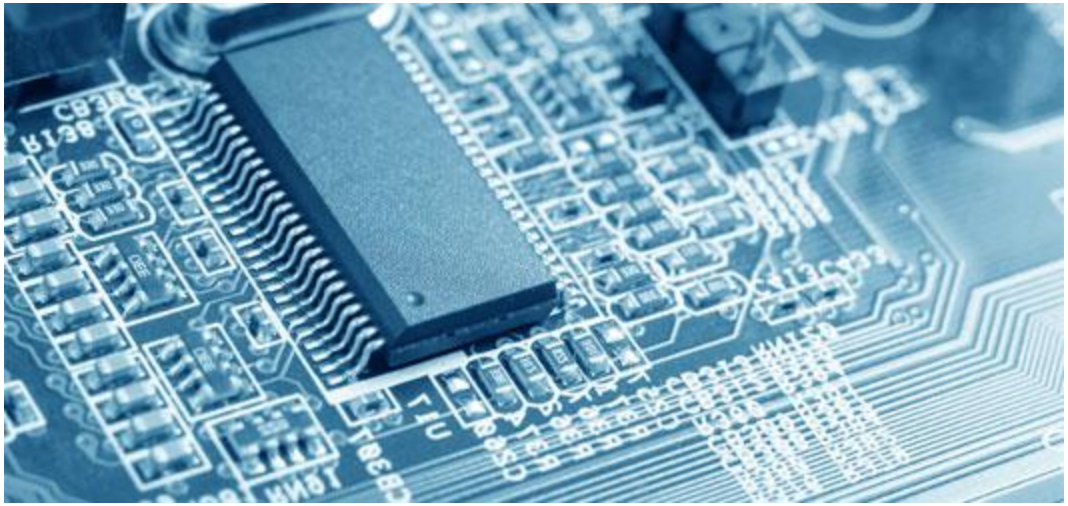
NAZARBAYEV UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY



ROBT305 - Embedded Systems

Lectures 10-11 – EDF Scheduling

29 September -
6 October, 2015

Course Logistics

Reading Assignment:
Chapter 5 of the Operating Systems Concept textbook **Chapter 3** of the Real-Time Systems Design and Analysis textbook
Homework Assignment #2 is out in Moodle and due to end of 4 October (Sunday) – extended deadline
Quiz #3 is on 1 October – Semaphores, Scheduling (except EDF)
Midterm exam is in class time on Thursday 8 October, 2015

Dynamic Priority Scheduling

- In contrast to fixed-priority algorithms, in dynamic priority schemes the priority of the task with respect to that of the other tasks changes as tasks are released and completed.
- One of the most well-known dynamic algorithm, **earliest-deadline-first** (EDF), deals with deadlines rather than execution times.
- The ready task with the earliest deadline has the highest priority at any point of time.

Earliest Deadline First (EDF)

Theorem [EDF Bound]

