CSCE 990: *Real-Time Systems*

Resource Sharing

Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/RealTimeSystems

Resources & Resource Access Control

(Chapter 8 of Liu)

- Until now, we have assumed that tasks are independent.
- We now remove this restriction.
- We first consider how to adapt the analysis discussed previously when tasks access **shared resources**.
- Later, in our discussion of distributed systems, we will consider tasks that have **precedence constraints**.

Shared Resources

- We continue to consider single-processor systems.
- We add to the model a set of ρ serially **reusable resources** $R_1, R_2, \ldots, R_\rho$, where there are v_i units of resource R_i.
  - » **Examples of resources:**
    - Binary semaphore, for which there is one unit.
    - Counting semaphore, for which there may be many units.
    - Reader/writer locks.
    - Printer.
    - Remote server.

Locks

- A job that wants n units of resource R executes a **lock request**, denoted L(R, n).
- It unlocks the resource by executing a corresponding **unlock request**, denoted U(R, n).
- A matching lock/unlock pair is a **critical section**.
- A critical section corresponding to n units of resource R, with an execution cost of e, will be denoted [R, n; e]. If n = 1, then this is simplified to [R; e].

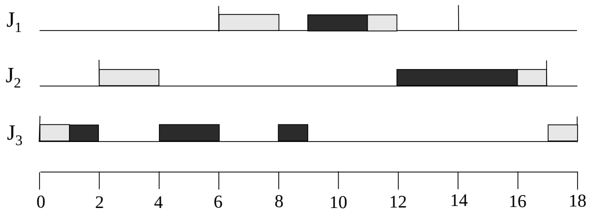
Locks (Continued)

- Locks can be **nested**.
- We will use notation like this:
  - » $[R_1; 14 [R_4, 3; 9 [R_5, 4; 3]]]$
- In our analysis, we will be mostly interested in **outermost critical sections**.
- **Note:** For simplicity, we only have one kind of lock request.
  - » So, for example, we can't actually distinguish between reader locks and writer locks.

Conflicts

- Two jobs have a **resource conflict** if some of the resources they require are the same.
  - » Note that if we had reader/writer locks, then notion of a "conflict" would be a little more complicated.
- Two jobs **contend** for a resource when one job requests a resource that the other job already has.
- The scheduler will always deny a lock request if there are not enough free units of the resource to satisfy the request.

Example

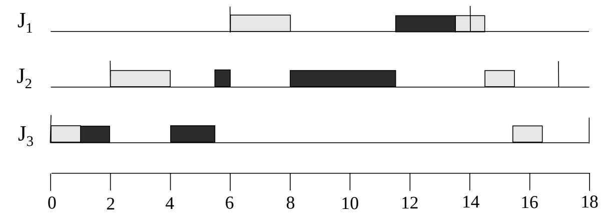


= access of single-unit resource R

Timing Anomalies

When tasks share resources, there may be timing anomalies.

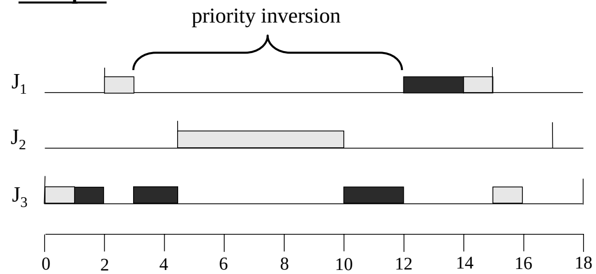
Example: Let us reduce J_3's critical section execution from 4 time units to 2.5. Then J_1 misses its deadline!



Priority Inversions

When tasks share resources, there may be priority inversions.

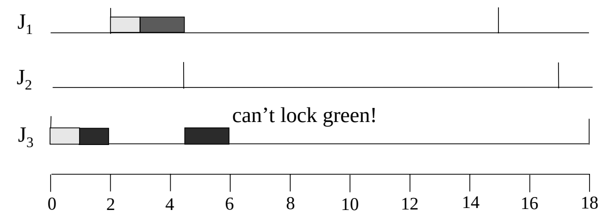
<u>Example:</u>



Deadlocks

When tasks share resources, deadlocks may be a problem.

<u>Example:</u> J_1 accesses green, then red (nested). J_3 accesses red, then green (nested).

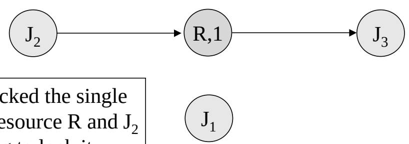


What's a <u>very</u> simple way to fix this problem?

Wait-for Graphs

We will specify blocking relationships using a **wait-for graph**.

Example:

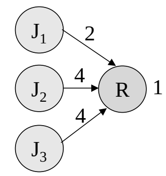


J_3 has locked the single unit of resource R and J_2 is waiting to lock it.

Question: Can we use a wait-for graph to determine if there is a deadlock?

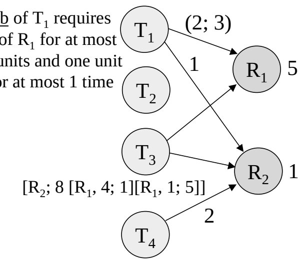
Specifying Resource Requirements

Resource requirements will be specified like this:



J_1 requires the single-unit resource R for 2 time units.

Each job of T_1 requires 2 units of R_1 for at most 3 time units and one unit of R_2 for at most 1 time unit.



Simple resource requirements are shown on edges. Complicated ones by the corresponding task.

Resource Access Control Protocols

- We now consider several protocols for allocating resources that control priority inversions and/or deadlocks.
- From now on, the term "critical section" is taken to mean "outermost critical section" unless specified otherwise.

