CENG 383

Real-Time Systems

Lecture 3

Theoretical Foundations of RTOS

Asst. Prof. Tolga Ayav, Ph.D.

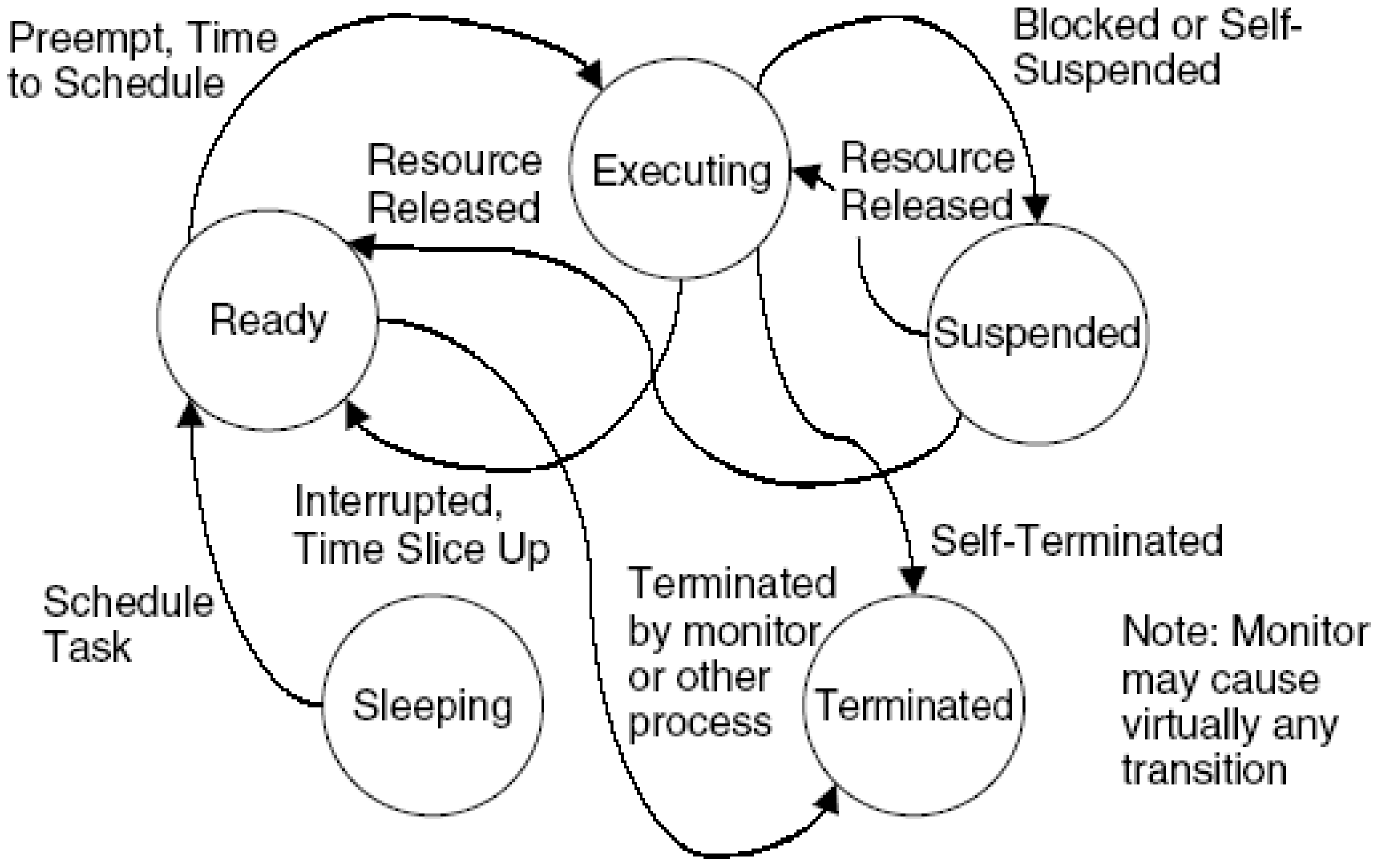
Department of Computer Engineering

İzmir Institute of Technology

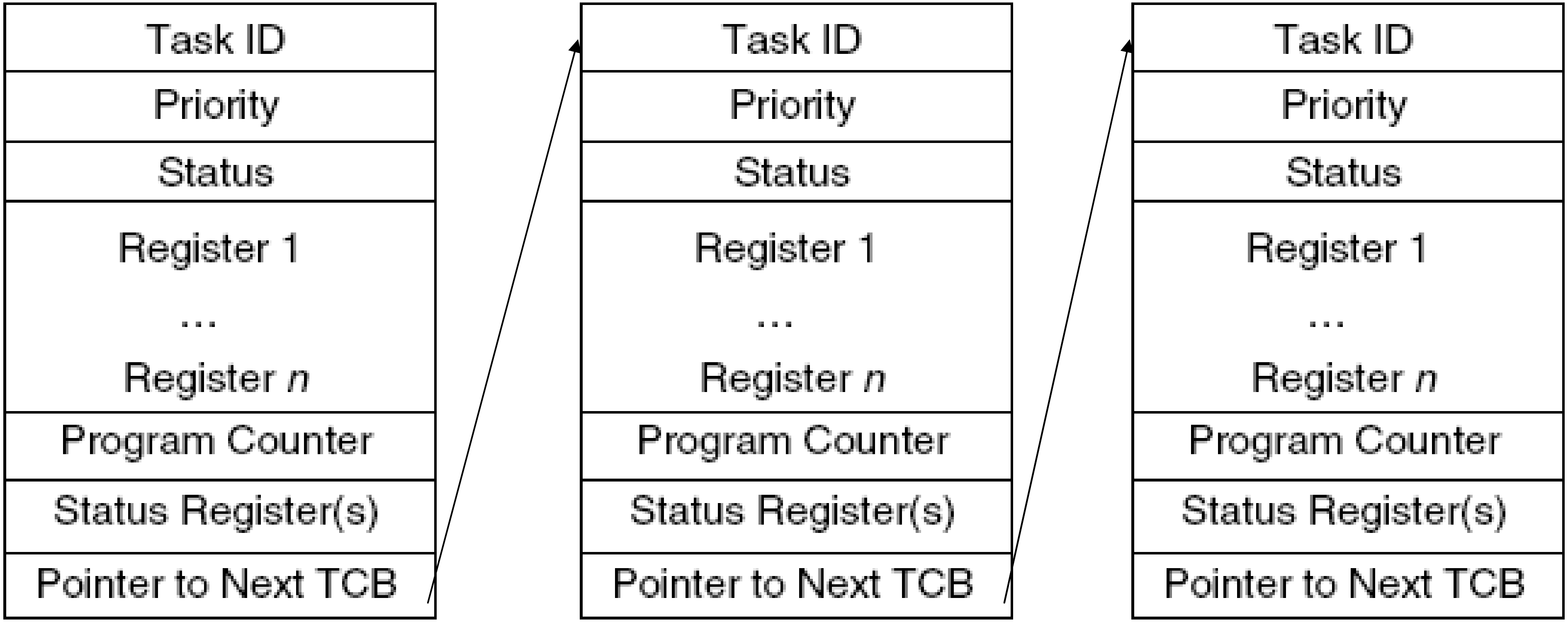
Task States

- *Executing*
- *Ready*
- *Suspended (or blocked)*
- *Dormant (or sleeping)*

Task State Diagram



Task Control Block



RT Scheduling

- Among many functions, scheduling is the most important function of a real-time kernel
- A realtime application is composed of as a set of coordinated tasks. We can categorize the task according to their activation:
  - *Periodic tasks*
  - *Sporadic tasks*
  - *Aperiodic tasks*
- Periodic tasks are started at regular intervals and has to be completed before some deadline.
- Sporadic tasks are appeared irregularly, but within a bounded frequency.
- Aperiodic tasks' parameters are completely unknown.

RT Tasks

- We can use the following quintuple to express task τ_i:
  - $< \tau_i, b_i, c_i, f_i, d_i >$

- b_i is begin time of τ_i
- c_i is computation time of τ_i
- d_i is the deadline
- f_i is the frequency (for sporadic tasks it's the bound)

- For schedulability, at least the following conditions must be met:
  - $c_i < d_i - b_i < 1/f_i$
  - $\sum c_i f_i \leq$ available resource

RT Tasks

- We can also categorize tasks according to their time criticality:
  - Hard real-time tasks
  - Soft real-time tasks
  - Non real-time tasks (background tasks)

