

Scheduling & RT Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time Scheduling
- Windows 2000 Scheduling
- Scheduler Algorithm Evaluation and Time Measurement

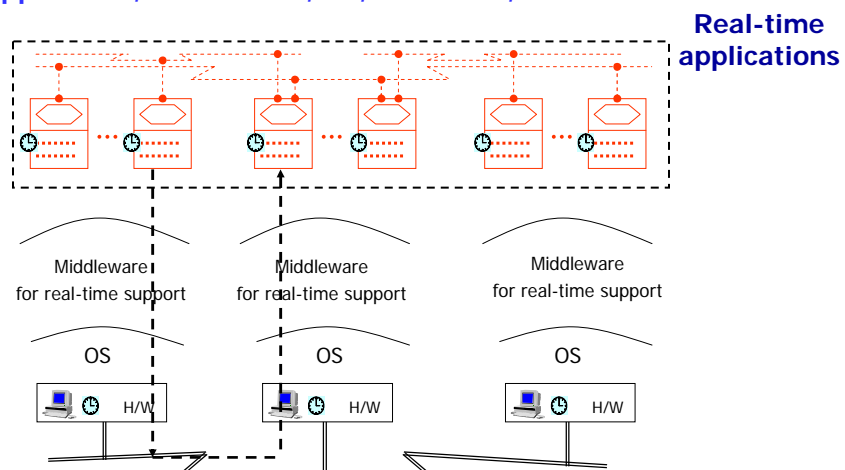
Jan-07 1

UCI
DREAM Lab



Factors impacting response times

- Application, Middleware, OS, Hardware, Comm Network

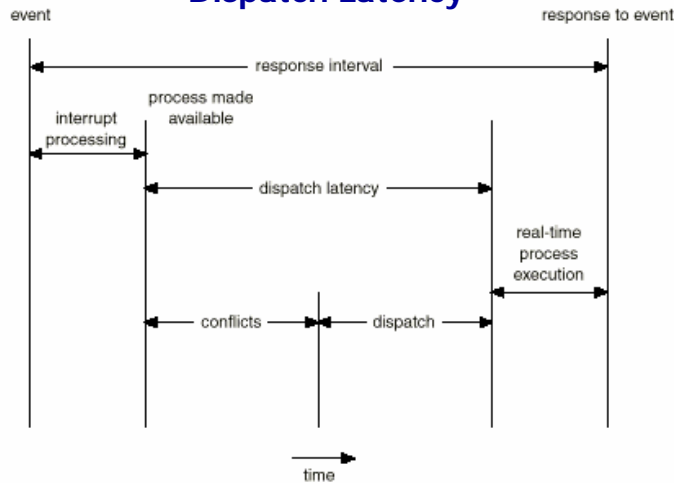


Jan-07 2

UCI
DREAM Lab



Dispatch Latency



- To reduce the dispatch latency: Insert preemption points in long-duration system calls +
- * In Solaris 2, the dispatch latency is
Over 100 ms with preemption disabled and
2ms with preemption enabled

Jan-07 5

UCI
DREAM Lab



Round Robin (RR)

- Each process gets a small unit of CPU time (*time-quantum* or *time-slice*), usually 10-100 milliseconds.
 - After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units at once.
 - No process waits more than $(n-1) * q$ time units.
- Performance
 - q large \Rightarrow FIFO
 - q small \Rightarrow processor sharing
 - q must be large with respect to context switch, otherwise overhead is too high.

Jan-07 6

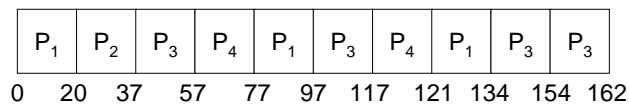
UCI
DREAM Lab



Ex. RR with Time Quantum = 20

<u>Process</u>	<u>Burst Time</u>
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:



- Typically, higher average turnaround than SJF (Shortest Job First), but better *response*.

Jan-07 7

UCI
DREAM Lab



Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer \equiv highest priority).
 - Preemptive
 - non-preemptive
- Problem: *Starvation*
 - low priority processes may never execute.
- Solution: *Aging*
 - as time progresses, the priority of the process that has been waiting increases.

