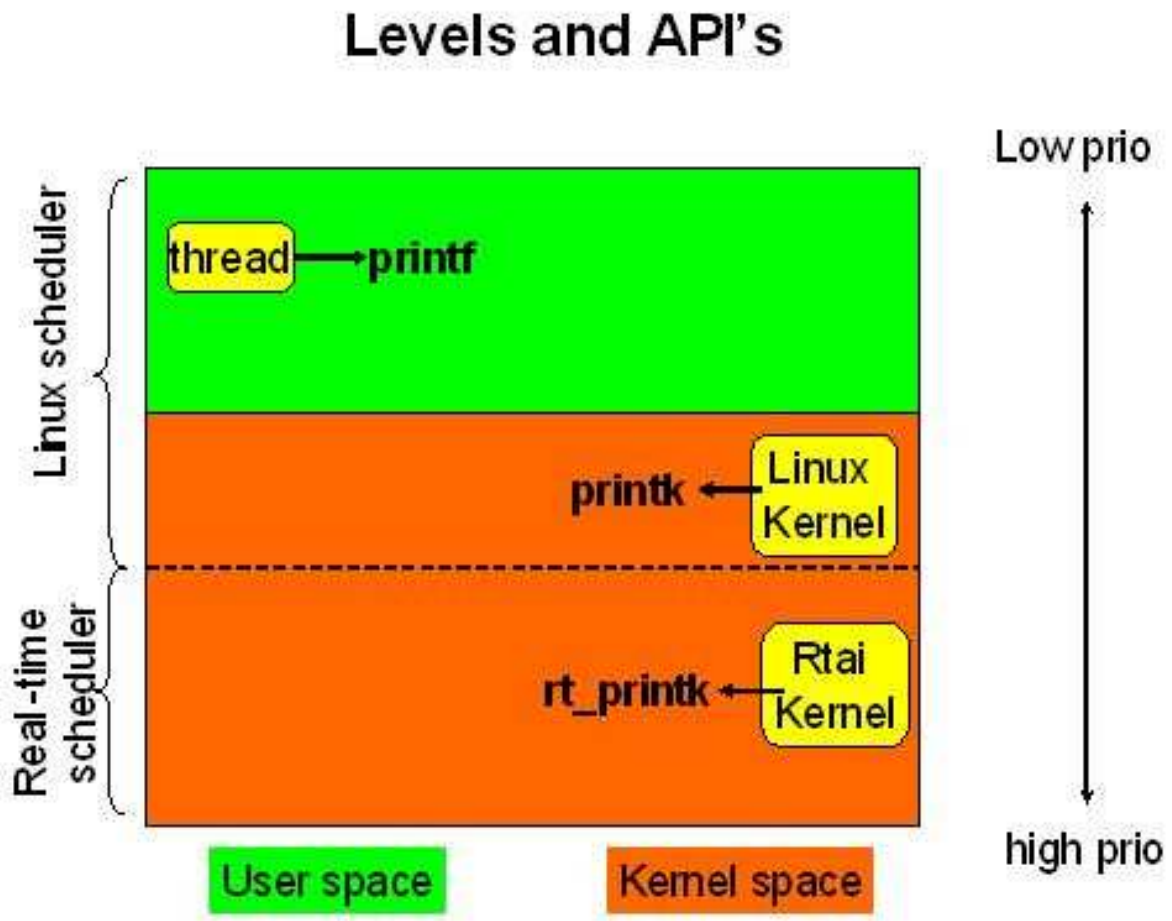

Levels and API's
Low prio
Linux scheduler
thread
printf
printk
Linux Kernel
Real-time scheduler
rt_printk
Rtai Kernel
high prio
User space
Kernel space

Figure 4:

How does RTAI work?

1. Capturing and redirecting of interrupts
   - Real Time Hardware Abstraction Layer (RTHAL)
   - Interrupt Descriptor Table (IDT), provides a set of pointers, which define to which processes each of the interrupts should be routed.
   - Whenever Linux tries to disable interrupts, the real-time kernel intercepts the request, records it, and returns to Linux.
   - When a hardware interrupt occurs,

the real-time kernel first determines where it is directed:

- Real-Time Task? Then schedule the task.
- Linux? Check the software interrupt flag:
- If enabled, invoke the appropriate Linux interrupt handler.
- If disabled, note that the interrupt occurred, and deliver it when Linux re-enables interrupts.
- Linux real time kernel relies on the "Linux Loadable Modules" mechanism to load RT components as kernel modules.

- The Real Time kernel, all its component parts, and the real time applications are all run in Linux kernel address space as kernel modules.
- Advantages: task switch time minimized, modularity
- Disadvantage: a bug can crash the whole system

2. Real time schedulers and services:
   - Module loading capability of Linux is used to provide real time schedulers, FIFOs, shared memory, and other services as they are needed.

- Services are implemented as kernel modules which can be loaded and unloaded by Linux commands **insmod, rmmod**
- Every time a real time service is required, **rtai** module is first loaded followed by other module(s) providing the desired service(s).
- Basic services are provided by four modules:

(a) rtai: the basic RTAI framework, plus interrupt dispatching and timer support.

(b) rtai_sched: the real-time,

pre-emptive, priority-based scheduler, chosen according to the hardware configuration.

(c) rtai_fifos: FIFOs and semaphores

(d) rtai_shm or mbuff: shared memory

Advanced features of RTAI such as LXRT, Pthreads, Pqueues and dynamic memory management are added via seperate modules.

3. Implementation of real time task:

- RT tasks are developed as kernel loadable modules
- In general real-time Linux tasks run with kernel modules (although

extended LXRT is changing this requirement) where they have direct access to the HAL and RTAI service modules.

Development Fundamentals

- Related commands: insmod, rmmod, lsmod, modinfo
- A real time task running as a kernel module under RTAI consists of three sections:
  1. Function *init_module()*: Invoked by *insmod* to prepare for later invocation of module's functions

Can be used to allocate required system resources, declare and start tasks etc.

2. Task specific code based on RTAI API
3. Function *cleanup_module()*
   Invoked by *rmmod* to inform kernel that the module's functions will not be called any more. A good place to release all of the system resources allocated during the lifetime of the module, stop and delete tasks etc.

A Simple Example

```
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void) {
printk(KERN_INFO "Hello World\n");
return 0;
}

void cleanup_module(void){
printk(KERN_INFO "Goodbye
          Cruel World!\n");
}
```

Compiling kernel modules:

- There is a new way of compiling kernel modules called **kbuild**.
- The build process for external loadable modules is now fully integrated into the standard kernel build mechanism.
- Details are available in linux/Documentation/kbuild/modules.txt file available on linux systems
- Kernel 2.6 introduced a new file naming convention for kernel modules with extension **.ko** instead of normal **.o**

A Sample MakeFile:

```
obj-m +=hello

all:
 make -C /lib/modules/$(shell uname -r)
                  /build M=$(PWD) modules

clean:
 make -C /lib/modules/$(shell uname -r)
                       /build M=$(PWD) clean
```

In the current versions of Linux (2.4 onwards) it is possible to use any suitable name for the *init* and *cleanup*

functions.
In order to do that one has to begin the name of functions with __init and __exit macros, then
use *module_init() and module_exit()* macros **after** defining these functions.