Nonpreemptive Critical Section Protocol

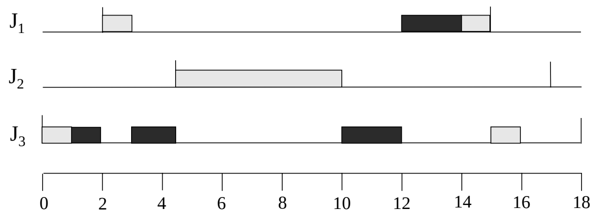
- The simplest protocol: just execute each critical section nonpreemptively.
- If tasks are indexed by priority (or relative deadline in the case of EDF), then task T_i has a **blocking term** equal to $\max_{i+1 \le k \le n} c_k$, where c_k is the execution cost of the longest critical section of T_k.
  - We've talked before about how to incorporate such blocking terms into scheduling analysis.
- **Advantage:** Very simple.
- **Disadvantage:** T_i's blocking term may depend on tasks that it doesn't even have conflicts with.

The Priority Inheritance Protocol

(Sha, Rajkumar, Lehoczky)

Observation: In a system with lock-based resources, priority inversion cannot be eliminated.

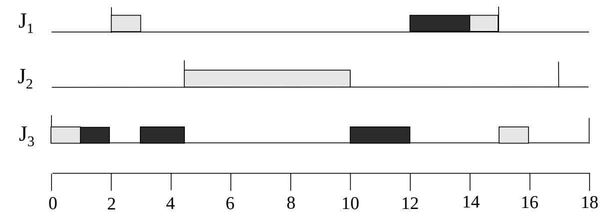
Thus, our only choice is to limit their duration. Consider again this example:



The Priority Inheritance Protocol

The problem here is not the low-priority job J_3 — it's the medium priority job J_2!

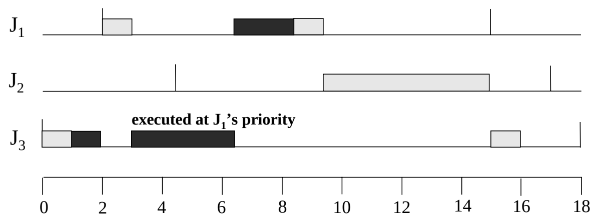
We must find a way to prevent a medium-priority job like this from lengthening the duration of a priority inversion.



The Priority Inheritance Protocol

Priority Inheritance Protocol: When a low-priority job blocks a high-priority job, it *inherits* the high-priority job's priority.

This prevents an untimely preemption by a medium-priority job.



PIP Definition

Each job J_k has an **assigned priority** (e.g., RM priority) and a **current priority** $\pi_k(t)$.

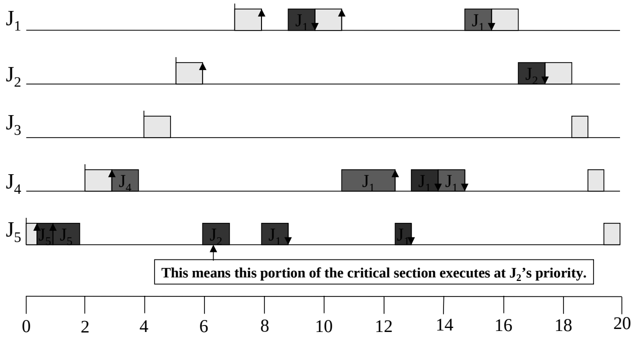
1. **Scheduling Rule:** Ready jobs are scheduled on the processor preemptively in a priority-driven manner according to their current priorities. At its release time t, the current priority of every job is equal to its assigned priority. The job remains at this priority except under the condition stated in rule 3.

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

3. **Priority-Inheritance Rule:** When the requesting job J becomes blocked, the job J_l that blocks J inherits the current priority of J. The job J_l executes at its inherited priority until it releases R (or until it inherits an even higher priority); the priority of J_l returns to its priority $\pi_l(t')$ at the time t' when it acquires the resource R.

A More Complicated Example

(This is slightly different from the example in Figure 8-8 in the book.)



Properties of the PIP

- We have two kinds of blocking with the PIP: **direct blocking** and **inheritance blocking**.
  - In the previous example, J_2 is directly blocked by J_5 over the interval [6,9] and is inheritance blocked by J_4 over the interval [11,15].
- Jobs can transitively block each other.
  - At time 11.5, J_5 blocks J_4 and J_4 blocks J_1.
- The PIP doesn't prevent deadlock.
- A jobs that requires v resources and conflicts with k lower priority jobs can be blocked for min(v,k) times, each for the duration of an outermost CS.
  - It's possible to do much better.

The Priority-Ceiling Protocol

(Sha, Rajkumar, Lehoczky)

- **Two key assumptions:**
  - The assigned priorities of all jobs are fixed (as before).
  - The resources required by all jobs are know *a priori* before the execution of any job begins.
- **Definition:** The **priority ceiling** of any resource R is the highest priority of all the jobs that require R, and is denoted $\Pi(R)$.
- **Definition:** The **current priority ceiling** $\Pi'(R)$ of the system is equal to the highest priority ceiling of the resources currently in use, or Ω if no resources are currently in use (Ω is a priority lower than any real priority).
  - **Note:** I've used ′ instead of ^ due to PowerPoint limitations.

PCP Definition

1. Scheduling Rule:

(a) At its release time t, the current priority $\pi(t)$ of every job J equals its assigned priority. The job remains at this priority except under the conditions of rule 3.

(b) Every ready job J is scheduled preemptively and in a priority-driven manner at its current priority $\pi(t)$.