Jan-07 8

UCI
DREAM Lab



Multiple-Processor Scheduling

- *Homogeneous processors* within a multiprocessor are considered here.
- *Load sharing*
 - separate ready Q for each processor vs. one common ready Q
- *Symmetric Multiprocessing (SMP)* – each processor makes its own scheduling decisions.
- *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing.

Jan-07	9
--------	---

UCI
DREAM Lab



Overview of Windows XP Scheduling Policies

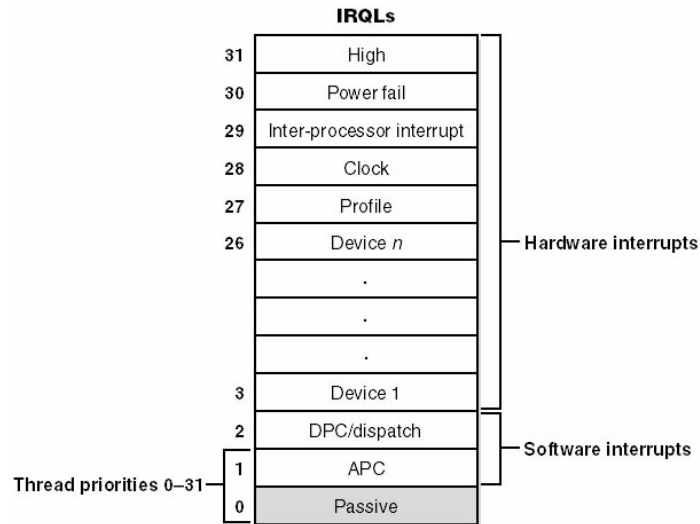
- Windows XP implements a **priority-driven**, **preemptive** scheduling systems.
- Windows XP scheduler treats the **thread** as the basic concurrent execution unit.
 - Suppose there are Process A with 8 threads and Process B with 2 threads with the same priority level, Process A may consume 80 % of CPU time and Process B may consume 20 % of CPU time.
- When a thread is selected to run, it runs for an amount of time called a **quantum**.

Jan-07	10
--------	----

UCI
DREAM Lab



Interrupt Priorities vs. Thread Priorities



Jan-07 11

UCI
DREAM Lab



Thread Dispatching

- Thread dispatching occurs at DPC/dispatch level and is triggered by any of the following events:
 - A thread becomes ready to execute
 - A thread leaves the running state because its time quantum ends, it terminates, or it enters a wait state
 - A thread's priority changes, either because of a system service call or because Windows XP itself changes the priority value.
 - The processor [affinity](#) of a running thread changes.

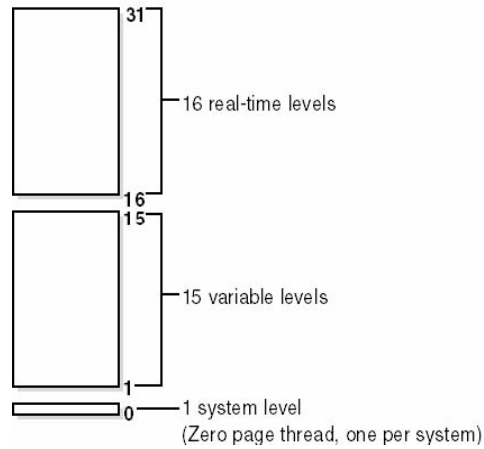
Jan-07 12

UCI
DREAM Lab



Thread Priority Levels

- Windows XP uses 32 priority levels, ranging from 0 through 31.
 - Sixteen **real-time** levels (16-31)
 - Fifteen **variable** levels (1-15)
 - One **system** level (0), reserved for the zero page thread



