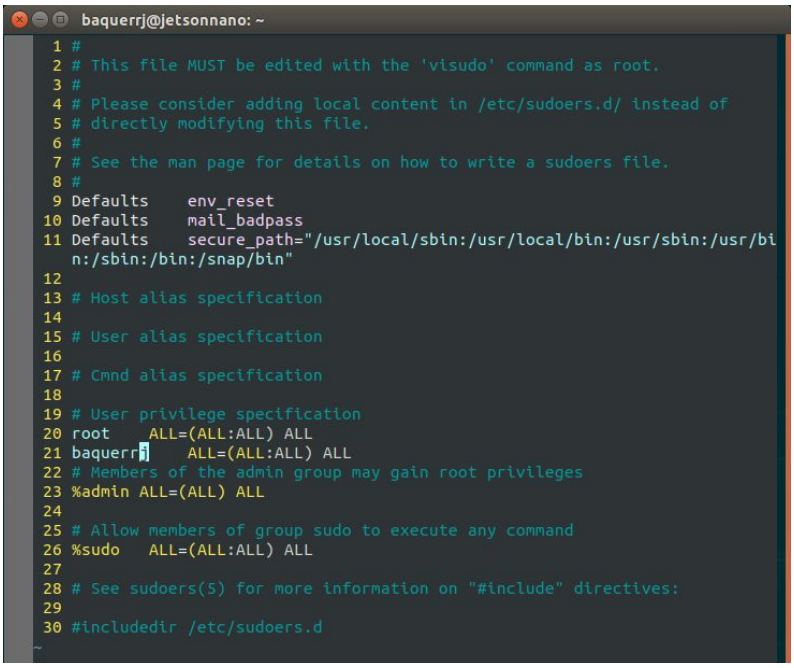
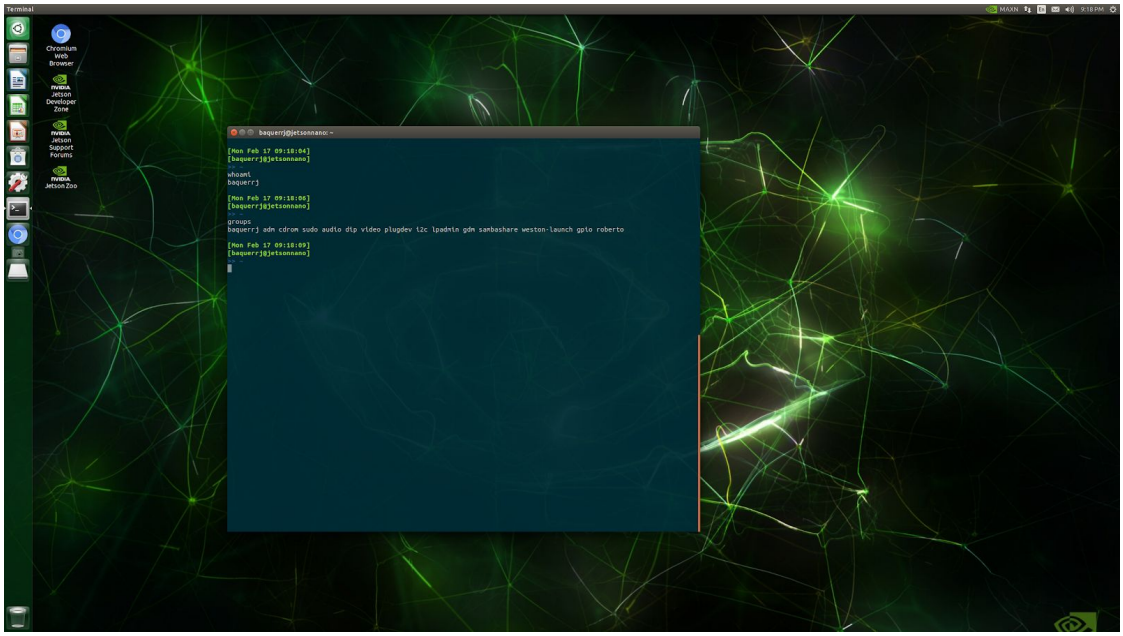
Chutao Wei
Roberto Baquerizo
February 8, 2020

Exercise 2

Problem 1

The screenshots below show the process of adding the user ‘baquerrj’ as a sudoer on the Jetson Nano Dev Kit using visudo. The second screenshot below shows the desktop after logging in as the newly created ‘baquerrj’ user.





Chutao Wei
Roberto Baquerizo
February 8, 2020
PROBLEM 2

The paper “Architecture of the Space Shuttle Primary Avionics Software System” describes what it claims is the most complex flight computer program ever developed. The Primary Avionics Software System (PASS) was developed by the IBM Federal Systems Division for NASA’s Space Shuttle program. The success of PASS underscores the benefits of establishing a well-structured system architecture at the start of the development process. The paper goes on to describe the structure of PASS itself, along with the architectural drivers for PASS and the subsystems and interfaces of PASS.

In addition to the critical timing and redundancy management requirements of the avionic system design, data processing system general purpose computer design and memory/CPU constraints; multi-computer redundancy management and synchronization; the operational sequencing/mode control and man/machine interface requirements; the applications functional and performance requirements; and design modularity and flexibility requirements were taken into account in the development of PASS, thereby driving the architecture.

The PASS architectural structure requires a blend of operational, reliability, and functional requirements along with several physical constraints. The impact of these considerations and constraints is evident in the memory structure of the PASS. The main memory was not large enough to accommodate all of the software required to satisfy all the requirements for the PASS. Therefore, memory was split up into a mass memory unit and a main memory with a capacity of 106K 32-bit words. The mass memory unit was a serial access device, rather than a direct-access storage device we would be more familiar with today. PASS is segmented into eight different phase/function combinations, each of which is identified with a unique operational sequence (OPS). The software that is needed to execute each OPS is loaded into the main memory of a General Purpose Computer (GPC) from the mass memory. Due to reliability and redundancy considerations there is a requirement for the code required to execute an OPS to be totally resident in each redundant GPC main memory. For those reasons, the PASS is stored in main memory in such a manner that minimizes the amount of code and data that must be transferred to main memory when a new OPS is initiated. Thus, each OPS has three parts in memory: the resident software that contains code and data common to all OPS loads; the major function base, which contains code and data common to major applications functions used in more than one OPS load; and the OPS overlay, which contains the code and data unique to an OPS load. Flight phase sequencing and memory structure was architected in such a way that in the best-case scenario only an OPS overlay would have to be loaded when a new OPS is initiated.

MAN/MACHINE INTERFACE

The man/machine interface of the PASS was structured to accommodate a knowledgeable user such that the crew could communicate with the computers to monitor and control the Orbiter avionics system with minimum time and effort. This interface was implemented as a substructure of the OPS, and facilitated by control segments. The OPS substructure consists of major modes, specialist

functions, and display functions. Control segments consist of a series of standardized logic blocks establishing the structure of an OPS, major mode, specialist functions, or display functions.

Major modes are substructured into blocks linked to CRT displays. Each OPS has one or more modes, and each mode has one or more blocks. Sequencing from one mode, or block, to another occurred automatically, event-driven by software, or by keyboard entry from the crew. The next substructure in the OPS are the specialist functions (SPEC). A SPEC is initiated within an OPS only upon keyboard entry from the crew, unlike the major modes. The SPEC executes independently and concurrently with other processes of that OPS. Like the major modes, the SPEC is also linked to the CRT displays. The final element of the OPS substructure are the display functions (DISP). A DISP is only used to monitor the results of a SPEC or major mode function, and so no processing function is initiated by the DISP.

Although flight software systems are characterized by their small size and limited number of functions, it became clear to the PASS architects that the requirements of the Shuttle avionics system would not be satisfied by a small and limited flight software. Shuttle computers would perform an increased number and variety of functions. In addition, there was the expectation that requirements could change, so they had to decide on an adaptable architecture. Synchronous architectures, with a relatively simple operating system or executive, have limited flexibility. Therefore, an asynchronous architecture was designed for the PASS.

The Flight Computer Operating System (FCOS) was responsible for the management and control of the internal resource of the GPC and external interfaces. The variety of jobs performed by the FCOS could be grouped into three major functions: (1) process management, (2), IO management, and (3) DPS configuration management.

Using a multitasking priority queue structure, the FCOS schedules and allocates CPU resources. It ensures that the highest priority system or application process is given the CPU resources it requires when it is ready to run. Under the process management umbrella, the FCOS controls the allocation of all internal computer resources.

Like the process management function controls the CPU, the IO management function controls the allocation of IO processor resources. The IO processor software of each GPC is considered part of the IO management function of the FCOS. The IO management function ensures that redundant computers receive identical data from the hardware, and it also performs IO fault isolation and correction.

The DPS configuration management function controls the loading of the GPC memories, sequencing, and the GPC and IOP operating states. It is responsible for hardware initialization and status checks. This function performs all transfers of code or data between the mass and main memories, including

program overlay and mass memory modification during flight. The configuration function also establishes the redundant computer set for support of critical mission phases. However, management of redundant set membership is a manual step performed by the crew, e.g. a GPC may be “voted” out, but to drop that computer from the redundant set requires crew confirmation.

System control performs the initialization and configuration control of DPS and the associated avionic data network. Other functions performed by system control include several system-leveL SPECs that the crew can utilize. These include resetting the master timing unit (MTU), initiating a memory dump, examining or changing a core location in memory, or changing the configuration of the DPS.

The user interface of the PASS consisted of keyboard entry and display, which were accomplished with on-board software and off-line processors. The off-line processors translate relevant data into parametric data tables for use by the on-board software. Through interpretation of user input commands received via keyboard entry and process control commands, the user interface performed the initiation, sequencing, and termination of other systems or applications processes. From the user interface, the crew could initiate any of the system-level SPECs discussed earlier to, for example, change the configuration of the DPS.

The on-board software that facilitates the user interface was developed for three major applications: (1) guidance, navigation, and control (GN&C), (2) vehicle system management (SM), and (3) vehicle checkout (VCO).

GN&C software determines vehicle position, velocity, and attitude. It performs sensor redundancy management, and provides the crew with isplays and data entry capabilities to monitor and control the avionics subsystems. It is implemented a cyclic closed loop application with tight timing and phasing relationships. The cyclic process, called the executive, controls the initiating and phasing of the principle functions and associated IO. Three executives are used to ensure all critical flight control processing is completed in a 40 millisecond minor cycle. The high frequency executive is scheduled at a high priority with an execution rate of 25Hz. This one initiates all principal function processes directly related to vehicle flight control. Mid-frequency and low-frequency executives are scheduled at lower priorities, and operate at rates of 6.25Hz down to 0.25Hz. Processes that run at 0.25Hz could be responsible for display updates.

The SM application provided the crew with status monitoring and controls for orbiter subsystems not directly involved with vehicle flight control. It monitors payload subsystems and alerts the crew if it detects any abnormalities. It employs multi-frequency executives as well. There is a 5Hz executive for data acquisition, another 5Hz executive dispatcher for special functions, and a 1Hz process for parameter preconditioning, and fault detection and annunciation.

Chutao Wei
Roberto Baquerizo
February 8, 2020

The VCO application was designed to support avionics system initialization and checkout under control of the crew. The primary functions of the VCO application were configured into three ground checkout OPSs and one in-flight checkout OPS. The VCO provided the software support for the testing, integration, and certification of the Orbiter avionics subsystems during vehicle preparation on the ground and in-flight orbit coast period prior to re-entry.

