Real-Time Computing

Definition of Real-Time Computing

- Real-time computing is computing in which the **time plays an essential role** in the defining the **correctness** of the system.

- This includes:
  - Computations that ***measure*** time
  - Computations that must ***meet*** **deadlines**
  - Computations that ***synchronize*** other computations ***based on time***

Obligatory Inspiring Example

http://www.youtube.com/watch?v=sTBIr65cL_E

- July 20, 1969, landing module 10,000 feet above the Moon:
  - Houston: “Eagle, you’re go for a landing. Altitude 1600 [feet]”

    ...
  - Houston: “One minute [of fuel left]”.

    ...
  - Lander: 100 feet, 3 1/2 down, 9 forward.”

    ...
  - Houston: “30 Seconds” [of fuel left]

    ...
  - Lander: “OK, engine stop.”

Example, continued

(from Briand and Roy,
Meeting deadlines in hard real-time systems, IEEE Press, 1999)

- "During the descent of the Eagle [landing module], an incorrect switch position caused the analog-to-digital conversion circuit of the rendezvous to send some **bursts of high-priority requests** to the computer.

- After 15 percent of the computer resources were tied up in responding to the spurious requests, **jobs began to miss their deadlines**.

- A hardware recovery mechanism **detected the timing fault** and restarted the computer."

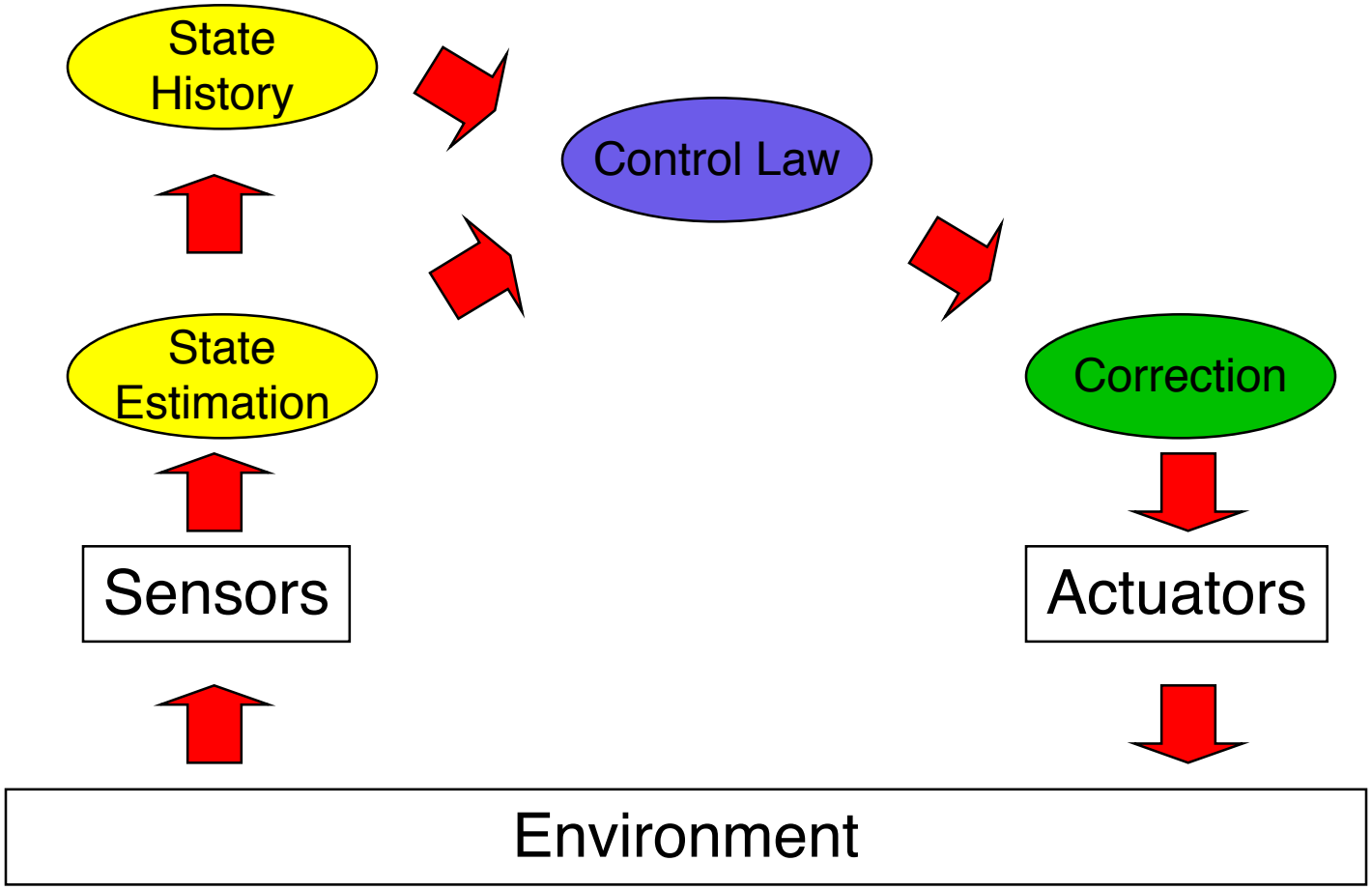
Control Loops

- Autonomous, or semi-autonomous vehicles or robots require control loops to govern their motion.
- Computing equipment may also require such loops (e.g. transfers to/from rotational media).
- Sporadic computational tasks may need to be handled by the same system.

Example of Real-Time: Control Loop

- A vehicle is governed by a **control law**, that takes input from sensors and produces output to actuators. The control law is based on a sampling and update rate.
- From **sensor** input, an **estimate** of position, velocity, etc. is computed.
- Based on the computed estimate, appropriate adjustment is made to **actuators**. For example, the adjustment may be based on difference between estimated state and desired state.
- **All computations must complete in time for the next sample.**

Example of Real-Time: Control Loop



Examples of Control Laws

- Open-loop vs. Closed-loop control
- Bang-Bang (“On-Off”) controller
- P controller (proportional controller)
- PI controller (proportional-integral controller)
- PID controller (proportional-integral-derivative controller)

Comparison of Control Laws

- Bang-Bang:

  {Measure error;
  Fully open the valve for a fixed time (or not)}*

- P controller (“proportional”)

  {Measure error;
  Open the valve in proportion to error}*

- PI controller (proportional-integral)

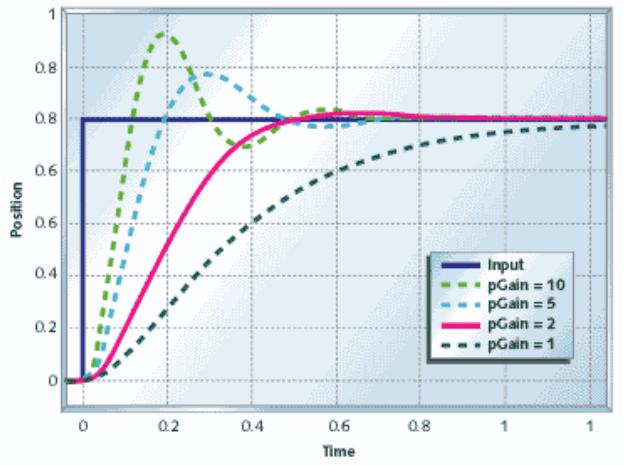
  {Measure error;
  Open the valve in proportion to integral of error}*

- PID controller (proportional-integral-derivative)

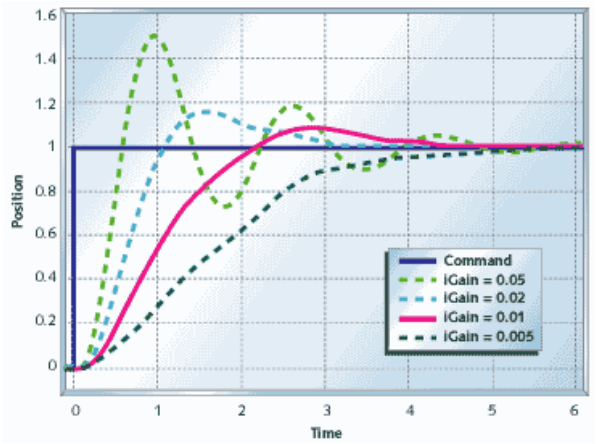
  Similar to PI, with added term based on derivative of output

P, PI, and PID Controller Responses

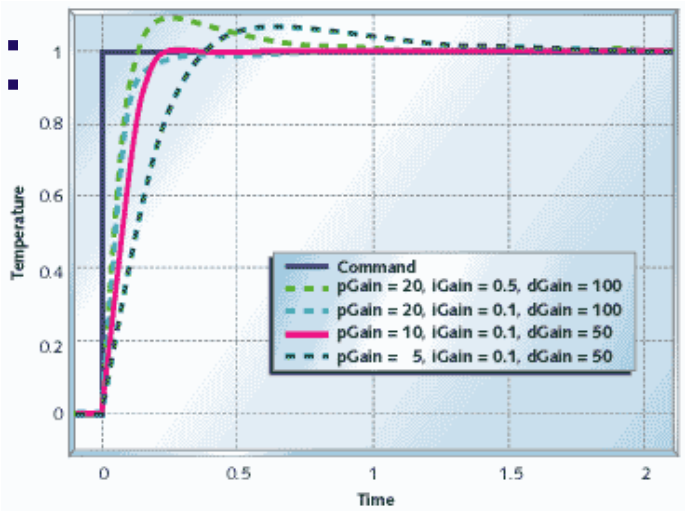
P:



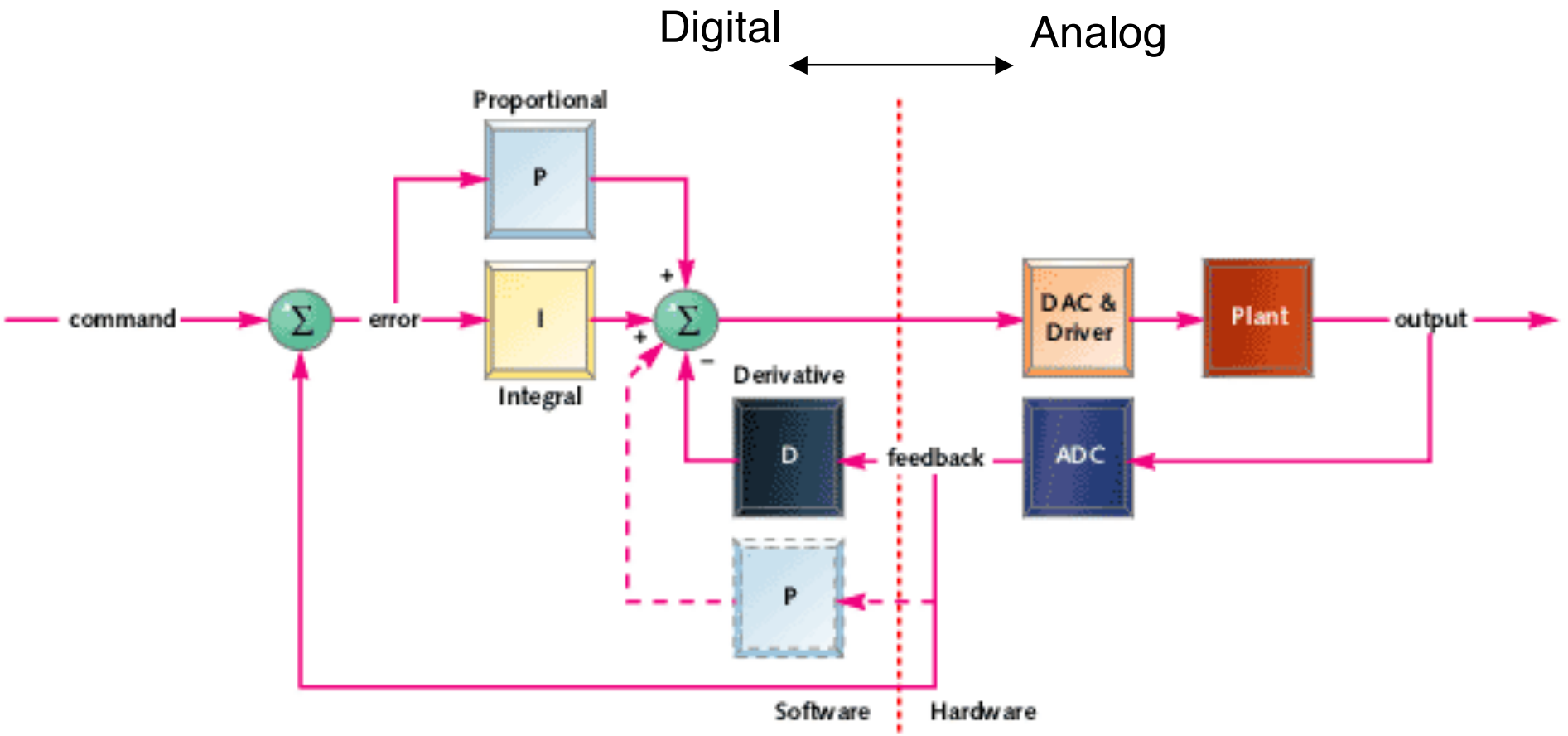
PI:



PID:



PID Controller



Sample PID Controller Code

```
typedef struct
  {
    double dState;        // Last position input
    double iState;        // Integrator state
    double iMax, iMin;    // Maximum and minimum allowable integrator state

    double        iGain,          // integral gain
                  pGain,          // proportional gain
                  dGain;          // derivative gain
  } SPid;

  double UpdatePID(SPid *pid, double error, double position)
  {
    double pTerm, dTerm, iTerm;

    pTerm = pid->pGain * error;   // calculate the proportional term

  // calculate the integral state with appropriate limiting

    pid->iState += error;

    if (pid->iState > pid->iMax) pid->iState = pid->iMax;
    else if (pid->iState < pid->iMin) pid->iState = pid->iMin;

    iTerm = pid->iGain * iState;  // calculate the integral term

    dTerm = pid->dGain * (pid->dState - position);

    pid->dState = position;

    return pTerm + dTerm + iTerm;
  }
```