Jan-07 13

UCI
DREAM Lab



Thread Priority Levels

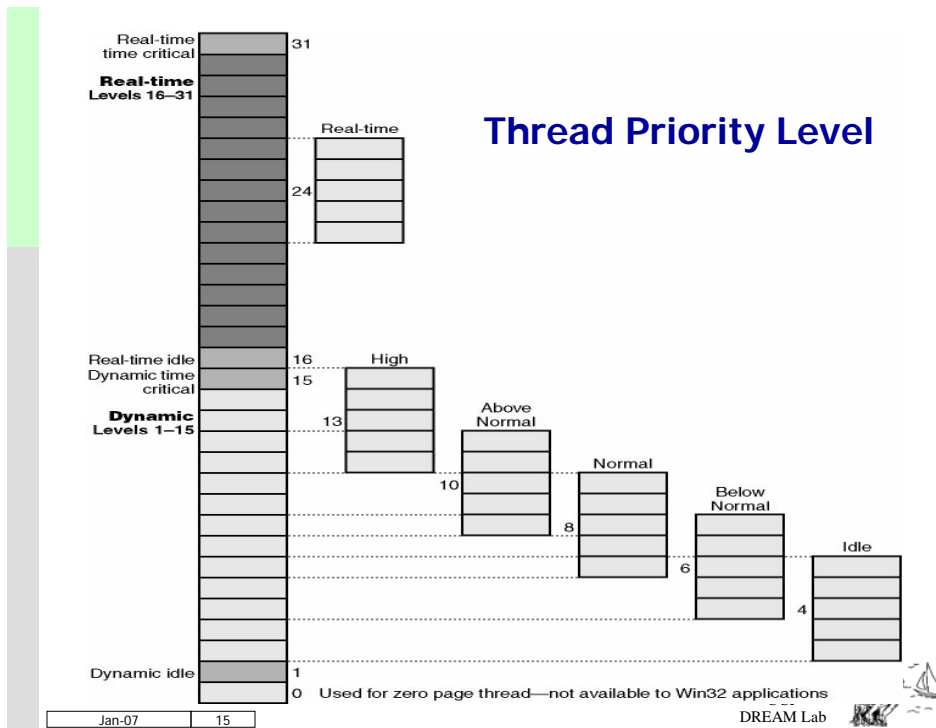
Process Priority Class

Relative Thread Priority	Idle	Below Normal	Normal	Above normal	High	Real-time
Time-critical	15	15	15	15	15	31
Highest	6	8	10	12	15	26
Above normal	5	7	9	11	14	25
Normal	4	6	8	10	13	24
Below normal	3	5	7	9	12	23
Lowest	2	4	6	8	11	22
Idle	1	1	1	1	1	16

Jan-07 14

UCI
DREAM Lab





Win32 Scheduling APIs

- [Suspend/ResumeThread](#)
- [Get/SetPriorityClass](#)
- [Get/SetThreadPriority](#)
- [Get/SetProcessAffinityMask, SetThreadAffinityMask](#)
- [Get/SetThreadPriorityBoost](#)
- [SetThreadIdealProcessor](#)
- [Get/SetProcessPriorityBoost](#)
- [SwitchToThread](#)
- [Sleep](#)
- [SleepEx](#)



Win32 Scheduling APIs

- **Suspend/ResumeThread:**
 - The *SuspendThread* function suspends the specified thread.
 - The *ResumeThread* function decrements a thread's suspend count. When the suspend count is decremented to zero, the execution of the thread is resumed.
- **Get/SetPriorityClass:**
 - The *GetPriorityClass* function retrieves the priority class for the specified process. This value, together with the priority value of each thread of the process, determines each thread's base priority level.
 - The *SetPriorityClass* function sets the priority class for the specified process. This value together with the priority value of each thread of the process determines each thread's base priority level.
- **Get/SetThreadPriority:**
 - The *GetThreadPriority* function retrieves the priority value for the specified thread. This value, together with the priority class of the thread's process, determines the thread's base-priority level.
 - The *SetThreadPriority* function sets the priority value for the specified thread. This value, together with the priority class of the thread's process, determines the thread's base priority level.