ADVANTAGES OF FREQUENCY EXECUTIVE

1. There is no overhead from context switching, because preemption of tasks is not supported in the frequency executive method. In the case of real-time threading where higher priority tasks can preempt lower priority tasks, the overhead of context switching is not zero and cause an overloading of the system if the utilization margin is not large enough.
2. As long as the number of tasks isn't too large and their associated frequencies too varied, the frequency executive method should be easier to implement than a real-time scheduler with real-time threading and preemption.
3. As the tasks in the frequency executive method run in a hyperloop, the resulting schedule is very deterministic and predictable, when compared to a real-time software scheduler implementing real-time threading with preemption.

DISADVANTAGES OF FREQUENCY EXECUTIVE

1. The frequency executive is not very flexible. If any of the parameters for a task change, or if a new task has to be added to the task table, the whole executive would likely have to be re-architected. In the case of a real-time threading scheduler, changing priorities for threads or adding a new thread is not as difficult.
2. In the frequency executive method, all tasks that could execute at the same time must be analyzed together in order to ensure there are enough resources available, and priorities are assigned ad-hoc in this way. In the case of a real-time threading scheduler, priorities are assigned according to the parameters of a task (computation time, deadline, period, etc.).
3. Because there is no preemption, a lower priority task that was initiated due to some event cannot be preempted by a higher priority task. This could lead to starvation of CPU resources.

Chutao Wei
Roberto Baquerizo
February 8, 2020
PROBLEM 3

Overall understanding of paper and key point articulation

First we need to define and understand what Cyclic Executive is. As its name suggests, Cyclic Executive is an infinite loop inside of main(). Its scheme is to run through a sequence of functions at a fixed interval. The loop inside of the main() is usually called the major cycle. There should be only one major cycle for each system. Major cycle consists of a fixed number of minor cycles, which are also periodic. Inside of each minor cycle, there are frames with variable durations. Each frame has a process associated with it.

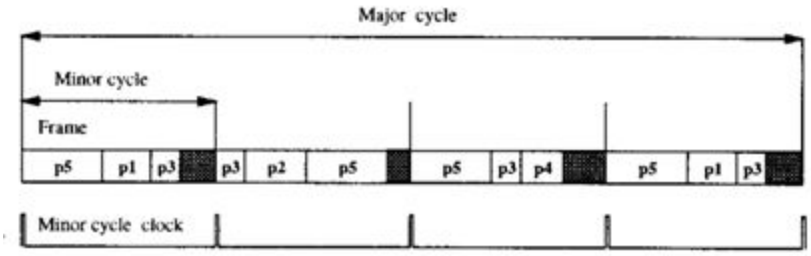


Fig. 1. Cyclic scheduler structure

The Figure above shows the relationship visually. Each minor cycle or frame has its associated deadline. In case of error or other events, exceptions are triggered. CPU runs exception handler instead of the normal code in the major cycle.

Comparison to Linux and RTOS approaches

Cyclic Executive keeps timing very strict in each of its loop hierarchy to achieve hard real time. A good part of Cyclic Executive is that you potentially have less context switching time. Programmers would have the ability to control the margin you want in each minor cycle. Since there isn't a large OS running behind all of this, programmers got more control over how to accomplish context switching. However, the system cannot have huge exception overload. Otherwise minor cycles can be easily overrun. The other downside is that the programmer needs to efficiently do the task scheduling yourself, based on the information you got for period, deadline and computation time. The CPU utilization probably would not end up very high.

Linux POSIX RT is still based on Linux but its scheduler is changed to SCHED_FIFO or SCHED_DEADLINE. SCHED_FIFO is the basically fixed priority first-in-first-out. SCHED_DEADLINE is the earliest deadline first. Linux POSIX RT does not provide a lot of other scheduling schemes for us. Linux is probably much familiar to most software programmers, which reduces development time. However, there are a lot of implementation details hidden by Linux POSIX

API. Sometimes, the methods implemented by Linux POSIX API should not be used for a real time system, due to unpredictable runtime.

Other RTOS are more flexible about how you go with scheduling and how to schedule tasks more efficiently. For example, FreeRTOS used fixed priority by default. You can change the scheduler if you wish. If I remembered correctly, there is task.c and task.h you can edit. All Freertos API are written based on the proposition that "FreeRTOS never performs a non-deterministic operation, such as walking a linked list, from inside a critical section or interrupt"[1]. Whereas, Linux is never based on this proposition. In RTOS, we evaluate determinism over CPU utilization in order to make sure our system reaching deadlines.

[1] https://www.freertos.org/freertos/quality.html

EXPLANATION OF CODE

COMPLETION-TIME TEST

This test is governed by the formula shown below.

$$a_n(t) = \sum_{j=1}^{n} \left\lceil \frac{t}{T_j} \right\rceil C_j$$

$a_n(t)$ is the total cumulative demand from the n tasks up to time t. Passing this test requires proving that $a_n(t)$ is less than or equal to the deadline for S_n, which proves that S_n is feasible. Proving this same property for all S from S_1 to S_n proves that the service set is feasible[2]. We need to run through all possible n value. The code explanation is added into the comments below

[2] Pratt, John_ Siewert, Sam - Real-time embedded components and systems, pg 87

```
int i, j; // i is n in the equation above, j is the same as the equation
U32_T an, anext; // an is an(t), anext is single term for particular n index

// assume feasible until we find otherwise
int set_feasible = TRUE;

// this is the for loop that goes
// through all a1(t), a2(t) ..... an(t) tests
for ( i = 0; i < numServices; i++ )

{
   // initalize them to be zeroes
   an    = 0;
   anext = 0;

   // start from Service0(S0) all the way to current i value
   for ( j = 0; j <= i; j++ )
   {
      // cumulative adding wect to an, to get the total
      // computation time up to i tasks
      an += wcet[ j ];
   }
   while ( 1 )
   {
      // add  wcet for task i
      anext = wcet[ i ];

      // this is the right hand side of the formula
      // it is the sum of the demand for all tasks from 0 to i-1
      for ( j = 0; j < i; j++ )
         anext += ceil( ( (double)an ) / ( (double)period[ j ] ) ) * wcet[ j ];
      // check if anext is the same as the total worst case excution time
      if ( anext == an )
         break;
      // else replace an with anext
      else
         an = anext;
   }
   // if demand is larger than deadline, no way this is feasible
   if ( an > deadline[ i ] )
   {
      set_feasible = FALSE;
   }
}
return set_feasible;
```

Chutao Wei
Roberto Baquerizo
February 8, 2020

Scheduling Point Test

According to Lehoczky, Sha, Ding theorem and Liu and Layland's paper, the worst case scenario is all services are requested at the same time. Thus, if we prove that the critical period is feasible, then the whole service set is feasible. The book gives a mathematical formula to determine this condition.

$$\forall i, 1 \le i \le n, \min \sum_{j=1}^{i} C_j \left\lceil \frac{(l)T_k}{T_j} \right\rceil \le (l)T_k$$

$$(k,l) \in R_i$$

$$R_i = \left\{ (k,l) \middle| 1 \le k \le i, l = 1, \ldots, \left\lfloor \frac{T_i}{T_k} \right\rfloor \right\}$$

- Where n is the number of tasks in the set S_i to S_n, where S_1 has higher priority than S_2, and S_n has higher priority than $S_{n>1}$.
- j identifies S_j, a service in the set between S_1 and S_n.
- k identifies S_k, a service whose l periods must be analyzed.
- l represents the number of periods of S_k to be analyzed.
- $\left\lceil \frac{(l)T_k}{T_j} \right\rceil$ represents the number of times S_j executes within l period of S_k.

  $C_j \left\lceil \frac{(l)T_k}{T_j} \right\rceil$ is the time required by S_j to execute within l periods of S_k—

Code is commented in the screenshot below.

```
int rc = TRUE, i, j, k, l, status, temp;
// iterate from highest to lowest priority
for ( i = 0; i < numServices; i++ )
{
   // status is false so that it does not break in the for loop
   status = 0;
   // loop through all 0<=k<=i, we start from 0
   // because service index from 0
   for ( k = 0; k <= i; k++ )
   {
      // left hand of inequalitym, also include the code
      // for figure out upper bound of l
      for ( l = 1; l <= ( floor( (double)period[ i ] / (double)period[ k ] ) ); l++ )
      {
         temp = 0;

         // time required by Sj to execute within l periods of Sk
         for ( j = 0; j <= i; j++ )
            temp += wcet[ j ] * ceil( (double)l * (double)period[ k ] / (double)period[ j ] );
         // if the sum of these times for the set of tasks is smaller than l
         // period of Sk, then the service set is feasible
         if ( temp <= ( l * period[ k ] ) )
         {
            status = 1;
            break;
         }
      }
      // just fall out of for loop since feasible
      if ( status )
         break;
   }
   // just fall out of for loop since feasible
   if ( !status )
      rc = FALSE;
}
// return success
return rc;
```

Chutao Wei
Roberto Baquerizo
February 8, 2020

Earliest Deadline First Scheduling Simulation

The program goes through second-by-second(the smallest time interval) check on the remaining computation times for all tasks and deadlines to make sure EDF is feasible.

```
// Delcare all variables used
U32_T lcm = getLcmOfPeriods( numServices, period );
U32_T min;
U32_T i;
U32_T k;
U32_T remainingComputationTime[ numServices ];
U32_T timeUntilDeadline[ numServices ];

// assign values for every service
for ( i = 0; i < numServices; ++i )
{
   remainingComputationTime[ i ] = wcet[ i ];
   timeUntilDeadline[ i ]        = deadline[ i ];
}

// We assume 1 second is the smallest interval for
// checking if the service is the earliest deadline
// every second.
// This for loop dose loop every second until LCM.
for ( k = 1; k <= lcm; k++ )
{
   // get the earliest deadline service
   min = getIndexOfMin( timeUntilDeadline, numServices );
   // now computation time for the earliest deadline
   // service is advanced by 1 second
   if ( remainingComputationTime[ min ] > 0 )
   {
      remainingComputationTime[ min ]--;
   }
   // loop over all possible services to update status
   for ( i = 0; i < numServices; i++ )
   {
```

```
        // This just loop over make sure when we reach
        // LCM, we set deadline back to LCM, so that
        // the test can complete without error
        if ( remainingComputationTime[ i ] == 0 )
        {
            timeUntilDeadline[ i ] = lcm;
        }
        if ( timeUntilDeadline[ i ] > 0 )
        {
            // decrement deadline for all tasks since
            // we advanced a second
            timeUntilDeadline[ i ]--;
        }
        // Check if all services are able to finish computing before
        // deadline
        if ( timeUntilDeadline[ i ] < remainingComputationTime[ i ] )
        {
            // if cannot finish computting before deadline, return FALSE
            return FALSE;
        }
        // update timeUntilDeadline and remaining computation time
        if ( k % period[ i ] == 0 )
        {
            timeUntilDeadline[ i ]         = deadline[ i ];
            remainingComputationTime[ i ] = wcet[ i ];
        }
    }
}
return TRUE;
```