Video Samples

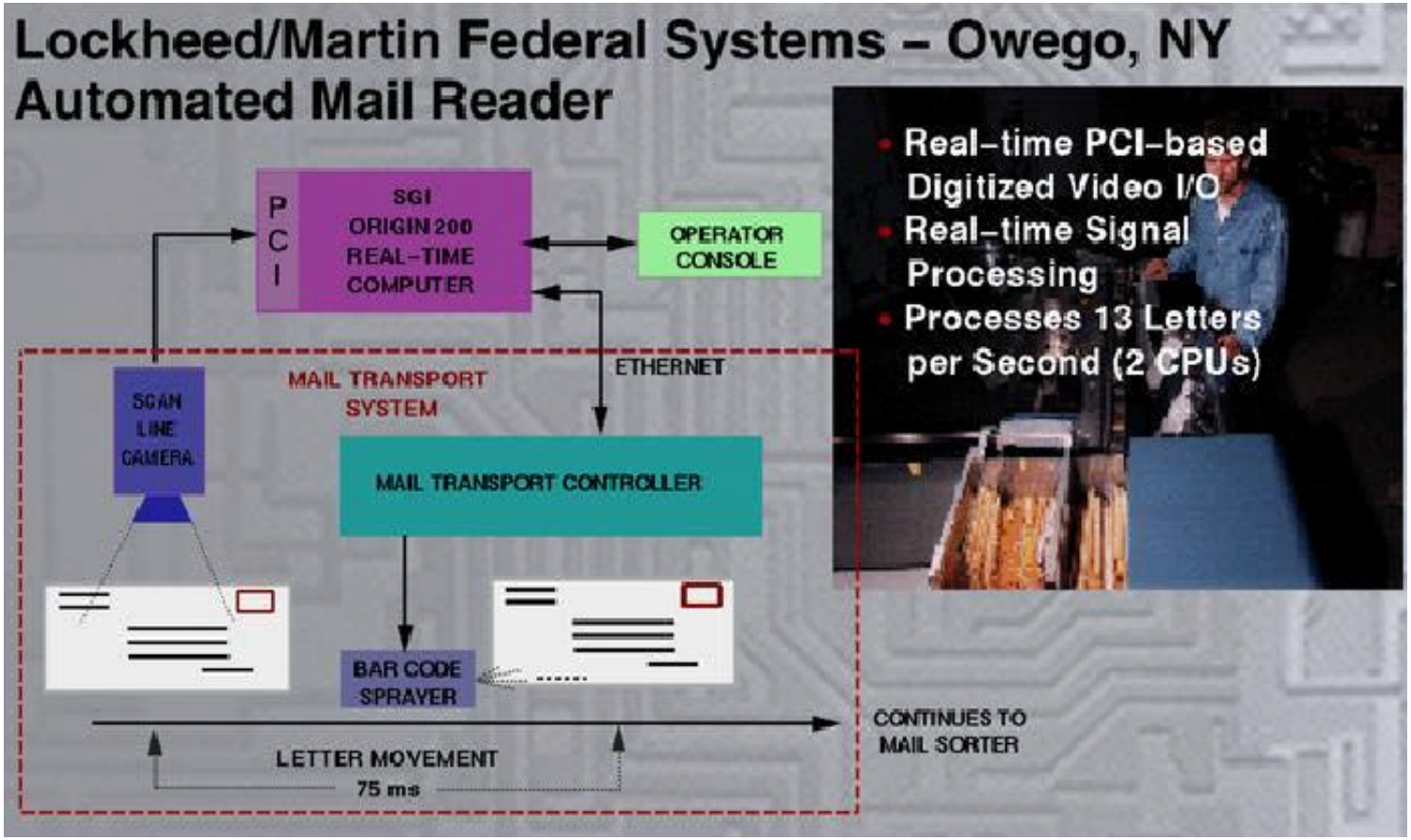
Search YouTube: “PID control”

http://www.youtube.com/watch?v=09JFUtREIro

http://www.youtube.com/watch?v=G6CBzx0x2Io&NR=1

Sample Real-Time Application

Automated Mail Reader



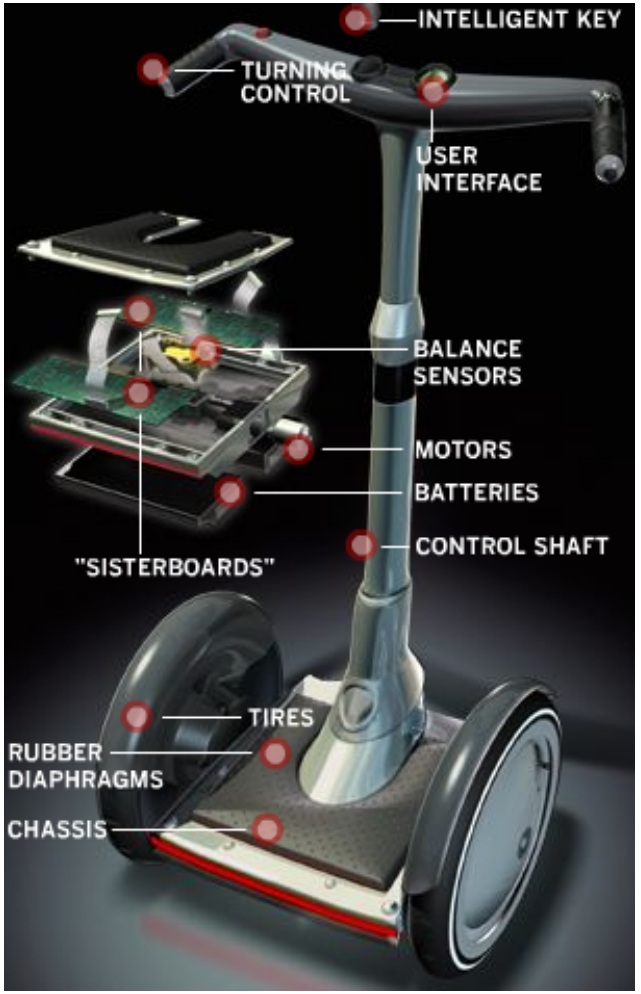
Sample Real-Time Application
Flight Simulator

Lockheed Martin:
Distributed Mission Training

Replacing flight hours with simulator hours

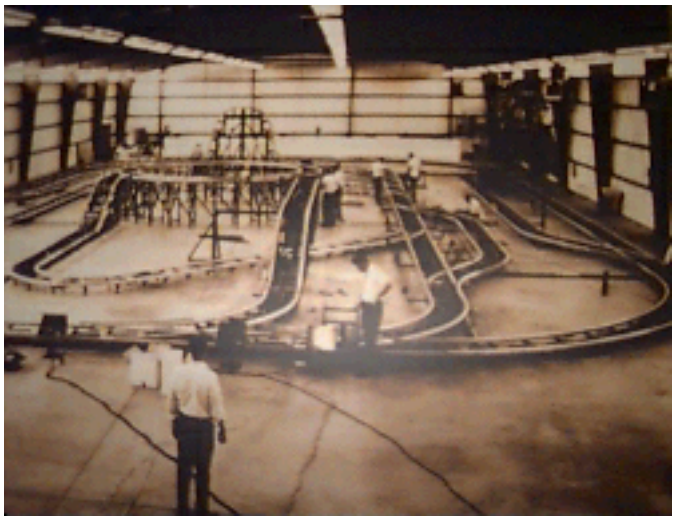
Sample Real-Time Application

Segway



Sample Real-Time Application

Airport Baggage Handler



BAE Automated Systems, Inc.

Sample Real-Time Application

Denver Airport Baggage Handler

- Contract of $193 million in June 1992 to begin work on the baggage-handling system.
- Involved 100 computers, 56 laser scanners, 400 radio systems.
- Baggage system failures:
  - Continued to unload bags despite jam on conveyor belt.
  - Loaded bags onto full carts, causing bags to fall onto tracks.
  - Bags wedged under carts due to timing problems.
  - Lost track of carts themselves, due to above types of incidents.
- Airport said to have lost **$1 million per day** upon opening.

Other Real-Time Computation Issues

- Database and networking access
- Image/video/speech processing
- Rendering
- Performance monitoring
- Fault monitoring
- Fault recovery
- Security checks, certification
- Integrity checking
- Logging
- Planning
- Garbage collection

Distinctions

- “Real-time” does not necessarily mean “real fast”.
- **Hard** real-time: **Deadlines** must be met in order for the system to be **correct**.
- **Soft** real-time: It is **desirable** for deadlines to be met, but if not, the system can still function, possibly with degraded performance.
- **Isochronous** real-time: Tasks should finish neither too late nor too early.

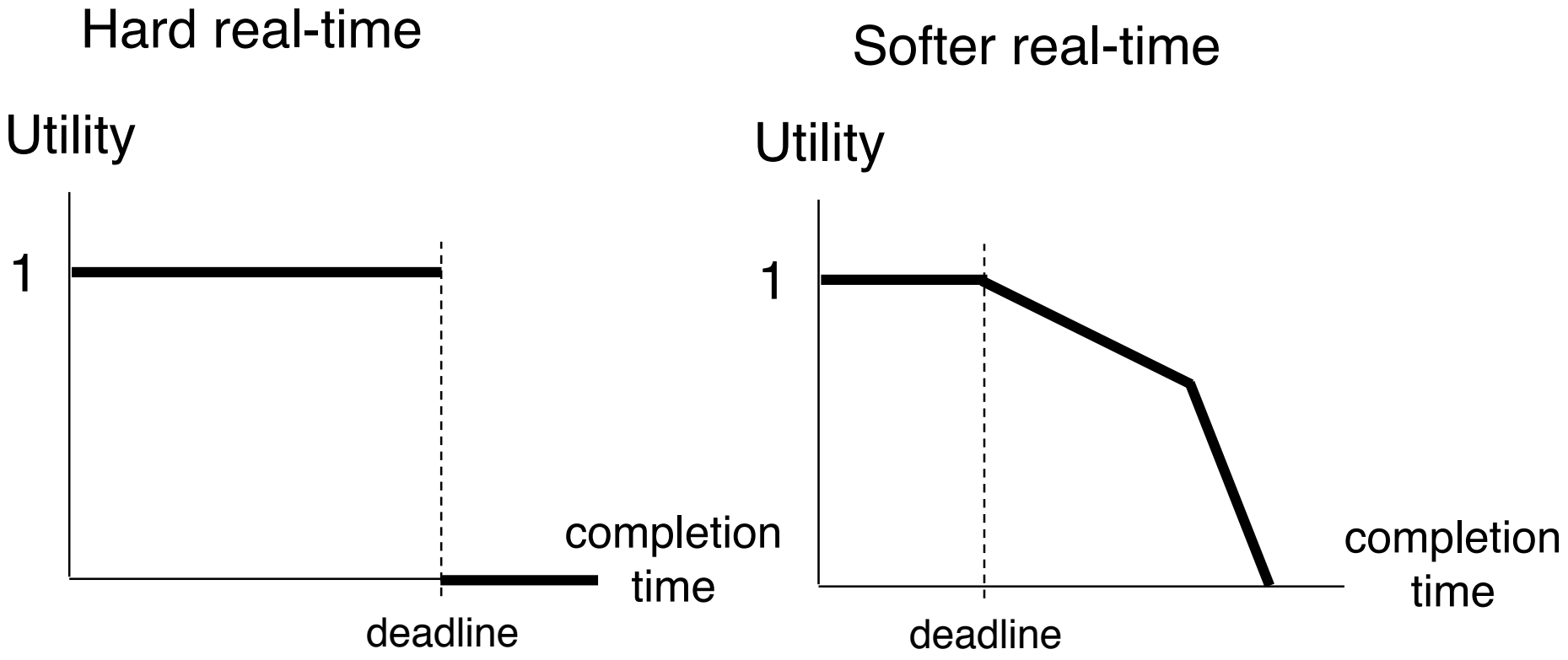
Hard Real-Time Examples

- Vehicular fuel or rendezvous problems
  - Lunar lander
  - Train scheduling
- Production deadlines
  - Newspaper
  - Live TV show
  - Graduation

Isochronous Real-Time Examples

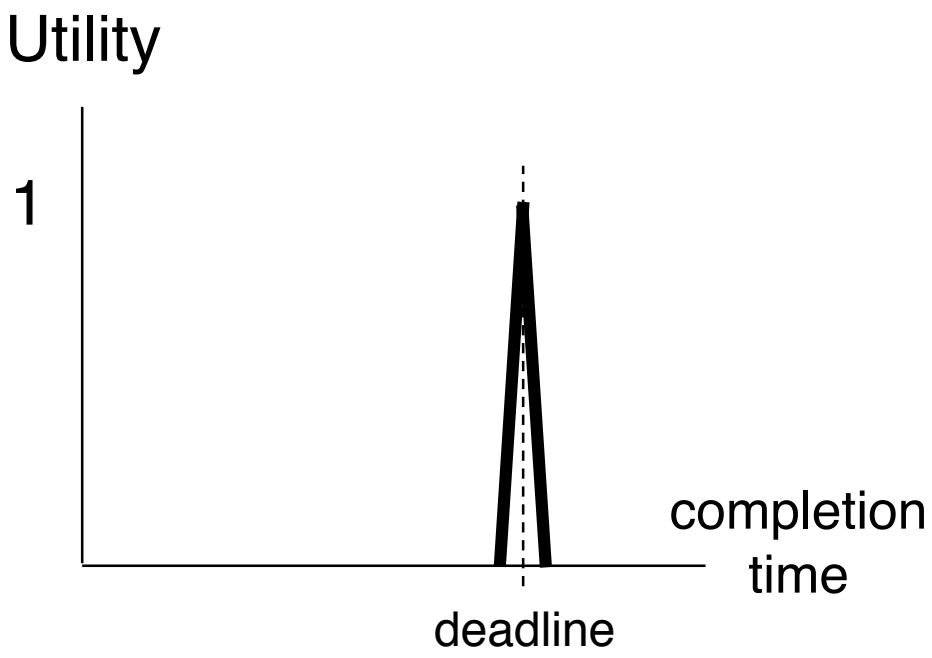
- Baggage handlers
- Assembly lines
- Writing to a disk
- Various communication protocols (e.g. FireWire, Industrial Ethernet, …)