Jan-07	17
--------	----

UCI
DREAM Lab



Win32 Scheduling APIs

- **Get/SetProcessAffinityMask, SetThreadAffinityMask:**
 - The *GetProcessAffinityMask* function retrieves the process affinity mask for the specified process and the system affinity mask for the system. A process affinity mask is a bit vector in which each bit represents the processor that a process is allowed to run on. A system affinity mask is a bit vector in which each bit represents the processor that are configured into a system. A process affinity mask is a proper subset of a system affinity mask. A process is only allowed to run on the processors configured into a system.
 - The *SetProcessAffinityMask* function sets a processor affinity mask for the threads of the specified process.
 - The *SetThreadAffinityMask* function sets a processor affinity mask for the specified thread. A thread affinity mask is a bit vector in which each bit represents the processors that a thread is allowed to run on. A thread affinity mask must be a proper subset of the process affinity mask for the containing process of a thread. A thread is only allowed to run on the processors its process is allowed to run on.

Jan-07	18
--------	----

UCI
DREAM Lab



Win32 Scheduling APIs

- **Get/SetThreadPriorityBoost :**
 - The *GetThreadPriorityBoost* function retrieves the priority boost control state of the specified thread.
 - The *SetThreadPriorityBoost* function enables or disables the ability of the system to temporarily boost the priority of a thread.
- **SetThreadIdealProcessor :**
 - The *SetThreadIdealProcessor* function sets a preferred processor for a thread. The system schedules threads on their preferred processors whenever possible.
- **Get/SetProcessPriorityBoost**
 - The *GetProcessPriorityBoost* function retrieves the priority boost control state of the specified process.
 - The *SetProcessPriorityBoost* function enables or disables the ability of the system to temporarily boost the priority of the threads of the specified process.

Jan-07	19
--------	----

UCI
DREAM Lab



Win32 Scheduling APIs

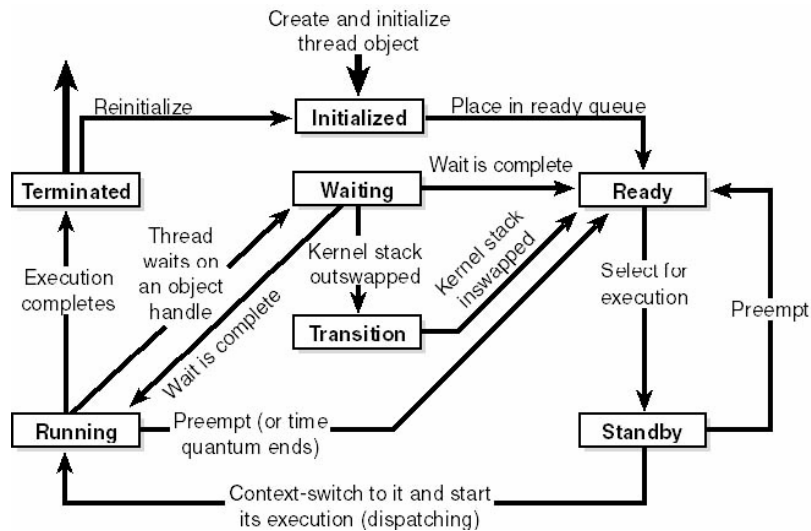
- **SwitchToThread :**
 - The *SwitchToThread* function causes the calling thread to yield execution to another thread that is ready to run on the current processor. The operating system selects the thread to yield to.
- **Sleep :**
 - The *Sleep* function suspends the execution of the current thread for the specified interval.
- **SleepEx :**
 - The *SleepEx* function suspends the current thread until one of the following occurs:
 - An I/O completion callback function is called
 - An asynchronous procedure call (APC) is queued to the thread.
 - The time-out interval elapses

Jan-07	20
--------	----

UCI
DREAM Lab



Thread States in Windows XP



Jan-07 21

UCI
DREAM Lab



Quantum in Windows XP

- Each thread has a integer quantum value that represents how long the thread can run until its quantum expires.
 - By default, **6** for Windows 2000 Professional and **36** on Windows 2000 Server.
- Each time the clock interrupts, the clock-interrupt routine deducts a fixed value (3) from the thread quantum.
 - By default, a thread runs for **2 clock intervals** on Windows 2000 Professional and for **12 clock intervals** on Windows 2000 Server.
 - The length of clock interval varies according to the hardware platform.
 - For example, the clock interval for most x86 uniprocessor is **10 milliseconds**, and for most x86 multiprocessors, 15 milliseconds.