Simple Task Model

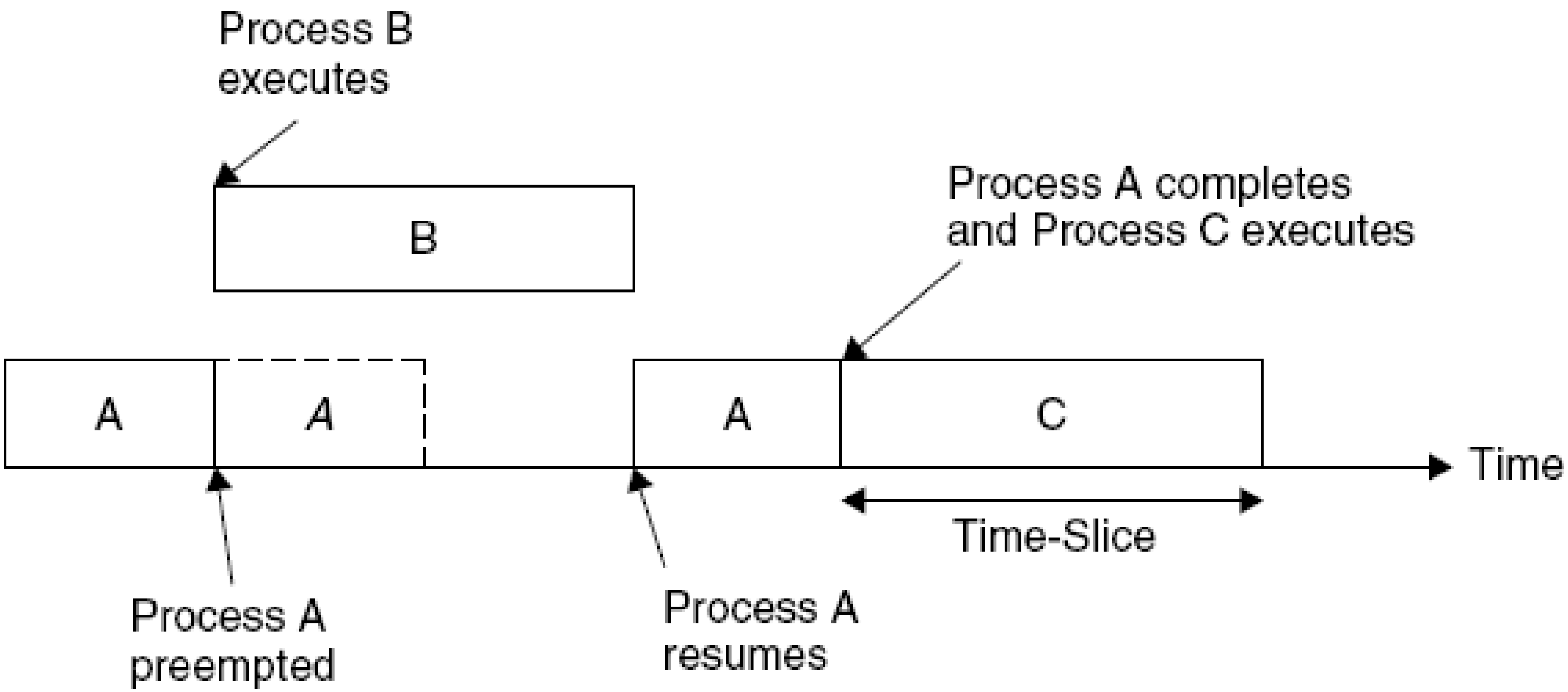
- All tasks in the task set are strictly periodic.
- The relative deadline of a task is equal to its period/frame.
- All tasks are independent; there are no precedence constraints.
- No task has any nonpreemptible section, and the cost of preemption is negligible.
- Only processing requirements are significant; memory and I/O requirements are negligible.

Scheduling Techniques

- Dynamic Scheduling
  - Static priority-driven preemptive scheduling(RM)
  - Dynamic priority-driven preemptive scheduling(EDF)
  - Adaptive scheduling(FC-EDF)
  - Round-Robin Scheduling
  - Cooperative Scheduling Techniques
  - ...
- Static Scheduling
  - AAA (algorithm architecture adequation)
  - ...

Round-Robin Scheduling

- Each executable task is assigned a fixed-time quantum called a time slice in which to execute.
- The task executes until it completes, or its execution time expires.
- Task switching occurs then.
- Round-robin systems can be combined with preemptive priority systems, yielding a kind of mixed system:



Cyclic Executives (1)