2. Allocation Rule: Whenever a job J requests a resource R at time t, one of the following two conditions occurs:

(a) R is held by another job. J's request fails and J becomes blocked.

(b) R is free.

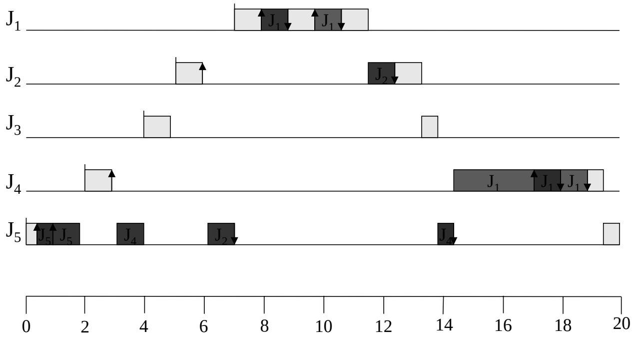
(i) If J's priority $\pi(t)$ is higher than the current priority ceiling $\Pi'(t)$, R is allocated to J.

(ii) If J's priority $\pi(t)$ is not higher than the ceiling $\Pi'(t)$, R is allocated to J only if J is the job holding the resource(s) whose priority ceiling equals $\Pi'(t)$; otherwise, J's request is denied and J becomes blocked.

3. Priority-Inheritance Rule: When J becomes blocked, the job J_l that blocks J inherits the current priority $\pi(t)$ of J. J_l executes at its inherited priority until it releases every resource whose priority ceiling is $\geq \pi(t)$ (or until it inherits an even higher priority); at that time, the priority of J_l returns to its priority $\pi(t')$ at the time t′ when it was granted the resources.

Example

(This is the PCP counterpart of our “complicated” PIP example.)



Properties of the PCP

- The PCP is not greedy.
  - For example, J_4 in the example is prevented from locking the green object, even though it is free.
- We now have three kinds of blocking:
  - » **Direct blocking** (as before).
    - For example, J_5 directly blocks J_2 at time 6.
  - » **Priority-inheritance blocking** (also as before).
    - This doesn’t occur in our example.
  - » **Priority-ceiling blocking** (this is new).
    - J_4 suffers a priority-ceiling blocking at time 3.

Two Theorems

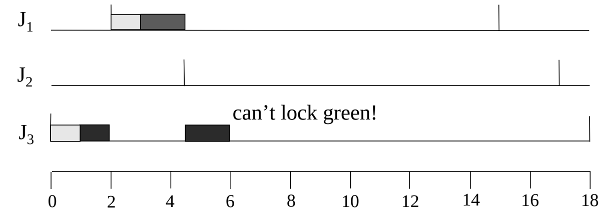
Theorem 8-1: When the resource accesses of a system of preemptive, priority-driven jobs on one processor are controlled by the PCP, deadlock can never occur.

Theorem 8-2: When the resource accesses of a system of preemptive, priority-driven jobs on one processor are controlled by the PCP, a job can be blocked for at most the duration of one critical section.

Deadlock Avoidance

With the PIP, deadlock could occur if nested critical sections are invoked in an inconsistent order. Here's an example we looked at earlier.

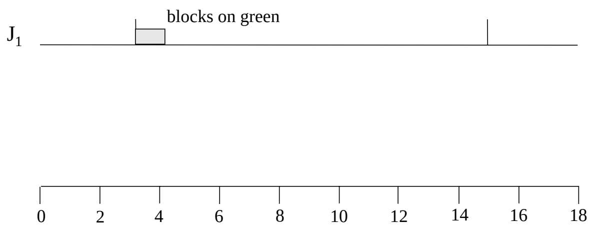
Example: J_1 accesses green, then red (nested). J_3 accesses red, then green (nested).



The PCP would prevent J_1 from locking green. **Why?**

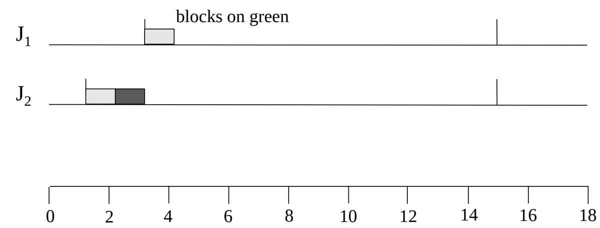
Blocking Term

Suppose J_1 blocks when accessing the green critical section and later blocks when accessing the red critical section.



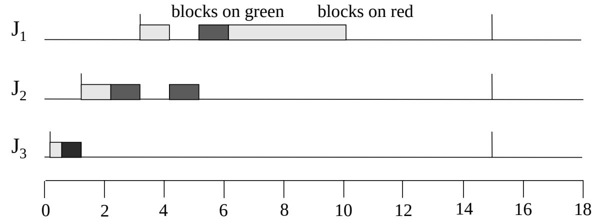
Blocking Term

For J_1 block on green, some lower-priority job must have held the lock on green when J_1 began to execute.



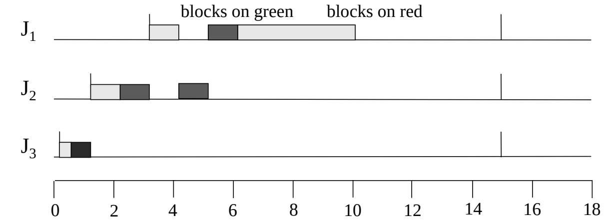
Blocking Term

For J_1 to later block on red, some lower-priority job must have held the lock on red when J_1 began executing.