Jan-07 22

UCI
DREAM Lab



Win2K/XP Scheduler Time Measurement (Example II)

- Objective:
 - Measure the context switching time among threads within the same process
- Win32 APIs to be used:
 - *CreateThread ()*: The **CreateThread** function creates a thread to execute within the virtual address space of the calling process.
 - *CreateEvent ()*: The CreateEvent function creates or opens a named or unnamed event object.
 - *OpenEvent ()*: The OpenEvent function opens an existing named event object.
 - *SetEvent ()*: The SetEvent function sets the specified event object to the signaled state.
 - *WaitForSingleObject ()*: The WaitForSingleObject function returns when one of the following occurs:
 - The specified object is in the signaled state.
 - The time-out interval elapses.

Jan-07	23
--------	----

UCI
DREAM Lab



Win2K/XP Scheduler Time Measurement (Example II)

- Main Thread: (major code is shown below)

```
while(1)
{
    QueryPerformanceCounter(&high_resolution_time1);
    SetEvent(hResumeChild);
    WaitForSingleObject(hResumeMain, INFINITE);
}
```
- Child Thread Creation: (Major code is shown below)

```
HANDLE hChild = CreateThread(
    NULL, 0, (LPTHREAD_START_ROUTINE)ChildThreadRoutine, NULL,
    0, NULL);
```
- Child Thread: (major code is shown below)

```
while(1)
{
    WaitForSingleObject(hResumeChild, INFINITE);
    QueryPerformanceCounter(&high_resolution_time2);
    CompTime(high_resolution_time2, high_resolution_time1);
    SetEvent(hResumeMain);
}
```

Jan-07	24
--------	----

UCI
DREAM Lab



Win2K/XP Scheduler Time Measurement (Example II)

- Measure the context switching time among threads within the same process
- Measurement Result:
 - Platform: WinXP, 600Mhz, 256M
 - Context switching time is typically around 1 microsecond.
In the worst case, it is 851 microsecond.

Jan-07	25
--------	----

UCI
DREAM Lab



Win2K/XP Scheduler Time Measurement (Example III)

- Objective:
 - Measure the context switching time among threads within different processes
- Process 1: (major code is shown below)

```
While(1)
{
    QueryPerformanceCounter(&high_resolution_time1);
    SetEvent(hResumeSecond);
    WaitForSingleObject(hResumeFirst, INFINITE);
    QueryPerformanceCounter(&high_resolution_time2);
    CompTime(high_resolution_time2, high_resolution_time1);
}
```

- Process 2: (major code is shown below)

```
While(1)
{
    WaitForSingleObject(hResumeSecond, INFINITE);
    SetEvent(hResumeFirst);
}
```

Jan-07	26
--------	----

UCI
DREAM Lab



Win2K/XP Scheduler Time Measurement (Example III)

- Measure the context switching time among threads within different processes
- Measurement Result:
 - Platform: WinXP, 600Mhz, 256M
 - Two context switching time is typically around 5 microseconds, so the average context switching time is $5/2 = 2.5$ microseconds. In the worst case, it is 970 microseconds.

Jan-07	27
--------	----

UCI
DREAM Lab



Real-Time Scheduling

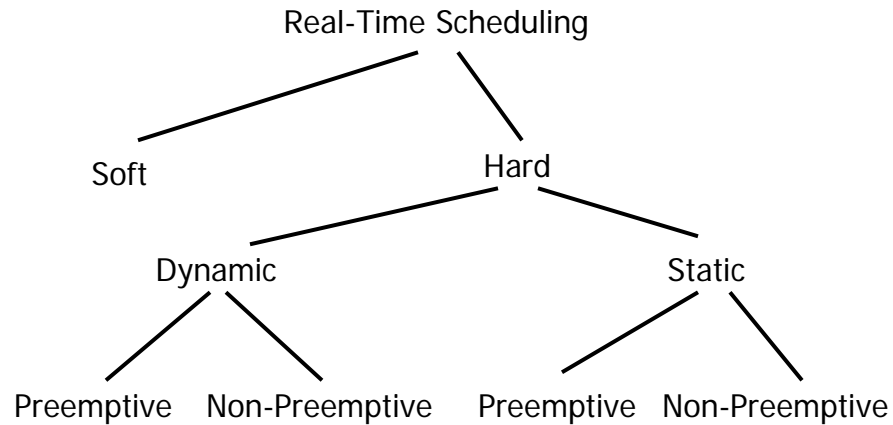
- The scheduling problem
 - A hard real-time system must execute a set of concurrent real-time tasks in such a way that all time-critical tasks meet their specified deadlines.
 - Every task needs computational & data resources to proceed.
 - Therefore, the scheduling problem is concerned with the allocation of the resources to satisfy all timing requirements.