Hardness Expressed as a **Utility** Function



Isochronous Real-time Using Utility

Isochronous real-time



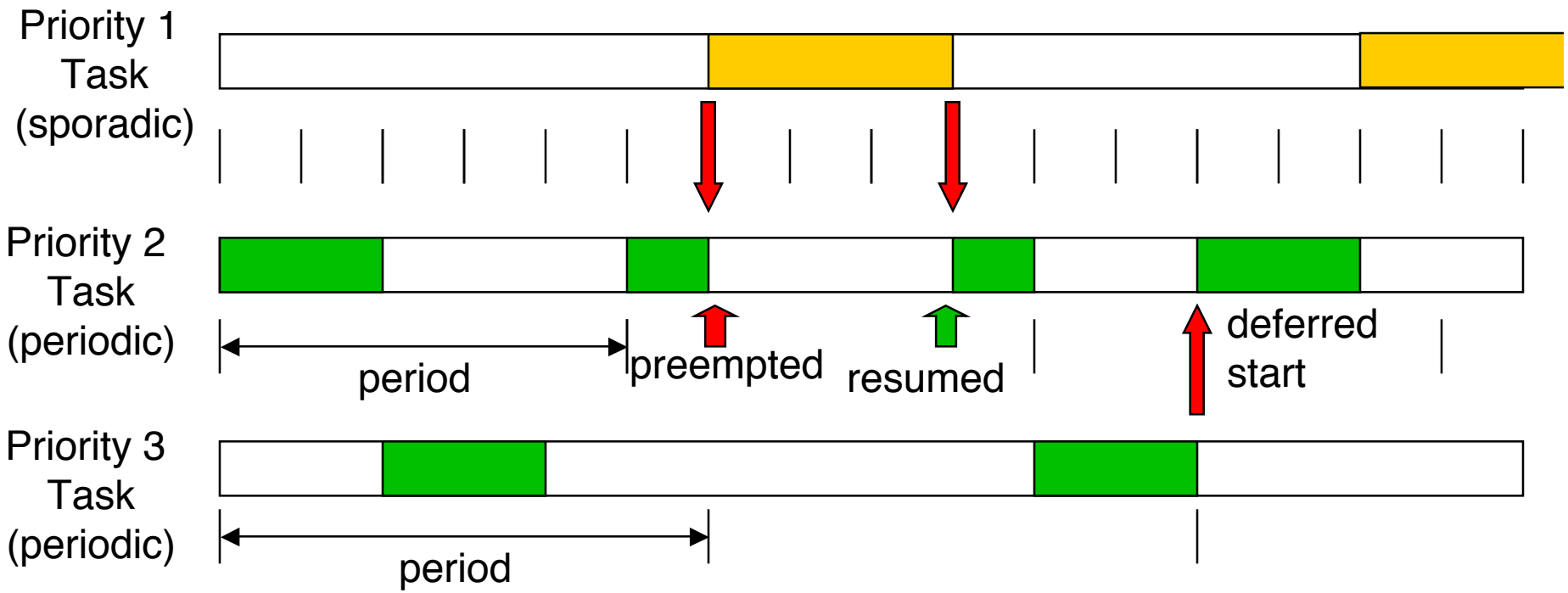
Real-Time may include Communication Considerations

- Obviously communication, as well as computation, must be taken into account if the system consists of multiple components.

Common Aspects of Real-Time Systems

- Deadlines
- Scheduled task start times
- Timeouts
- Periodic & Aperiodic (Sporadic, Episodic) tasks
- Priorities among tasks
- Preemption/resumption of lower priority tasks (interrupts)

Example Real-Time Task Considerations



Operating Systems Supporting Preeemptive Multitasking

Current versions of

- Windows
- Mac OSX
- Linux
- iOS
- Android

Real-Time Operating Systems (RTOSs)

- RTLinux (alternatively, RTAI)
- VxWorks (Wind River Systems, Inc.)
- QNX (POSIX.1 certified).
- Solaris, when generated with real-time features
- Windows CE, (alt. Windows XP Embedded)
- RTMX (POSIX RT extensions to free BSD)

Real-Time Programming Languages

- Adaptation of existing language
  - Industrial Real-Time Fortran (1981)
  - Ada (see also “Ravenscar profile”, 1995)
  - Real-Time Java: RTSJ (late 1990’s)
- Languages designed especially for the purpose
  - PEARL (Process and Experiment Automation Realtime Language, 1990)
  - Esterel (tick-based)
  - Lustre (dataflow)

Modeling Languages for Real-Time

- Uppaal (Timed Automata)
- UML
- SPIN (?)

Example of a (simplified) Real-Time Modeling Problem Statement

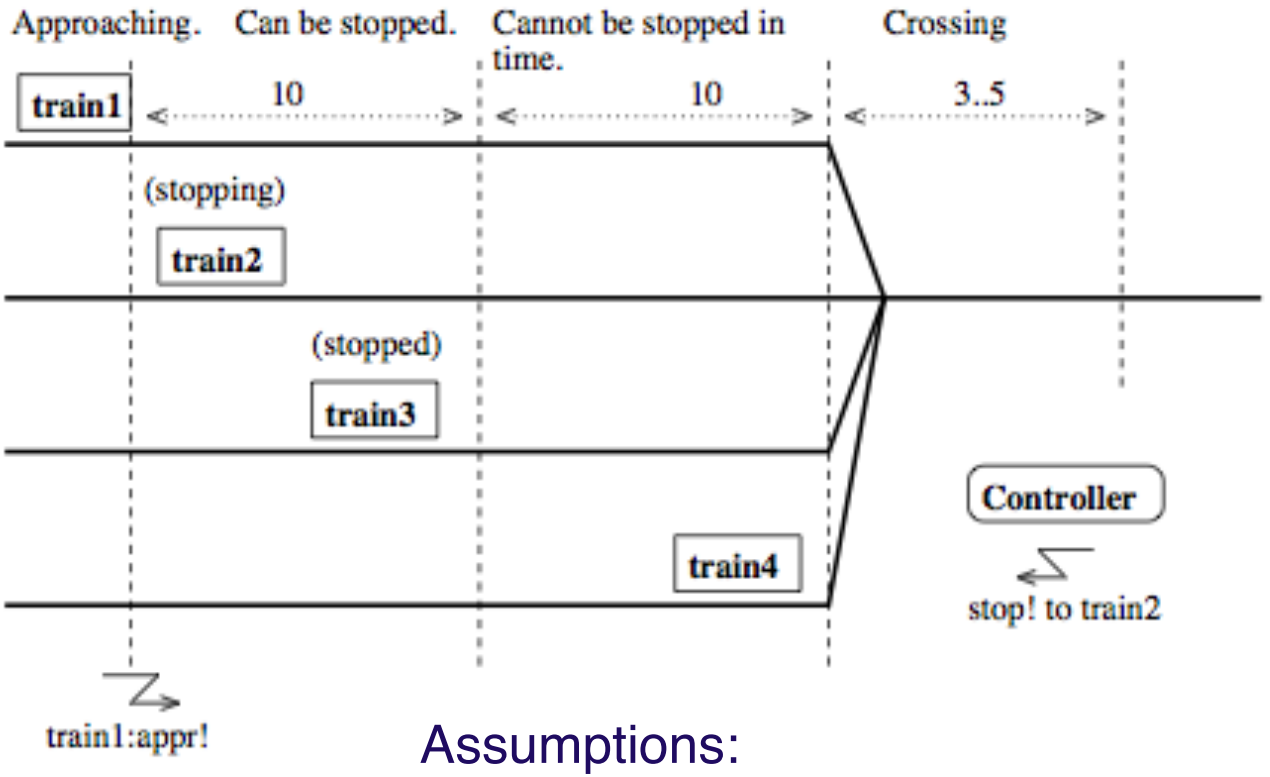
- Several trains begin on separate tracks, but need to **share** a common segment of track at some point.
- A **sensor** is installed on each track, indicating that a train is **approaching** the shared segment.
- Another sensor indicates when a train has **cleared** the shared segment.
- A **controller** gets **signals** from the sensors.
- The trains get stop and go **signals** from the controller.

continued next slide

Real-Time Modeling Problem Statement continued

- A train **cannot** be stopped instantaneously. There is some non-zero time (say 10 units) that it requires to stop when going at nominal speed.
- A train takes between 3 and 5 units to clear the shared segment once it has entered.
- Assuming no break-downs, a train should be able to cross within a finite-time from arriving at the approach sensor.
- A train shouldn't be kept waiting unnecessarily or indefinitely, e.g. when the segment is clear.
- Design the controller for N trains (N = 4, say). (Some form of queuing will be needed to ensure fairness.)

Possible 4-Train Scenario



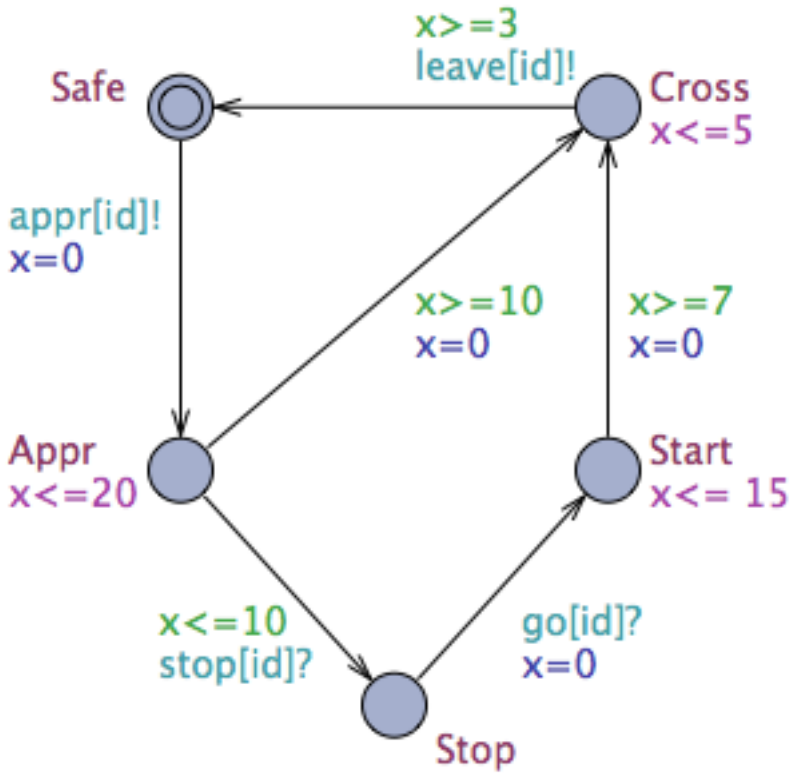
Assumptions:

10 time units are required to stop the train.

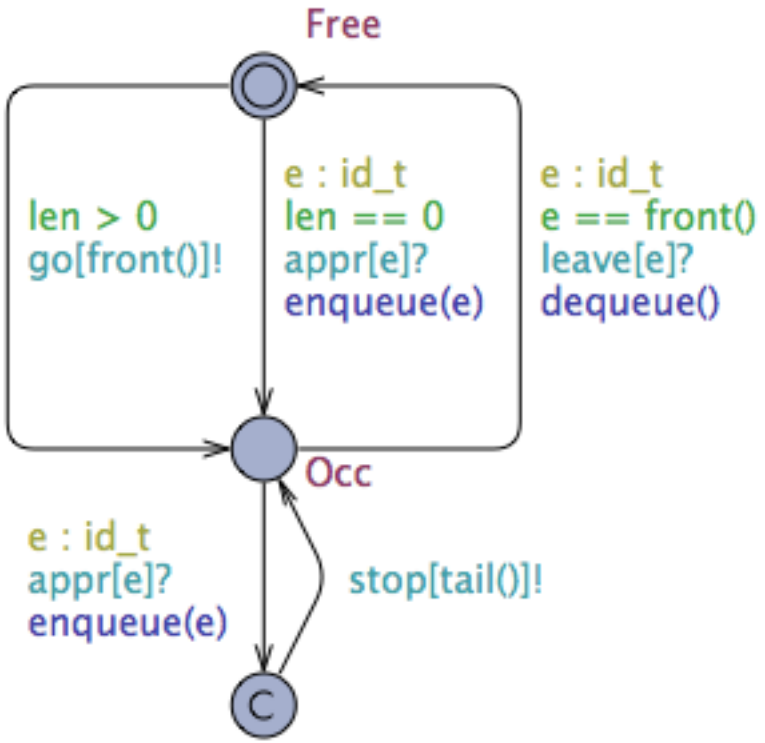
Between 3 and 5 units are required to cross the shared segment.

An Uppaal Model of a Controller

One Train (x is a clock):



Controller:



appr[] is an array of channels

Scheduling Algorithms

- Choices of performance metric to meet requirement:
  - **Average** response time over all tasks
  - **Weighted** sum of times to complete
  - **Maximum** lateness
  - **Number** of late tasks
- each of these to be minimized.

Scheduling

A Few Examples

Caution: These are mostly for 1-processor systems.

Scheduling to Meet **Deadlines**

- Assume a set of tasks $\{T_i\}$
- Associate with each task T_i:
  - computation time C_i
  - deadline D_i
- Suppose we want to minimize **maximum lateness**

Scheduling to Meet Deadlines

- **Criterion:** Minimize maximum lateness
- 1-processor case
- **Jackson's Rule**:

  EDF (Earliest Deadline First):

  "Execute tasks in order of increasing deadlines."

Proof that Jackson's Rule works

(typical of reasoning used in proofs)

- Let S' be a schedule that executes the tasks in some order **other** than by Jackson's Rule.
- Let S be a schedule that instead executes, in order of increasing deadline, some pair of tasks that were out-of-order in S'.
- We want to show that the maximum lateness of S is $\leq$ that of S'.

Proof that Jackson's Rule works

- In S' there are two tasks T_a and T_b such that $D_a \leq D_b$ but T_b precedes T_a.