Other examples of macros

- __initdata: To initialize data
- MODULE_LICENSE(): use **GPL**
- MODULE_DESCRIPTION(): To describe what the module does
- MODULE_AUTHOR(): Name of the person who wrote code for the

module

- MODULE_SUPPORTED_DEVICES(): Declares what type of devices the module supports

Example

```
#include <linux/module.h>/*Every module requires it*/
#include <linux/kernel.h>/*KERN_INFO needs it*/
#include <linux/init.h>/* Required by macros*/

#define DRIVER_AUTHOR "Asghar Bokhari"
#define DRIVER_DESC "SE 4AA3/4GA3 example4"

static char *my_string __initdata = "dummy";
static int my_int __initdata = 4;

/* Init function with user defined name*/
static int __init hello_4_init(void)
{
 printk (KERN_INFO  "Hello %s world, number %d\n",my_string, my_int);
 return 0;
}


/* Exit function with user defined name*/
static void __exit hello_4_exit(void)
{
```

```
 /* printk (KERN_INFO "Goodbye cruel world %d \n", my_int);*/
 printk(KERN_INFO "Goodbye cruel world 4\n");
}

/*Macros to be used after defining init and exit functions*/
module_init(hello_4_init);
module_exit(hello_4_exit);

MODULE_LICENSE("GPL"); /* Avoids kernel taint message*/

MODULE_AUTHOR(DRIVER_AUTHOR); /* Who wrote this module? */
MODULE_DESCRIPTION(DRIVER_DESC); /* What does this module do */
MODULE_SUPPORTED_DEVICE("testdevice");/*This module */
                            /* uses /dev/testdevice.*/
```

Passing Commandline Arguments

- Command line arguments can be passed to modules but not with *argv*, *argc*
- First declare variables that will be used to store values passed on

commandline

- Then set them up using macro module_param(name, type, permissions)
- At run time *insmod* will fill up the variables with values passed

Example

```
static int my_int = 5; (initialize defaults)
module_param(my_int, int, S_IRUSER | S_IWUSER | S_IRGRP | S_IROTH)

OR

module_param(my_int, int, 0000);
MODULE_PARAM_DESC(my_int, "An integer");

There are macros for passing array or string via commandline
module_param_array() and module_param_string()
Details in linux/moduleparams.h
```

Relevent sections of the documents provided by Lineo Education Services:
File Day1.pdf:
Part: 2 (except those sections specific to ver 2.4 e.g. sec 2.4, 2.5, 2.8). For section 2.9 follow instructions in lab 4 document.
Part 3, 4, 5, 6, 9 and 10 (Use programming examples for guidance only. They may not work in 2.6 kernel without modifications.)
File Day2.pdf: Parts 1 and 2
File Day3.pdf: Parts 1, 2 and 3

Temporal Parameters

In order to meet the real time requirements of hard real-time tasks, it is assumed that many parameter of these tasks are known at all times. Some of these parameters are described below:

Number of Tasks:(n) The number of tasks in the system are known in advance.

1. In many embedded sytems, the number of tasks is fixed as long as the system remains in an

operation mode.

2. The number of tasks may change when the system enters a new mode and the number of tasks in the new mode is also known.
3. In some systems the number of tasks may change as tasks are added or deleted while the system executes, still the number of tasks with hard timing constraints is known at all times.

Release Time or Arrival Time: $(r_{i,j})$

Absolute deadline: (d_i)

Response Time The time span

between the task activation and its completion.

Relative Deadline: (D_i) Maximum allowable response time of a job is its relative deadline.

Execution Time: (e_i) The actual amount of time required by a job to complete its execution may vary for many reasons. What can be determined a priori through analysis and measurements is the max and min amounts of time to complete execution. e_i normally refers to the maximum time.

Periodic Tasks In this task model

computation or data transmission is executed repeatedly at regular or semi-regular time intervals in order to provide a function of the system on continuing basis.

This model fits accurately many of the hard real time applications such as digital control, real time monitoring, and constant bit-rate voice/video transmission.

Periods and Phases of Periodic Tasks:

A Period p_i of a periodic task T_i is the minimum length of all time intervals between release times of consequitive tasks.

Phase of a Task ϕ_i: The release time $r_{i,1}$ of a task T_i is called the phase of T_i i. e. $\phi_i = r_{i,1}$.
The first instances of several tasks may be released simultaneously. They are called in phase and have a zero phase.

CPU Utilization The CPU utilization or time-loading factor, U is a measure of the percentage of non-idle processing. A system is said to be time-overloaded if $U > 100\%$. U is calculated by summing the contribution of utilization factors for each (periodic

or aperiodic) task. The utilization factor u_i for a task T_i with execution time e_i and period p_i is given by:

$$u_i = e_i/p_i$$

And for a system with n tasks the overall system utilization is

$$U = \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} e_i/p_i$$

Aperiodic or Sporadic Tasks

Tasks whose release times are not known in advance. They are normally released in response to an external event e. g. sensitivity setting of a radar seveillance system by an

operator (soft), change of auto-pilot to manual mode (hard)

Precedence Constraints

If certain tasks are constrained to execute in a particular order (consuming data produced by previous task or other timing constraints) they are said to have **precedence constraints**.
Tasks that can execute in any order are called **independent**.

Preemptivity of Tasks

A task is preemptable if its execution can be suspended any time to allow execution of other jobs.

Typical Task Model

In order to simplify some of the scheduling policies used in real-time systems, a simple task model will be assumed in further discussions. This model assumes that:

1. All tasks in the task set are strictly periodic.
2. The relative deadline of a task is

equal to its period (if not specified specifically).

3. All tasks are independent i. e. there are no precedence constraints.
4. No task has any non-preemptible section and the cost of preemption is negligible.
5. Only processing requirements are significant; memory and I/O requirements are ngligible.

Round-Robin Scheduling

This method is commonly used for time shared applications where every

task that is ready for execution joins a FIFO queue

- The job at the head of the queue executes for at most one time slice.
- If it does not commplete by the end of the time slice, it is preempted after its context is saved and is placed at the end of the queue to wait for its next turn.
- This approach achieves a fair allocation of the CPU to tasks of the same priority and are generally not suitable for real-time applications.

- Round robin systems can be combined with preemptive systems to get a kind of mixed system that works as shown in

Fig(3.6 of textbook).

Timer-Driven Scheduler

- parameters of jobs with hard deadlines known
- off-line static-schedule - specify exactly when each job executes
- All deadlines are surely met under normal conditions

- Sophisticated algorithms can be used
- consider 4 jobs $T_1 = (4, 1)$, $T_2 = (5, 1.8)$, $T_3 = (20, 1)$, $T_4 = (20, 2)$
- What is total utilization?
- Want to construct schedule for these processes, how long should it be (how many entries?)
- Each period should divide the cycle - hyperperiod; how can it be ensured?
- The maximum number of jobs N, in a hyperperiod H is:

$$N = \sum_{i=1}^{n} H/p_i$$

- in above example $H = 20$, and $N = 11$

```
For one of the possible schedules, the
table may have the following entries:
Time      Process
0         T1
1         T3
2         T2
3.8       I
4         T1
...........
...........
19.8      I
```

```
Psudocode from (Jane Liu)
Input: Stored schedule (t_k, T(t_k))
       for k = 0, 1, 2, ... N - 1
Task SCHEDULER:
   set the next decision point i and
       table entry k to 0;
   set the timer to expire at t_k;
   do for ever:
      accept time interrupt;
      if an aperiodic job is excuting,
         pre-empt the job;
      current task T = T(t_k);
      increment i by 1;
      compute the next table entry
            k = i mod (N);
```

```
    set the timer to expire at
            floor(i/N)*H + t_k;
    if the current task T is I,
       let the job at the head of
          the aperiodic queue execute;
    else, let the task T execute;
    sleep;
end SCHEDULER
```

Cyclic Executive CE

- a schedule that is not ad hoc; it has some structure
- scheduling decisions made at regular

intervals rather than at arbitrary times.

- timeline is partioned into minor cycles called frames.
- a non-repeating set of minor cycles makes up a major cycle.
- The operations are implemented as procedures, and are placed in a pre-defined list covering every minor cycle.
- When a minor cycle begins, the timer task calls each procedure in the list.
- long operations must be manually broken to fit frames.

- Every frame has a length, f, called the frame size.
- The schedule is written for one hyperperiod and can be repeated for subsequent periods.
- The hyperperiod is equal to the LCM of of the periods of processes allocated to a processor.

$$Hyperperiod = lcm(p_1, p_2, p_3, ...p_n)$$

- As the scheduling decisions are made only at the begining of every frame, there is no preemption within a frame.
- The phase of each task is a non-negative integer multiple of the

frame size i. e. the first instance of every task is released at the beginning of some frame.

- In addition to chosing which process to execute the scheduler carries out monitoring and enforcement actions at the beginning of the frame, particularly it checks if every job scheduled in the frame has been released and is ready for execution.
- The scheduler also checks if there is any overrun and takes error handling action if necessary.

Frame Size Constraints

- Ideally, frames must be sufficiently long so that every task can start and complete execution within a frame. In this way no task will be preempted.
- This requires that the frame size f is larger than the execution time e_i of every task, T_i.