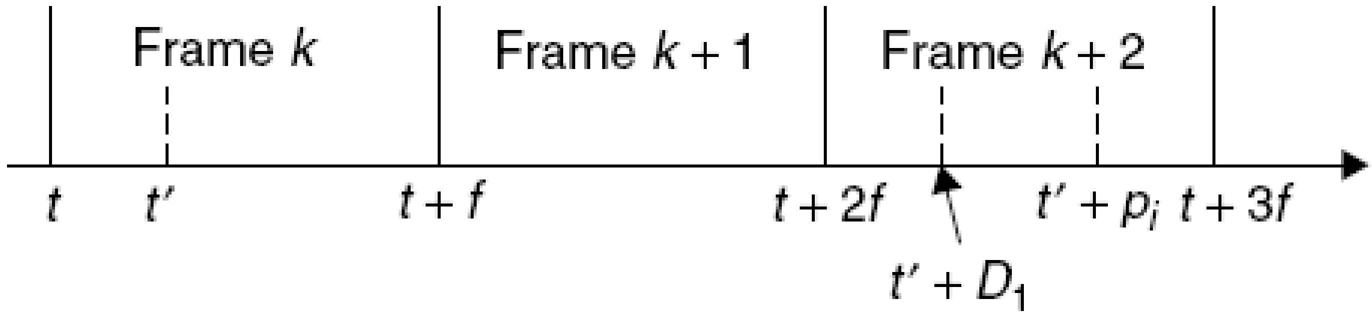
- Execution of periodic tasks on a processor according to a pre-run time schedule.
- CE is a table of procedure calls, where each task is a procedure, within a single do loop.
- The major cycle is the minimum time required to execute tasks allocated to the processor, ensuring that the deadlines and periods of all processes are met.

$$Hyperperiod = LCM(T_1, T_2, \ldots, T_n)$$

- Scheduling decisions are made at the beginning of each frame. No preemption within each frame.
- Frames must be sufficiently long so that every task can start and complete within a single frame.

$$R_1 : f \geq \max_{1 \leq i \leq n} c_i$$

Cyclic Executives (2)



- 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:

$$R_2 : \left\lfloor \frac{T_i}{f} \right\rfloor - \frac{T_i}{f} = 0$$

- In order to ensure that every task completes by its deadline, frames must be small so that between the release time and deadline of every task, there is at least one frame. The following relation is derived for a worst-case scenario, which occurs when the period of a process starts just after the beginning of a frame and, consequently, the process cannot be released until the next frame.

$$R_3 : 2f - \gcd(T_i, f) \le D_i$$

Example Frame Calculation

τ_i	T_i	C_i	D_i
τ_2	15	1	14
τ_3	20	2	26
τ_4	22	3	22

$$R1 \; : \forall i f \geq C_i \Rightarrow f \geq 3$$

$$R2 \; : \lfloor T_i / f \rfloor - T_i / f = 0 \Rightarrow f = 2, 3, 4, 5, 10, \ldots$$

$$R3 \; : 2f - gcd(T_i, f) \leq D_i \Rightarrow f = 2, 3, 4, 5$$

Possible value of f could be any of the values of 3, 4 and 5.

Rate-Monotonic Scheduling

Assumptions:

1. Simple task model: No interprocess communication and all tasks are periodic
2. Tasks have priorities which are inversly proportional to their periods.
3. Tasks' deadlines are equal to their periods.
4. A high priority task may preempt lower priority tasks.

Liu and Layland (1973) proved that for a set of *n* periodic tasks with unique periods, a feasible schedule that will always meet deadlines exists if the CPU utilization is:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(\sqrt[n]{2} - 1)$$

Where C_i is the computation time of a task i, T_i is the deadline of task i and *n* is the number of tasks.

For example, for n=2, $U \leq 0.8284$

Rate-Monotonic Scheduling

When number of tasks approaches to infinity, this utilization bound will converge to:

$$\lim_{n \to \infty} n(\sqrt[n]{2} - 1) = \ln 2 \approx 0.693147\ldots$$

Example:

Task	Execution Time	Period
τ1	1	8
τ2	2	5
τ3	2	10

$$\frac{1}{8} + \frac{2}{5} + \frac{2}{10} = 0.725$$

$$U = 3(2^{\frac{1}{3}} - 1) = 0.77976\ldots$$

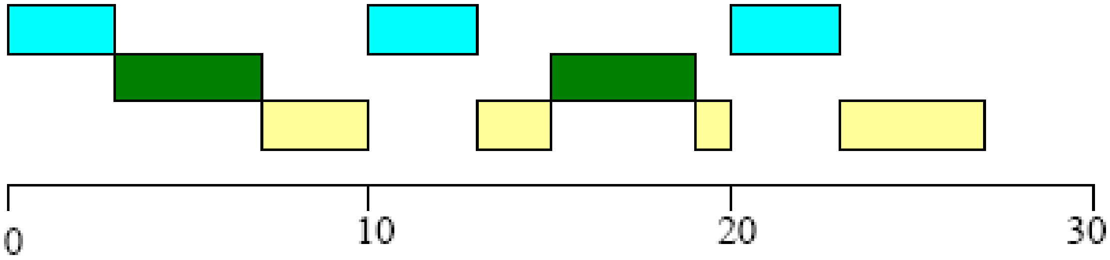
$$0.725 < 0.77976\ldots$$

Thus, the system is schedulable

Sample Task Scheduling

- Three periodic tasks: A, B, C
- T is period, D is deadline and C is execution time
- uniprocessor