Jan-07	28
--------	----

UCI
DREAM Lab



Classification of Scheduling Algorithm



Jan-07 29

UCI
DREAM Lab



Dynamic vs. Static

- **Dynamic Scheduling**
 - A scheduler is called **dynamic** (or **on-line**) if it makes its scheduling decisions at run time, selecting one out of the current set of ready tasks.
 - It is flexible and adapt to an evolving task scenario
 - The run-time overhead involved in finding a schedule can be substantial
- **Static Scheduling**
 - A scheduler is called **static** (or **pre-run-time**) if it makes its scheduling decisions at compile time.
 - It needs complete prior knowledge about the task-set characteristics.
 - The run-time overhead is small.
- **Hybrid**

Jan-07 30

UCI
DREAM Lab



Preemptive vs. Non-preemptive

- **Preemptive scheduling**
 - The current executing task may be preempted, i.e., interrupted, if a more urgent task requests service.
- **Non-preemptive scheduling**
 - The current executing task will not be interrupted until it decides on its own to release the allocated resources – normally after completion.

Jan-07	31
--------	----

UCI
DREAM Lab



Schedulability Test

- A test that determines whether a set of ready tasks can be scheduled in such a way that each task meets its deadline
 - *Exact* schedulability test
 - The test can determine exactly whether a set of ready tasks are schedulable or not.
 - Known as a NP-complete problem
 - *Necessary* schedulability test
 - The negative test result means that a set of tasks are definitely not schedulable.
 - *Sufficient* schedulability test
 - The positive test result means that a set of tasks are definitely schedulable.
- **Optimal** scheduler
 - If a set of tasks can be scheduled to meet all deadlines under a certain scheduler, then the set of tasks can also be scheduled to meet all deadlines under an optimal scheduler.

Jan-07	32
--------	----

UCI
DREAM Lab



Periodic vs. Sporadic tasks

- Task request time
 - The point in time when a request for a task execution is made.
- Periodic task
 - A task of which all future request times are known *a priori* by adding multiples of the known period to the initial request time if the initial request time is determined.
- Sporadic task
 - A task whose request times are not known *a priori*.
 - To be schedulable in a uniprocessor system, there must be a minimum interval between any two request times of sporadic tasks.
- Aperiodic task
 - A task that has no constraint on the request times of task activation

Jan-07 33

UCI
DREAM Lab



A Necessary Schedulability Test for a set of periodic tasks

- Assuming there is a task set, $\{T_i\}$, of periodic tasks with periods p_i , deadline interval d_i , and execution time c_i .
 - *Deadline interval*: the difference between the deadline of a task and the task request time.
 - *Laxity* (l_i): the difference $d_i - c_i$
- Utilization factor ($\mu_i = c_i / p_i$)
 - The percentage of time the task requires service from a processor
- A necessary schedulability test of a set of periodic tasks:
 - The sum of the utilization factors must be less or equal to n , where n is the number of available processors.

$$\mu = \sum c_i / p_i \leq n$$

Jan-07 34

UCI
DREAM Lab



Static Scheduling

- In static or pre-runtime scheduling, a feasible schedule of a set of tasks is calculated offline.
- The schedule must guarantee all deadlines, considering the resources, precedence, and synchronization requirements of all tasks.
- A **static schedule** is a periodic time-triggered schedule.
- One of the weakness of static scheduling is the assumption of strictly periodic tasks.
 - Although the majority of tasks in hard real-time applications is periodic, there are also sporadic requests for service that have hard deadline requirements.