Chutao Wei
Roberto Baquerizo
February 8, 2020

Least Laxity First Scheduling Simulation

This program is almost the same as EDF simulation. Instead of associating priority with deadline, we associate priority with laxity, which is pretty much the only thing we changed in code.

```
// Delcare all variables used
U32_T lcm = getLcmOfPeriods( numServices, period );
U32_T min;
U32_T i;
U32_T k;
U32_T remainingComputationTime[ numServices ];
U32_T timeUntilDeadline[ numServices ];
U32_T laxity[ numServices ];
// assign values for every service
for ( i = 0; i < numServices; ++i )
{
   remainingComputationTime[ i ] = wcet[ i ];
   timeUntilDeadline[ i ]        = deadline[ i ];
}
// We assume 1 second is the smallest interval for
// checking if the service is the earliest deadline
// every second.
// This for loop dose loop every second until LCM.
for ( k = 1; k <= lcm; ++k )
{
   // compute laxity for every services
   for ( i = 0; i < numServices; ++i )
   {
      laxity[ i ] = timeUntilDeadline[ i ] - remainingComputationTime[ i ];
   }
   // find the least laxity service
   min = getIndexOfMin( laxity, numServices );

   // Now computation time for the earliest deadline
   // service is advanced by 1 second, so remaining is
   // subtracted by 1 second
   if ( remainingComputationTime[ min ] > 0 )
   {
      remainingComputationTime[ min ]--;
   }
```

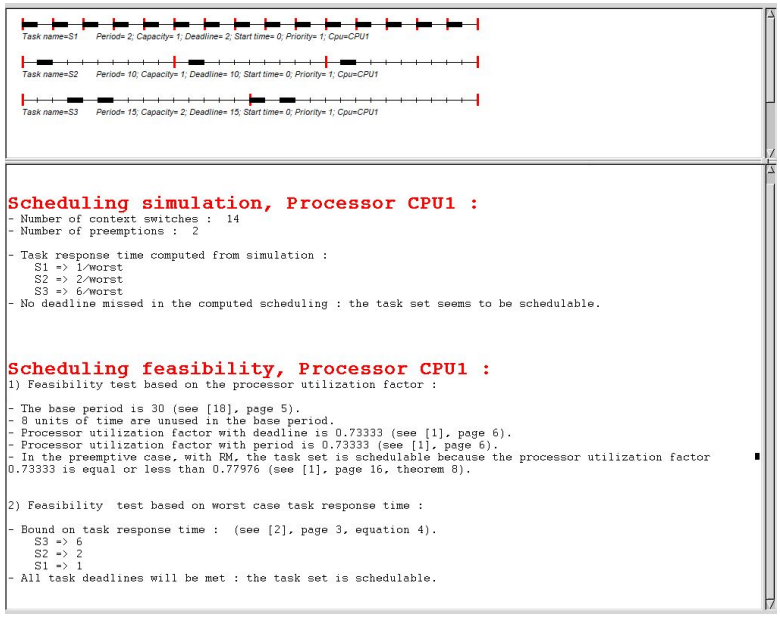
```
    // loop over all possible services to update all service status
    for ( i = 0; i < numServices; i++ )
    {
        // This just loop over make sure when we reach
        // LCM, we set deadline back to LCM, so that
        // the test can complete without error
        if ( remainingComputationTime[ i ] == 0 )
        {
            timeUntilDeadline[ i ] = lcm;
        }
        // Update deadline
        if ( timeUntilDeadline[ i ] > 0 )
        {
            // decrement deadline for all tasks since
            // we advanced a second
            timeUntilDeadline[ i ]--;
        }
        // Check if all services are able to finish computing before
        // deadline
        if ( timeUntilDeadline[ i ] < remainingComputationTime[ i ] )
        {
            // if cannot finish computting before deadline, return FALSE
            return FALSE;
        }
        // update timeUntilDeadline and remaining computation time if
        // service is requested again
        if ( k % period[ i ] == 0 )
        {
            timeUntilDeadline[ i ]        = deadline[ i ];
            remainingComputationTime[ i ] = wcet[ i ];
        }
    }
}
return TRUE;
```

Chutao Wei
Roberto Baquerizo
February 8, 2020
Cheddar Analysis for Service Sets

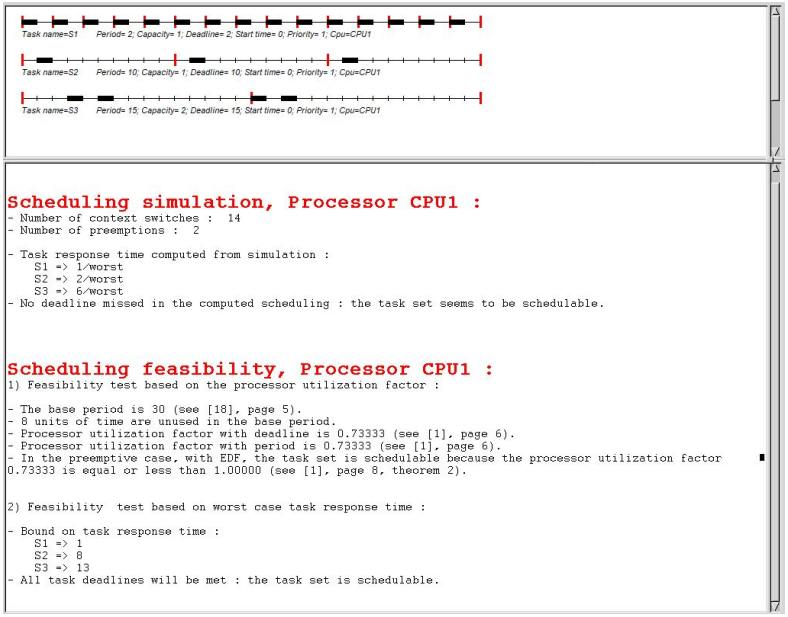
Service Set 0

Example 0	T1	2	C1	1	U1	0.5	LCM =	30
	T2	10	C2	1	U2	0.1		
	T3	15	C3	2	U3	0.133333	Utot =	0.733333

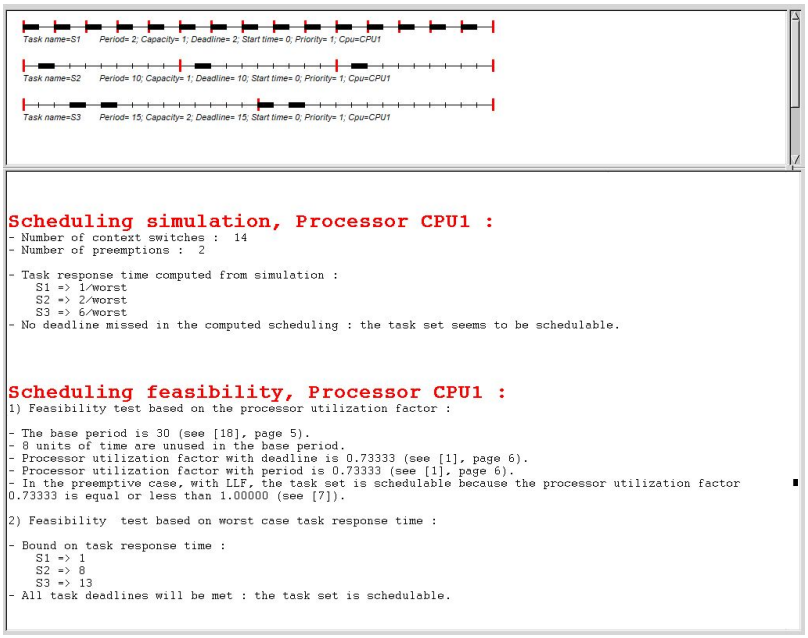
Rate Monotonic Analysis



Earliest Deadline First



Chutao Wei
Roberto Baquerizo
February 8, 2020
Least Laxity First



Scheduling simulation, Processor CPU1 :

- Number of context switches : 14
- Number of preemptions : 2

- Task response time computed from simulation :
  S1 => 1/worst
  S2 => 2/worst
  S3 => 6/worst
- No deadline missed in the computed scheduling : the task set seems to be schedulable.

Scheduling feasibility, Processor CPU1 :

1) Feasibility test based on the processor utilization factor :

- The base period is 30 (see [18], page 5).
- 8 units of time are unused in the base period.
- Processor utilization factor with deadline is 0.73333 (see [1], page 6).
- Processor utilization factor with period is 0.73333 (see [1], page 6).
- In the preemptive case, with LLF, the task set is schedulable because the processor utilization factor 0.73333 is equal or less than 1.00000 (see [7]).

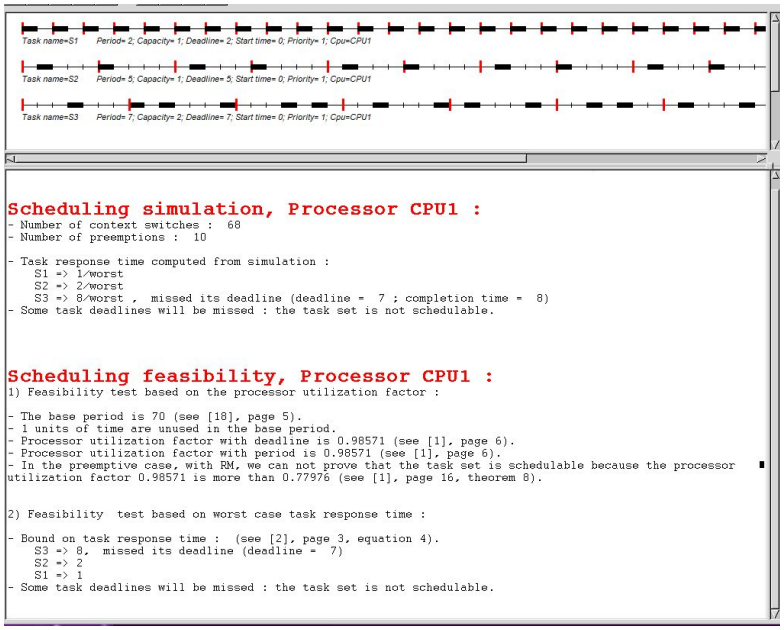
2) Feasibility test based on worst case task response time :

- Bound on task response time :
  S1 => 1
  S2 => 8
  S3 => 13
- All task deadlines will be met : the task set is schedulable.