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

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

EDF Example 2

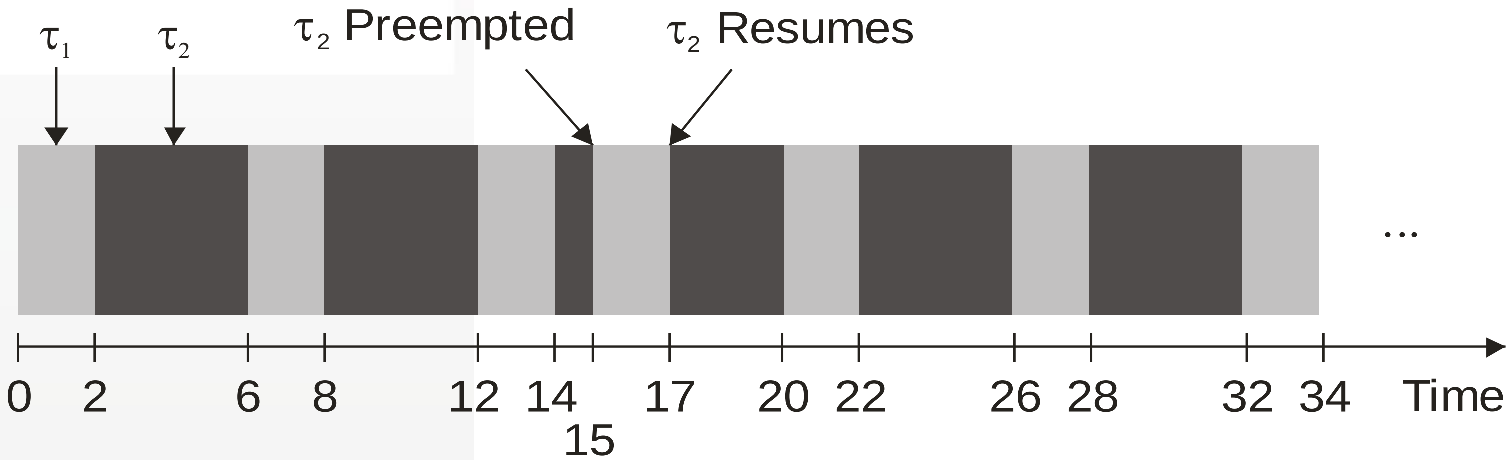
Task set

τ_i	e_i	p_i
τ_1	2	5
τ_2	4	7

EDF Example 2

Task set

τ_i	e_i	p_i
τ_1	2	5
τ_2	4	7



EDF schedule for task set shown. Although τ_1 and τ_2 release simultaneously, τ_1 executes first because its deadline is earliest. At $t = 2$, τ_2 can execute. Even though τ_1 releases again at $t = 5$, its deadline is not earlier than τ_3's. This sequence continues until time $t = 15$ when τ_2 is preempted as its deadline is later ($t = 21$) than τ_1's ($t = 20$). τ_2 resumes when τ_1 completes.

EDF Example

Task	Execution time, e_i	Period, p_i
P_1	25	50
P_2	35	80

- Failed to meet RM scheduling requirements
- Priorities are assigned according to deadlines: the earlier the deadline, the higher the priority; the later the deadline, the lower the priority

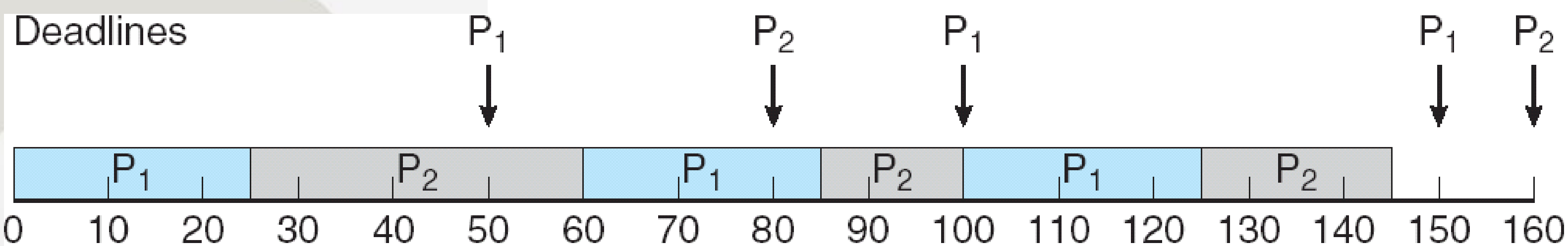
EDF Example

Task	Execution time, e_i	Period, p_i
P_1	25	50
P_2	35	80

- Failed to meet RM scheduling requirements
- Priorities are assigned according to deadlines:

  the earlier the deadline, the higher the priority;