Task	T	D	C
A	10	10	3
B	15	15	4
C	30	30	10



Earliest-Deadline-First Scheduling

- Priorities are changed dynamically
- Task with the earliest deadline gets the highest priority
- Unless RM, utilization may go up to 100%

RM vs.EDF

- EDF is more flexible and achieves better utilization.
- RM is more predictable especially in overload conditions:the same lower-priority tasks miss deadlines every time.
- İn EDF, it is difficult to predict which tasks will miss their deadlines during overloads.
- RM tends to need more preemption.
- EDF only preempts when an earlier-deadline task arrives.

For further discussion, read:

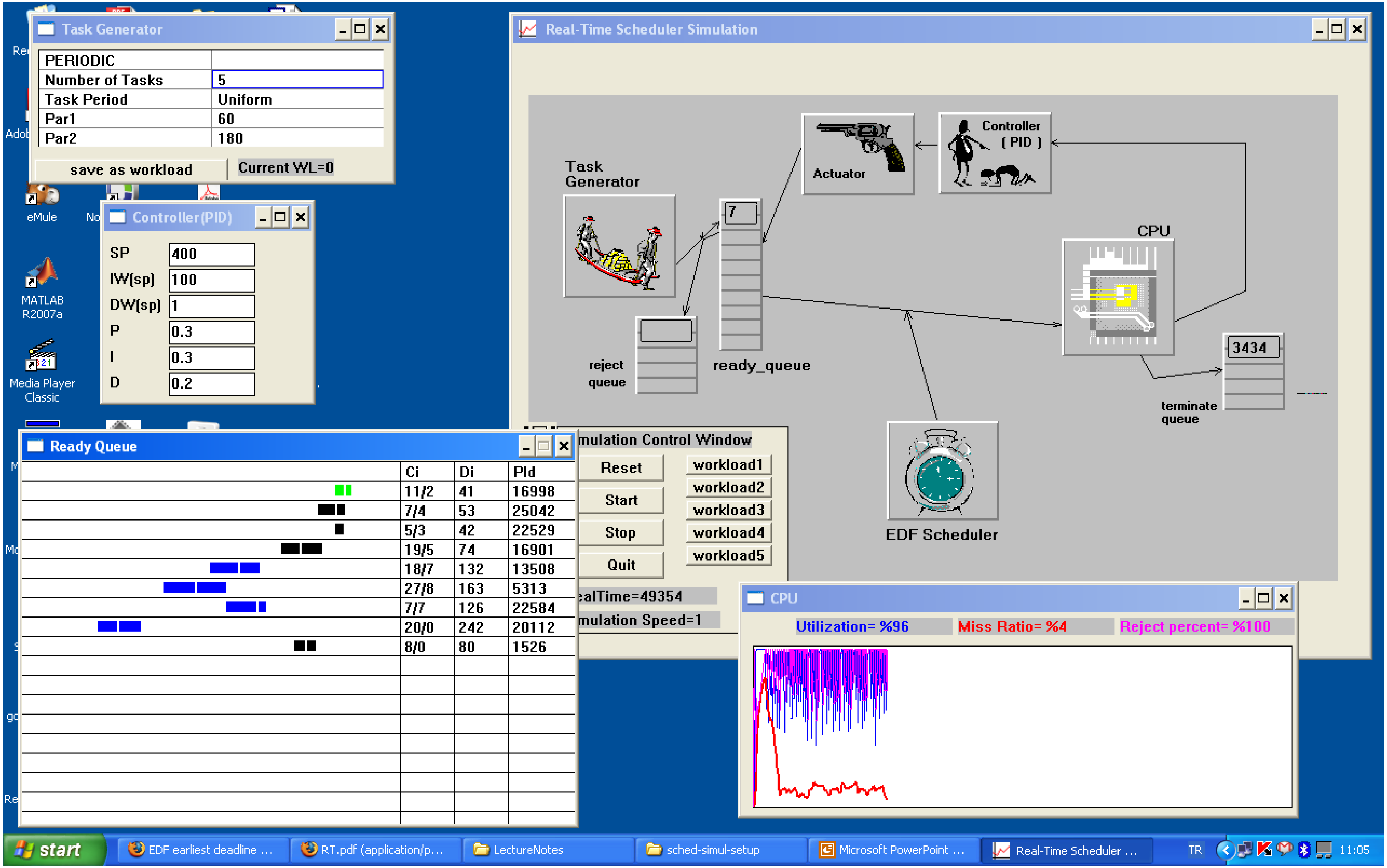
Buttazzo, G. C. 2005. Rate monotonic vs. EDF: judgment day. Real-Time Syst. 29, 1 (Jan. 2005), 5-26.