Blocking Term

Whichever way J_2 and J_3 are prioritized (here, J_2 has priority over J_3), we have a contradiction. **Why?**



Some Comments on the PCP

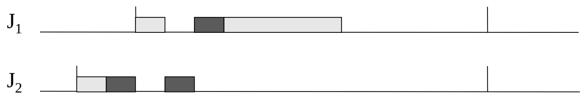
- When computing blocking terms, it is important to carefully consider all three kinds of blockings (direct, inheritance, ceiling).
  - » See the book for an example where this is done systematically (Figure 8-15).
- With the PCP, we have to pay for extra two context switches per blocking term.
  - » Such context switching costs can really add up in a large system.
  - » This is the motivation for the Stack Resource Policy (SRP), described next.

Stack-based Resource Sharing

- So far, we have assumed that each task has its own runtime stack.
- In many systems, tasks can share a run-time task.
- This can lead to memory savings because there is less fragmentation.

Stack-based Resource Sharing (Cont’d)

- If tasks share a runtime stack, we clearly cannot allow a schedule like the following. (**Why?**)



- We must delay the execution of each job until we are sure all the resources it needs are available.

Stack Resource Policy

(Baker)

0. Update of the Current Ceiling: Whenever all the resources are free, the ceiling of the system is Ω. The ceiling $\Pi'(t)$ is updated each time a resource is allocated or freed.

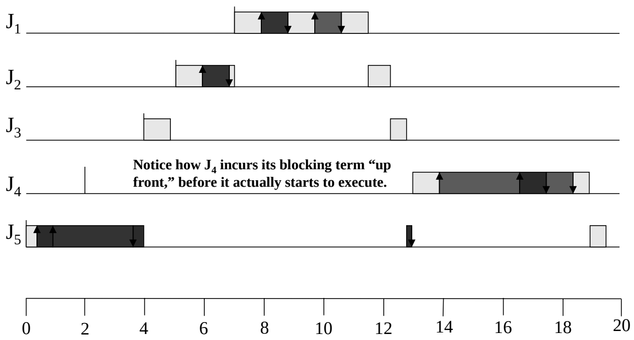
1. Scheduling Rule: After a job is released, it is blocked from starting executing until its assigned priority is higher than the current ceiling $\Pi'(t)$ of the system. At all times, jobs that are not blocked are scheduled on the processor in priority-driven, preemptive manner according to their assigned priorities.

2. Allocation Rule: Whenever a job requests a resource, it is allocated the resource.

Note: Can be implemented using a single runtime stack, but this isn’t *required*.

Example

(This is the SRP counterpart of our “complicated” example.)



Properties of the SRP

- No job is ever blocked once its execution begins.
  - » Thus, there can never be any deadlock.
- The blocking term calculation is the same as with the PCP.
  - » Convince yourself of this!
  - » One difference, though: With the SRP, a job is blocked only before it begins execution, so extra context switches due to blockings are avoided.

Scheduling, Revisited

We have already talked about how to incorporate blocking terms into scheduling conditions.

For example, with **TDA** and **generalized TDA**, we changed our time-demand function by adding a blocking term. For TDA, we got this:

$$w_i(t) = e_i + b_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil \cdot e_k \qquad \text{for } 0 < t \le \min(D_i, p_i)$$

For **EDF**-scheduled systems, we stated the following utilization-based condition:

$$\sum_{k=1}^{n} \frac{e_k}{\min(D_k, p_k)} + \frac{b_i}{\min(D_i, p_i)} \le 1$$

A Closer Look at Dynamic-Priority Systems

- It turns out that this EDF condition is not very tight.
- We now cover a paper by Jeffay that presents a much tighter condition.
  - » Although it may not seem like it on first reading, Jeffay's paper basically reinvents the SRP, but for dynamic-priority systems.
  - » However, the scheduling analysis for dynamic-priority systems given by Jeffay is much better than that found elsewhere.

Scheduling Sporadic Tasks with Shared Resources

(Jeffay)

- In the model of this paper, each task T_i is partitioned into n_i distinct **phases**.
  - In each phase, either no resource is required or exactly one resource is required.
  - If resource R_k is required by T_i's j^{th} phase, then we denote this by $r_{ij} = k$, where $1 \le k \le m$.
  - If no resource is required, then $r_{ij} = 0$.

Single-Phase Systems

In a **single-phase system**, each task is either a critical section that accesses some resource, or a non-critical section that accesses no resource.

Notation: Each task T_i will be denoted by $(s_i, (c_i, C_i, r_i), p_i)$ where:

- s_i is its **release time**;
- c_i is its **minimum execution cost**;
- C_i is its **maximum execution cost**;
- r_i indicates which (if any) **resource** is accessed;
- p_i is its **period**.

Definition: We let P_i denote the period of the "shortest" task that requires resource R_i, i.e., $P_i = \min_{1 \le j \le n} (p_j \mid r_j = i)$.

Necessary Scheduling Condition

Theorem 3.2: Let **T** = {T_1, T_2, …, T_n} be a system of single-phase, sporadic tasks with relative deadlines equal to their periods such that the tasks in **T** are indexed in non-decreasing order by period (i.e., if i < j, then $p_i \le p_j$). If **T** is schedulable on a single processor, then:

1) $\sum_{i=1}^{n} \frac{C_i}{p_i} \le 1$

2) $\left(\forall i : 1 \le i \le n \wedge r_i \ne 0 :: \left(\forall L : P_{r_i} < L < p_i :: L \ge C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor \cdot C_j \right) \right)$

Compare this to the feasibility condition we had for nonpreemptive EDF, which is repeated on the following slide.