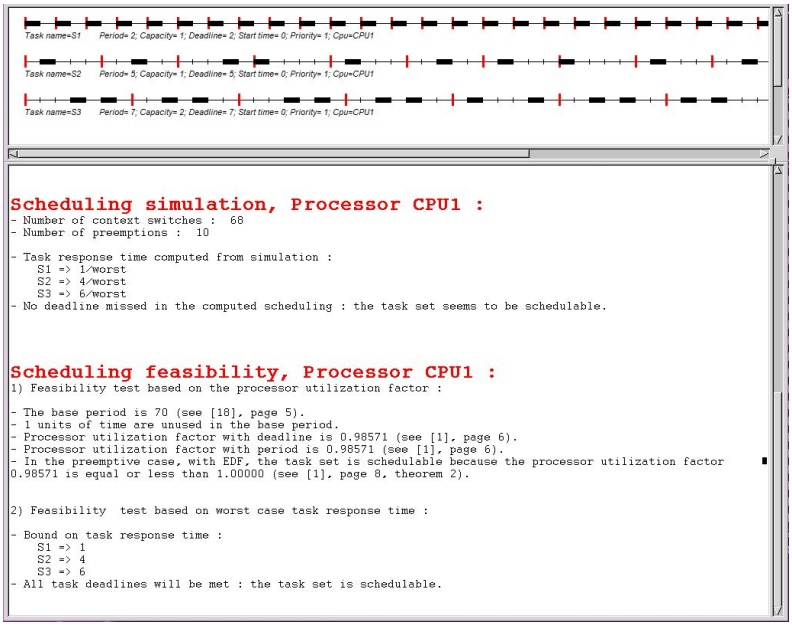
Chutao Wei
Roberto Baquerizo
February 8, 2020
SERVICE SET 1

Example 1	T1	2	C1	1	U1	0.5	LCM =	70
	T2	5	C2	1	U2	0.2		
	T3	7	C3	2	U3	0.285714	Utot =	0.985714

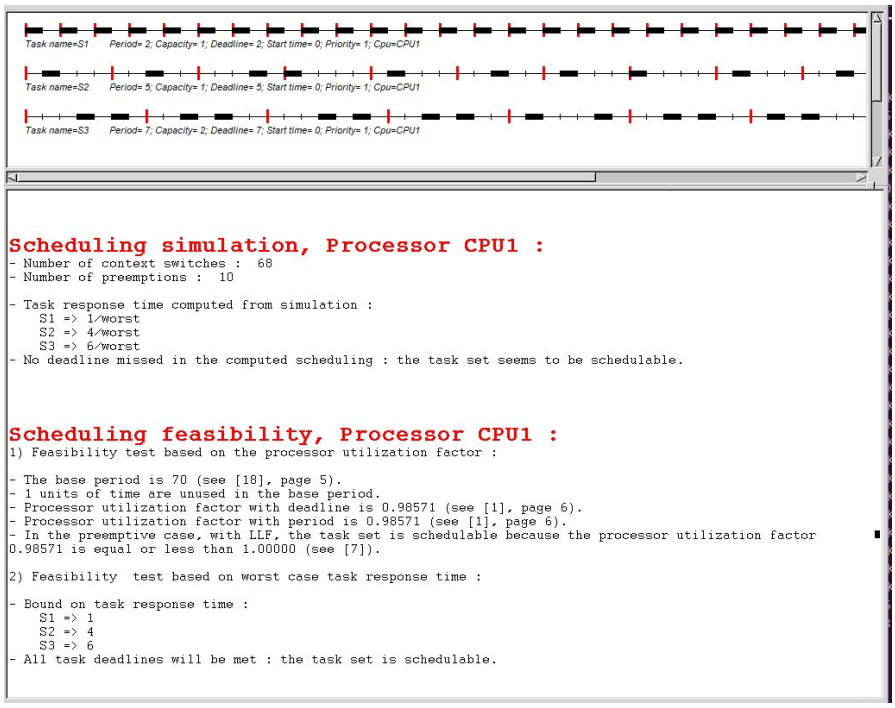
Rate Monotonic Analysis



Earliest Deadline First



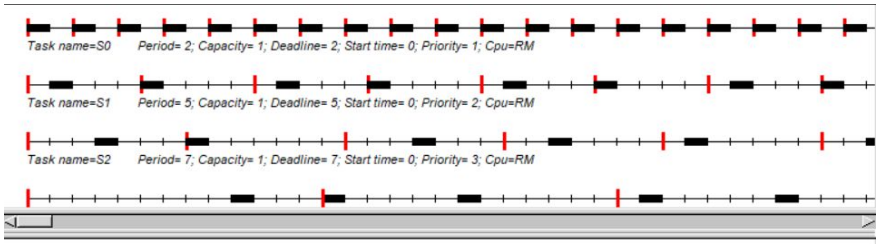
Chutao Wei
Roberto Baquerizo
February 8, 2020
Least Laxity First



Chutao Wei
Roberto Baquerizo
February 8, 2020
SERVICE SET 2

Example 2	T1	2	C1	1	U1	0.5	LCM =	910
	T2	5	C2	1	U2	0.2		
	T3	7	C3	1	U3	0.142857		
	T4	13	C4	2	U4	0.153846	Utot =	0.996703

Rate Monotonic Analysis

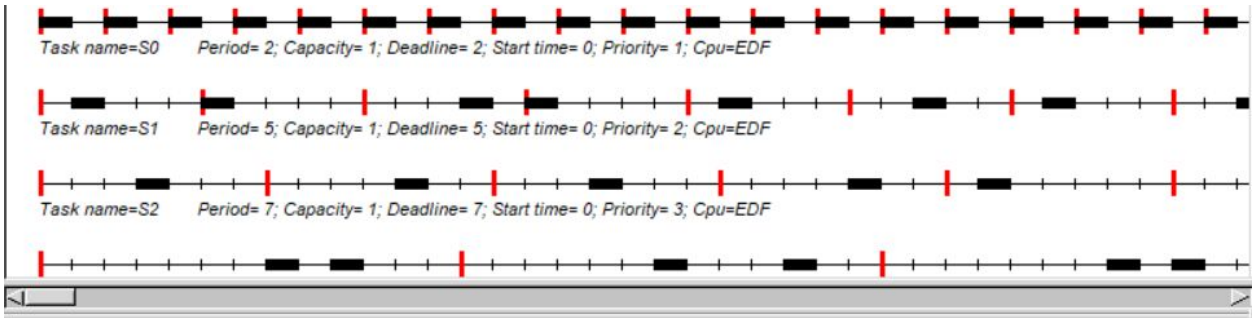


Scheduling simulation, Processor RM :

- Number of context switches : 904
- Number of preemptions : 70

- Task response time computed from simulation :
  S0 => 1/worst
  S1 => 2/worst
  S2 => 4/worst
  S3 => 16/worst , missed its deadline (deadline = 13 ; completion time = 14), missed its deadline (deadline = 26 ; completion time = 28), missed its deadline (deadline = 39 ; completion time = 40), missed its deadline (deadline = 52 ; completion time = 54), missed its deadline (deadline = 65 ; completion time = 68), missed its deadline (deadline = 78 ; completion time = 80), missed its deadline (deadline = 117 ; completion time = 118), missed its deadline (deadline = 377 ; completion time = 378), missed its deadline (deadline = 403 ; completion time = 404), missed its deadline (deadline = 416 ; completion time = 418), missed its deadline (deadline = 429 ; completion time = 430), missed its deadline (deadline = 663 ; completion time = 664), missed its deadline (deadline = 676 ; completion time = 678), missed its deadline (deadline = 689 ; completion time = 690), missed its deadline (deadline = 767 ; completion time = 768)
- Some task deadlines will be missed : the task set is not schedulable.

Earliest Deadline First

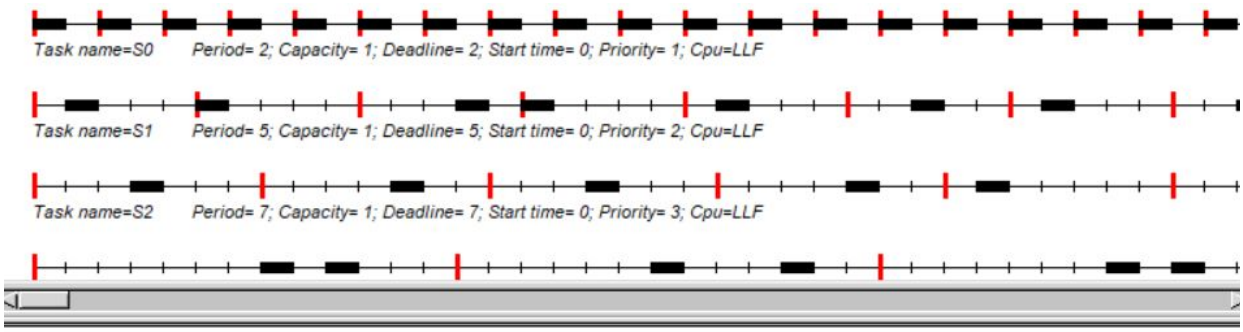


Scheduling simulation, Processor EDF :

- Number of context switches : 904
- Number of preemptions : 70

- Task response time computed from simulation :
  S0 => 1/worst
  S1 => 4/worst
  S2 => 6/worst
  S3 => 12/worst
- No deadline missed in the computed scheduling : the task set seems to be schedulable.

Chutao Wei
Roberto Baquerizo
February 8, 2020
Least Laxity First



Scheduling simulation, Processor LLF :

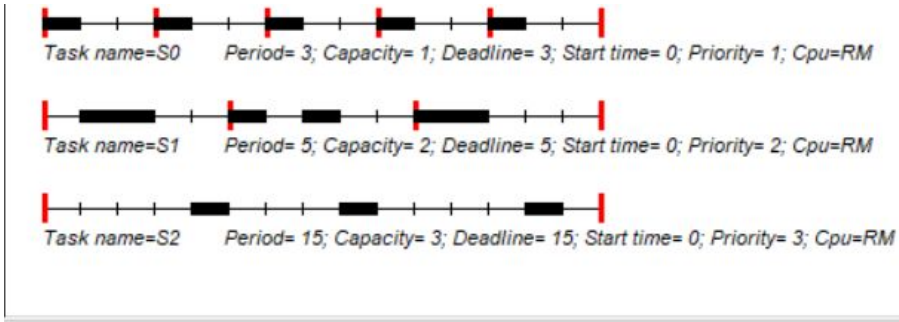
- Number of context switches : 904
- Number of preemptions : 70

- Task response time computed from simulation :
  - S0 => 1/worst
  - S1 => 4/worst
  - S2 => 6/worst
  - S3 => 12/worst
- No deadline missed in the computed scheduling : the task set seems to be schedulable.

Chutao Wei
Roberto Baquerizo
February 8, 2020
SERVICE SET 3

Example	T1	3	C1	1	U1	0.33	LCM =	15
	T2	5	C2	2	U2	0.4		
	T3	15	C3	3	U3	0.2	Utot =	0.93

Rate Monotonic Analysis

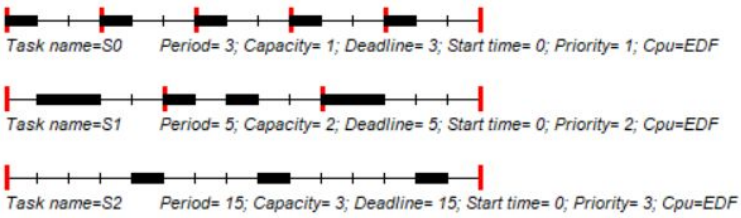


Scheduling simulation, Processor RM :

```
- Number of context switches :  11
- Number of preemptions :  3

- Task response time computed from simulation :
    S0 => 1/worst
    S1 => 3/worst
    S2 => 14/worst
- No deadline missed in the computed scheduling : the task set seems to be
schedulable.
```

Earliest Deadline First

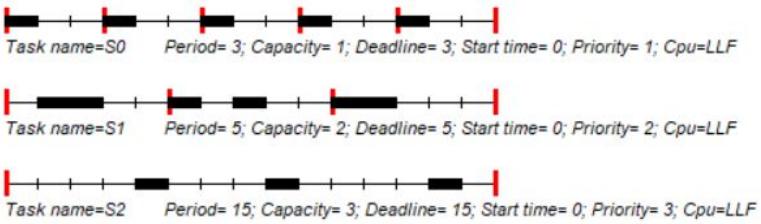


Scheduling simulation, Processor EDF :

```
- Number of context switches :  11
- Number of preemptions :  3

- Task response time computed from simulation :
    S0 => 1/worst
    S1 => 3/worst
    S2 => 14/worst
- No deadline missed in the computed scheduling : the task set seems to be
schedulable.
```

Chutao Wei
Roberto Baquerizo
February 8, 2020
Least Laxity First



Scheduling simulation, Processor LLF :

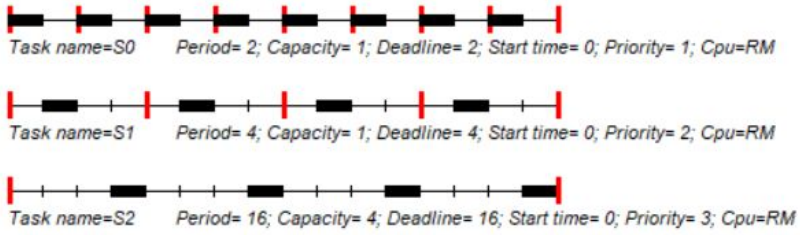
- Number of context switches : 11
- Number of preemptions : 3

- Task response time computed from simulation :
  S0 => 1/worst
  S1 => 3/worst
  S2 => 14/worst
- No deadline missed in the computed scheduling : the task set seems to be schedulable.