DOI= http://dx.doi.org/10.1023/B:TIME.0000048932.30002.d9

Imprecise Computations

- In case that digital signal processing algorithms are performed, The system may get overloaded.
- For example, a Taylor series expansion (perhaps using look-up tables for function derivatives) can be terminated early, at a loss of accuracy, but with improved performance.
- Tasks can be divided into two parts: Mandatory and Optional.
- Various methods: Sieve, Multiple-versions etc.
- For example, a digital filtering task might be implemented as 4 versions such that each version has different filter characteristics, contributions and consequently computation requirements.
- In overload conditions, the scheduler may schedule shorter versions.
- Requires an adaptive scheduling algorithm.
- Their applications can be seen on network systems.
- See Stankovic's works for further details.

FC-EDF Scheduler Simulator



Critical Regions

```
⋮
printf("I am Task A");
⋮
```

||

```
⋮
printf("I am Task B");
⋮
```

```
I am I am Task B Task A
```

Semaphores

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

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

```
void Task_A(void)
{
    P(S);
    printf("I am Task_A");
    V(S);
}
```

```
void Task_B(void)
{
    P(S);
    printf("I am Task_B");
    V(S);
{
```

Counting Semaphores

If there are more than one resources:

```
void MP(int S)
{
    S=S-1;
    while (S < 0);
}
```

```
void MV(int S)
{
    S=S+1
}
```

Deadlock

```
Task_A
   :
  P(S)
  use resource 1
   :
   P(R)
   stuck here
   use resource 2

   V(R)
   V(S)
   :
```

```
Task_B

    :
    P(R)
    use resource 2
    :

    P(S)
    stuck here
    use resource 1
    V(S)
    V(R)

    :
```

Four conditions are necessary for deadlock:
1. Mutual exclusion
2. Circular wait
3. Hold and wait
4. No preemption

Eliminating any of them will prevent deadlock from occuring.

POSIX

POSIX is the IEEE's Portable Operating System Interface for Computer Environments.The standard provides compliance criteria for operating system services and is designed to allow applications programs to write applications that can easily port across operating systems.

For further details, read POSIX.pdf

RT-Linux

- **RT-Linux is an operating system, in which a small real-time kernel co-exists with standard Linux kernel**

– The real-time kernel sits between *standard Linux kernel* and the *h/w*.

– The standard Linux kernel sees this real-time layer as actual h/w

– The real-time kernel *intercepts all hardware interrupts*.

• Only for those RTLinux-related interrupts, the appropriate ISR is run.