Jan-07 35

UCI
DREAM Lab



Rate Monotonic Scheduling Framework (Rarely applicable in practice)

- A dynamic preemptive algorithm based on static task priorities
- Assumptions
 - i. The requests for all tasks of the task set $\{T_i\}$, for which hard deadlines exist, are periodic.
 - ii. All task are independent of each other. There exists no precedence constraints or mutual exclusion constraints between any pair of tasks
 - iii. The deadline interval of every task T_i is **equal** to its period p_i .
 - **A severe restriction ! Holds very rarely in practice !!**
 - iv. The required maximum computation time of each task c_i is known *a priori* and is constant.
 - v. The time required for context switching can be ignored.
 - vi. The sum of the utilization factor μ of the n tasks satisfies :

$$\mu = \sum c_i / p_i \leq n(2^{1/n} - 1)$$

Jan-07 36

UCI
DREAM Lab



Rate Monotonic Scheduling Framework (Rarely applicable in practice) (cont)

- The rate monotonic policy involves assignment of static priorities based on the task periods.
 - The task with the shortest period gets the highest static priority.
- At run time, the dispatcher selects the task request with the highest static priority.
- If all assumptions are satisfied, the rate monotonic algorithm guarantees that all tasks will meet their deadlines.
- The algorithm is an optimal static priority assignment algorithm for single processor systems which run the special class of task sets satisfying assumptions i - vi.
- If the task periods are multiples of the period of the highest priority task (a severely restrictive case), the sufficient condition (vi) can be relaxed to become the following necessary & sufficient condition:

$$\mu = \sum c_i / p_i \leq 1$$

- For the task sets not satisfying assumption iii, very little is known about the performance of this algorithm.

Jan-07 37

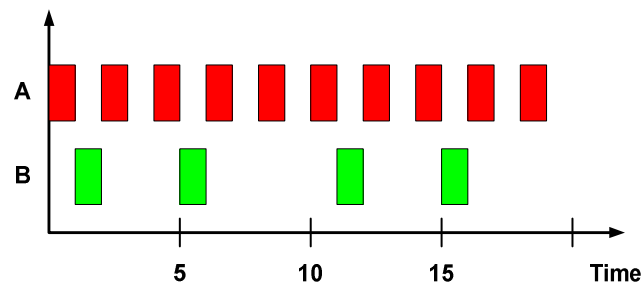
UCI
DREAM Lab



Rate Monotonic Scheduling Framework

Task	Deadline interval	Execution Time
A	2	1
B	5	1

$$\mu = \sum c_i / p_i = \frac{1}{2} + \frac{1}{5} = 0.7 \leq 2 * (2^{1/2} - 1) \approx 0.83$$



Jan-07 38

UCI
DREAM Lab



Early-Deadline-First (EDF) Scheduling Framework

- This algorithm is an optimal dynamic preemptive algorithm for single processor systems which run a **special highly restrictive class of task sets** (i.e., the deadline interval of every task T_i is **equal** to its period p_i).
- Assumptions
 - ~ v. the same as those for rate monotonic algorithm.
 - iv. The processor utilization factor can go up to 1, even when the task periods are not multiples of the smallest period.
- The task with the earliest deadline is assigned the highest dynamic priority.
- The dispatcher selects the task request with the highest priority.
- For the task sets not satisfying assumption iii, very little is known about the performance of this algorithm.