Chutao Wei
Roberto Baquerizo
February 8, 2020
Service Set 4

Example 4	T1	2	C1	1	U1	0.5	LCM =	16
	T2	4	C2	1	U2	0.25		
	T3	16	C3	4	U3	0.25	Utot =	1

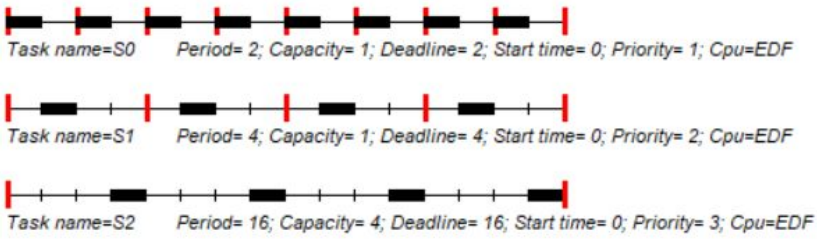
Rate Monotonic Analysis




```
Scheduling simulation, Processor RM :
- Number of context switches :  15
- Number of preemptions :  3

- Task response time computed from simulation :
    S0 => 1/worst
    S1 => 2/worst
    S2 => 16/worst
- No deadline missed in the computed scheduling : the task set seems to be
schedulable.
```

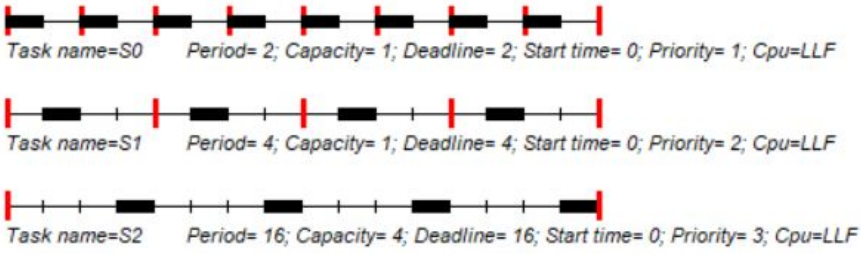
Earliest Deadline First




```
Scheduling simulation, Processor EDF :
- Number of context switches :  15
- Number of preemptions :  3

- Task response time computed from simulation :
    S0 => 1/worst
    S1 => 2/worst
    S2 => 16/worst
- No deadline missed in the computed scheduling : the task set seems to be
schedulable.
```

Chutao Wei
Roberto Baquerizo
February 8, 2020
Least Laxity First



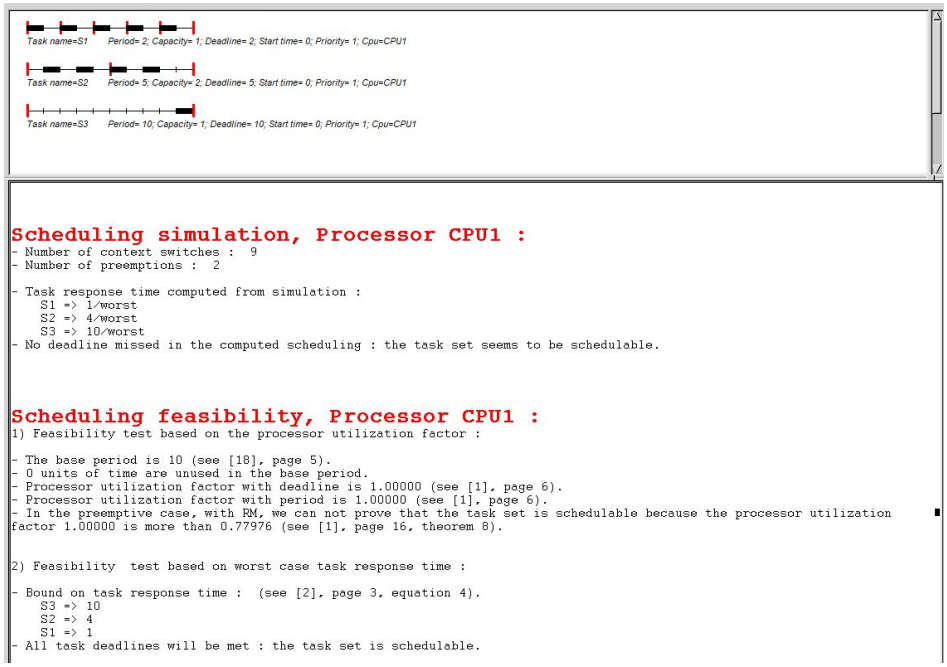

```
Scheduling simulation, Processor LLF :
- Number of context switches :  15
- Number of preemptions :  3

- Task response time computed from simulation :
    S0 => 1/worst
    S1 => 2/worst
    S2 => 16/worst
- No deadline missed in the computed scheduling : the task set seems to be
schedulable.
```

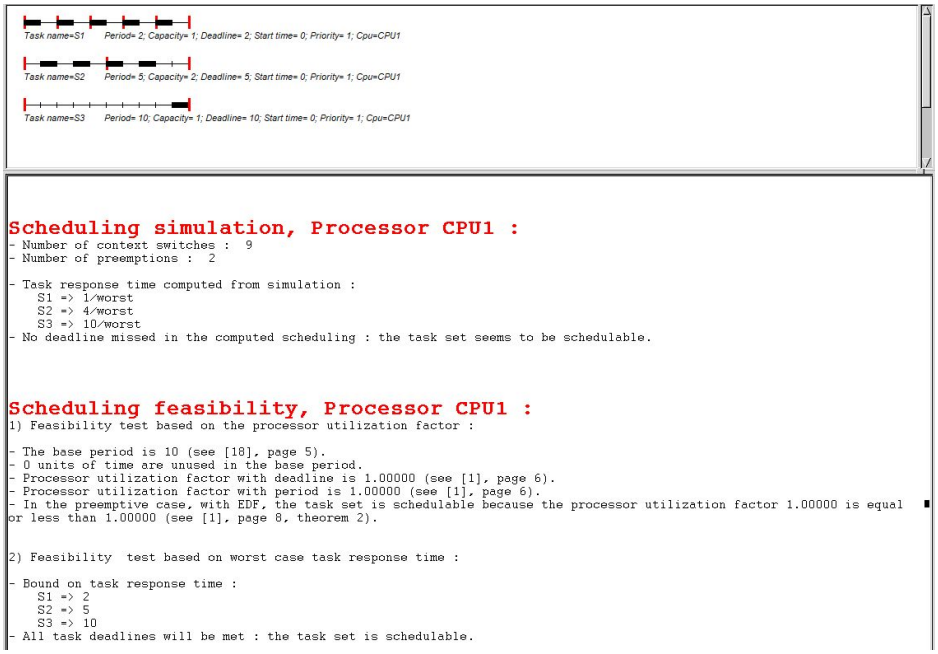
Chutao Wei
Roberto Baquerizo
February 8, 2020
SERVICE SET 5

Example 5	T1	2	C1	1	U1	0.5	LCM =	10
	T2	5	C2	2	U2	0.4		
	T3	10	C3	1	U3	0.1	Utot =	1

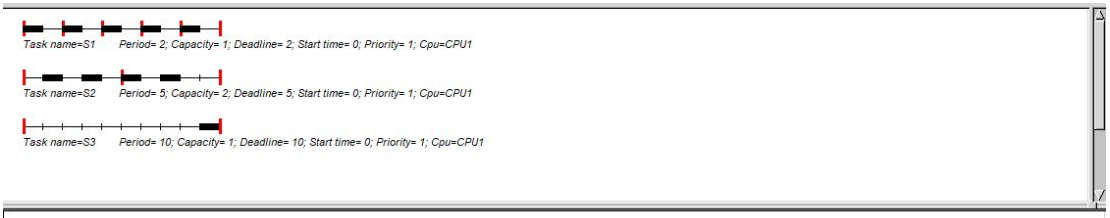
Rate Monotonic Analysis



Earliest Deadline First



Chutao Wei
Roberto Baquerizo
February 8, 2020
Least Laxity First



Scheduling simulation, Processor CPU1 :

- Number of context switches : 9
- Number of preemptions : 2

- Task response time computed from simulation :
  S1 => 1/worst
  S2 => 4/worst
  S3 => 10/worst
- No deadline missed in the computed scheduling : the task set seems to be schedulable.

Scheduling feasibility, Processor CPU1 :

1) Feasibility test based on the processor utilization factor :

- The base period is 10 (see [18], page 5).
- 0 units of time are unused in the base period.
- Processor utilization factor with deadline is 1.00000 (see [1], page 6).
- Processor utilization factor with period is 1.00000 (see [1], page 6).
- In the preemptive case, with LLF, the task set is schedulable because the processor utilization factor 1.00000 is equal or less than 1.00000 (see [7]).

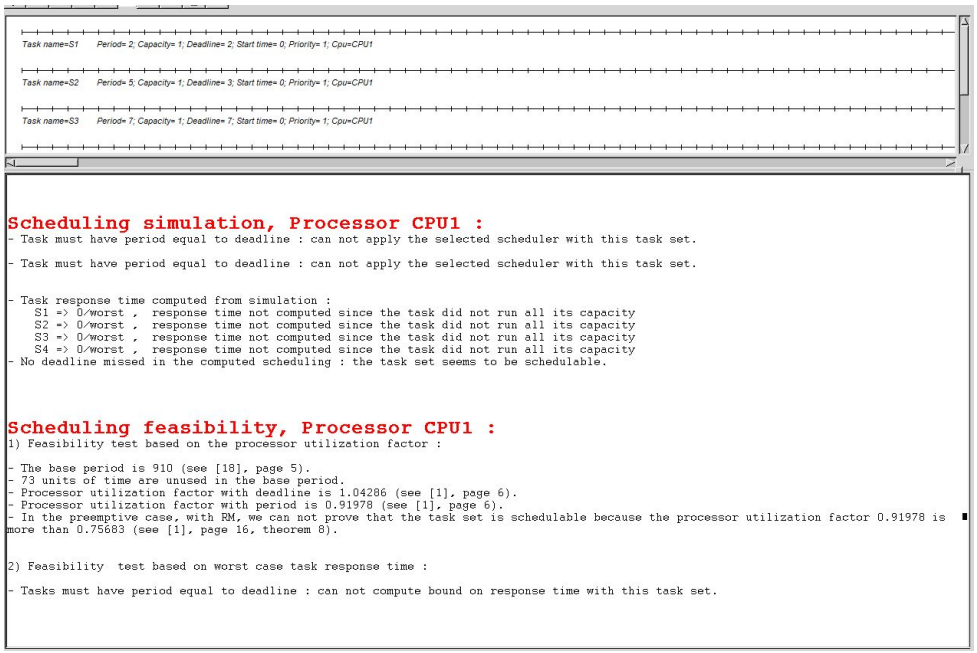
2) Feasibility test based on worst case task response time :

- Bound on task response time :
  S1 => 2
  S2 => 5
  S3 => 10
- All task deadlines will be met : the task set is schedulable.