  the later the deadline, the lower the priority



EDF Scheduling

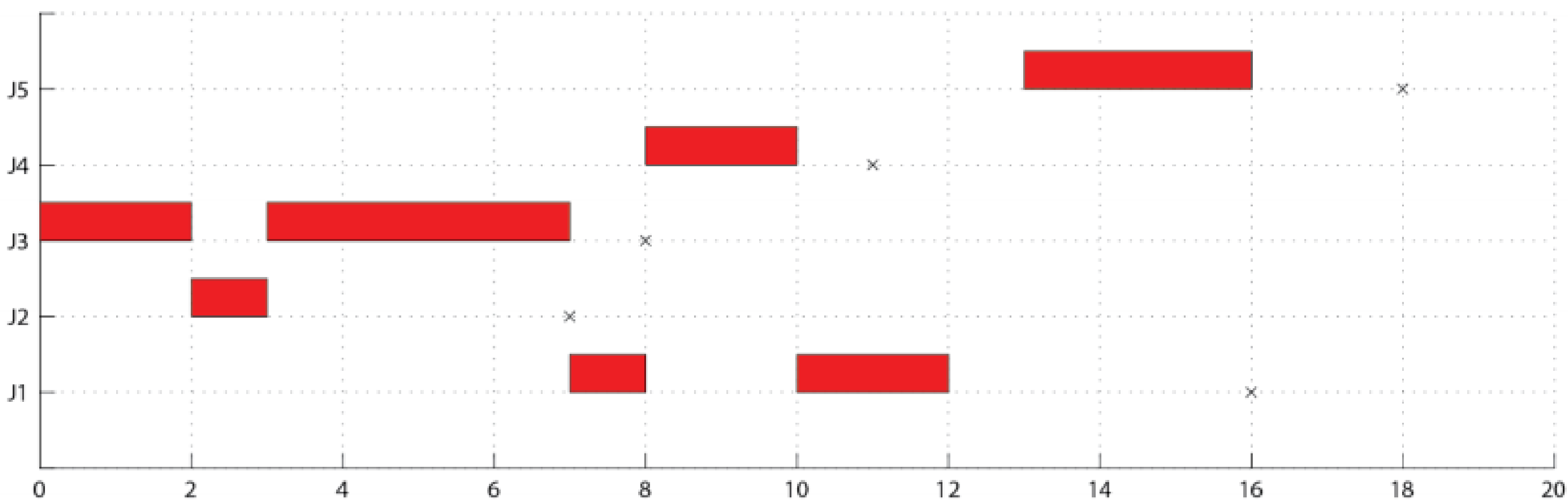
Given are five tasks with arrival times, execution times and deadlines according to the following table.

Task	Release time, r_i	Exec. Time, e_i	Deadline, d_i
J1	0	3	16
J2	2	1	7
J3	0	6	8
J4	8	2	11
J5	13	3	18

Determine the EDF schedule.

EDF Scheduling

Task	Release time, r_i	Exec. Time, e_i	Deadline, d_i
J1	0	3	16
J2	2	1	7
J3	0	6	8
J4	8	2	11
J5	13	3	18