Non-preemptive EDF, Revisited

Theorem : Let **T** = {T_1, T_2, …, T_n} be a system of independent, periodic tasks with relative deadlines equal to their periods such that the tasks in **T** are indexed in non-decreasing order by period (i.e., if i < j, then $p_i \le p_j$). **T** can be scheduled by the non-preemptive EDF algorithm if:

1) $\sum_{i=1}^{n} \frac{e_i}{p_i} \le 1$

2) $\left(\forall i : 1 \le i \le n :: \left(\forall L : p_1 < L < p_i :: L \ge e_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor \cdot e_j \right) \right)$

Remember, we showed this condition is also <u>necessary</u> for sporadic tasks.

Proof Sketch of Theorem 3.2

Given our previous discussion of nonpreemptive EDF, Theorem 3.2 should be pretty obvious.

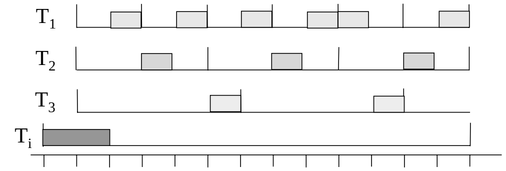
Clearly, if **T** is schedulable, total utilization must be at most one, i.e., condition (1) must hold.

Condition (2) accounts for the worst-case blocking that can be experienced by each task T_i.

Remember, with nonpreemptive EDF, the "worst-case" pattern of job releases occurs when a job of some T_i begins executing (**non-preemptively!**) one time unit before some tasks with smaller periods begin releasing some jobs.

Proof Sketch (Continued)

Here's an illustration:



Moreover, with sporadic tasks, such releases are always possible, and thus if **T** is schedulable, then it is *necessary* to ensure no deadline is missed in the face of job releases like this.

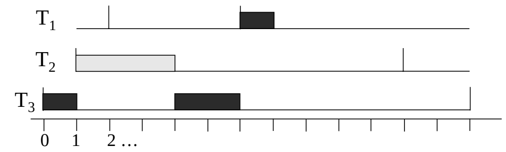
In a single-phase system, we have the same kind of necessary condition, but now a task may only be blocked by a task that accesses a common resource.

EDF-DDM

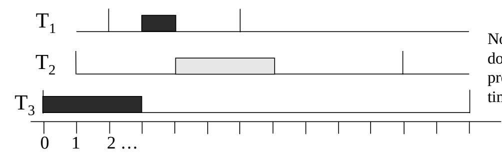
- Our goal now is to define a scheduling algorithm for which the conditions of Theorem 3.2 are necessary.
- Since EDF is optimal in the absence of resources, it makes sense to look at some variant of EDF.
- Remember with the PIP, PCP, and SRP, the idea is to raise a lower-priority job's priority when a blocking occurs.
- With EDF, raising a priority means temporarily "shrinking" the job's deadline.
- The resulting scheme is called **EDF with dynamic deadline modification**.

Example

Here's what can happen without dynamic deadline modification:



Here's the corresponding schedule with dynamic deadline modification:



Notice that T_2 does not preempt T_1 at time 1.

EDF/DDM Definition

- Let t_r be the time when job J of task T_i is released, and let t_s be the time job J starts to execute.
- In the interval $[t_r, t_s)$, J's deadline is $t_r + p_i$, just like with EDF.
  - » This is called J's **initial deadline**.
- At time t_s, J's deadline is changed to $\min(t_r + p_i, (t_s + 1) + P_{r_i})$.
  - » This is called J's **contending deadline**.

Sufficient Condition for EDF/DDM

Theorem 3.4: Let $\mathbf{T} = \{T_1, T_2, \ldots, T_n\}$ be a system of single-phase, sporadic tasks with relative deadlines equal to their periods such that the tasks in **T** are indexed in non-decreasing order by period (i.e., if $i < j$, then $p_i \leq p_j$). The EDF/DDM discipline will succeed in scheduling **T** if conditions (1) and (2) from Theorem 3.2 hold.

Thus, by Theorem 3.2, (1) and (2) are **feasibility conditions**.

Not surprisingly, the proof of Theorem 3.4 is very similar to the corresponding proof we did for nonpreemptive EDF systems.

Proof of Theorem 3.4

Suppose conditions (1) and (2) hold for **T** but a deadline is missed. Let t_d be the earliest point in time at which a deadline is missed.

There are two cases.

<u>Case 1:</u> No job with an <u>initial</u> deadline after time t_d is scheduled prior to time t_d. The analysis is just like with preemptive EDF.

As before, let t_{-1} be the last "idle instant". (This is denoted t_0 in the paper, but I've used t_{-1} to be consistent with previous proofs.)

Because a deadline is missed at t_d, demand over $[t_{-1}, t_d]$ exceeds $t_d - t_{-1}$. In addition, this demand is at most $\sum_{j=1,..,n} \lfloor (t_d - t_{-1})/p_j \rfloor \cdot C_j$.

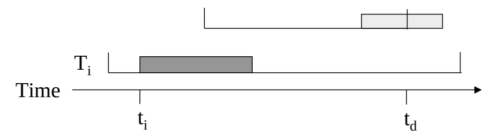
Thus, we have $t_d - t_{-1} < \sum_{j=1,..,n} \lfloor (t_d - t_{-1})/p_j \rfloor \cdot C_j \leq \sum_{j=1,..,n} [(t_d - t_{-1})/p_j] \cdot C_j$.

This implies utilization exceeds one, which contradicts condition (1).

Proof (Continued)

<u>Case 2:</u> Some job with an <u>initial</u> deadline after time t_d is scheduled prior to time t_d.

Let T_i be the task with the last job with an initial deadline after t_d that is scheduled prior to t_d. Then, we have the following:

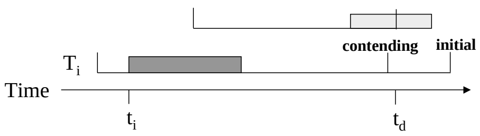


Let us bound the processor demand in $[t_i, t_d]$ … (This is where things start to get a little different from the nonpreemptive EDF proof.)