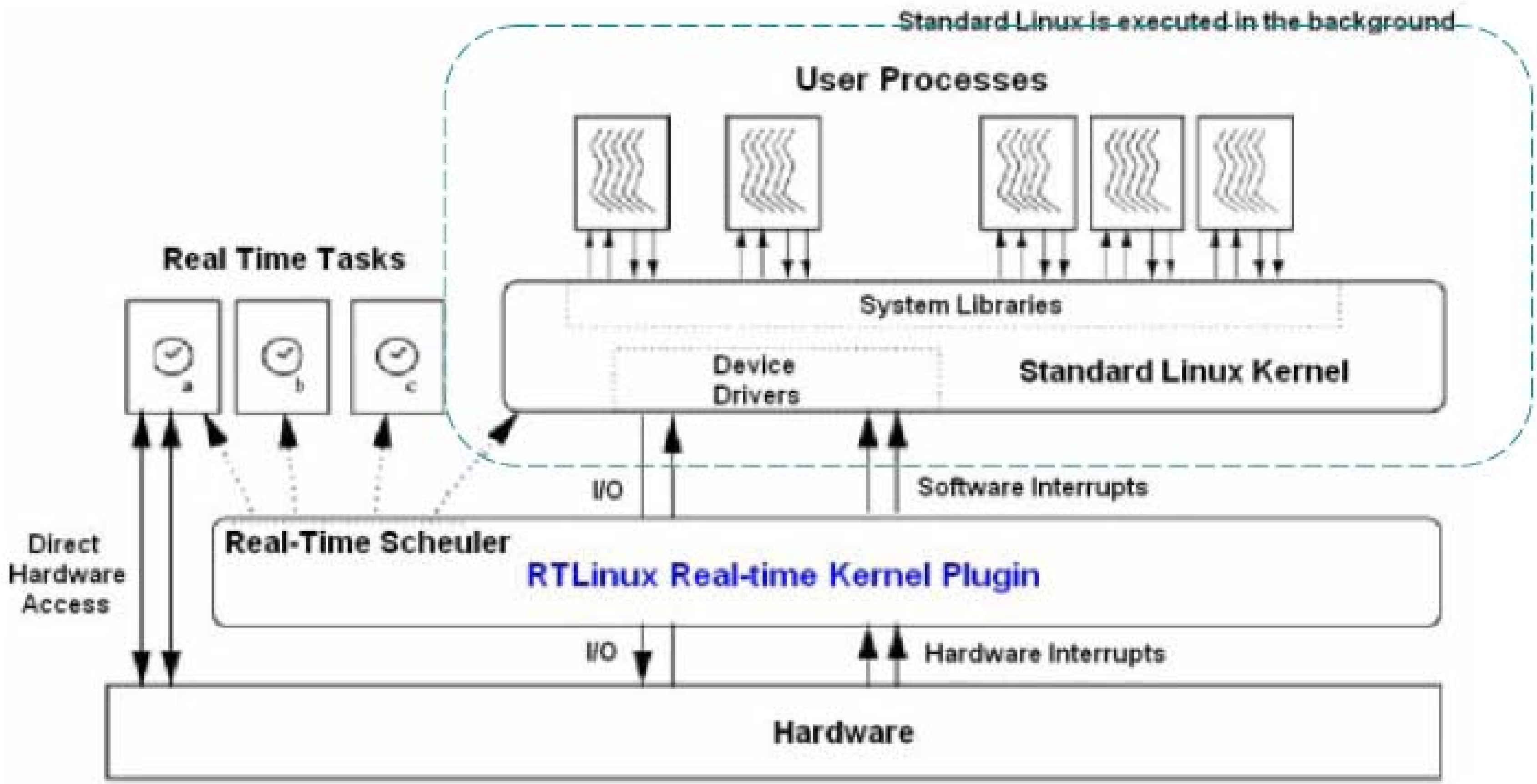
• All other interrupts are held and passed to the standard Linux kernel as software interrupts when the standard Linux kernel runs.

– The real-time kernel assigns the *lowest priority* to the *standard Linux kernel*. Thus the realtime tasks will be executed in real-time

– user can create realtime tasks and achieve correct timing for them by deciding on scheduling algorithms, priorities, execution freq, etc.