- Let L_i be the *lateness* of task T_i, defined as
  $L_i = F_i - D_i$
  = Finish time - Deadline

- In schedule S, the combined lateness of T_a and T_b is:
  $\max(L_a, L_b)$

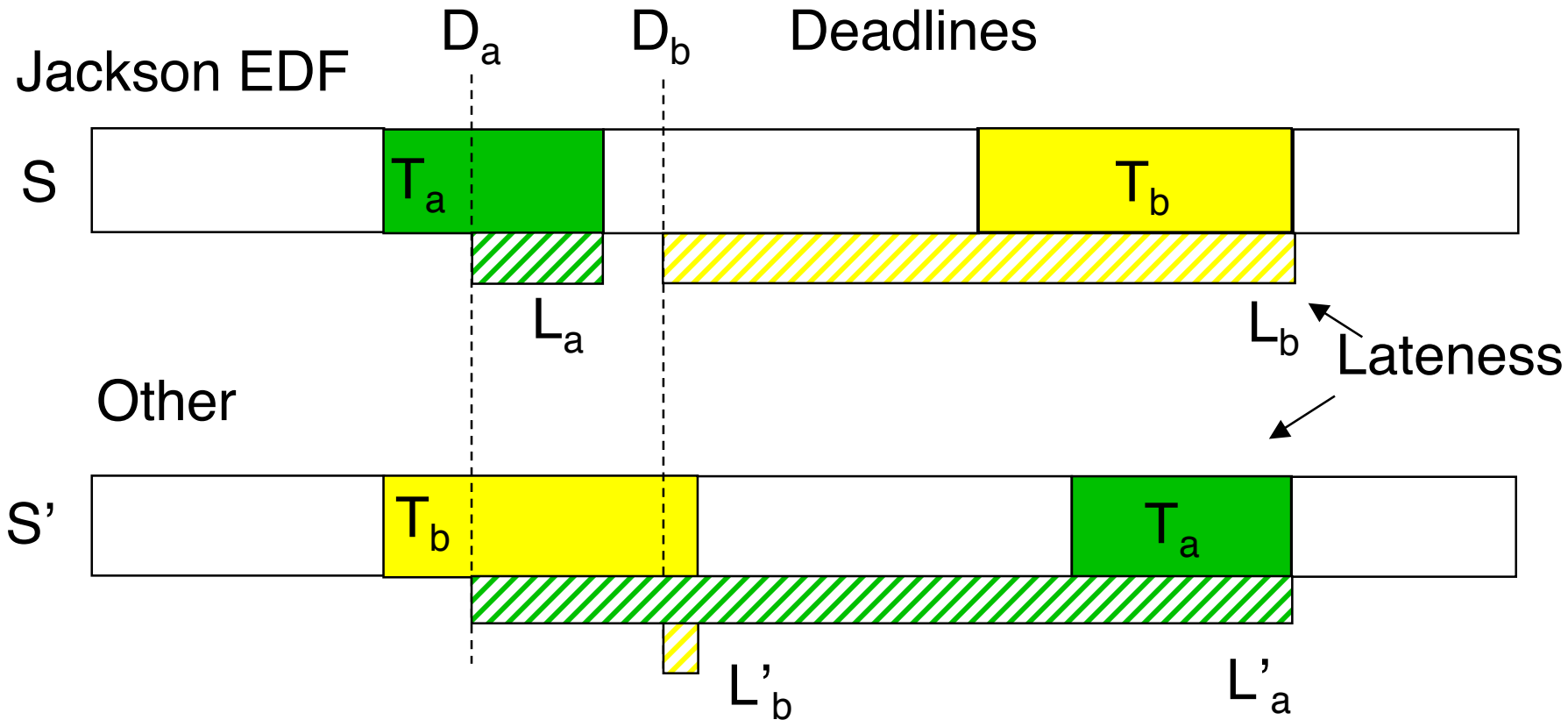
Proof that Jackson's Rule works

- We want to show that:
  $\max(L_a, L_b) \leq \max(L'_a, L'_b)$ (*)
  where the ' designates S' vs. S.

- Consider two cases:
  - $L_a < L_b$
  - $L_a \geq L_b$

- We will show that the inequality (*) holds in either case.

Jackson's Rule with $L_a < L_b$



Proof of Jackson's Rule

- Case $L_a < L_b$:

$\max(L_a, L_b) = L_b$ since $L_a < L_b$

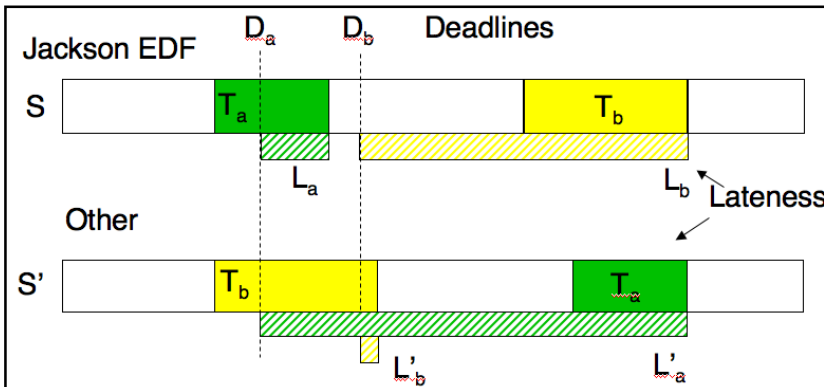
$= F_b - D_b$ def'n of L

$= F'_a - D_b$ a and b are flipped in S' vs S

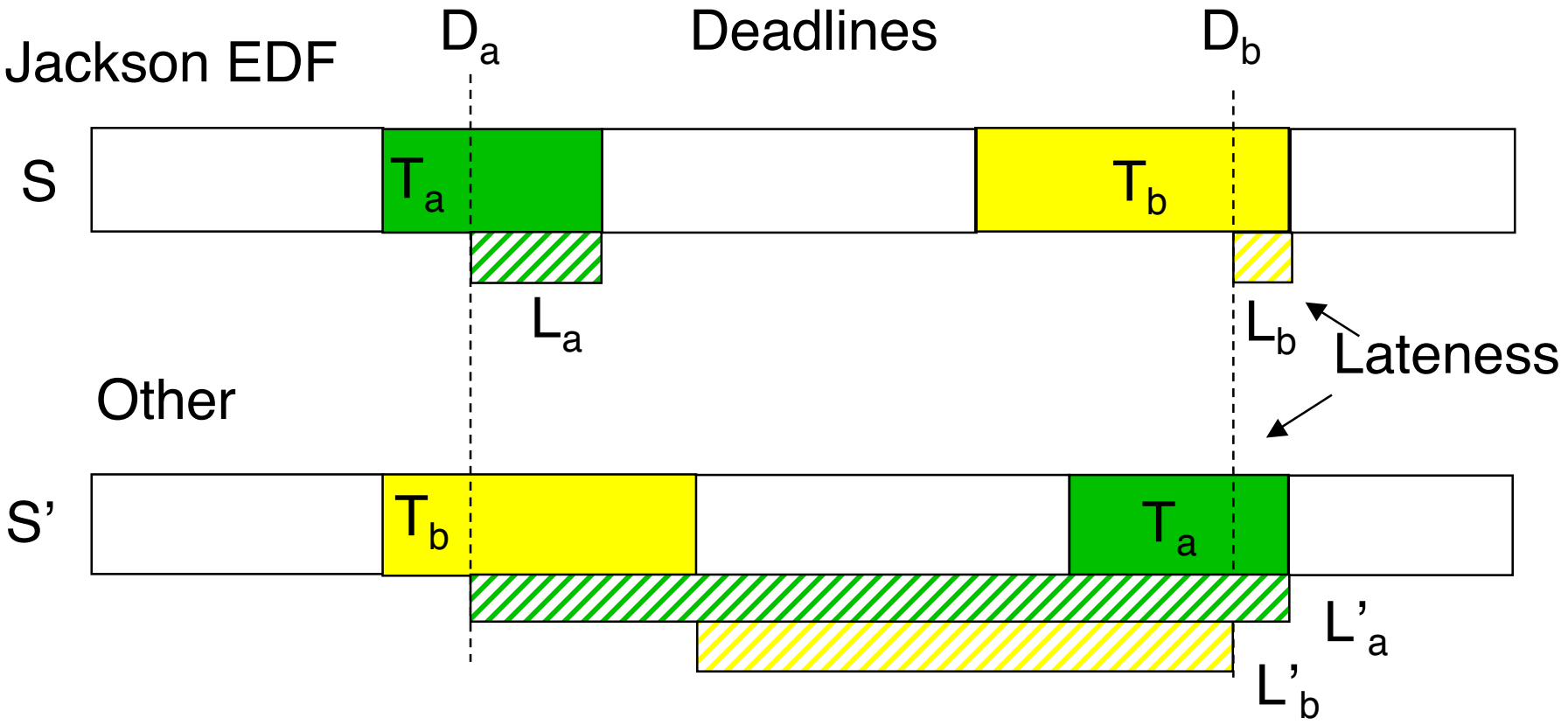
$\leq F'_a - D_a$ assumption $D_a \leq D_b$

$= L'_a$

$\leq \max(L'_a, L'_b)$



Jackson's Rule with $L_a \geq L_b$



Proof of Jackson's Rule

- Case $L_a \geq L_b$:

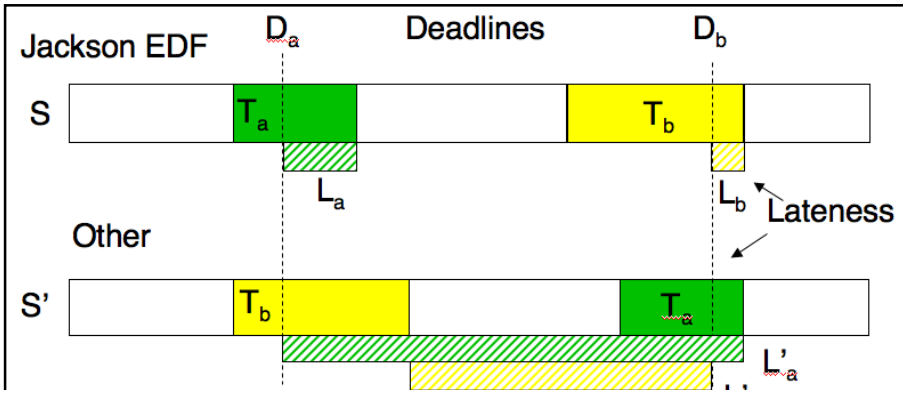
$\max(L_a, L_b) = L_a$ since $L_a \geq L_b$

$= F_a - D_a$ def'n of L

$< F'_a - D_a$ T_b precedes T_a in S'

$= L'_a$ def'n of L

$\leq \max(L'_a, L'_b)$ def'n of max



Corollary to Jackson's Rule

- Assume that tasks are *numbered in order of increasing deadlines* D_i.
- Let C_i be the corresponding computation times.
- Then all tasks can be executed so as to meet their deadlines provided that

$$(\forall\ i)\ \Sigma(C_i,\ k = 1 \text{ to } i) \leq D_i$$

Limitation of Jackson's Rule

- The set of all tasks is not always presented in advance.
- New tasks might arrive at arbitrary times.
- To minimize maximum lateness in this broader setting, it may be necessary to *preempt* a task already being executed.
- This issue is addressed by **Horn's rule**.

Horn's Rule (1974)

- Arrange execution, using **preemption** if necessary, so that:
  - **At every instant,** the task with the **current earliest deadline** is being executed.

- Horn's rule can be proved to **minimize maximum lateness** in a manner similar to the proof of Jackson's rule.

Horn's Rule (1974)

- Horn's rule is based on preemptability of tasks and no inserted idle time.

- If these are not allowed, then Horn's rule does **not** minimize maximum lateness.

Terminology

- “Arrival time” of a task: The earliest time at which scheduling a task is desired.
- also called the “Release time”.

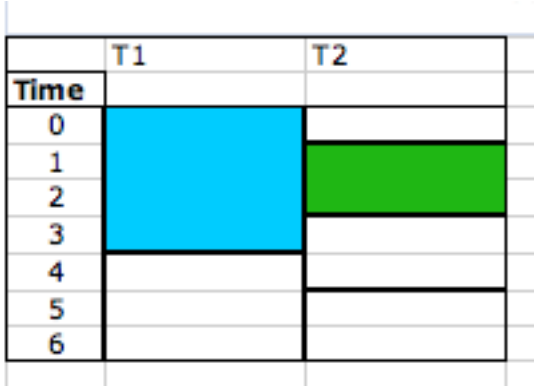
Horn's Rule (1974)

- Example where **preemption or inserted idle time are essential**:

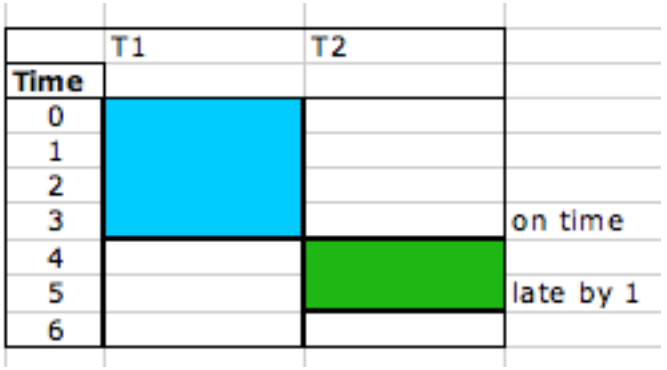
Task	Time	Deadline	Arrival
T_1	4	7	0
T_2	2	5	1

Horn's Rule (1974)

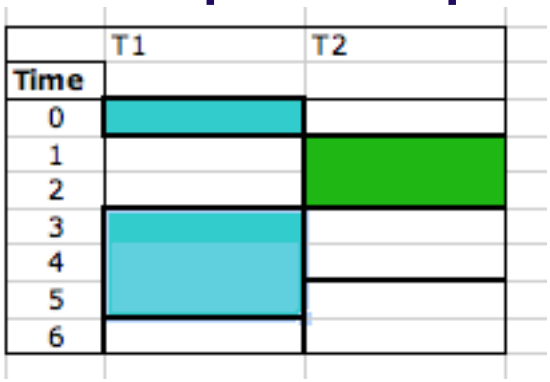
arrivals



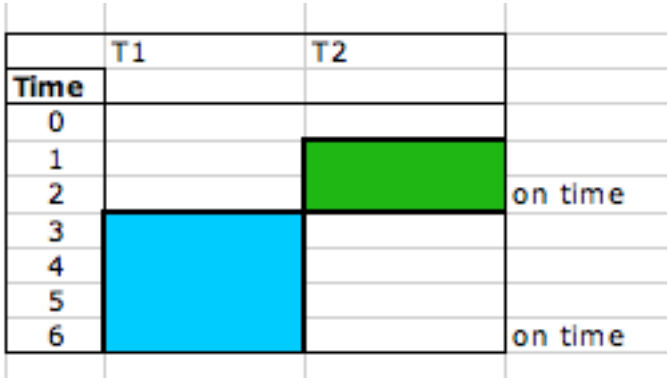
Horn's rule



Ok with preemption



Ok with inserted idle



Hard Problem

- Finding an **optimal non-preemptive** schedule when arrival times can be arbitrary is NP-hard.