Earliest Deadline First

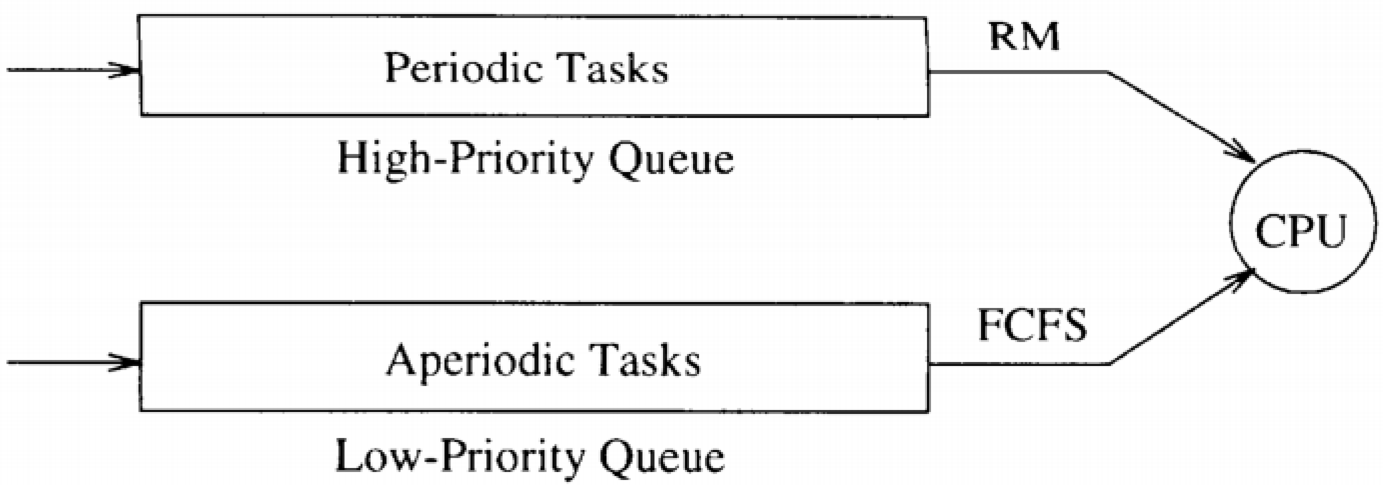
- EDF is more flexible and achieves better utilization than RM.
- However, the timing behavior of a system scheduled according to a fixed-priority algorithm is more predictable than that of a system scheduled according to a dynamic-priority-algorithm.
- In case of overloads, RM is stable in the presence of missed deadlines; the same lower priority tasks miss deadlines every time. There is no effect on higher priority tasks.
- When tasks are scheduled using EDF, it is difficult to predict which tasks will miss their deadlines during overloads.
- A good overrun management scheme is thus needed for such dynamic priority algorithms employed in systems where overload conditions cannot be avoided.

Problem of Mixed Task Sets

- In many applications, there are as well aperiodic as periodic tasks.
- Periodic tasks: time-driven, execute critical control activities with hard timing constraints aimed at guaranteeing regular activation rates.
- Aperiodic tasks: event-driven, may have hard, soft, nonreal-time requirements depending on the specific application.
- Sporadic tasks: Offline guarantee of event-driven aperiodic tasks with critical timing constraints can be done only by making proper assumptions on the environment; that is by assuming a maximum arrival rate for each critical event.