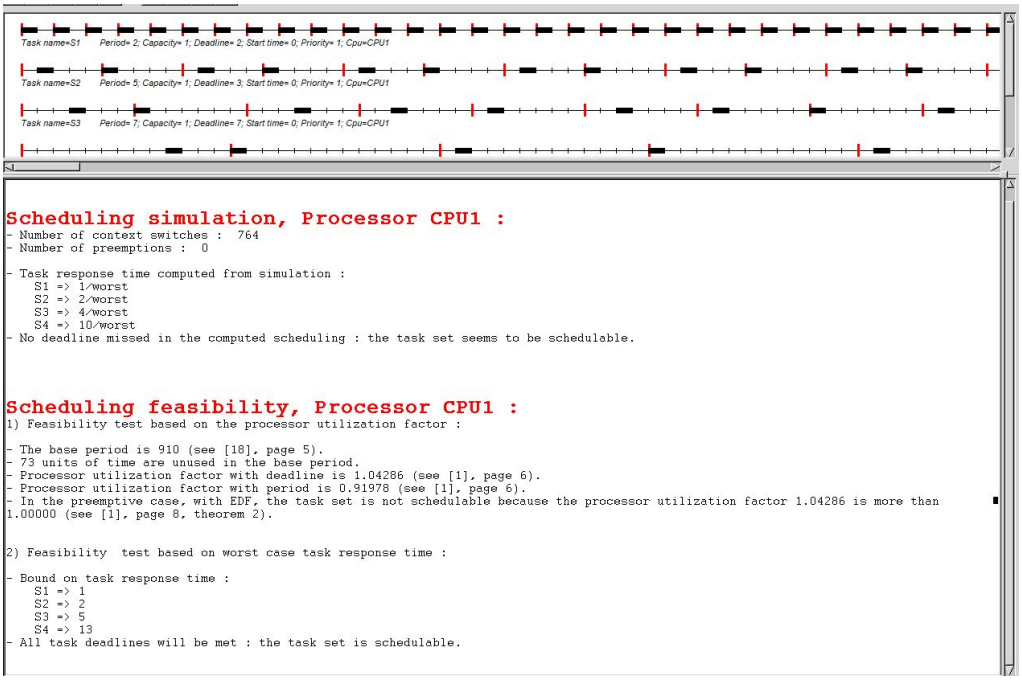
Chutao Wei
Roberto Baquerizo
February 8, 2020
SERVICE SET 6

Example 6	T1	2	C1	1	U1	0.5	LCM =	70	For DM	D1	2
	T2	5	C2	1	U2	0.2				D2	3
	T3	7	C3	1	U3	0.142857				D3	7
	T4	13	C4	2	U4	0.153846	Utot =	0.996703		D4	15

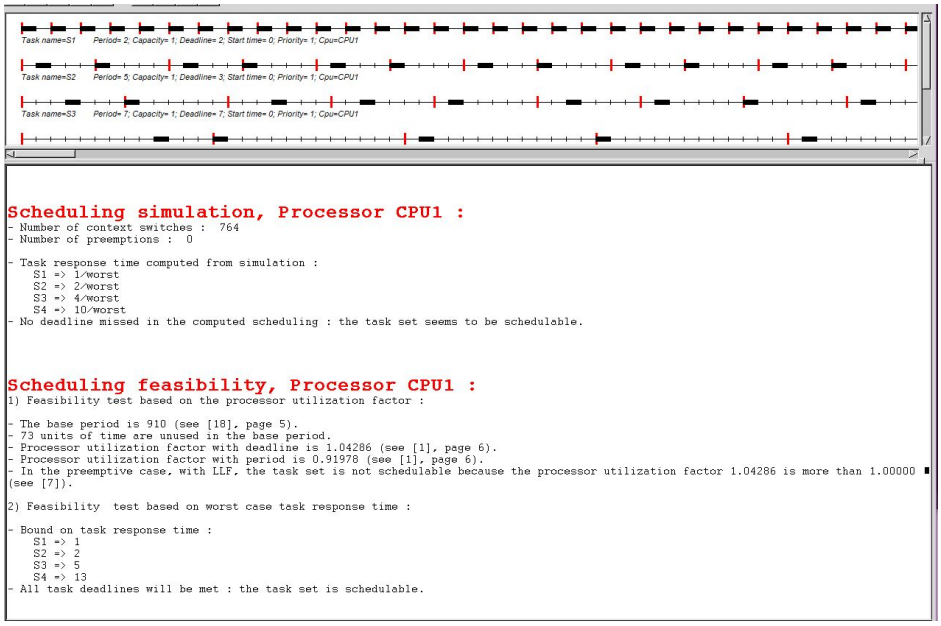
Rate Monotonic Analysis



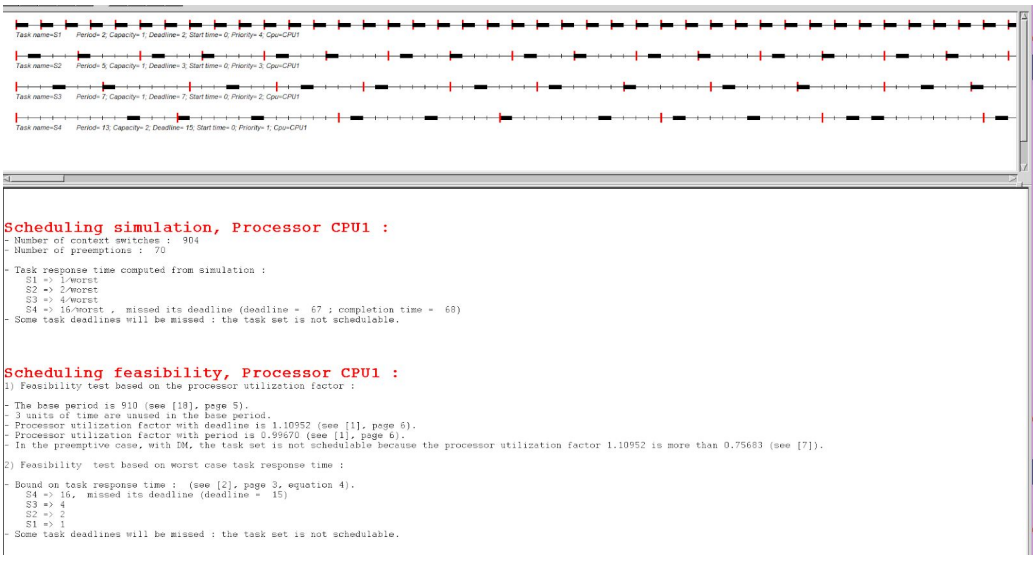
Earliest Deadline First



Chutao Wei
Roberto Baquerizo
February 8, 2020
Least Laxity First



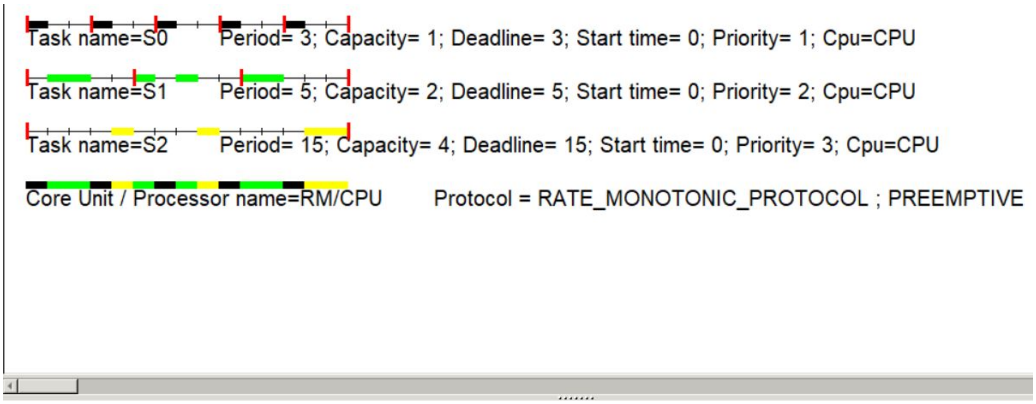
Deadline Monotonic Scheduler



Chutao Wei
Roberto Baquerizo
February 8, 2020
SERVICE SET 7

Example 7	T1	3	C1	1	U1	0.33	LCM =	15
	T2	5	C2	2	U2	0.4		
	T3	15	C3	4	U3	0.26667	Utot =	1.00

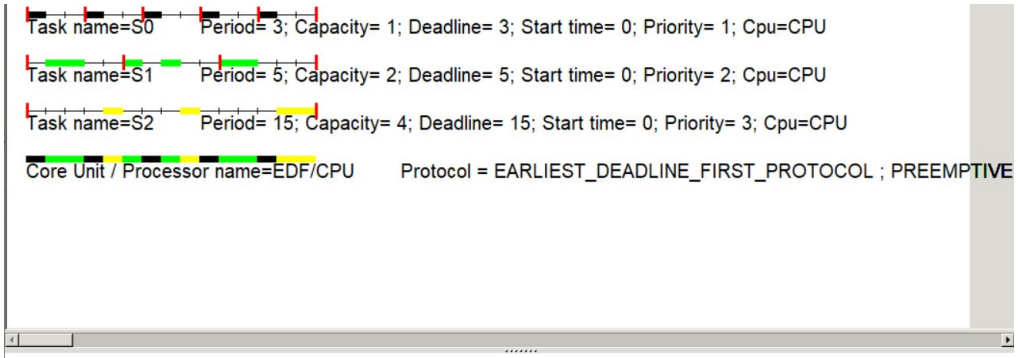
Rate Monotonic Analysis



Scheduling simulation, Processor CPU :
- Number of context switches : 11
- Number of preemptions : 3

- Task response time computed from simulation :
  S0 => 1/worst
  S1 => 3/worst
  S2 => 15/worst
- No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibility interval.

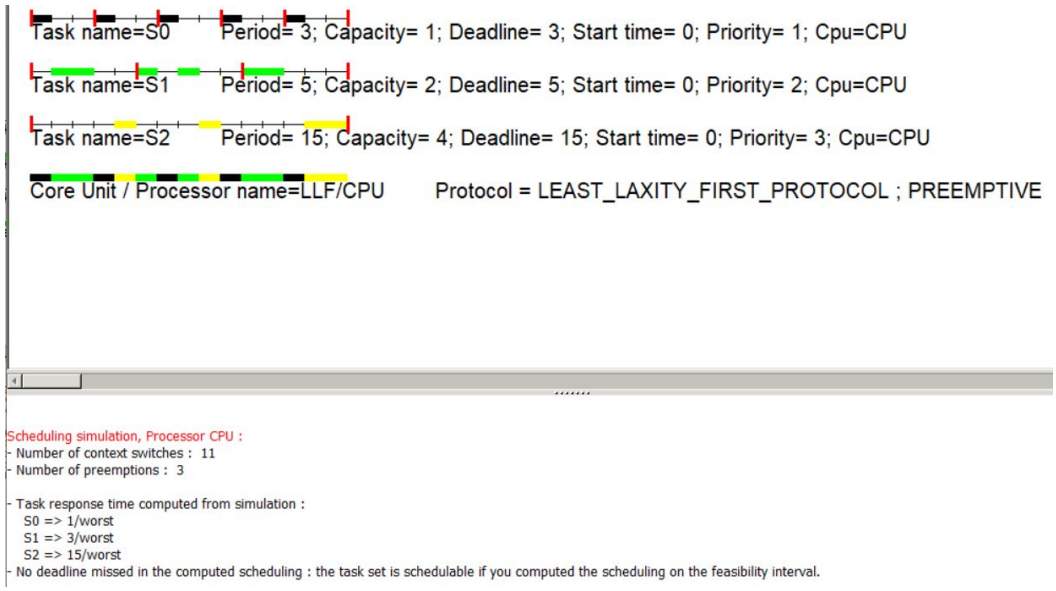
Earliest Deadline First



Scheduling simulation, Processor CPU :
- Number of context switches : 11
- Number of preemptions : 3

- Task response time computed from simulation :
  S0 => 1/worst
  S1 => 3/worst
  S2 => 15/worst
- No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibility interval.