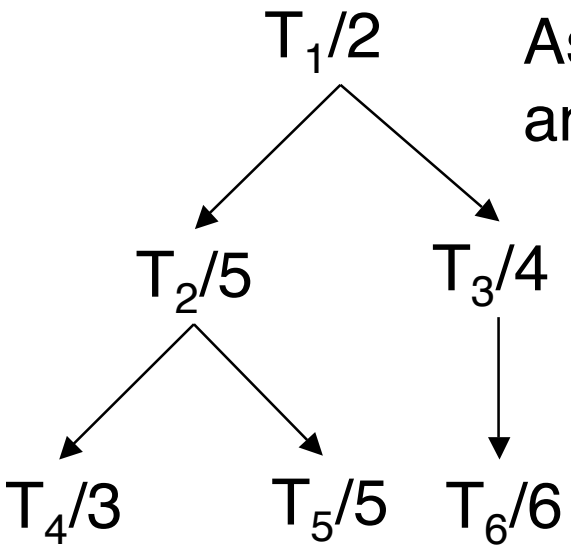
- An enumerative, branching, algorithm can be used, which will generally be exponential.

Lawler's Algorithm (1973)

- Schedules a set of tasks with identical arrivals on **one processor** subject to **precedence constraints**.

- Minimizes maximum lateness for 1 processor, among all **non-preemptive** algorithms.

Example: Lawler's Algorithm



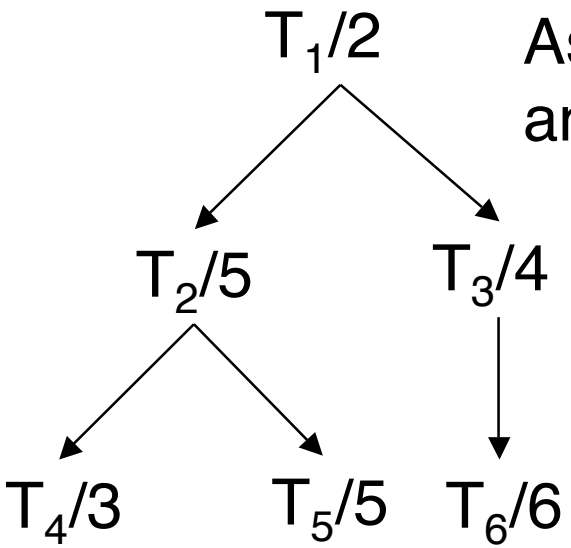
Assume **all computation times are 1** and deadlines are as shown.

Lawler's Algorithm (1973)

- Build a **stack**, selecting tasks with *latest* deadline first (LDF), *subject to* precedence constraints.

- Execute the tasks in **order of popping** from the stack.

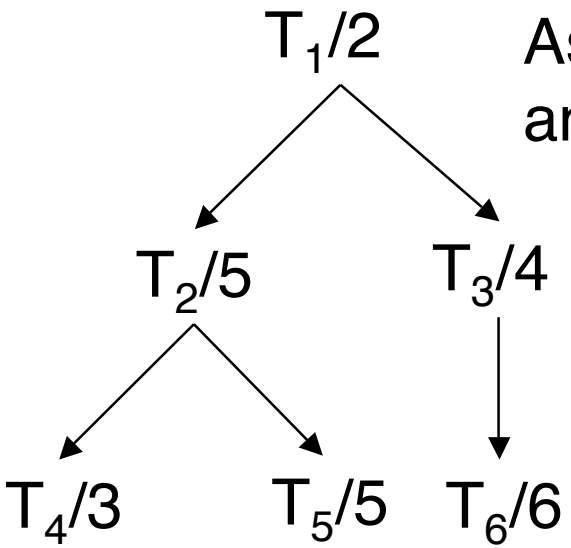
Example: Lawler's Algorithm



Assume **all computation times are 1** and **deadlines as shown** after /s.

Stacking order (LDF, obeying precedence):
$T_6/6$, $T_5/5$, $T_3/4$, $T_4/3$, $T_2/5$, $T_1/2$

Example: Lawler's Algorithm



Assume all computation times are 1 and deadlines are as shown.

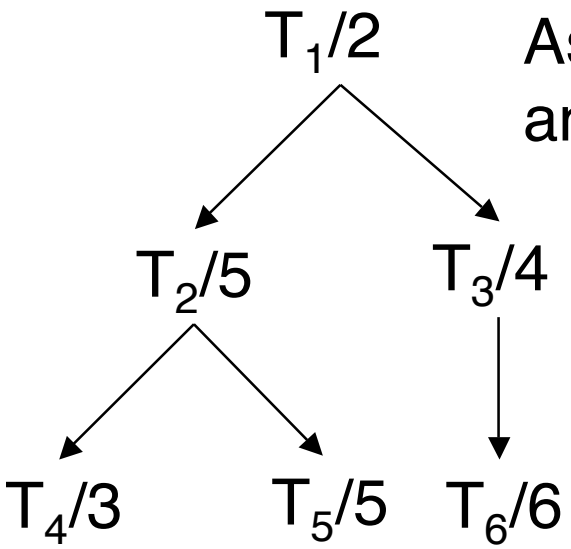
Stacking order (LDF, obeying precedence):
$T_6/6$, $T_5/5$, $T_3/4$, $T_4/3$, $T_2/5$, $T_1/2$

Execution order:

	$T_1/2$,	$T_2/5$,	$T_4/3$,	$T_3/4$,	$T_5/5$,	$T_6/6$
Completion:	1	2	3	4	5	6

No task is late!

Example: Lawler's Algorithm



Assume all computation times are 1 and deadlines are as shown.

Stacking order (LDF, obeying precedence):
$T_6/6$, $T_5/5$, $T_3/4$, $T_4/3$, $T_2/5$, $T_1/2$

Execution order:

	$T_1/2$	$T_2/5$	$T_4/3$	$T_3/4$	$T_5/5$	$T_6/6$
Completion:	1	2	3	4	5	6

Using EDF, the order would be: $T_1/2$, $T_3/4$, $T_2/5$, $T_4/3$, $T_5/5$, $T_6/6$

Completion:	1	2	3	4	5	6

One task would be late, so max. lateness = 1.

Proof of Lawler's Algorithm

Assume that the schedule obtained by Lawler's algorithm is not optimal. Let S^* denote an optimal schedule with maximal j, where j is the first iteration in Lawler's algorithm such that it chooses a job i different from the job i^* in this optimal schedule S^*. Due to the definition of j, job i must be processed before i^* in schedule S^*. We obtain a new schedule S' from S^* by inserting job i directly after job i^* and leaving the other jobs unchanged. Schedule S' is still feasible since job i is processed at the same position by Lawler's algorithm. The only job which completes later now is job i, so the other jobs cannot increase the maximum cost. But by definition of Lawler's algorithm, the completion time cost of job i is in S' not larger than the cost of job i^* in S^*. Thus, the new schedule S' is still optimal, a contradiction to the maximality of j. □

Other Algorithms

- Other scheduling methods may be found in these references:
  - G.C. Buttazo
    Hard Real-Time Computing Systems
    Kluwer Academic Publishers, 1977.
  - P. Brucker
    Scheduling Algorithms, 5th Edition
    Springer Verlag, 2007

Periodic Tasks

- Many realtime systems are based on **periodic** tasks, wherein each task T_i has:
  - Computation time C_i
  - Period P_i
  - Phase ϕ_i
- The meaning of "period" is that, for each i, T_i must execute **once every** P_i units.

Phase

- The meaning of “phase” is that, for each i, the earliest time at which T_i is available within the current period is at relative time ϕ_i from the start of the period.

- It is common to use the assumption that the phases are all 0.

“Release Time”

- Instead of using phase, “release time” is used to mean “the time at which a task is next available for scheduling”; the “release” is releasing the task ***into*** the system, not out of it.

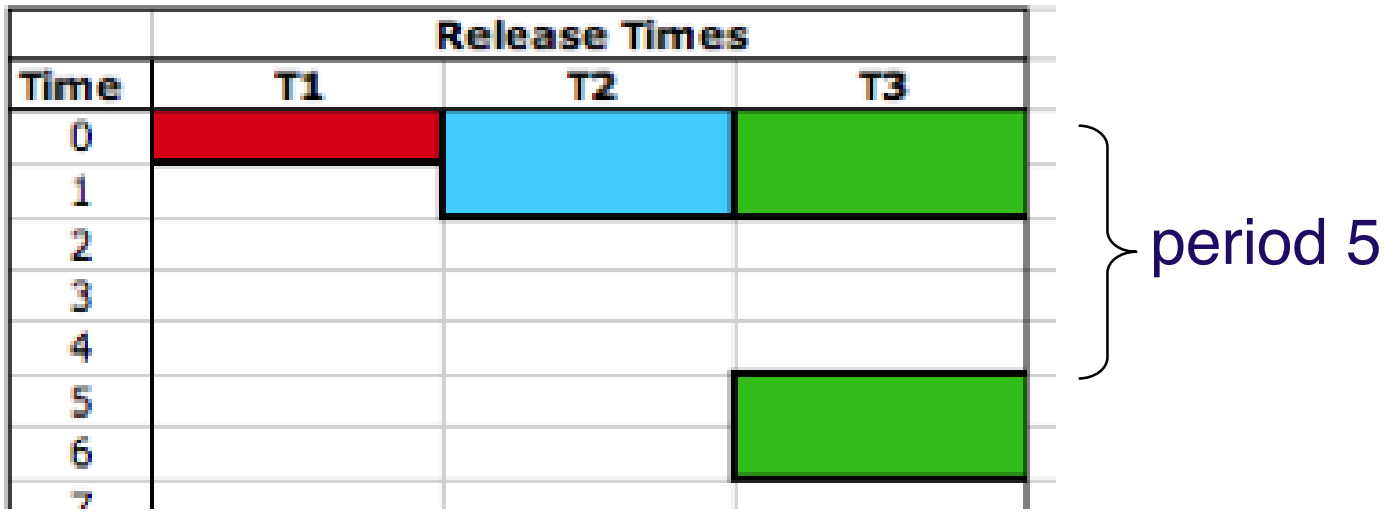
- **Unless we indicate otherwise**, the release time for a task will coincide with the **beginning of the period** for the task, i.e. a periodic task with duration 2 and period 20 will be “released” at times 0, 20, 40, ...

Effective Deadline

- The end of the period (plus the phase) becomes the effective deadline for a periodic task.

Example

The effective deadline for the first instance of T3 having period 5 is 5.



Preemptability

- Preemptability is a common assumption in real-time systems:

  A lower priority task may need to be preempted to give the processor to a higher priority one.

- Typically it is assumed that preempted tasks can be **resumed** at the point preempted, with no loss of information.

Preemption Costs

- In general, there will be a cost (delay) associated with preempting a task.
- For now, we assume that this cost is negligible.

Utilization
(don't confuse with utility function)

- The **utilization**, in a system of periodic tasks, by a task is defined as its

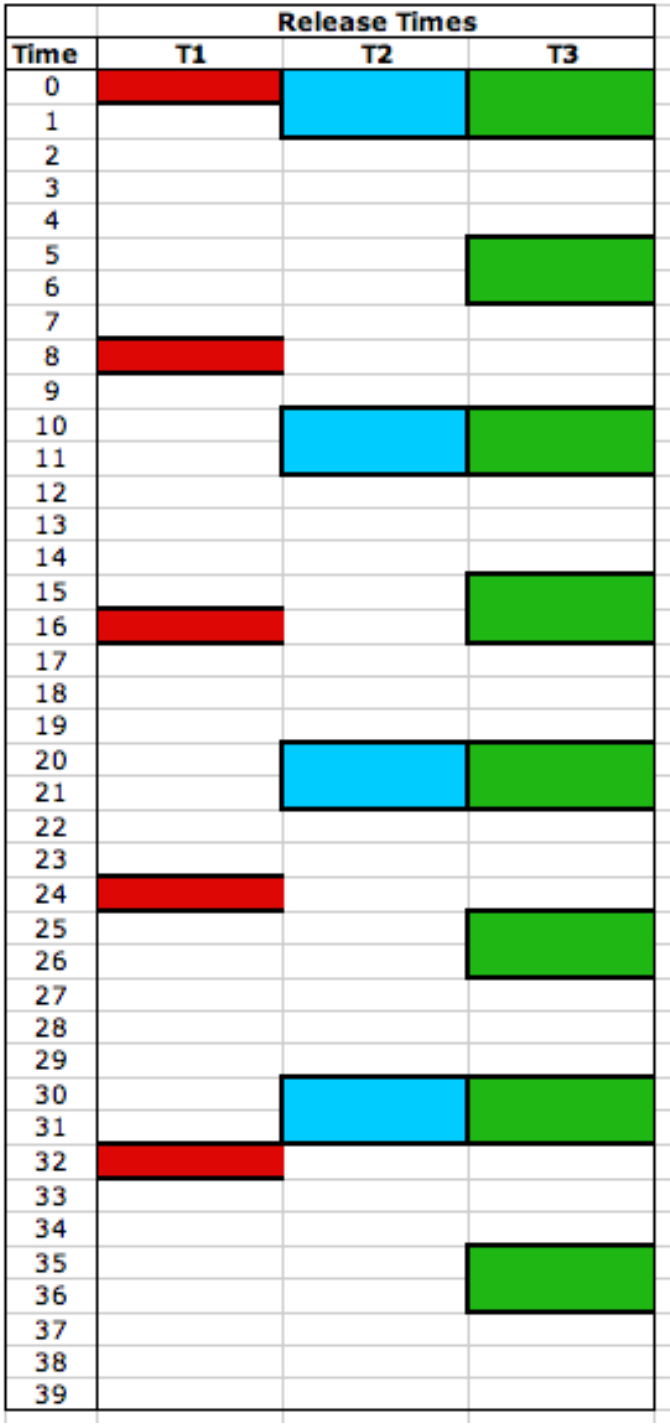
$$\frac{\text{compute time}}{\text{period}}$$

- A **necessary** condition for a set of period tasks to be schedulable is that the **sum** of their utilizations be ≤ 1.