Proof (Continued)

Case 2a: T_i's contending deadline is less than or equal to t_d.

This means T_i must be a resource requesting task. We have the following:



The proof for this subcase is very much like Case 2 in the nonpreemptive EDF proof (we get a contradiction of condition (2)).

Proof (Continued)

- Observe the following:
  - » Other than task T_i, no task with a period greater than or equal to $t_d - t_i$ executes in the interval $[t_i, t_d]$.
    - Such a task would contradict our choice of T_i.
  - » Other than T_i, no task that executes in $[t_i, t_d]$ could have been invoked at time t_i.
  - » The processor is fully utilized in $[t_i, t_d]$.

Proof (Continued)

From these facts, we conclude that demand over $[t_i, t_d]$ is less than or equal to

$$C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{t_d - (t_i + 1)}{p_j} \right\rfloor \cdot C_j.$$

Let $L = t_d - t_i$. We have $p_i > L > P_{r_i}$. (**Why?**) Also,

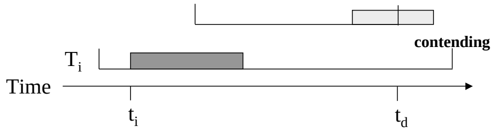
$$L < C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L - 1}{p_j} \right\rfloor \cdot C_j.$$

This contradicts condition (2).

Proof (Continued)

Case 2b: T_i's contending deadline is greater than t_d.

This means either T_i doesn't request any resource or $(t_i + 1) + P_{r_i} > t_d$. We have the following:

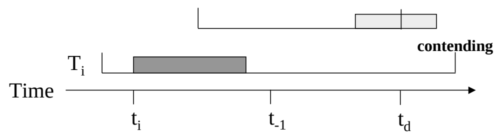


T_i is preemptable by any job whose period lies with $[t_i, t_d]$. (**Why?**)

Proof (Continued)

Let $t_{-1} > t_i$ be the later of the end of the last idle period in $[t_i, t_d]$ or the time T_i last stops executing prior to t_d.

All invocations of tasks occurring prior to t_{-1} with deadlines less than or equal to t_d must have completed executing by t_{-1}. (**Why?**)



As in Case 1, we can show that demand over $[t_{-1}, t_d]$ exceeds $t_d - t_{-1}$, which implies that condition (1) is violated.

Multi-Phase Systems

Notation: In a multi-phase system, each task T_i is denoted by $(s_i, (c_{ij}, C_{ij}, r_{ij}), p_i)$, $1 \le i \le n$, $1 \le j \le n_i$, where:

- s_i is its **release time**;
- n_j is the **number of phases** in each job of T_i;
- c_{ij} is the **minimum execution cost** of the j^{th} phase;
- C_{ij} is the **maximum execution cost** of the j^{th} phase;
- r_{ij} indicates which (if any) **resource** is accessed in the j^{th} phase;
- p_i is its **period**.

Definition: We let $P_{r_{ik}} = \min_{1 \le j \le n} (p_j \mid r_{jl} = r_{ik}$ for some l in the range $1 \le l \le n_j)$.

Definition: The **execution cost** of T_i is $E_i = \sum_{k = 1,\ldots,n_i} C_{ik}$.

Necessary Scheduling Condition

Theorem 4.1: Let **T** = {T_1, T_2, …, T_n} be a system of multi-phase, sporadic tasks with relative deadlines equal to their periods such that the tasks in **T** are indexed in non-decreasing order by period (i.e., if $i < j$, then $p_i \le p_j$). If **T** is schedulable on a single processor, then:

1) $\sum_{i=1}^{n} \frac{E_i}{p_i} \le 1$

2) $\left(\forall i, k : 1 \le i \le n \wedge 1 \le k \le n_i \wedge r_{ik} \ne 0 ::\right.$

$$\left(\left(\forall L : P_{r_{ik}} < L < p_i - S_{ik} :: L \ge C_{ik} + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor \cdot E_j \right)\right)$$

where $S_{ik} = \begin{cases} 0 & \text{if } k = 1 \\ \sum_{j=1}^{k-1} c_{ij} & \text{if } 1 < k \le n_i \end{cases}$

Ugh!

EDF/DDM for Multi-phase Systems

- Let t_r be the time when job J of task T_i is released, and let t_{sk} be the time job J's k^{th} phase starts to execute.
- In the interval [t_r, t_s), J's deadline is $t_r + p_i$, just like with EDF.
- At time t_{sk}, J's deadline is changed to min($t_r + p_i$, (t_{sk} + 1) + $P_{r_{ik}}$).
- When one of J's phases completes, its deadline immediately reverts to $t_r + p_i$.
- Note that this algorithm prevents a job from beginning execution until all the resources it requires are available, i.e., this is just a dynamic-priority SRP.

Sufficient Condition for EDF/DDM

Theorem 4.3: Let **T** = {T_1, T_2, …, T_n} be a system of multi-phase, sporadic tasks with relative deadlines equal to their periods such that the tasks in **T** are indexed in non-decreasing order by period (i.e., if $i < j$, then $p_i \le p_j$). The EDF/DDM discipline will succeed in scheduling **T** if conditions (1) and (2) from Theorem 4.1 hold.

Thus, by Theorem 4.1, (1) and (2) are **feasibility conditions** for multi-phase, sporadic task systems.

We will not cover the proofs of Theorems 4.1 and 4.3 in class, but you should read through them in the paper.