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

- In order to keep the length of the cyclic schedule as short as possible, the frame size, f, should be chosen so that the hyperperiod has an integer number of frames. (the sheduling decisions are taken at the

beginning of each frame and if a frame does not end with the end of hyperperiod, we cannot start repeating the schedule)

- This condition is met when f divides the period p_i of at least one task T_i:

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

- In order to make it possible for the scheduler to determine whether every task completes by its deadline, the frame size should be sufficiently small so that between the release time and the deadline of every job, there is at least one full frame.

Refer to fig 3.7 of textbook
t denotes the biginning of kth frame in which task T_i is released at time t'. In order to have one full frame between release time and deadline of the job:

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

The difference $t' - t$ is at least equal to $gcd(p_i, f)$, which result in the third constraint:

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

Example1:
Choose frame size for (4, 1, 4), (5, 1.8, 5), (20, 1, 20), (20, 2, 20)

$$C_1 :\rightarrow f \geq 2$$

$$Hyperperiod = lcm(4, 5, 20, 20) = 20$$

$$C_2 :\rightarrow f = 2, 4, 5, 10 and 20$$

C_3 (satisfied for:

$$f = 2, P_1 = 4) \rightarrow$$

$$4 - gcd(4, 2) = 2 \leq 4 < D_i$$

C_3 (satisfied for:

$$f = 2, P_2 = 5) \rightarrow$$

$$4 - gcd(5, 2) = 3 \leq 5 < D_i$$

C_3 (satisfied for:

$$f = 2, P_3 = 20) \rightarrow$$

$$4 - gcd(20, 2) = 2 \leq 20 < D_i$$

C_3 (satisfied for:

$$f = 2, P_4 = 20) \rightarrow$$

$$4 - gcd(20, 2) = 2 \leq 20 < D_i$$

C_3 (Satisfied for:

$$f = 4, P_1 = 4) \rightarrow$$

$$8 - gcd(4, 4) = 4 = D_i$$

C_3 (Not Satisfied for:

$$f = 4, P_2 = 5) \rightarrow$$

$$8 - gcd(5, 4) = 7 > D_i$$

C_3 (Not Satisfied for:

$$f = 5, P_1 = 4) \rightarrow$$

$$10 - gcd(4, 5) = 9 > D_i$$

C_3 (Not satisfied for:

$$f = 10, P_1 = 4) \rightarrow$$

$$20 - gcd(4, 10) = 18 > D_i$$

similar for $f = 20$.

We must choose the frame size as 2.

The schedule is shown in figure 5

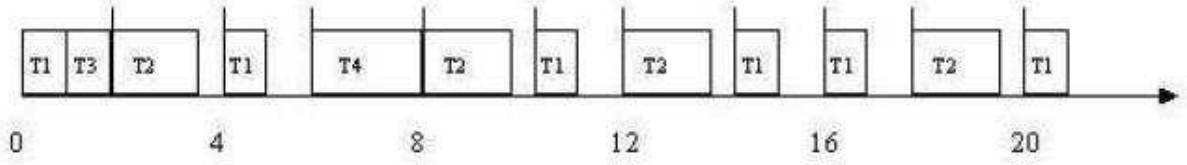


Figure 5: Cyclic executive example1 schedule

Example2:

Consider the tasks:

$$T_1 = (4, 1), T_2 = (5, 2, 7), T_3 = (20, 5)$$

Constraint C_1 dictates $f \geq 5$, but constraint C_3 requires $f \leq 4$

In this situation we must partion the task with large execution time into several sub-jobs.

The resulting system has five jobs:

$$T_1 = (4, 1), T_2 = (5, 2, 7), T_{31} = (20, 1),$$

$$T_{32} = (20, 3), T_{33} = (20, 1)$$

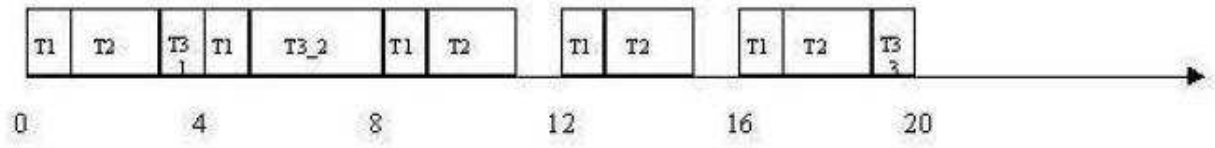


Figure 6: Cyclic executive example2 schedule

Why not split T_3 into

$T_{31}(20, 2) and T_{32}(20, 3)$?

1. T_1 with a period of 4 must be scheduled in each frame of size 4
2. T_2 with a period of 5 must be scheduled in 4 out of five frames
3. This leaves only 1 frame with 3 units of time for T_3, other frames have only 1 unit of time and cannot have a job with execution time of 2.

Three kinds of decisions:

1. choose a frame size
2. partition jobs into slices

3. place slices into frames

Try to partition the job into as few slices as necessary to meet frame size constraints but if there is no feasible schedule split the job into smaller slices.

```
Psudocode for a
cyclic executive (Jane Liu)
Input:Store schedule:
L(k) for k = 0, 1,...F-1
(Where F is the number of frames)
Aperiodic job queue
Task CYCLIC_EXECUTIVE:
 current time t = 0;
```

```
current frame k = 0;
do forever
 accept clock interrupt at time tf;
 currentblock = L(k);
 t = t + 1;
 k = t mod F;
 if the last job is not completed,
      take appropriate action;
 if any of the slices in
     current block is not
     released, take necessary action;
 wake up the periodic task server
     to execute slices in currentblock;
 sleep until periodic task
                      server finishes;
```

```
  while aperiodic job queue non-empty,
      wake up job at the head of queue;
  sleep until aperiodic job completes;
  remove aperiodic job from queue;
  endwhile;
   sleep until next clock interrupt;
 enddo;
end CYCLIC_EXECUTIVE
```

Response time of aperiodic jobs

- Aperiodic jobs scheduled in

background after all hard deadline jobs in a frame are completed

- released in response to an event and deserve a better response time.
- The delaying strategy makes the system less responsive
- Completing a hard deadline job early has no advantage
- How to make system more responsive? **Slack stealing**
- For this scheme to work, every periodic job slice must be scheduled in a frame that ends no later than its deadline

- Consider frame K
- Let the time allocated to all job slices in this frame be t_k
- The slack time available in this frame is $f - t_k$
- If there are jobs in aperiodic queue, the scheduler can let those jobs execute in this slack time at the beginning of a frame.
- If there are no jobs in aperiodic queue, the next slice of periodic job is executed
- At the end of execution of each periodic job slice the scheduler checks if an aperiodic job is

available, and runs it, as long as some slack time is available for the current frame.

Example:

Handling frame overruns

- Execution time of a job, input data dependent
- Transient hardware fault
- Undetected software bug
- Abort the job at the beginning of next frame and log the incident
- Preempt the job when its allocated

time is finished, and let it complete in slack time

- Let the job continue execution until completion and delay the rest of the jobs also

Multiprocessor Scheduling
Construct a global schedule that specifies on which processor the job executes in addition to when it executes.

Pros and Cons of clock driven scheduling

Pros:

- Conceptual simplicity: Complex dependencies, communication delays and resource contentions can be taken care of
- Timing constraints can be checked and enforced at frame boundaries.
- Preemption cost can be kept small by having appropriate frame sizes.
- Easy to validate: Execution times of slices known a priori.

  Cons:
- Release time of all jobs must be fixed
- Difficult to maintain.

- Does not allow to integrate hard and soft deadlines.