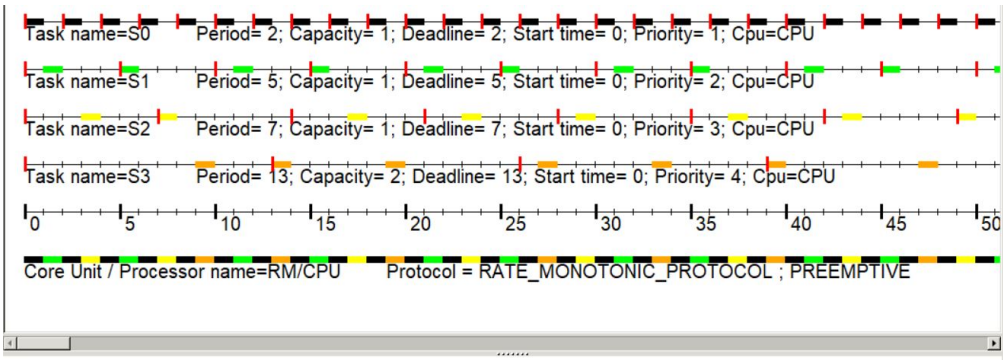
Chutao Wei
Roberto Baquerizo
February 8, 2020
Least Laxity First



Chutao Wei
Roberto Baquerizo
February 8, 2020
SERVICE SET 8

Example 8	T1	2	C1	1	U1	0.5	LCM =	70
	T2	5	C2	1	U2	0.2		
	T3	7	C3	1	U3	0.14286		
	T4	13	C4	2	U4	0.15385	Utot =	0.9967

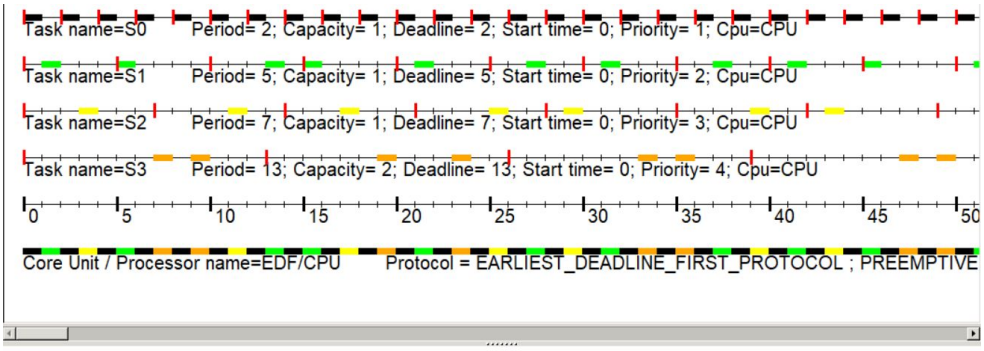
Rate Monotonic Analysis



Scheduling simulation, Processor CPU :
- Number of context switches : 904
- Number of preemptions : 70

- Task response time computed from simulation :
  S0 => 1/worst
  S1 => 2/worst
  S2 => 4/worst
  S3 => 16/worst , missed its deadline (absolute deadline = 13 ; completion time = 14), missed its deadline (absolute deadline = 26 ; completion time = 28), missed its deadline
- Some task deadlines will be missed : the task set is not schedulable.

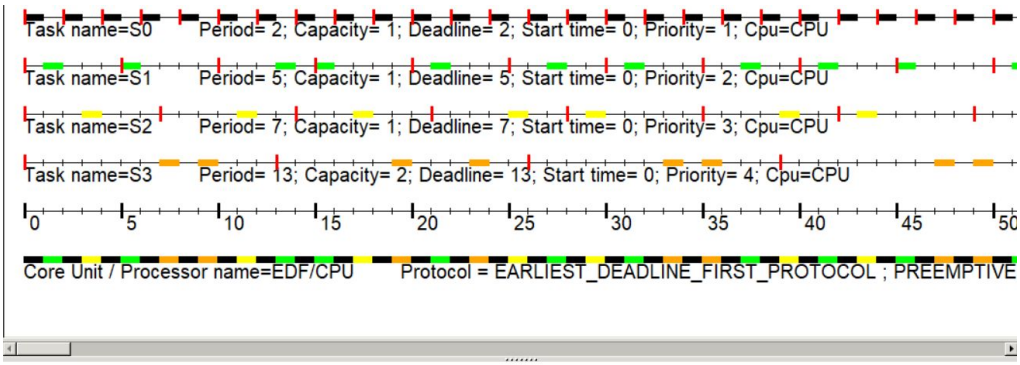
Earliest Deadline First



Scheduling simulation, Processor CPU :
- Number of context switches : 904
- Number of preemptions : 70

- Task response time computed from simulation :
  S0 => 1/worst
  S1 => 4/worst
  S2 => 6/worst
  S3 => 12/worst
- No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibility interval.

Chutao Wei
Roberto Baquerizo
February 8, 2020
Least Laxity First



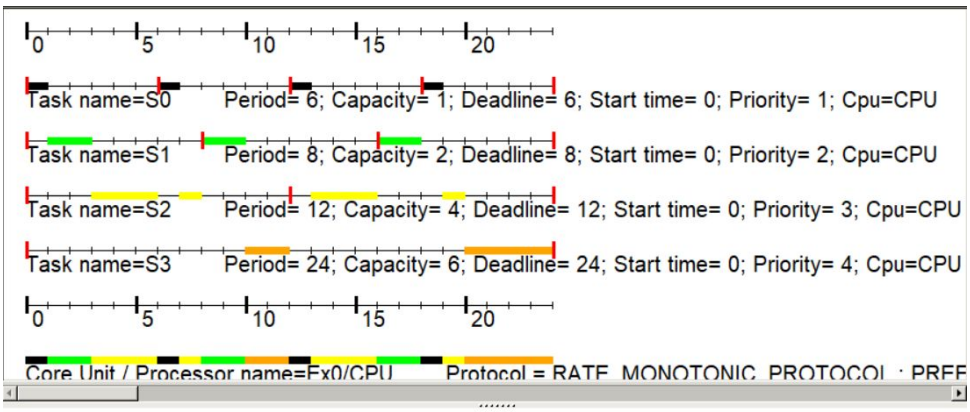
Scheduling simulation, Processor CPU :
- Number of context switches : 904
- Number of preemptions : 70

- Task response time computed from simulation :
  S0 => 1/worst
  S1 => 4/worst
  S2 => 6/worst
  S3 => 12/worst
- No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibility interval.

Chutao Wei
Roberto Baquerizo
February 8, 2020
SERVICE SET 9

Example 9		harmonic	f0 multiple									LCM/T
	f3	0.16667	4	T1	6	C1	1	U1	0.16667	LCM =	24	4
	f2	0.125	3	T2	8	C2	2	U2	0.25			3
	f1	0.08333	2	T3	12	C3	4	U3	0.33333			2
	f0	0.04167	1	T4	24	C4	6	U4	0.25	Utot =	1	1

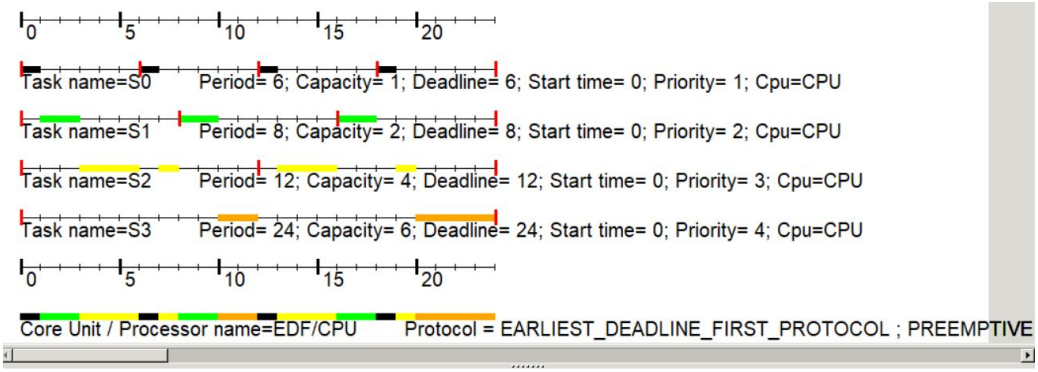
Rate Monotonic Analysis



Scheduling simulation, Processor CPU :
- Number of context switches : 12
- Number of preemptions : 3

- Task response time computed from simulation :
  S0 => 1/worst
  S1 => 3/worst
  S2 => 8/worst
  S3 => 24/worst
- No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibility interval.

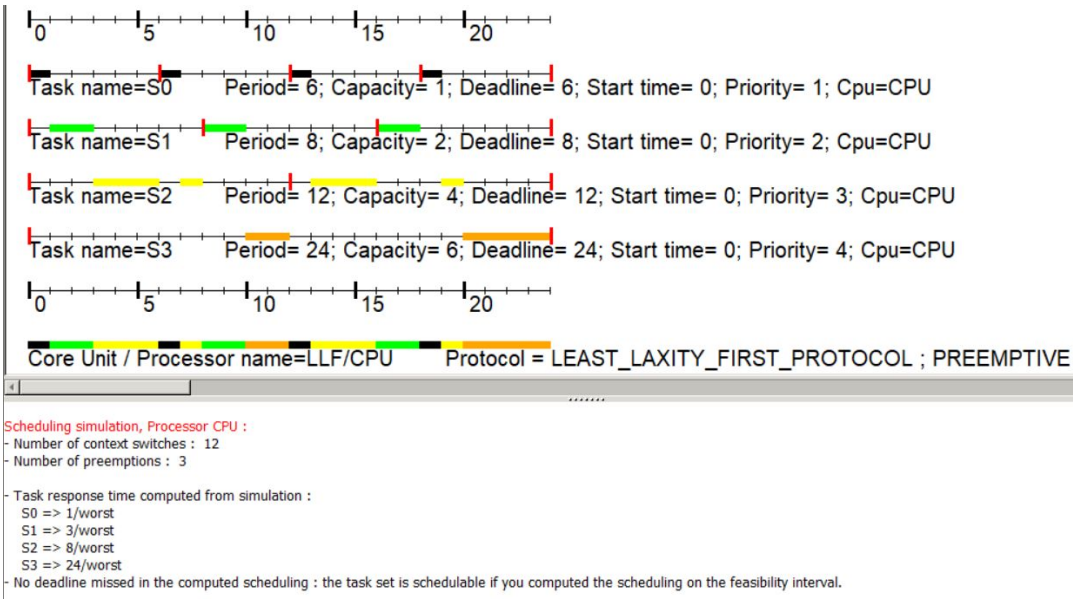
Earliest Deadline First



cheduling simulation, Processor CPU :
Number of context switches : 12
Number of preemptions : 3

Task response time computed from simulation :
S0 => 1/worst
S1 => 3/worst
S2 => 8/worst
S3 => 24/worst
No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibility interval.