An Alternative to Critical Sections

- Critical sections are often used to implement software shared objects.
  - » **Example:** producer/consumer buffer.
- Such objects actually can be implemented without using critical sections or related mechanisms.
- Such shared-object algorithms are called **nonblocking algorithms.**
- **Bottom Line:** We can avoid priority inversions altogether when implementing software shared objects.

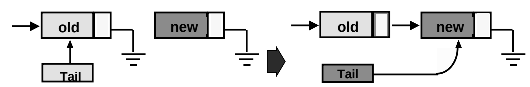
Nonblocking Algorithms

- Two variants:
  - » **Lock-Free:**
    - Perform operations “optimistically”.
    - Retry operations that are interfered with.
  - » **Wait-Free:**
    - No waiting of any kind:
      - No busy-waiting.
      - No blocking synchronization constructs.
      - No unbounded retries.
- Recent research at UNC has shown how to account for lock-free and wait-free overheads in scheduling analysis.
- First, some background …

Lock-Free Example

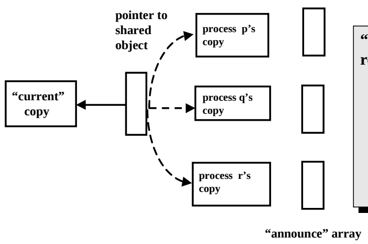
```
type  Qtype = record v: valtype; next:  pointer to Qtype end
shared var  Tail:  pointer to Qtype;
local var  old, new: pointer to Qtype

procedure Enqueue (input: valtype)
    new := (input,  NIL);
   repeat    old := Tail
   until  CAS2(&Tail, &(old->next), old, NIL, new, new)
```



Wait-Free Algorithms

(Herlihy's Helping Scheme)



Algorithm:

"announce" operation;
retry until done:
create local copy of the object;
apply all announced operations on local copy;
attempt to make local copy the "current" copy using a strong synchronization primitive

Can only retry once!

Disadvantage: Copying overhead.

Using Wait-Free Algorithms in Real-time Systems

- On uniprocesors, helping-based algorithms are not very attractive.
  - » Only high-priority tasks help lower-priority tasks.
    - – Similar to **priority inversion**.
  - » Such algorithms can have high overhead due to copying and having to use costly synchronization primitives.
    - – Some wait-free algorithms avoid these problems and *are* useful.
    - – **Example:** "Collision avoiding" read/write buffers.
- On the other hand, on multiprocessors, wait-free algorithms may be the best choice.

Using Lock-Free Objects on Real-time Uniprocessors

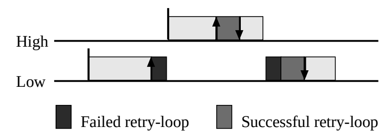
(Anderson, Ramamurthy, Jeffay)

- **Advantages of Lock-free Objects:**
  - » No priority inversions.
  - » Lower overhead than helping-based wait-free objects.
  - » Overhead is charged to low-priority tasks.
- **But:**
  - » Access times are **potentially unbounded**.

Scheduling with Lock-Free Objects

On a uniprocessor, lock-free retries really aren't unbounded.

A task fails to update a shared object only if **preempted** during its object call.



Can compute a bound on retries by counting preemptions.

RM Sufficient Condition

Assume **rate-monotonic** priority assignment.

Sufficient Scheduling Condition:

$$\left(\forall i :: \left(\exists t : 0 < t \le p_i :: \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j + \sum_{j=1}^{i-1} \left\lceil \frac{t}{p_j} \right\rceil s \le t \right)\right)$$

In this condition, **s** is the time to update a lock-free object (one retry loop iteration).

We are assuming at this point that all retry loops have the same cost.

Proof of RM Condition

The proof strategy should be very familiar to you by now.

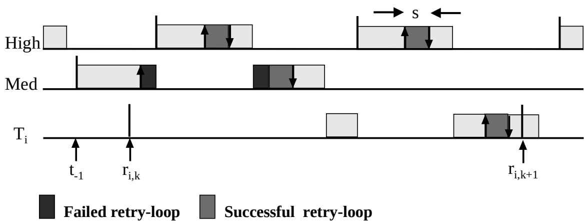
To Prove: If a task set is not schedulable, then the sufficient condition does not hold, i.e.,

$$\left(\exists i :: \left(\forall t : 0 < t \le p_i :: \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j + \sum_{j=1}^{i-1} \left\lceil \frac{t}{p_j} \right\rceil s > t \right)\right)$$

Setting Up the Proof…

Let the k[th] job of T_i be the first to miss its deadline.

Let t_{-1} be the latest "idle instant" before $r_{i,k+1}$.



Intuition

If a task set is not schedulable, then at all instants t in $(t_{-1}, r_{i,k+1}]$,

> the demand placed on the processor by T_i and higher-priority tasks in $[t_{-1}, t)$ is greater than the available processor time in $[t_{-1}, t)$.

Suppose not:

- Case $t \in (t_{-1}, r_{i,k}]$: Contradicts choice of t_{-1}.
- Case $t \in (r_{i,k}, r_{i,k+1}]$: T_i's deadline at $r_{i,k+1}$ is not missed.

Finishing the Proof ...