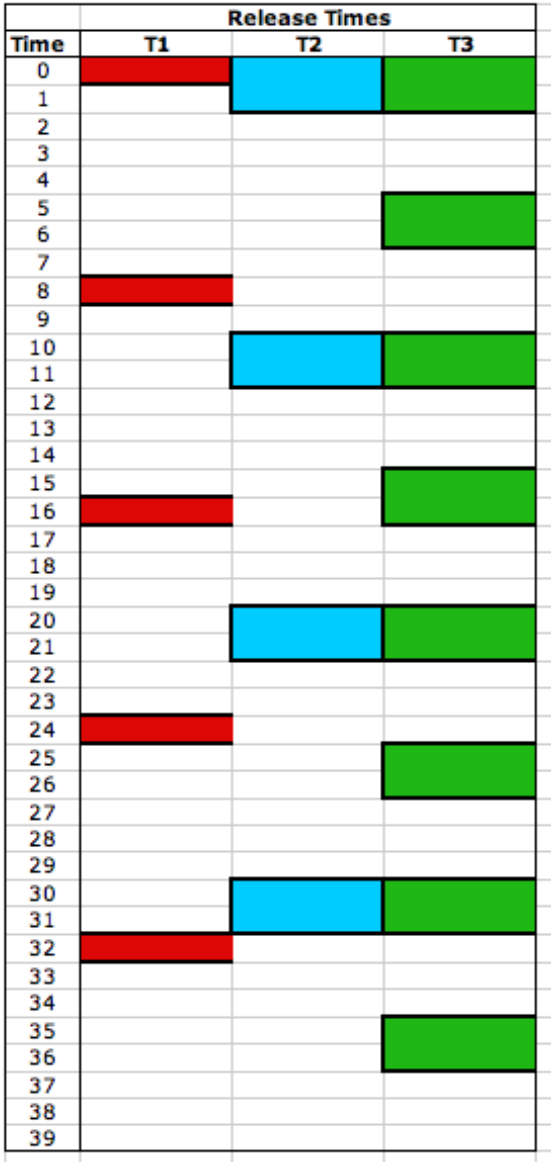
Example
Can this set be scheduled periodically?

Task	Time	Period
T1	1	8
T2	2	10
T3	2	5

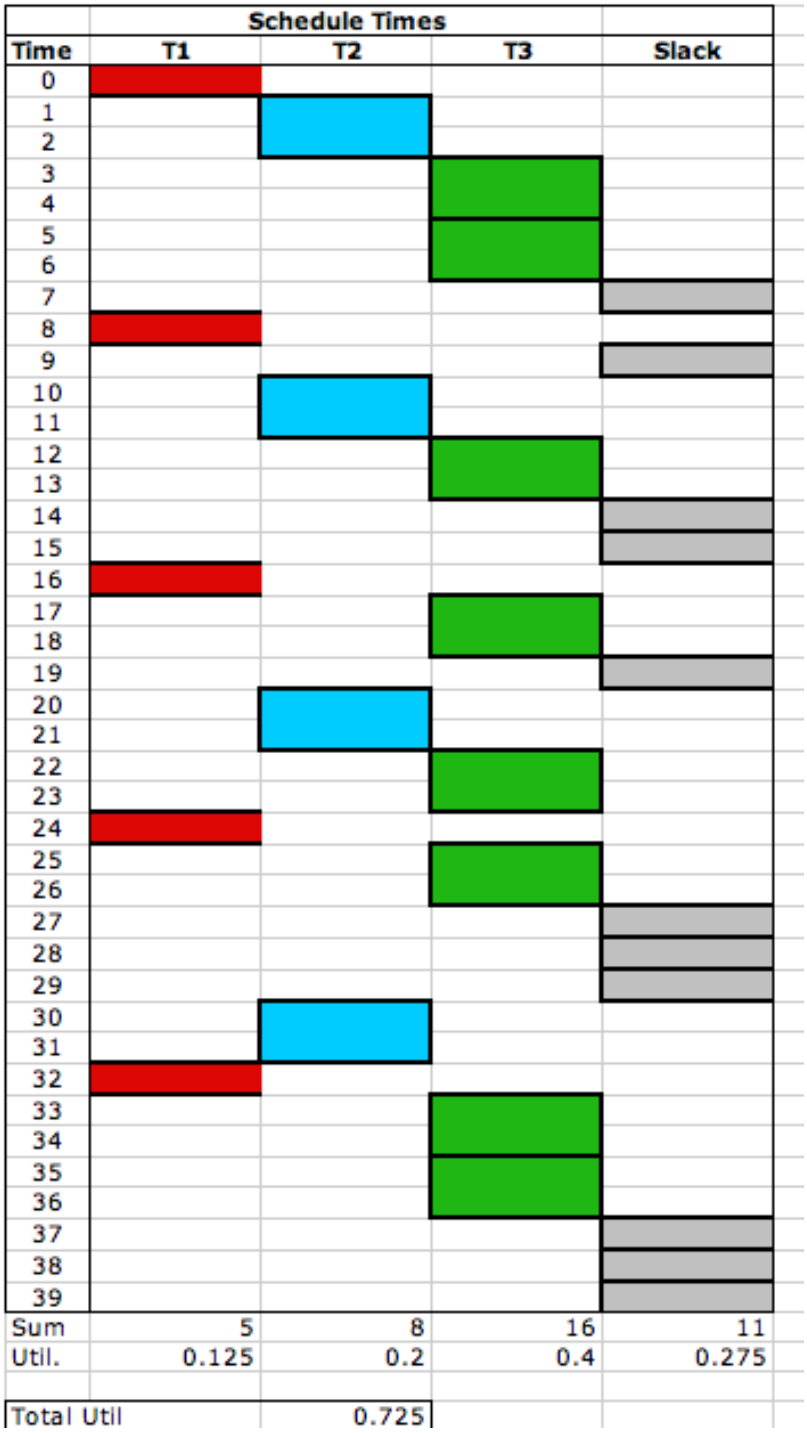
First plot release times.



Then adjust start times to avoid overlap.



	Release Times		
Time	T1	T2	T3
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			
27			
28			
29			
30			
31			
32			
33			
34			
35			
36			
37			
38			
39			



	Schedule Times			
Time	T1	T2	T3	Slack
0				
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
20				
21				
22				
23				
24				
25				
26				
27				
28				
29				
30				
31				
32				
33				
34				
35				
36				
37				
38				
39				
Sum	5	8	16	11
Util.	0.125	0.2	0.4	0.275
Total Util		0.725		

EDF Revisited

- Recall Horn's rule: Preemptive EDF (Earliest Deadline First)
- It works for ***arbitrary*** arrivals.
- Therefore it will work for periodic tasks as well.

“Dynamic Priority”

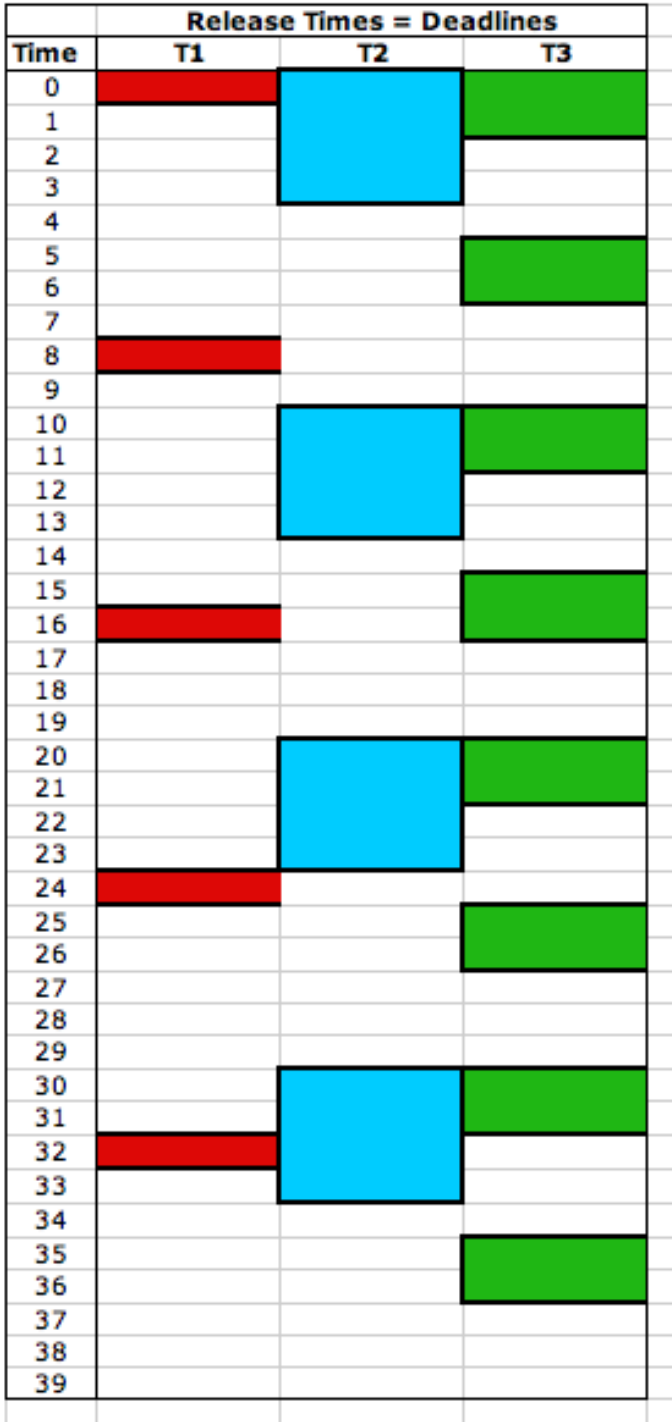
- EDF is *dynamic* because the *relative* priorities will depend on what is being executed *when* a new task arrives. This could be:
  - A task with a longer period but a nearer deadline.
  - A task with a shorter period but a more distant deadline.

Example
Can this set be scheduled periodically?