Event-Driven Systems

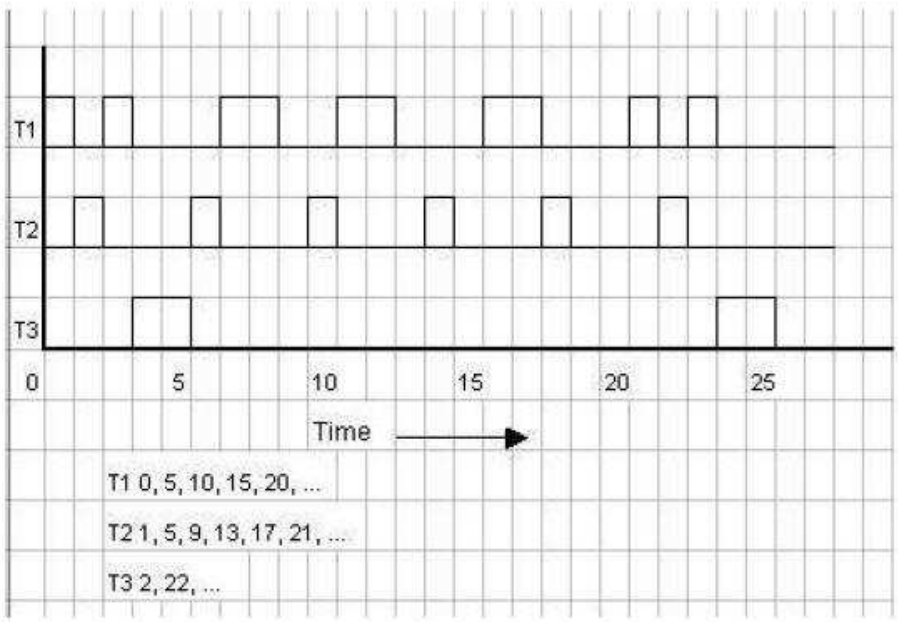
An event-driven design uses real-time I/O completion or timer events to trigger schedulable tasks. Many real-time Linux systems follow this model.
Scheduling based on job priorities - Static, Dynamic

Rate Monotonic (RM) Scheduling Algorithm

- One of the most popular algorithms.
- Uniprocessor static-priority preemptive approach.
- At any time instant, an RM scheduler executes the instance of the ready task that has the highest priority.
- The priority of a task is inversly related to its period i. e. if task T_i has a period p_i, and another task T_j has a period p_j, then T_i has a higher priority than T_j if $p_i < p_j$.
  If two or more tasks have the same period, the the scheduler selects one of these jobs at random.

Example

Consider the following periodic tasks:
$T_1(0, 5, 2, 5)$, $T_2(1, 4, 1, 4)$,
$T_3(2, 20, 2, 20)$



- At $t = 0$, T_1 is the only ready task
- At $t = 1$, T_2 is also available and has a higher priority as $p_2 < p_1$

- At $t = 2$, T_3 arrives but it has the lowest priority, so waits until T_1 finishes
- At $t = 5$, second instances of both T_1 and T_2 arrive but T_2 executes first
- Third instance of T_2 arrives at $t = 9$ and that of T_1 at $t = 10$
- So on

Schedulability Analysis

Determining if a specific set of tasks satisfying certain criteria can be successfully scheduled (completing execution of every task by its specified

deadline) using a specific scheduler.

Schedulability Test

is used to validate that a given application can satisfy its specified deadlines when scheduled according to a specific scheduling algorithm. This test is often done at compile time, before the computer system and its tasks start execution.

Optimal Scheduler

is one which may fail to meet the

deadline of a task, only if no other scheduler can meet it. Note that "Optimal" in real-time scheduling does not necessarily mean "fastest average response time" or "shortest average waiting time".

Shedulability Tests for RM Sheduler

Test 1:

- n periodic processes that are indenpendent and preemtable
- $D_i \geq p_i$ for all processes

- Periods of all processes are integer multiples of each other
- A necessary and sufficient condition for such tasks to be scheduled on a uniprocessor using RM algorithm:

$$U = \sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1$$

Example

Consider tasks: $(4, 1), (2, 1), (8, 2)$

$p_1 = 2p_2$, $p_3 = 4p_2 = 2p_1$

The task set belongs to the special class of tasks for which the above schedulability test applies

Now $U = 1/4 + 1/2 + 2/8 = 1 \leq 1$
Therefore this task set is RM schedulable

Test 2:

If the tasks have arbitrary periods, a sufficient but not necessary schedulability condition is :

$$U \leq n(2^{1/n} - 1)$$

That is there may be task sets with a utilization greater than $n(2^{1/n} - 1)$ that are schedulable by RM algorithm.

Test 3:(Shin)

A sufficient and necessary condition for scheduability by RM algorithm can be derived as follows:
Consider a set of tasks $(T_1, T_2 ... T_i)$ with $(p_1 < p_2 < p_3 < ..... < p_i)$. Assume all tasks are in phase. The moment T_1 is released, the processor will interrupt anything else it is doing and start processing this task as it has the highest priority (lowest period). Therefore the only condition that must be satisfied to ensure that T_1 can be feasibly scheduled is that:

$$e_1 \leq p_1$$

This is clearly a necessary and

sufficient condition.
The task T_2 will be executed successfully if its first iteration can find enough time over the time interval $(0, p_2)$ that is not used by T_1. (p_2 is the period of T_2 and the first instance of T_2 must complete before the second instance arrives p_2 time after the first instance that arrives at zero time)
Suppose T_2 finishes at t. The total number of instances of task T_1 released over the time interval $(0, t)$ is $\lceil \frac{t}{p_1} \rceil$
If T_2 is to finish at t, then every

instance of task T_1, released during time interval $(0, t)$, must be completed and in addition there must be e_2 time available for execution of T_2, i. e. the following condition must be satisfied:

$$t = \lceil \frac{t}{p_1} \rceil e_1 + e_2$$

All we need is to find some $t \in (0, p_2)$ satisfying this condition.

How to find such a t??

Note that every interval has infinite number of points, so we cannot exhaustively check for every possible t.

Consider t/p_1, it only changes at multiples of p_1, with jumps of e_1. So if we can find an integer k, such that

the time $t = k * p_1 \geq ke_1 + e_2$ and $kp_1 \leq p_2$, we have the necessary and sufficient condition for T_2 to be schedulable under the RM algorithm. That is we we only need to check if

$$t \geq \lceil \frac{t}{p_1} \rceil e_1 + e_2$$

for some value of t that is multiple of p_1 such that $t \leq p_2$.

Now consider task T_3. It is sufficient to show that its first instance completes before the arrival of its next instance at p_3. If T_3 completes its execution at at t, then by an argument

similar to that for T_2, we must have:

$$t = \lceil \frac{t}{p_1} \rceil e_1 + \lceil \frac{t}{p_2} \rceil e_2 + e_3$$

T_3 is schedulable iff there is some $t \in (0, p_3)$ such that the above condition is satisfied. Again the right side of above equation changes in multiples of p_1 and p_2. It is therefore sufficient to check that

$$t \geq \lceil \frac{t}{p_1} \rceil e_1 + \lceil \frac{t}{p_2} \rceil e_2 + e_3$$

is satisfied for some t that is a multiple of p_1 and/or p_2, such that $t \leq p_3$

Test 3 for schedulability under RM algorithm can now be stated as: The

time demand function

$$w_i(t) = \sum_{k=1}^{i} \lceil \frac{t}{p_k} \rceil e_k \leq t$$
$$0 \leq t \leq p_i$$

holds for any time instant t chosen as follows:

$$t = kp_j, j = 1, ...i$$

and

$$k = 1, ... \lfloor \frac{p_i}{p_j} \rfloor$$

iff the task T_i is RM-schedulable.
If $p_i \neq D_i$, we replace p_i with $\min(D_i, p_i)$ in the above expression.

Example

Determine if the following set of tasks is RM schedulable: (50, 10), (80, 15), (110, 40), (190, 50)

Deadline Monotonic (DM) Algorithm

- Another fixed priority scheduler
- Priorities are based on relative deadlines: shorter the deadline higher the priority
- if every task has the period equal to relative deadline, same as RM
- For arbitrary deadlines, DM

algorithm performs better than RM algorithm

- It may sometime produce a feasible schedule when RM fails
- RM algorithm always fails if DM algorithm fails.

Sporadic Tasks

So far we considered only periodic tasks. In case there are sporadic task they can be handled in several different ways;
Treat them as periodic with a period equal to the minimum inter-arrival

time between the release of successive sporadic tasks.
Define a fictitious periodic task of highest priority and some fictitious execution time. If a sporadic task is not available, this method results in wasting CPU time.
An approach to avoid waste of CPU time is **deferred server**, where the server starts the periodic task of highest priority if a sporadic task is not available, but if it does become available later, the periodic task is preempted.