Jan-07 39

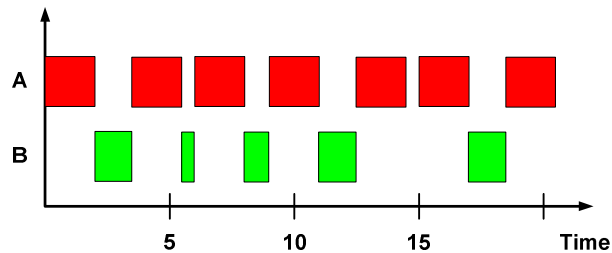
UCI
DREAM Lab



Early-Deadline-First (EDF) Scheduling Framework

Task	Deadline interval	Execution Time
A	3	2
B	5	1.5

$$\mu = \sum c_i / p_i = \frac{2}{3} + \frac{1.5}{5} \approx 0.97 \leq 1$$

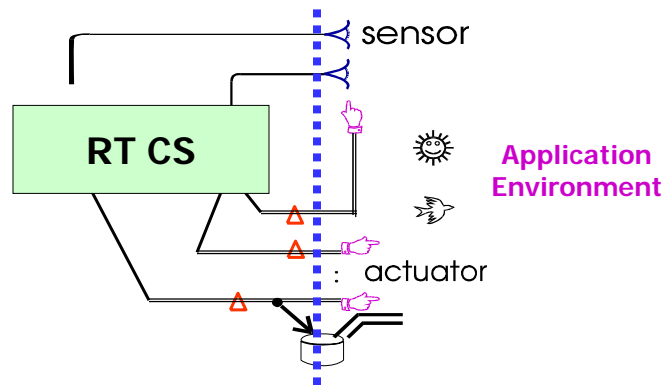


Jan-07 40

UCI
DREAM Lab



How do we determine **deadlines** and **intermediate deadlines** ?



Why has **EDF** been practiced rarely ?

Jan-07 41

UCI
DREAM Lab



Least-Laxity-First (LLF) Scheduling Framework

- This algorithm is another optimal dynamic preemptive algorithm for single processor systems which run a **special highly restrictive class of task sets** (i.e., the deadline interval of every task T_i is **equal** to its period p_i).
- Assumptions
 - ~ vi. the same as those of EDF algorithm
- **Laxity** (l_i): the difference between the time remaining before the deadline and the remaining execution time of a task T_i . ($d_i - c_i$).
- The task with the least laxity is assigned the highest dynamic priority and the dispatcher selects the task request with the highest priority.
- In multiprocessor systems, neither the EDF or the LL algorithm is optimal, although the LL algorithm can handle task scenarios which the EDF algorithm cannot handle.

Jan-07 42

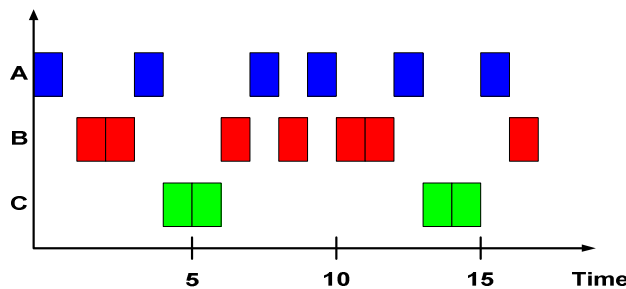
UCI
DREAM Lab



Least-Laxity-First (LLF) Scheduling Framework

Task	Deadline interval	Execution Time
A	3	1
B	5	2
C	9	2

$$\mu = \sum c_i / p_i = \frac{1}{3} + \frac{2}{5} + \frac{2}{9} \approx 0.96 \leq 1$$



Jan-07 43

UCI
DREAM Lab



A Flaw in Pure Least-Laxity-First (LLF) Preemptive Scheduling

- Pure LLF can create an **oscillation**.

Task	Deadline interval	Execution Time	Initial Laxity
A	5	2	3
B	6	2	4
C	10	4.5	5.5

- After Task A executes for one time-unit, what are the laxities for all tasks ?
- A practical variation: **Semi-preemptive LLF**
 - Make a selection of a running task periodically (i.e., after each time-slice) by using LLF.

Jan-07 44

UCI
DREAM Lab



More Realistic General Situations -- Cooperating Tasks & Distributed Systems

- It is difficult to guarantee tight deadlines by dynamic scheduling techniques in a single processor multi-tasking system if mutual exclusion and precedence constraints among the tasks must be considered.
- The situation is more complex in a distributed system, where non-preemptive access to the communication medium must be controlled.

Jan-07	45
--------	----

UCI
DREAM Lab



References

- [1] Inside NT : Inside NT's Interrupt Handling
<http://www.winntmag.com/Articles/Index.cfm?IssueID=25&ArticleID=298>
- [2] Inside the Windows NT Scheduler, Part 1
<http://www.winntmag.com/Articles/Index.cfm?ArticleID=302>

Jan-07	46
--------	----

UCI
DREAM Lab