Chutao Wei
Roberto Baquerizo
February 8, 2020
Least Laxity First



Scheduling simulation, Processor CPU :
- Number of context switches : 12
- Number of preemptions : 3

- Task response time computed from simulation :
  S0 => 1/worst
  S1 => 3/worst
  S2 => 8/worst
  S3 => 24/worst
- No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibility interval.

```
./bin/modified_feasibility_tests
******** Completion Test Feasibility Example
Ex-0 U=0.73 (C1=1, C2=1, C3=2; T1=2, T2=10, T3=15; T=D): FEASIBLE
Ex-1 U=0.99 (C1=1, C2=1, C3=2; T1=2, T2=5, T3=7; T=D): INFEASIBLE
Ex-2 U=1.00 (C1=1, C2=1, C3=1, C4=2; T1=2, T2=5, T3=7, T4=13; T=D): INFEASIBLE
Ex-3 U=0.93 (C1=1, C2=2, C3=3; T1=3, T2=5, T3=15; T=D): FEASIBLE
Ex-4 U=1.00 (C1=1, C2=1, C3=4; T1=2, T2=4, T3=16; T=D): FEASIBLE
Ex-5 U=1.00 (C1=1, C2=2, C3=1; T1=2, T2=5, T3=10; T=D): FEASIBLE
Ex-6 U=1.00 (C1=1, C2=1, C3=1, C4=2; T1=2, T2=5, T3=7, T4=13; T/=D): FEASIBLE
Ex-7 U=1.00 (C1=1, C2=2, C3=4; T1=3, T2=5, T3=15; T=D): FEASIBLE
Ex-8 U=1.00 (C1=1, C2=1, C3=1, C4=2; T1=2, T2=5, T3=7, T4=13; T=D): INFEASIBLE
Ex-9 U=1.00 (C1=1, C2=2, C3=4, C4=6; T1=6, T2=8, T3=12, T4=24; T=D): FEASIBLE


******** Scheduling Point Feasibility Example
Ex-0 U=0.73 (C1=1, C2=1, C3=2; T1=2, T2=10, T3=15; T=D): FEASIBLE
Ex-1 U=0.99 (C1=1, C2=1, C3=2; T1=2, T2=5, T3=7; T=D): INFEASIBLE
Ex-2 U=1.00 (C1=1, C2=1, C3=1, C4=2; T1=2, T2=5, T3=7, T4=13; T=D): INFEASIBLE
Ex-3 U=0.93 (C1=1, C2=2, C3=3; T1=3, T2=5, T3=15; T=D): FEASIBLE
Ex-4 U=1.00 (C1=1, C2=1, C3=4; T1=2, T2=4, T3=16; T=D): FEASIBLE
Ex-5 U=1.00 (C1=1, C2=2, C3=1; T1=2, T2=5, T3=10; T=D): FEASIBLE
Ex-6 U=1.00 (C1=1, C2=1, C3=1, C4=2; T1=2, T2=5, T3=7, T4=13; T/=D): INFEASIBLE
Ex-7 U=1.00 (C1=1, C2=2, C3=4; T1=3, T2=5, T3=15; T=D): FEASIBLE
Ex-8 U=1.00 (C1=1, C2=1, C3=1, C4=2; T1=2, T2=5, T3=7, T4=13; T=D): INFEASIBLE
Ex-9 U=1.00 (C1=1, C2=2, C3=4, C4=6; T1=6, T2=8, T3=12, T4=24; T=D): FEASIBLE


******** EDF Scheduling Example
Ex-0 U=0.73 (C1=1, C2=1, C3=2; T1=2, T2=10, T3=15; T=D): FEASIBLE
Ex-1 U=0.99 (C1=1, C2=1, C3=2; T1=2, T2=5, T3=7; T=D): FEASIBLE
Ex-2 U=1.00 (C1=1, C2=1, C3=1, C4=2; T1=2, T2=5, T3=7, T4=13; T=D): FEASIBLE
Ex-3 U=0.93 (C1=1, C2=2, C3=3; T1=3, T2=5, T3=15; T=D): FEASIBLE
Ex-4 U=1.00 (C1=1, C2=1, C3=4; T1=2, T2=4, T3=16; T=D): FEASIBLE
Ex-5 U=1.00 (C1=1, C2=2, C3=1; T1=2, T2=5, T3=10; T=D): FEASIBLE
Ex-6 U=1.00 (C1=1, C2=1, C3=1, C4=2; T1=2, T2=5, T3=7, T4=13; T=D): FEASIBLE
Ex-7 U=1.00 (C1=1, C2=2, C3=4; T1=3, T2=5, T3=15; T=D): FEASIBLE
Ex-8 U=1.00 (C1=1, C2=1, C3=1, C4=2; T1=2, T2=5, T3=7, T4=13; T=D): FEASIBLE
Ex-9 U=1.00 (C1=1, C2=2, C3=4, C4=6; T1=6, T2=8, T3=12, T4=24; T=D): FEASIBLE


******** LLF Scheduling Example
Ex-0 U=0.73 (C1=1, C2=1, C3=2; T1=2, T2=10, T3=15; T=D): FEASIBLE
Ex-1 U=0.99 (C1=1, C2=1, C3=2; T1=2, T2=5, T3=7; T=D): FEASIBLE
Ex-2 U=1.00 (C1=1, C2=1, C3=1, C4=2; T1=2, T2=5, T3=7, T4=13; T=D): FEASIBLE
Ex-3 U=0.93 (C1=1, C2=2, C3=3; T1=3, T2=5, T3=15; T=D): FEASIBLE
Ex-4 U=1.00 (C1=1, C2=1, C3=4; T1=2, T2=4, T3=16; T=D): FEASIBLE
Ex-5 U=1.00 (C1=1, C2=2, C3=1; T1=2, T2=5, T3=10; T=D): FEASIBLE
Ex-6 U=1.00 (C1=1, C2=1, C3=1, C4=2; T1=2, T2=5, T3=7, T4=13; T=D): FEASIBLE
Ex-7 U=1.00 (C1=1, C2=2, C3=4; T1=3, T2=5, T3=15; T=D): FEASIBLE
Ex-8 U=1.00 (C1=1, C2=1, C3=1, C4=2; T1=2, T2=5, T3=7, T4=13; T=D): FEASIBLE
Ex-9 U=1.00 (C1=1, C2=2, C3=4, C4=6; T1=6, T2=8, T3=12, T4=24; T=D): FEASIBLE
```

Chutao Wei
Roberto Baquerizo
February 8, 2020
Comments on Cheddar Simulation Results and Output of Feasibility Tests

Ex0:

This example actually passes the RM least upper bound test. The least upper bound is 0.78, which is less than 0.733333 the total utilization. So RM for sure pass. Cheddar shows the same.

Ex1:

Both scheduling point and completion time tests did not pass. RM is not schedulable due to S2 having a higher priority than S3. S3 misses its deadline. This is shown in cheddar screenshot For EDF and LLF, both pass. Because both of them take deadlines into account.

Ex2:

Both scheduling point and completion time tests did not pass. This one adds another service S4. RM fails due to the same reason. The higher priority service runs before a lower priority service when the lower priority service is about to miss the deadline. Both EDF and LLF pass.

Ex3:

All three pass. S3 period is the lcm of S1's and S2's period.

Ex4:

This is a harmonic case. T = 2, 4, 8. freq = ½, ¼, ⅛. All three policies pass. All the policies have the same timing diagram.

Ex5:

This is a harmonic case, too. T = 2, 4, 8. freq = ½, ¼, ⅛ This is a case where all 3 schemes are feasible. All of them provide the same timing diagram. Utilization is 100%. Cheddar output results are the same as the output of .c programs.

Ex6:

Same as Ex8, except that the deadline is not the same as period. So RM policy does not apply here. EDF and LLF both pass with U = 100%. Cheddar output results are the same as the output of .c programs. The example did DM policy instead of EDF or LLF. DM policy is feasible in this case. We also did EDF and LLF. Both of them pass.

Ex7:

This is a case where all 3 schemes are feasible. All of them provide the same timing diagram. Cheddar output results are the same as the output of .c

programs. Utilization is 100%. This example shows that EDF and LLF could potentially have different timing diagrams.

Ex8:

This service set has the largest LCM (910). EDF and LLF both pass feasibility tests. RM policy failed because S2 has a higher priority than S3. The S3 failed to complete before its deadline.

Ex9:

All three scheduling policies are feasible, proved by both the .c program Roberto wrote and cheddar. Cheddar output results are the same as the output of .c programs. Since the Scheduling Point Test and Completion Time Test both passed. We expect that RM is feasible. Also due to the service set being harmonic, we usually expect that RM, EDF, and LLF all have the same timing diagram. However, in this case, the LLF timing diagram is a little different than RM and EDF at the later half portion. which means that EDF and LLF should be feasible, too.

Chutao Wei
Roberto Baquerizo
February 8, 2020
PROBLEM 5

THREE CONSTRAINTS MADE ON THE RM LUB DERIVATION

1. For the assumption that deadlines consist of run-ability constraints only to hold, a small but possibly significant amount of buffering hardware must exist for each peripheral function.
2. The deadline for a task must be equal to its time period.
3. The time required for context switching is not taken into account. In the derivation, the time that the CPU takes to halt the execution of one task and restore another is instantaneous. In reality, this time could be significant, so it is desirable that there is a large enough margin in CPU utilization.

THREE ASSUMPTIONS MADE ON THE RM LUB DERIVATION

1. The tasks are independent in that requests for a certain task do not depend on the initiation or the completion of requests for other tasks.
2. Deadlines consist of run-ability constraint only, i.e. each task must be completed before the request for it occurs.
3. The run-time, i.e. the execution time, for each task is constant for that task and does not vary with time.

THREE KEY DERIVATION STEPS IN THE RM LUB DERIVATION

1. I did not understand how equation $C_1 + C_2 \leq T_1$ (2) implies $\lfloor T_2/T_1 \rfloor \times C_1 + C_2 \leq T_2$ (1) after the proof of Theorem 1, prior to stating Theorem 2.
   After reading through Chapter 3 of the RTES book, it became clear that equation (1) states that the computation time of S_2 and the total time used up by S_1 must be less than or equal to the request period of S_2. With that interpretation of (1) and the assumption that $T_1<T_2$ makes it obvious that (2) means that the execution time of S_2 and a single execution of S_1 must be less than or equal to the request period of S_1.
2. In the proof of Theorem 3, Liu and Layland consider two cases for the relationship between the run-times and periods for two tasks. I did not understand how the relation for Case 2 arose $C_1 \geq T_2 - T_1 \times \lfloor T_2/T_1 \rfloor$. It became evident, after reading through Chapter 3 of the RTES book, that the relation states that the computation time for S_1 must be at least as long as the request period of S_2 minus the number of times S_1 executes inside the time-span of T_2.
3. In the proof of Theorem 4, Liu and Layland write down an equation for the run-time of the m^{th} task: $C_m = T_m - 2(C_1 + C_2 + \ldots + C_{m-1})$. I did not understand where that relation was used in the subsequent derivation steps.
   After introducing $g_i = (T_m - T_i)/T_i \ (i = 1, 2, .., m)$, that relation is actually used in deriving equation 4 on page 54, noting that the relation is just twice the finite sum of the run-times for tasks 1 through m-1 subtracted from the request period of the m^{th} task.