1 Dynamic-priority Scheduling: Earliest-deadline First (EDF)

- In this algorithm the task priorities are not fixed but change depending upon the closeness of their **absolute deadlines**.
- The processor always executes the task whose absolute deadline is the earliest. (Note that the absolute deadline is the arrival time of a task plus its relative deadline).
- If more than one tasks have the

same absolute deadlines, EDF randomly selects one for execution next.

Example (textbook page 97 fig. 3.10.)

EDF is optimal

EDF is an optimal uniprocessor scheduling algorithm, which means that if EDF cannot feasibly schedule a task set on a uniprocessor, there is no other scheduling algorithm that can.

EDF Schedulability Tests:

Test 1:

A set of n periodic tasks, each of whose relative deadline equals its period, can be feasibly scheduled by EDF iff

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

Test 2:

No simple test is available in the case where the relative deadlines do not equal to their periods. In such cases the best course of action is to develop a schedule using EDF algorithm to

see if all deadlines are met over a given interval of time.

A sufficient condition for such cases is:

$$\sum_{i=1}^{n} \frac{e_i}{min(D_i, p_i)} \leq 1$$

- Only sufficient condition - if it fails the task set may or may not be EDF schedulable.
- If $D_i \geq p_i$, it reduces to the test discussed above.
- If $D_i < p_i$, the equation represents only a sufficient condition.

Comparison of RM and EDF

Algorithms

- EDF is more flexible and has a better utilization than RM.
- Timing behaviour of a system scheduled with RM algorithm is more predictable
- In case of overload RM is stable in the presence of missed deadlines: same lower priority tasks miss the deadlines every time and there is no effect on higher priority tasks.
- In case of EDF, it is difficult ot predict which tasks will miss their deadlines during overloads. Also

note that a late task that has already missed its deadline has a higher priority than a task whose deadline is still in the future.

2 Other Considerations

So far we assumed that all tasks are independent and can be preempted any time, however this assumption may be unreasonable from a practical viewpoint. We now discuss the effects of task synchronization and how to avoid blocking that may arise in

uniprocessors when concurrent tasks use shared resources.

Inter-task communication

- Task interaction common in applications
- Tasks share resources,
- Some resources can only be used by one task at a time
- Mechanisms required to to allow tasks to communicate, share resources and synchronize activity.

Buffering Data

Producer consumer processes need to synchronize
Double buffers
Ring buffers

Mailboxes

OS provided inter task communication mechanism
A mutually agreed upon memory location that one or more tasks can use to pass data.

Critical Regions

- In most cases resources can be used by one task at a time
- Mutual exclusion
- Use of a resource cannot be interrupted - serially reusable
- Code that interacts with serially reusable resources - critical section
- If two tasks enter the same critical region simultaneously, error will result.
- Example of ATM or printer
- When two or more processes are

competing to use the same resource - race condition - conflict

- How to ensure exclusivity?

Semaphores

Most common method of protecting critical regions

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

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

```
Process 1
.
.
.
P(s)
critical region
V(S)
.
.
.
```

```
Process2
.
.
.
.
P(s)
critical region
 V(s)
.
.
```

Counting Semaphores

Deadlocks

```
Task A                  Task B

.                           .
.                           .
.                           .
P(S)                        P(R)
use resource 1              use res 2
.                           .
.                           .
```

```
.                          .
P(R)                       P(S)
stuck here                 stuck here
use resoure 2              use res 1
.                          .
.                          .
V(R)                       V(S)
V(S)                       V(R)
.                          .
.                          .
```

Non-preempt-able Tasks with Precedence Constraint

- Task precedence graph: Node

represents a task, directed edge represents precedence relationship

- $T_i \rightarrow T_j$ means T_i must be completed before T_j
- Create a precedence graph such that tasks with no in-edges are listed first.
- If two or more tasks can be listed next, select the one with earliest deadline.
- Execute tasks one at a time following this order

Example

$$T_1(5,2), T_2(10,3), T_3(7,2),$$

$$T_4(18, 8), T_5(25, 6), T_6(28, 4)$$

The tasks have the following precedence constraints:

$$T_1 \rightarrow T_2, T_1 \rightarrow T_3$$

$$T_2 \rightarrow T_4, T_2 \rightarrow T_6$$

$$T_3 \rightarrow T_4, T_3 \rightarrow T_5$$

$$T_4 \rightarrow T_6$$

How to schedule these one-instance tasks?

Resource Access Control

- one processor

- n serially reuseable resources $R_1, R_2, ..R_n$
- typically used by processes in a mutually exclusive manner without preemption
- a resource once allocated to a job cannot be used by another job until the previous job frees it
- Resources that can be used by more than one jobs at the same time (e. g. file) are modeled as a resource type that has many units, each used in a mutually exclusive manner.

Mutual Exclusion and Critical Sections

- When a job wants to use η_i units of resource R_i, it executes a lock to request them, denoted by $L(R_i, \eta_i)$. An unlock is denoted by $U(R_i, \eta_i)$
- In case R_i has just one unit, a simpler notation $L(R_i), U(R_i)$ is used.
- A segment of a job that begins at a lock and ends at a matching unlock is called a critical section, denoted by $[R; t]$
- A critical section that is not

included in any other critical sections is called outermost critical section

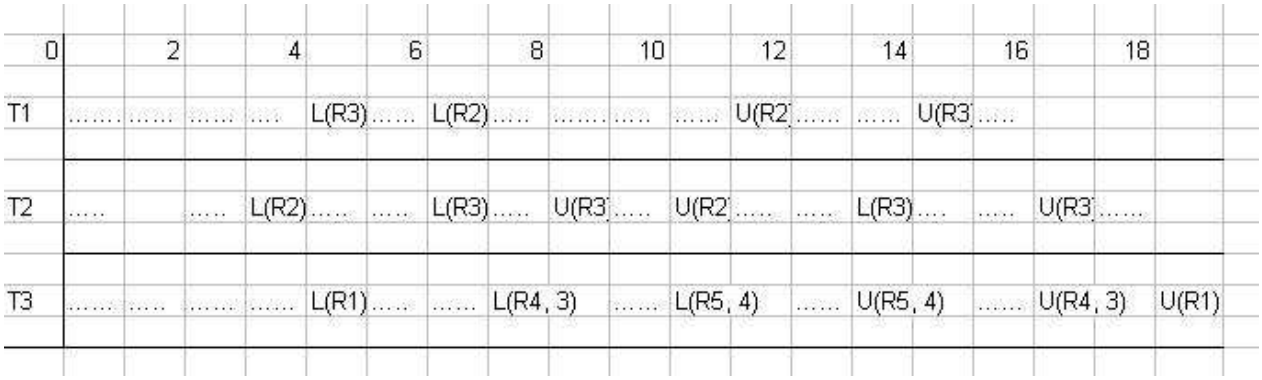


Figure 7: Locks and CS's

Resource Contention

- If more than one jobs require same resource they are said to be in conflict or contention
- The scheduler always denies a request if enough units of the required resource are not free
- The lock request $L(R_i, \eta_i)$ fails and the process that requested the lock is blocked and looses the processor
- It is moved out of the ready queue until the required resource becomes available, when it is placed back in the ready queue.