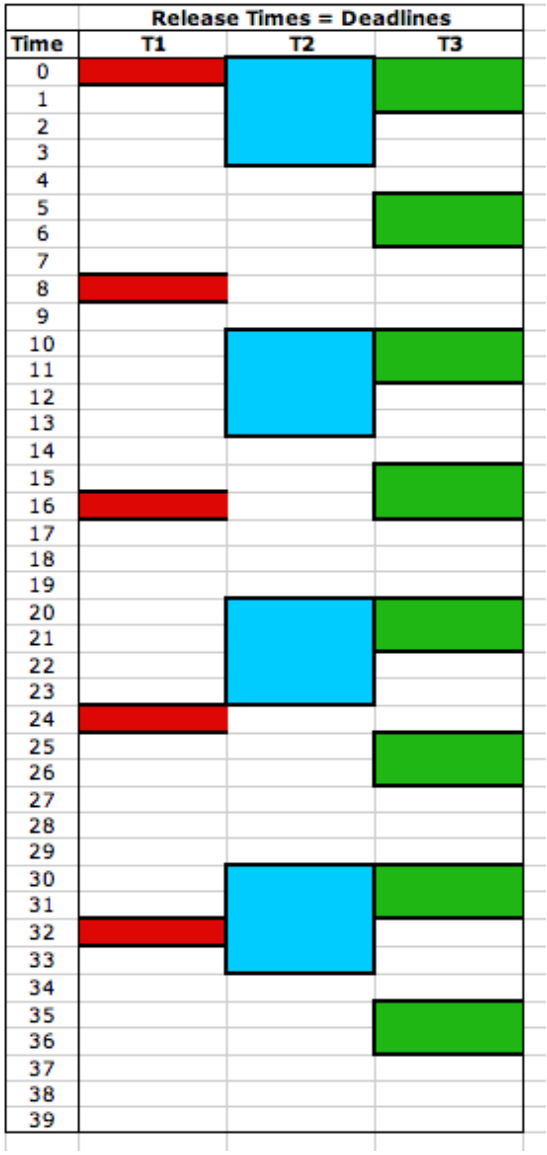
Task	Time	Period
T1	1	8
T2	2	10
T3	4	5

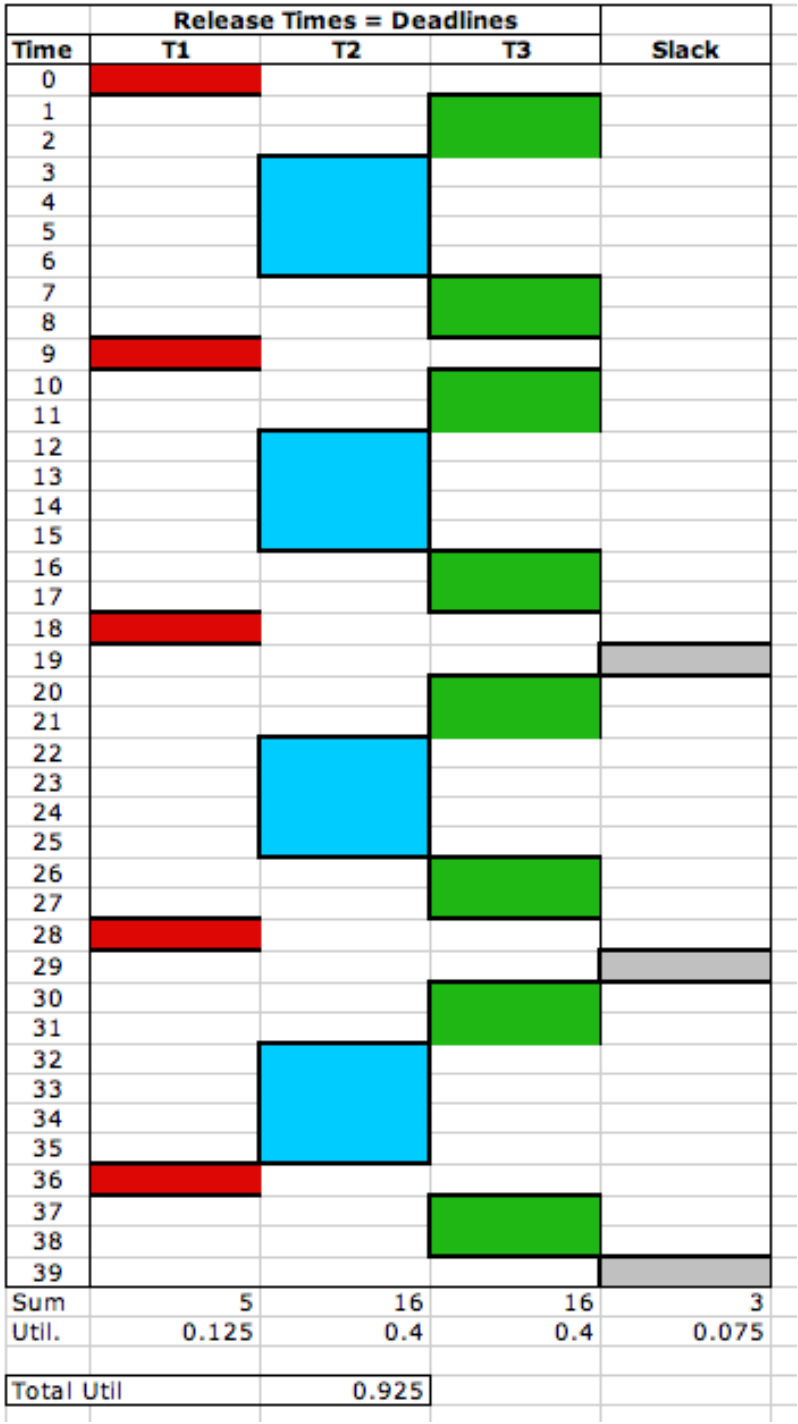
Again plot release times.

Again, note also that release time of the next instance is the deadline for the current instance.



Then adjust start times to avoid overlap.





Utilization Bounds for Dynamic Priority Assignment

- For dynamic priority assignment, any system with a **total utilization** ≤ 1 can be scheduled, assuming no overhead in preemption.

- EDF is adequate for constructing such a schedule.

Static Priority Systems

- Being able to assign static priorities among tasks simplifies the scheduling problem.
- However, a static priority system might not be able to achieve the same total utilization as a dynamic priority one.

Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment

Landmark Paper

C. L. LIU

Project MAC, Massachusetts Institute of Technology

AND

JAMES W. LAYLAND

Jet Propulsion Laboratory, California Institute of Technology

ABSTRACT. The problem of multiprogram scheduling on a single processor is studied from the viewpoint of the characteristics peculiar to the program functions that need guaranteed service. It is shown that an optimum fixed priority scheduler possesses an upper bound to processor utilization which may be as low as 70 percent for large task sets. It is also shown that full processor utilization can be achieved by dynamically assigning priorities on the basis of their current deadlines. A combination of these two scheduling techniques is also discussed.

KEY WORDS AND PHRASES: real-time multiprogramming, scheduling, multiprogram scheduling, dynamic scheduling, priority assignment, processor utilization, deadline driven scheduling

CR CATEGORIES: 3.80, 3.82, 3.83, 4.32

Journal of the Association for Computing Machinery, Vol. 20, No. 1, January 1973, pp. 46–61.

“Critical Instant”

- Liu and Layland define a **critical instant** as one at which a task release will have the largest response time.

- **Theorem 1**: A critical instant for any task occurs whenever the task is released along with releases of all higher priority tasks.

Proof of L&L Theorem 1

- Let $\tau_1, \ldots, \tau_m$ denote the tasks ordered from highest to lowest priority, with periods $T_1, \ldots, T_m$, respectively.
- Consider a request for τ_m occurring at time t_1.
- The next request for τ_m will be at $t_1 + T_m$.

Proof of L&L Theorem 1

- Suppose that in the interval $I = [t_1, t_1 + T_m]$ requests for higher priority τ_i occur, at times t_2, $t_2 + T_i$, $t_2 + 2T_i$, …, $t_2 + kT_i$ for some k, where $t_2 \geq t_1$.
- The preemption of τ_m by τ_i may cause some delay in the completion of τ_m.
- Advancing t_2 (moving it closer to t_1), can at best delay the completion of τ_m, by moving another instance of into interval I.
- Therefore the delay in completion τ_m of is largest when t_2 coincides with t_1.

Proof of L&L Theorem 1

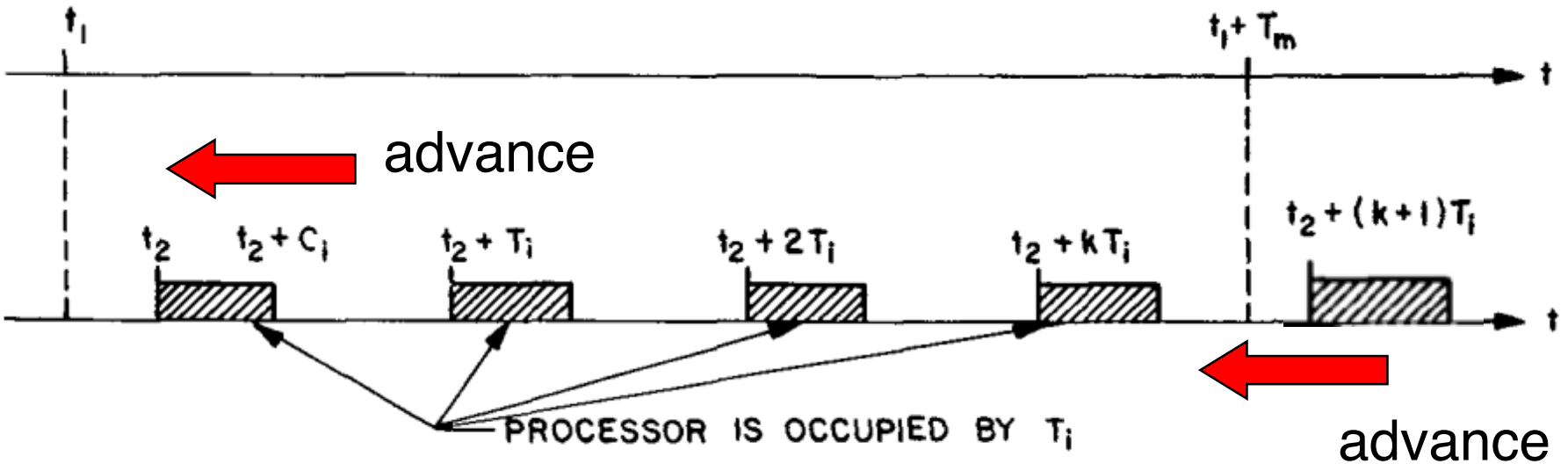


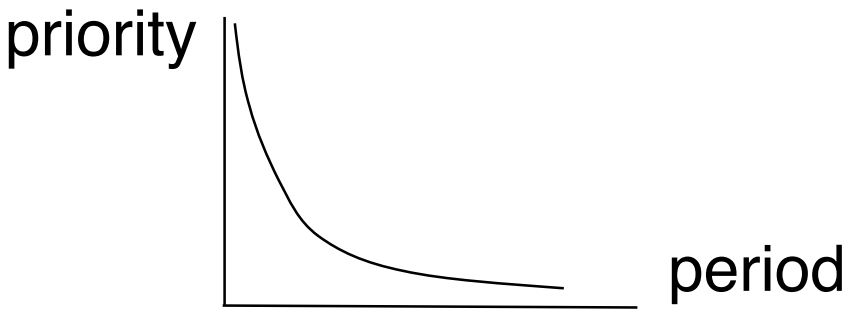
FIG. 1. Execution of τ_i between requests for τ_m

Proof of L&L Theorem 1

- By induction, the worst case, i.e. a critical instant, is when the release times of all higher priority tasks coincide with the release of this task.

Rate-Monotonic Assumption (RMA)

- In a static priority system, a task that is preemptable by another has ***lower priority***.
- RMA says that tasks with **shorter periods** should generally have **higher priority**.



- Possible rationale: lower priority tasks can be preempted, then resumed, to allow higher priority tasks to meet their deadlines, which occur with greater frequency.

Rate-Monotonic Assumption (RMA)

- In a static priority system, a task that is preemptable by another has ***lower priority***.
- RMA says that tasks with **shorter periods** should generally have **higher priority**.
- The next slides give a quantitative rationale.

“Critical Time Zone”

- A critical time zone for a task is the interval between a critical instant for the task and the end of the response to the release of the task.

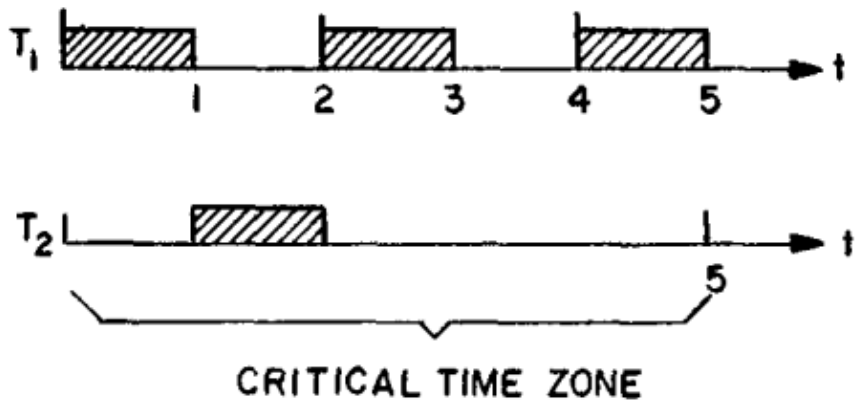
Example of why RM is good

- Consider two tasks both with computation time 1,
  - one having period 2,
  - the other period 5.

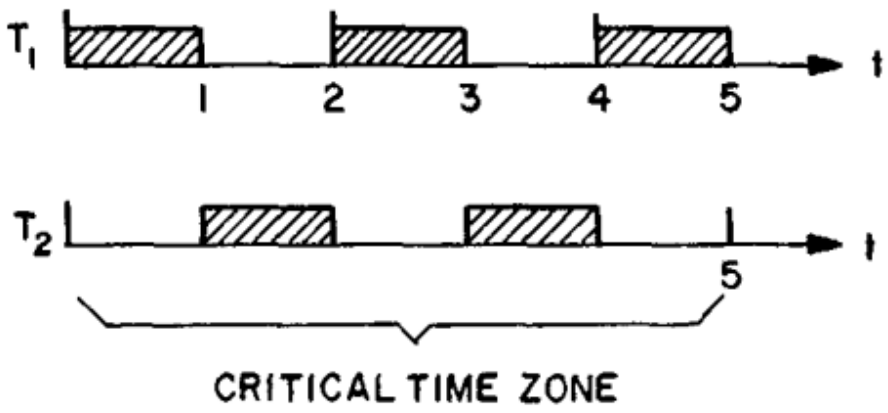
- We want to see why the period 2 task should have higher priority.

If period 2 task has higher priority, the computation time for the period 5 task can be increased to 2 with no impact.

higher priority



higher priority

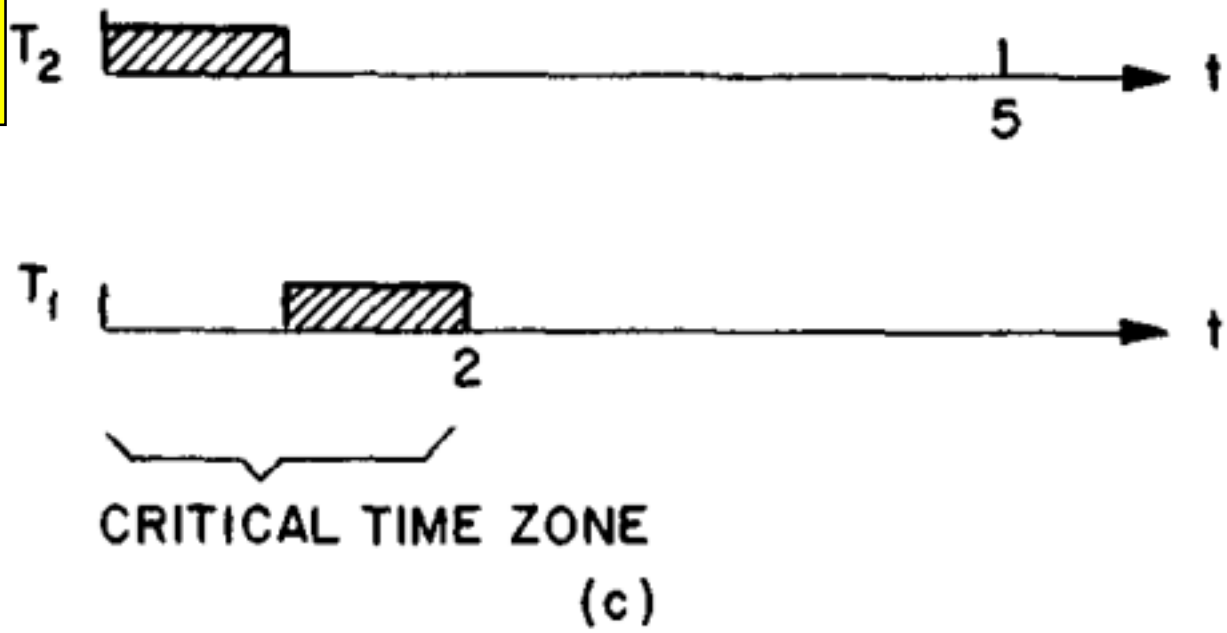


extend T_2 to longer compute

(b)

If period 5 task has higher priority, the computation time for neither task can be increased with no impact.