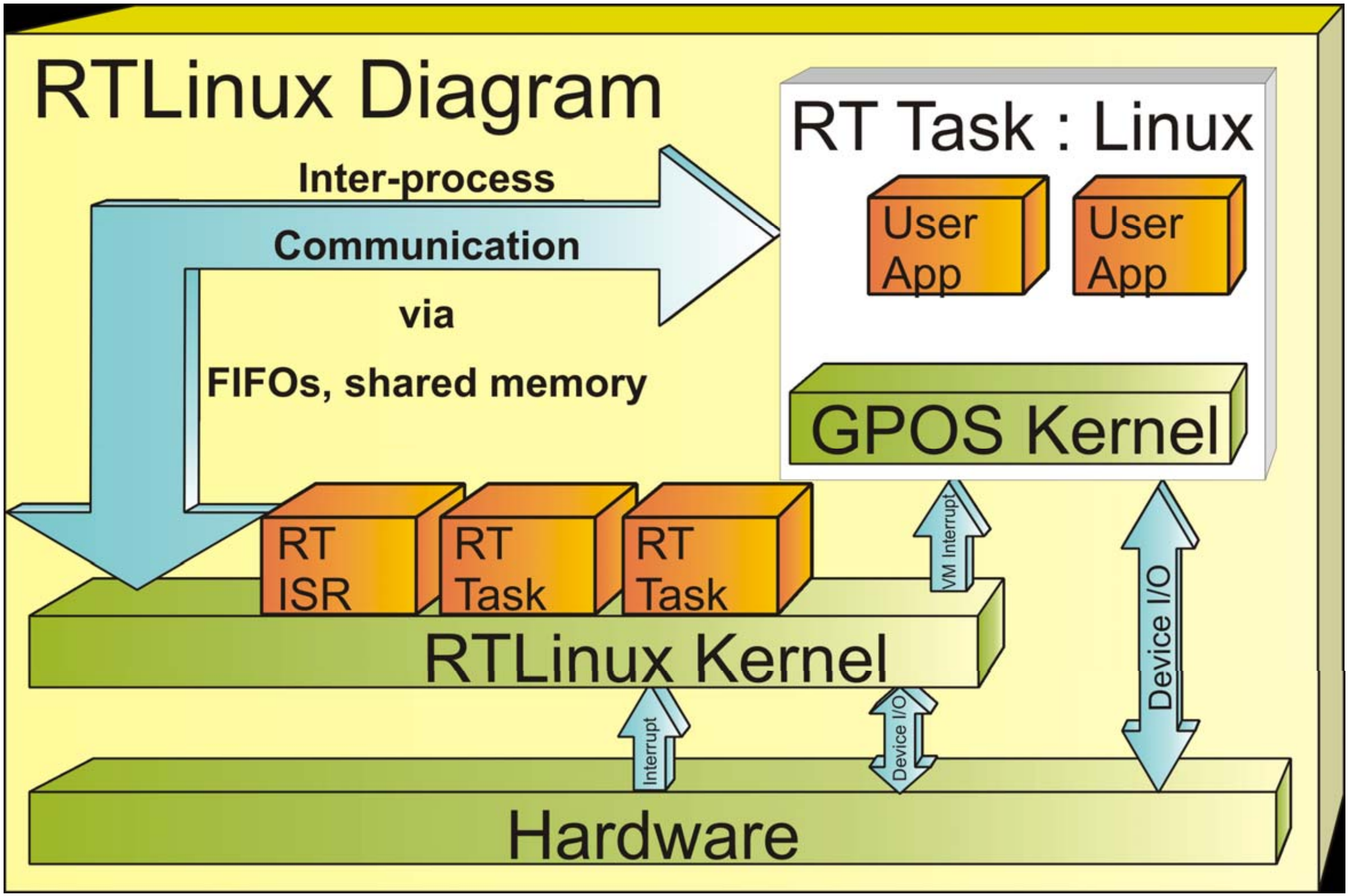
– Realtime tasks are *privileged* (that is, they have direct access to hardware), and they do *NOT use virtual memory*.

RT-Linux



Standard Linux is executed in the background
User Processes
Real Time Tasks
a
b
c
System Libraries
Device
Drivers
Standard Linux Kernel
I/O
Software Interrupts
Direct
Hardware
Access
Real-Time Scheuler
RTLinux Real-time Kernel Plugin
I/O
Hardware Interrupts
Hardware

RT-Linux



RTLinux Diagram
Inter-process
Communication
via
FIFOs, shared memory
RT Task : Linux
User App
User App
GPOS Kernel
RT ISR
RT Task
RT Task
VM Interrupt
Device I/O
RTLinux Kernel
Interrupt
Device I/O
Hardware

Scheduler

- **RT-Linux contains a dynamic scheduler**
- **RT-Linux has many kinds of schedulers**
  – The EDF (Earliest Deadline First) scheduler
  – Rate-monotonic scheduler
- **Real-time FIFOs**
  – RT-FIFOs are used to pass information between real-time process and ordinary Linux process.
  – RT-FIFOs are designed to never block the real-time tasks.
  – RT-FIFOs are, like realtime tasks, never page out. This eliminates the problem of unpredictable delay due to paging.

Time Resolution

- If the kernel was patched with UTIME, we could schedule processes with microsecond resolution.
- Running rtlinx-V3.0 Kernel 2.2.19 on the 486 allows stable hard real-time operation. Giving:

• 15 microseconds worst case jitter.

– **10 microseconds event resolution.**
– **17 nanoseconds timer resolution.**
– **6 microseconds interrupt response time. (This value was measured on interrupts on the parallel port)**

• High resolution timing functions give nanosecond

resolution (limited by the hardware only)

Linux v.s. RTLinux

- **Linux Non-real-time Features**

– Linux scheduling algorithms are not designed for real-time tasks
  - Provide good *average* performance or throughput

– Unpredictable delay
  - Uninterruptible system calls, the use of interrupt disabling virtual memory support (context switch may take hundreds of microsecond).

  – Linux Timer resolution is coarse, 10ms

  – Linux Kernel is Non-preemptible.

- **RTLinux Real-time Features**

– Support real-time scheduling

– Predictable delay (by its small size and limited operations)

– Finer time resolution

– Preemptible kernel

– No virtual memory support