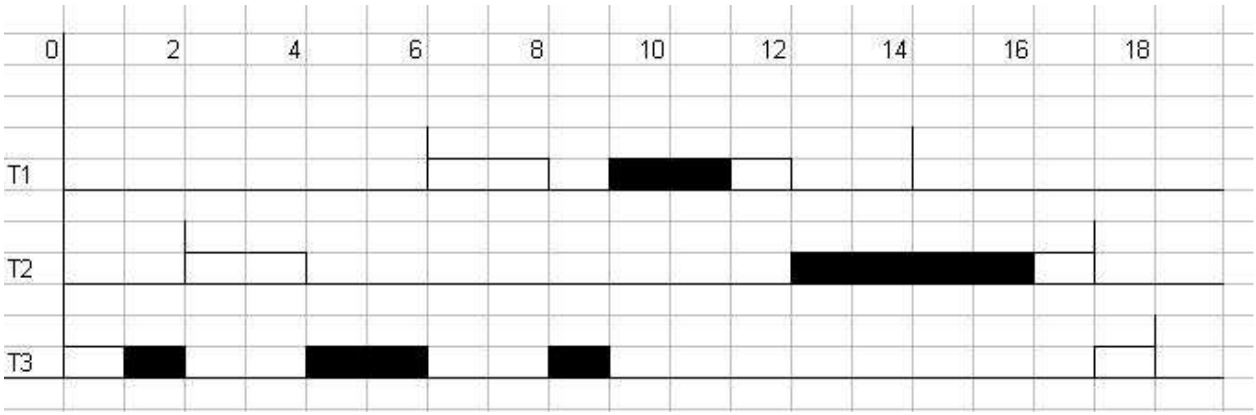


Figure 8: Resource Contention and Priority Inversion

- Consider
  $T_1(6, 7, 5, 8), T_2(2, 15, 7, 15),$
  $T_3(0, 18, 6, 18)$
- Assume EDF scheduling algorithm
- T_1 has the highest priority and T_3 the lowest priority
- The tasks T_1, T_2, T_3 each has critical sections $[R; 2], [R; 4], [R; 4],$

respectively

- Figure 8 shows a section of the schedule, where black boxes indicate critical sections of each task
- It shows how resource contention can delay completion of a higher priority task.
- Tasks T_1, T_2 could complete by time 11 and 14 if there was no resource contention

Priority Inversion

What really happened on Mars?

http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/Mars_

- The above example shows that a higher priority task can be blocked by a lower priority task due to resource contention
- This is because even if the taks are preemtable, resources are allocated on non-preemptive basis
- This phenomenon is called priority inversion (ref. time intervals (4, 6) and (8, 9))

Timing Anomalies

Priority inversion may result in timing anamalies i. e. some tasks may not be able to meet their deadlines. Reduce CS of T_3 to 2.5

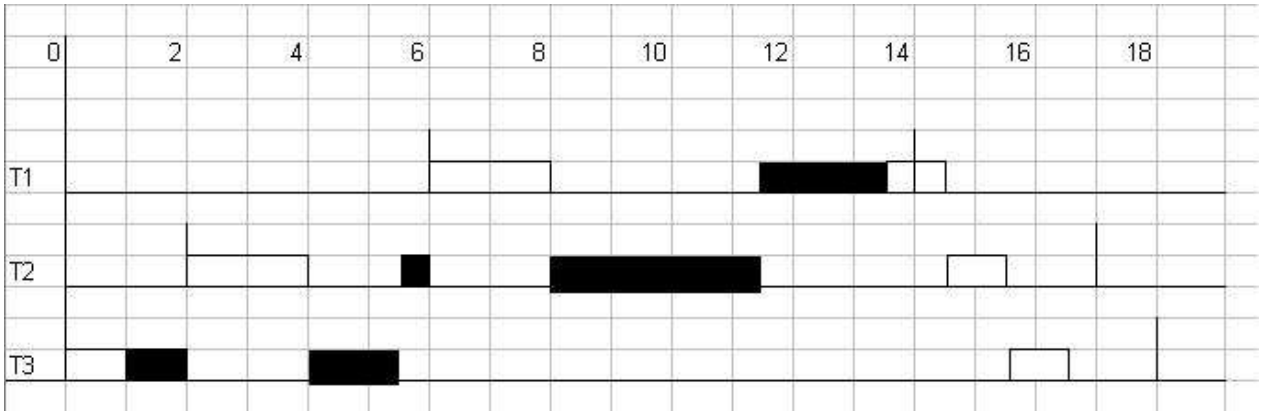


Figure 9: Timing Anomalies

Without a good resource access control protocol, duration of a priority inversion may be infinite

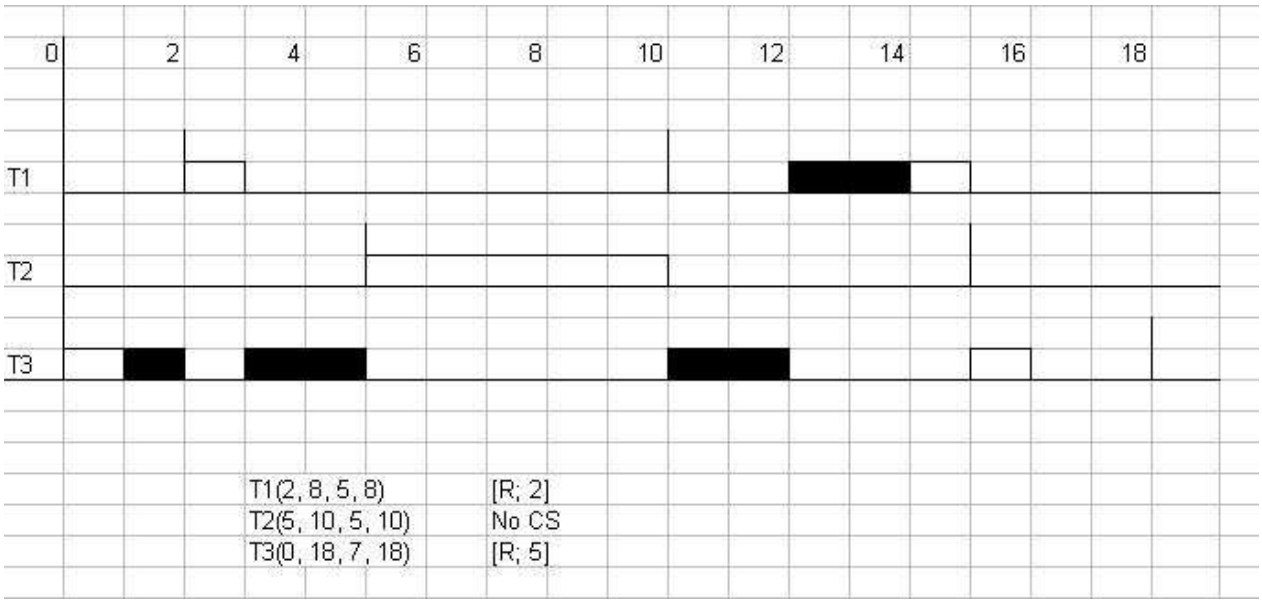


Figure 10: Uncontrolled Priority Inversion

Non preemptivity of resource allocation can also cause deadlocks e. g. when there are two jobs that both require resources X and Y.

Resource Access Protocols

- A set of rules that govern:
- When and under what conditions each request for a resource is granted?
- How to schedule the tasks that require resources?

Non-preemptive Critical Section Protocol (NPCS)

- Schedule all critical sections non-preemptively

- While a task holds a resource it executes at a priority higher than the priorities of all tasks
- In general uncontrolled priority inversion never occurs as a higher priority task is blocked only if it is released when some lower priority job is in critical section
- Once the blocking critical section completes, no lower priority task can get the processor and acquire any resource until the higher priority task completes.

Advantages

- Does not need prior knowledge about resource requirements of tasks
- Simple to implement
- Can be used in both static and dynamic priority schedulers
- Good protocol when most critical sections are short and most tasks conflict with one another

Disadvantages

- Every task can be blocked by every lower priority task that requires

some resource, even without a resource conflict

Priority Inheritance Protocol

Has most of the advantages of NPCS, however it does not avoid deadlocks

Assigned Priority: A priority that is assigned to a task according to the scheduling algorithm used.

Current Priority: The priority of a task at a given time - may differ from the assigned priority and may vary with time.

Inherited priority: The current priority of a task may be raised to the higher priority of another task. The task with lower assigned priority can then execute at the higher priority and is said to inherit the higher priority.

Scheduling Rule: 1. Tasks are scheduled on a processor according to current priorites

2. At release time, the current priority of each task is equal to its assigned priority
3. The current priority of a task changes according to priority

inheritance rule.

Allocation Rule: When a task requests a resource R at time t
(a) if R is free, it is allocated to the task until it releases it
(b) if R is not free, the request is denied and the requesting task is blocked

Priority Inheritance Rule: When a task T_1 is blocked due to non availability of a resource that it needs, the task T_2 that holds the resource and consequently blocks T_1 inherits the current priority of task T_1.