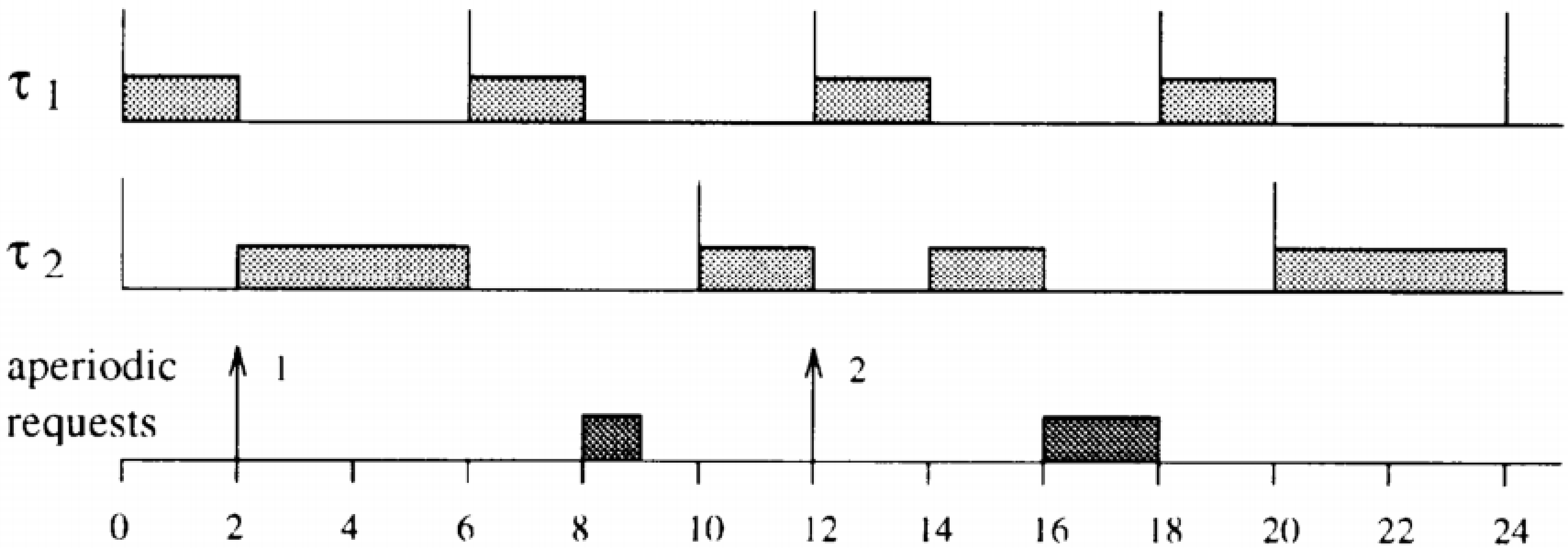
Background Scheduling

- *Simple solution* for RM and EDF scheduling of periodic tasks:
  - Processing of aperiodic tasks in the background, i.e. if there are no periodic request.
  - Periodic tasks are not affected.
  - Response of aperiodic tasks may be prohibitively long and there is no possibility to assign a higher priority to them.



Background Scheduling

- *Example* (rate monotonic periodic schedule):



POSIX Upgrade 1b: Real-Time Extensions

- Priority Scheduling
- Real-Time Signals
- Clocks and Timers
- Semaphores
- Message Passing
- Shared Memory
- Asynch and Synch I/O
- Memory Locking
- Mostly compliant OS: Linux
- Some compliant: VxWorks RTOS

POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines scheduling classes for real-time threads:

1. **SCHED_FIFO** - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
2. **SCHED_RR** - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority
3. **SCHED_OTHER** – undefined and system specific

- Defines two functions for getting and setting scheduling policy:

1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

POSIX Real-Time Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Unable to get policy.\n");
   else {
      if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR) printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO)
printf("SCHED_FIFO\n");
   }
```

POSIX Real-Time Scheduling API (Cont.)

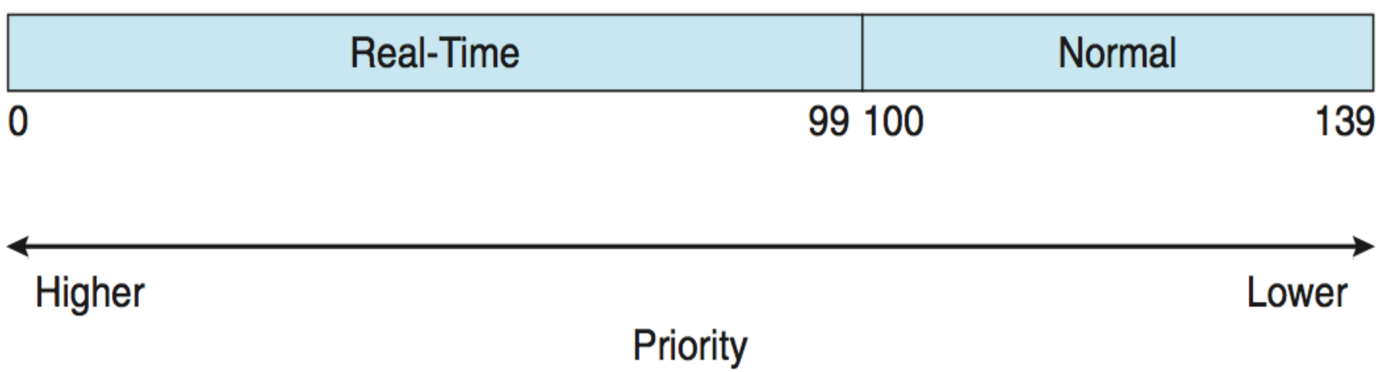
```
    /* set the scheduling policy - FIFO, RR, or OTHER */
    if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
       fprintf(stderr, "Unable to set policy.\n");
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i],&attr,runner,NULL);
    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
       pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```

Linux Scheduling

- ***Completely Fair Scheduler*** (CFS)
- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - 2 scheduling classes included, others can be added
    1. default
    2. real-time
- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



Any Questions?