For any t in $(t_{-1}, r_{i,k+1}]$, the following holds.

available processor time in $[t_{-1}, t)$

< demand due to T_i and higher-priority jobs in $[t_{-1}, t)$

= demand due to job releases of T_i and higher-priority tasks
+ demand due to failed loop tries in T_i and higher-priority tasks

≤ $\sum_{j=1\ldots,i}$ (number of jobs of T_j released in $[t_{-1}, t)$) $\cdot\, e_j$
+ $\sum_{j=1\ldots,i-1}$(number of preemptions T_j can cause in T_i and higher-priority tasks) $\cdot$ (cost of failed loop try)

… Finishing the Proof

Hence, for any t in $(t_{-1}, r_{i,k+1}]$,

$$t - t_{-1} < \sum_{j=1}^{i} \left\lceil \frac{t - t_{-1}}{p_j} \right\rceil e_j + \sum_{j=1}^{i-1} \left\lceil \frac{t - t_{-1}}{p_j} \right\rceil s.$$

Replacing $t - t_{-1}$ by t´ in $(0, r_{i,k+1} - t_{-1}]$,

$$\boxed{t' < \sum_{j=1}^{i} \left\lceil \frac{t'}{p_j} \right\rceil e_j + \sum_{j=1}^{i-1} \left\lceil \frac{t'}{p_j} \right\rceil s.}$$

EDF Sufficient Condition

Assume **earliest-deadline-first** priority assignment.

Sufficient Condition:

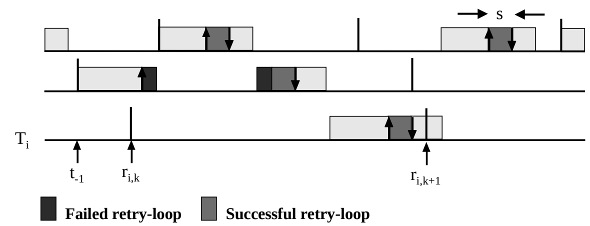
$$\sum_{j=1}^{N} \frac{e_j + s}{p_j} \leq 1$$

As before … **To Prove:** If a task set T is not schedulable, then

$$\sum_{j=1}^{N} \frac{e_j + s}{p_j} > 1$$

Setting Up the Proof...

Same set-up as before…



Intuition

If a task set is not schedulable, then the demand placed on the processor in $[t_{-1}, r_{i,k+1})$ by jobs with deadlines at or before $r_{i,k+1}$ is greater than the available processor time in $[t_{-1}, r_{i,k+1}]$.

Finishing the Proof ...

available processor time in $[t_{-1}, r_{i,k+1}]$

< demand due to jobs with deadlines $\le r_{i,k+1}$

= demand due to releases of those jobs
+ demand due to failed loop tries in those jobs

$\le$ $\Sigma_{j=1,\ldots,N}$ [number of jobs of T_j with deadlines at or before $r_{i,k+1}$ released in $[t_{-1}, r_{i,k+1})$] $\cdot e_j$
+ $\Sigma_{j=1,\ldots,N}$ (number of preemptions T_j can cause in such jobs) $\cdot$ (cost of failed loop try)

… Finishing the Proof

Hence,

$$r_{i,k+1} - t_{-1} < \sum_{j=1}^{N} \left\lceil \frac{r_{i,k+1} - t_{-1}}{p_j} \right\rceil e_j + \sum_{j=1}^{N} \left\lceil \frac{r_{i,k+1} - t_{-1}}{p_j} \right\rceil s,$$

which implies,

$$r_{i,k+1} - t_{-1} < \sum_{j=1}^{N} \frac{r_{i,k+1} - t_{-1}}{p_j} e_j + \sum_{j=1}^{N} \frac{r_{i,k+1} - t_{-1}}{p_j} s.$$

Canceling $r_{i,k+1} - t_{-1}$ yields

$$\boxed{1 < \sum_{j=1}^{N} \frac{e_j}{p_j} + \sum_{j=1}^{N} \frac{s}{p_j}.}$$

Comparison of Lock-Free & Lock-Based

- It can be shown analytically that lock-free wins over lock-based if:
  - » (lock-free access cost) ≤ (lock-based access cost)/2.
    - For many objects, this will be the case, because with a lock-based implementation, you get one object access for the price of many (due to all the kernel objects that have to be accessed).
- Breakdown utilization experiments involving randomly-generated task sets show that lock-free is *likely* to win if:
  - » (lock-free access cost) ≤ (lock-based access cost).

Better Scheduling Conditions

- Previous conditions perform poorly when retry loop costs vary widely.
- Also, they over-count interferences (not *every* preemption causes an interference).
- **Question:** How to incorporate different retry loop costs?
- **Answer:** Use **linear programming**.
  - » Can apply linear programming to both RM and EDF (and also DM).
  - » We only consider RM here.

Linear-Programming RM Condition

(Anderson and Ramamurthy)

Definition: $$E_i(t) \equiv \sum_{j=1}^{i} \sum_{v=1}^{w(j)} \sum_{l=1}^{j-1} m_l^{j,v}(t) s_l^{j,v}$$

w(j) - Number of phases of T_j.

$m_l^{j,v}(t)$ - Number of interferences in T_j's v^{th} phase due to T_l in an interval of length t.

$s_l^{j,v}$ - Cost of one such interference.

Approach: View $E_i(t)$ as a linear expression, where $m_l^{j,v}(t)$ are the variables.

Maximize $E_i(t)$ subject to some constraints.

LP RM Condition (Continued)

Example Constraints (the easy ones):

$$\left(\forall i, j : j < i :: \sum_{v=1}^{w(i)} m_j^{i,v}(t) \le \left\lceil \frac{t+1}{p_j} \right\rceil\right)$$

$$\left(\forall i :: \sum_{j=1}^{i} \sum_{v=1}^{w(j)} \sum_{l=1}^{j-1} m_l^{j,v}(t) \le \sum_{j=1}^{i-1} \left\lceil \frac{t+1}{p_j} \right\rceil\right)$$

Let $E_i'(t)$ be an upper bound on $E_i(t)$ obtained by linear programming.

RM Condition: $$\left(\exists t : 0 < t \le p_i :: \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil e_j + E_i'(t-1) \le t\right)$$