T_2 executes at the inherited priority until it releases R

At this time the priority of T_2 returns to the priority that it held when it acquired the resource R.

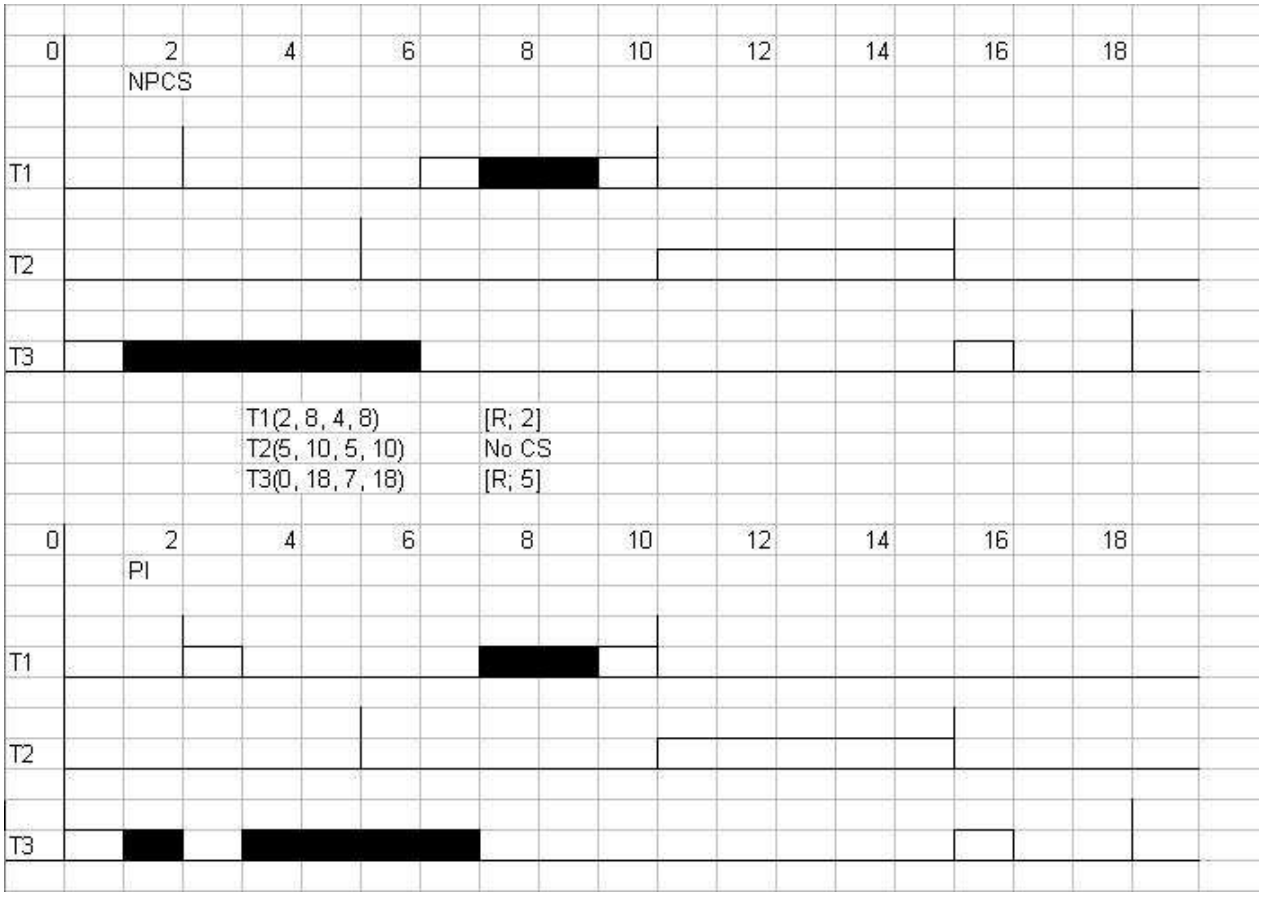


Figure 11: NPCS/Priority Inheritance

Priority Ceiling Protocol

Extend priority inheritance protocol to prevent deadlocks and to further reduce the blocking time.

Assumptions:

(1) Assigned priorities of all jobs are fixed

(2) Resource requirements of all jobs are known before their execution begins.

New Terms

Priority ceiling of a resource: By assumptions (1, 2) above, we know the priorities of a set of tasks that will

use a particular resource. The highest priority of this set is assigned to the resource. Denoted by $\pi(R_i)$

Priority ceiling of a system: At any given time a set of resources are being used, the highest priority ceiling of this resource set is called the priority ceiling of the system, denoted by $\widehat{\pi}(t)$.

Scheduling Rule: (a) The current priority of every task is equal to its assigned priority at the time of release except under conditions stated in priority inheritance rule.

(b) Every ready job is scheduled preemptively according to its current priority

Allocation Rule: A request for a resource R by a task, results in one of the following two conditions:

(a) If R is held by another task, the request fails and the requesting task is blocked

(b) If R is free then:

1. If the requesting task's priority is higher than the current priority ceiling $\widehat{\pi}(t)$ of the system, R is allocated to it.
2. If the current priority of the

requesting task is not higher than $\widehat{\pi}(t)$, the request is denied and the task is blocked.

Except if the requesting task is holding the resource(s) whose priority ceiling $\pi(R)$is equal to the priority ceiling of the system $\widehat{\pi}(t)$, in which case the resource is allocated to the requesting task.

Priority Inheritance Rule: When a task T_1 gets blocked, the task T_2 that blocks it inherits the current priority of T_1.

T_2 executes at the inherited priority until it releases every resource

whose priority ceiling is equal to or higher than the inherited priority of T_2

At this time the priority of T_2 returns to the priority that it held when it acquired the resource R.

Note: Rule 2 assumes that only one task holds all the resources with priority ceiling equal to $\widehat{\pi}(t)$

Rule 3 assumes that only one task is responsible for another task's request being denied because it holds the requested resource or a resource with a priority ceiling $\widehat{\pi}(t)$.

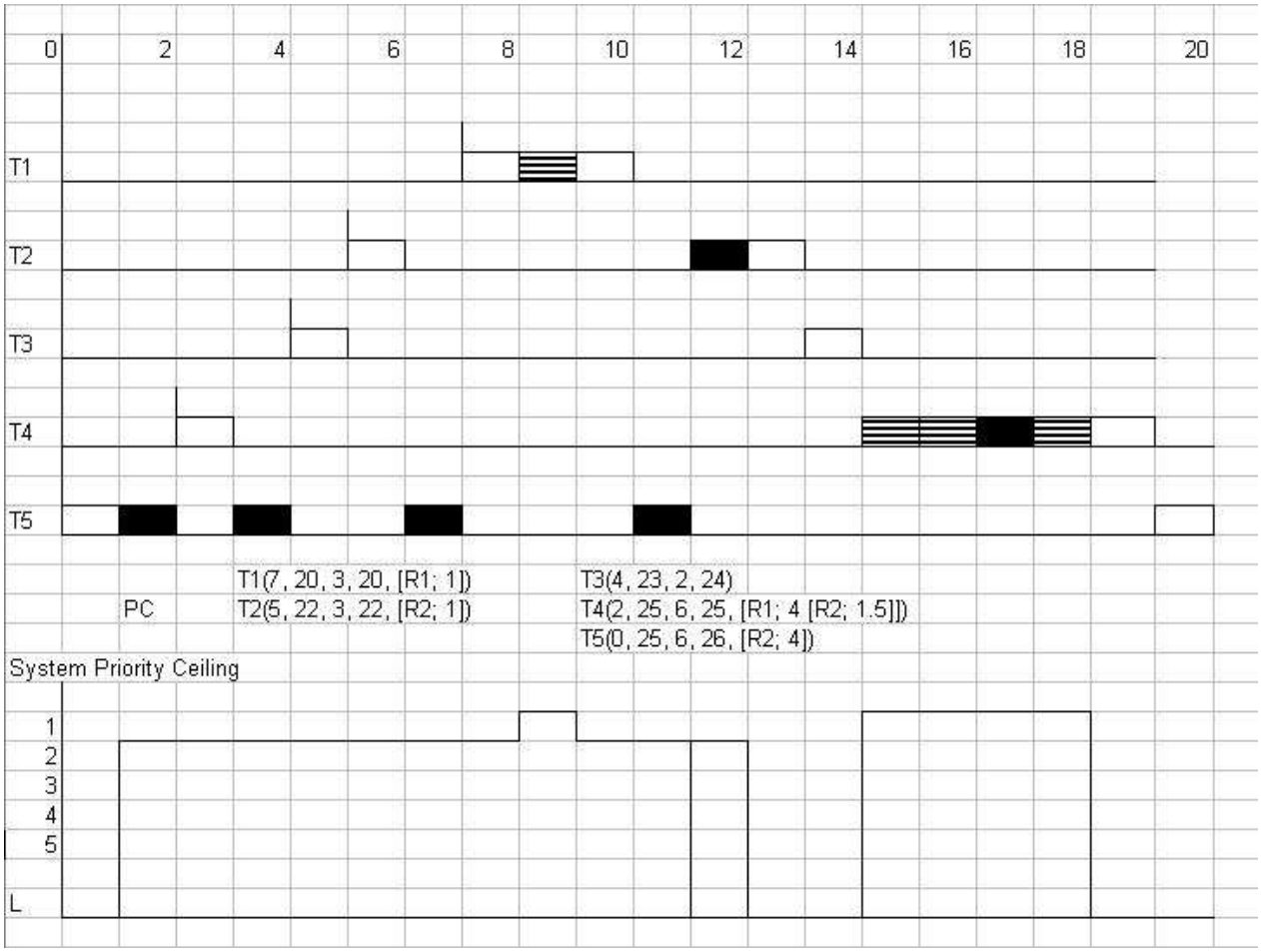


Figure 12: Priority Ceiling

Soft Real-Time (SRT) Scheduling Concepts (Ref: Multiamedia systems by Steinmetz, Ralf; Nahrstedt, Klara)

- Based on traditional real-time systems and their models discussed earlier
- They aim to satisfy soft real-time requirements
- Several differences:
  - SRT systems designed to run on general purpose platforms to support multimedia applications compared to special purpose platforms for HRT system

- SRT systems generally co-exist with a large number of other time-sharing tasks
- A number of SRT applications with varying timing and resource requirements may be required to co-exist and cooperate in order to share limited physical resources

- Two basic directions:

1. Reservation based systems: provide timing guarantees even in overload situations
2. Adaptation based systems: provide the best possible timing guarentees but achieve dynamic

re-allocation of resources, if necessary and deliver graceful degradation of multi-media applications

- What determines the timings that a SRT system must guarentee? - QoS
- What is Quality of Service (QoS)?
  - ISO defined QoS as a concept for specifying how good an offered networking service is
  - Quality of service indicates the defined and controlling behaviour of a service expressed through quantitative measurable parameters.

- Thge result is a "quality-controllable service"
- Layering of QoS, QoS specification languages and programming, QoS management, relation between QoS and resource management
- For networked multimedia systems QoS concept was extended to a layered architecture
- Figure on the board
- This model differentiates among users, application, system and individual device components
- Perceptual QoS:

    - Description of perceptive qualities (good, bad, excellent)
    - Window size (big , small)
    - Response time (interactive, batch)
    - Security (high, low)
    - Description of pricing choices: Users can specify range of price they are willing to pay for desired service
  - Application QoS:
    - Media quality (frame rate, frame resolution and their end-to-end delay, jitter)
    - Media relations: transformation,

intra frame synchronization

  * Adaptation rules

- System QoS:
  * Quantitative criteria: Bits per sec, number of errors, task processing time, task period
  * QoS parameters at system level include throughput, delay, response time, data rate, data corruption
  * Qualitative criteria: interstream synchronization, ordered data delivery, error-recovery mechanism, scheduling mechanism etc.

- QoS parameter values and services classes:
  - Guaranteed Service specified by deterministic parameter values at certain times:

$$QoS : T \rightarrow R$$

    where
    T is a time domain representing the lifetime of a service during which QoS should hold and
    R is the domain of positive real numbers representing QoS parameter value
  - The overall QoS deterministic bounds can be specified by:

* A single value (average, contractual, threshold or target values)
* A pair of values $[QoS_{min}, QoS_{max}]$ (e.g. min and average, lowest and target values)

$$Qos_{min} \leq QoS(t) \leq QoSmax$$

* A triple of values e.g. best value QoS_{max}, average value QoS_{ave} and worst value QoS_{min}

– A Pridictable Service is based on past network behaviour

$$B_p = \frac{1}{n} \sum_{i}^{n} B_i$$

where B_i is a past history value

  - The pridictable service class could promise to provide bandwidth B up to B_p
  - Best Effort Service: based on either no or partial guarantees
  - Most of the current computing and communication services are best effort services
  - Multimedia services require guaranteed service class or at least a predictive service class.
- The minimal reservation policy:
  - Based on minimul QoS values
  - Large violations of timing guarantees possible due variations

in multimedia data

  - May be useful multimedia threads have constant data size (Motion JPEG)

- The average reservation policy:
  - Based on average QoS values
  - May cause some timing violations
  - May be useful for multimedia with data causing occeasional processing violations - normally acceptable
- The maximum reservation policy:
  - Based on worst case QoS values
  - Will guarantee all timing

requirements - hard RTS spectrum

- May be used if there are multimedia strings that need strong guarantees (sharing medical info)

Reservation based SRT systems:

- Provides timing guarentees for multimedia applications in any load situation
- Two important steps:
  1. Reservation of CPU bandwidth with its mechanisms and rules
     - Need a CPU broker that

provides:

- Schedulability Test based on the scheduling policy
- CPU reservation operation that reserves CPU bandwidth according to a given policy. The granted reservations are registered in a despatch table
- Quality of service calculation to prepare scheduling and performance parameters required by the CPU scheduler to satisfy new admitted requests

2. Scheduling of CPU bandwidth with its mechanisms and rules

  - Need an SRT CPU Scheduler for:
  - Scheduling mechanisms based on scheduling policies
  - Up-Call Mechanisms for overrun / underrun cases (issue events to applications and the broker for adjustments in resource allocation or negotiation of new timing guarentees)

- Reservation Mechanisms:
- Application thread specifies its timing QoS parameters to the broker through an API
- For example period p or its cpu

utilization U within the period (p = 50ms, U= 40% task needs 20ms in a period of 50ms)

- The broker first performs the schedulability test to confirm if the process can be admitted or not
- If admitted, its timing requirements is guaranteed
- The admitted thread is inserted into the waiting queue with other SRT threads (by adjusting priority)
- The broker computes a new schedule and writes it into the dispatch table.

- If the thread is not admitted, it either exits and tries later to get admitted or can execute as best effort thread with no timing guarantees
- CPU bandwidth partitioned (say 70/30) into for SRT and bst effort threads
- The broker may be implemented as a background daemon process with superuser priviliges running at time sharing dynamic user priority level
- The SRT CPU Schedular process runs at the highest fixed priority so that it is able to preempt any other

SRT or Non RT process

- It wakes up periodically and checks the dispatch table to decide which process needs to be scheduled and dispatched next.

Adaptation based SRT systems

- Usually need to adapt and deal with overrun situations
- Why do overruns occur? - processes indicate only minimal / average requirements - bursts and overloads during processing
- Adjust and gracefully adapt all SRT

processes or

- Adjust only the process that experiences overrun and may cause delays to other processes
- Two types of adaption based systems:
  1. Systems with no reservation
     - No reservation concept
     - Adaptation relies on adaptive applications
     - SRT applications are scheduled with higher priorities
     - In case of overload dynamic feedback exists to adaptive applications to allow them to

gracefully adapt to current load

2. Adaptive reservation systems
   - Relies on changing the reservations for SRT threads when overruns occur and deadlines are violated.
   - The reservation can be adapted by CPU broker or by the application

Comparasion

- The reservation based systems provide better handling of timing guarantees

- Reservation usually based on worst case QoS values
- Lower resource utilization - inflexible allocation
- Violation of fairness possible
- Adaptation based systems show high resource utilization and fairness
- Timing guarentees in underload and normal load situations
- Adaptation in overload situations
- Possible violations of deadlines in overload situations
- Dependency on adaptive capabilities of running applications