higher priority



cannot extend T_1 to longer compute

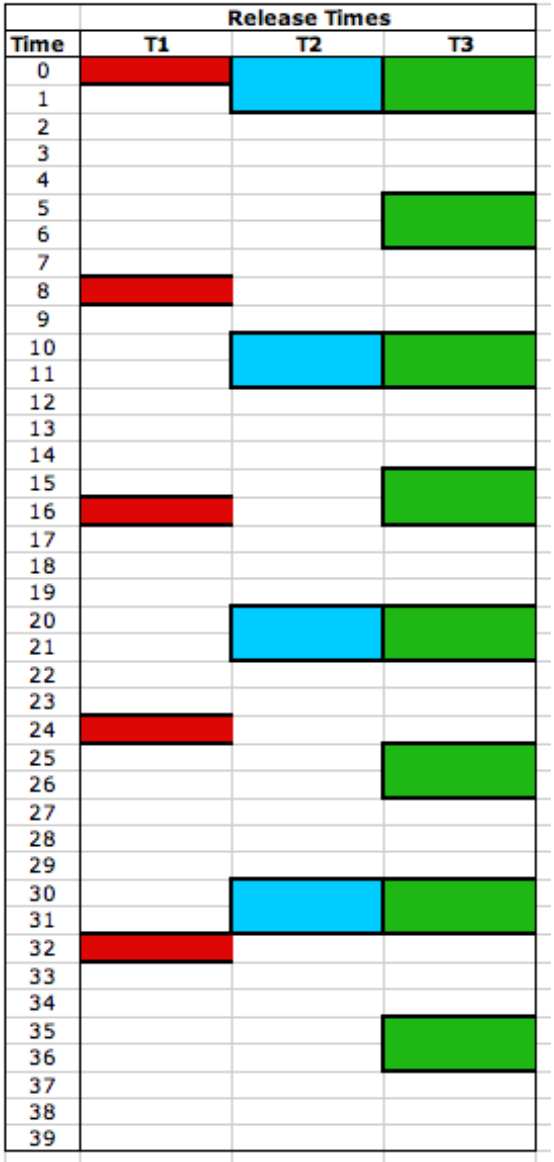
Rate-Monotonic Scheduling

- Deterministic deadlines are exactly equal to periods.
- Static priorities (the task with the highest static priority that is runnable immediately **preempts** all other tasks).
- Static priorities assigned according to the rate monotonic conventions (tasks with shorter periods/deadlines are given higher priorities)

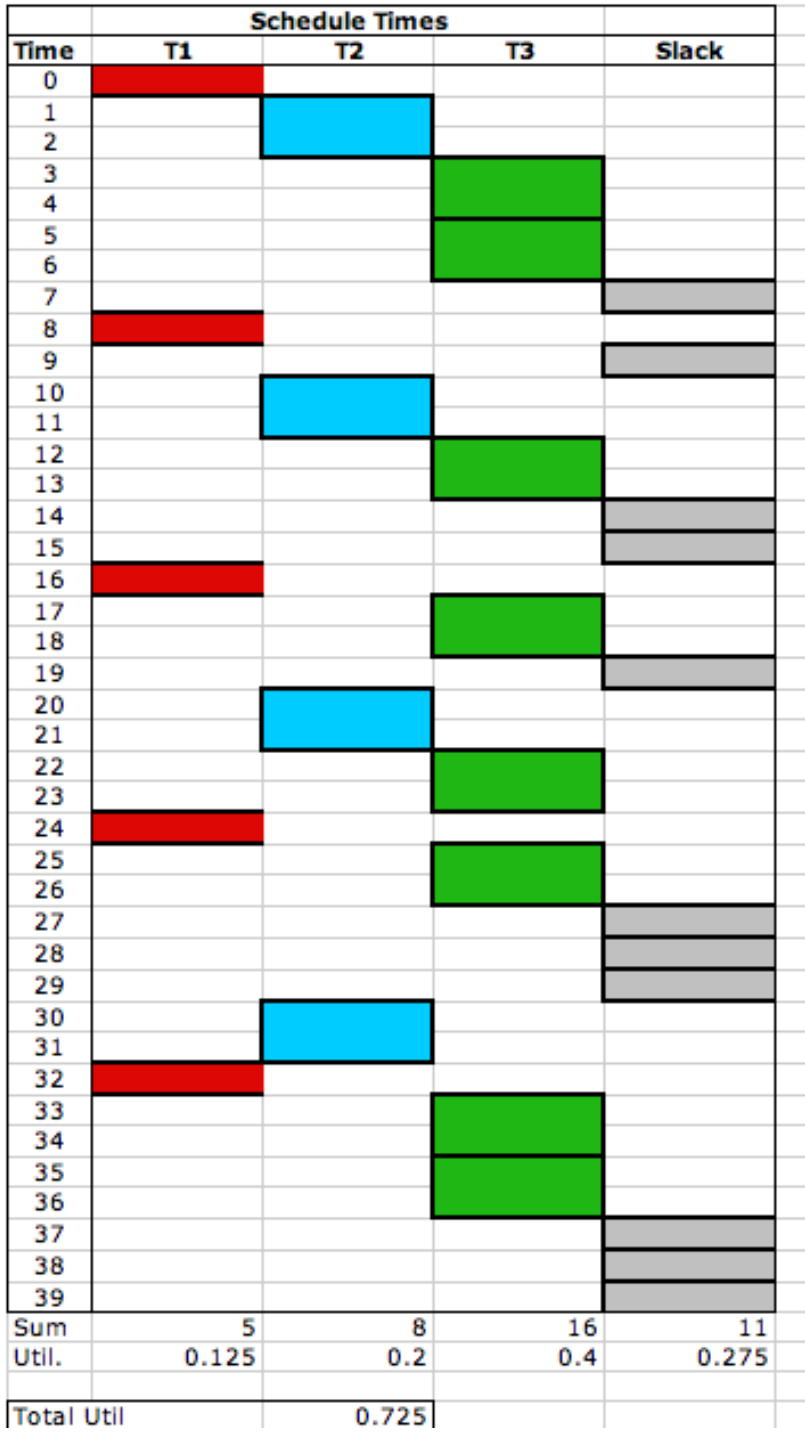
Example
Does this have a rate-monotonic schedule?

Task	Time	Period
T1	1	8
T2	2	10
T3	2	5

The previous schedule was already rate-monotonic.



Release Times			
Time	T1	T2	T3
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			
27			
28			
29			
30			
31			
32			
33			
34			
35			
36			
37			
38			
39			



Schedule Times				
Time	T1	T2	T3	Slack
0				
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
20				
21				
22				
23				
24				
25				
26				
27				
28				
29				
30				
31				
32				
33				
34				
35				
36				
37				
38				
39				
Sum	5	8	16	11
Util.	0.125	0.2	0.4	0.275
Total Util		0.725		

Theorem (Liu & Layland, 1973)

- A set of n tasks is rate-monotonically schedulable on 1 processor, provided:

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

U is the *total periodic utilization*

- The condition is **sufficient**, but not necessary.

Numeric Values for Liu & Layland Rule

tasks	bound on total utilization
1	1
2	0.828427
3	0.779763
4	0.756828
8	0.724062
16	0.707472
32	0.700709
64	0.696914
…	
∞	ln(2) ≈ 0.693147

Example
(L&L rule sufficiency)

- $P_1 = 100$, $P_2 = 150$,
  $C_1 = 20$, $C_2 = 30$.
- Consider an interval of time [0, 300], during which T_1 must complete $\lceil 300/100 \rceil = 3$ times and T_2 must complete 2 times.
- L&L rule computes $20/100 + 30/150 = 120/300 \leq 0.828$, so these tasks can be scheduled RM.

- What schedule realizes the requirements?

task	T_1	T_2	T_1	T_2	T_1
start	0	20	100	150	200
end	20	50	120	180	220

Example
(L&L rule not necessary)

- Consider adding a third task:
  $P_1 = 100$, $P_2 = 150$, $P_3 = 210$,
  $C_1 = 20$, $C_2 = 30$, $C_3 = 80$.

- The Liu and Layland rule computes total utilization: 20/100 + 30/150 + 80/210 = 0.780952 which is not realizable for 3 tasks (0.7796).

- Since the Liu and Layland rule is sufficient, but not necessary, there still might be a schedule.

- Can you construct one?

Example

- $P_1 = 100$, $P_2 = 150$, $P_3 = 210$,
  $C_1 = 20$, $C_2 = 30$, $C_3 = 80$.

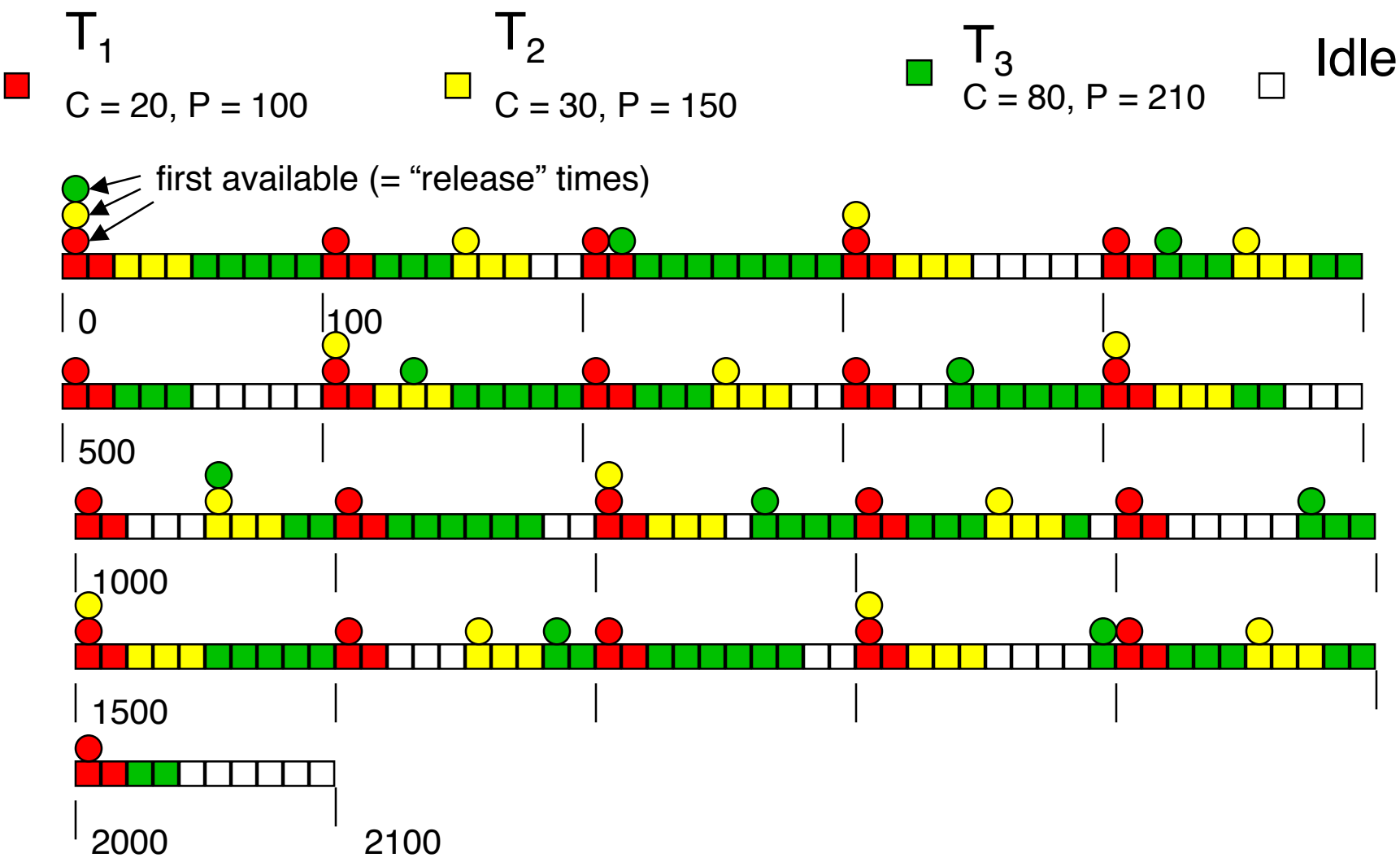
- Consider an interval of time [0, 2100], (2100 is the lcm of 100, 150, and 210) during which T_1 must complete 21 times, T_2 14 times, and T_3 10 times.

- The total time required is 21*20 + 14*30 + 10*80 = 420+720+800 = 1940 $\leq$ 2100. So it is at least *plausible* that there is a schedule.

- On the next page, we construct such a schedule.

A Rate-Monotonic Schedule

(each box = 10 time units)



Critical Zone Theorem
Lehoczky, Sha, and Ding (1987)

- Under RM scheduling, if **each** task can meet its **first** deadline, then all deadlines can be met.

RMS is Pessimistic

- The RMS 69% utilization bound covers the worst case.
- For a randomly chosen task set, Lehoczy, Sha, and Ding derive a better bound of 88% **average** utilization.

2nd Theorem of Liu & Layland

- If a set of tasks is schedulable under **any static priority** scheme, then it is schedulable using rate-monotonic scheduling.
- In other words, rate-monotonic is **optimal** among *static* priority schemes.
- (But dynamic priority schemes such as premptive EDF can do better.)

Lehoczky, Sha, and Ding (1987) **Sharpens** Liu and Layland Criterion

Periods: $T_1 \le T_2 \le \ldots \le T_n$ focus on tasks $\tau_1, \ldots, \tau_i$,

then the expression

$$W_i(t) = \sum_{j=1}^{i} C_j \cdot \lceil t/T_j \rceil,$$

gives the cumulative demands on the processor made by these tasks over [0,t] when 0 is a critical instant. For later notational convenience, we define

$$L_i(t) = W_i(t)/t,$$

$$L_i = \min_{\{0 < t \le T_i\}} L_i(t),$$

$$L = \max_{\{1 \le i \le n\}} L_i.$$

Lehoczky, Sha, and Ding (1987) Sharpens Liu and Layland Criterion

Theorem 1: Given periodic tasks $\tau_1, \ldots, \tau_n$,

1. τ_i can be scheduled for all task phasings using the rate monotonic algorithm if and only if $L_i \leq 1$.

2. The entire task set can be scheduled for all task phasings using the rate monotonic algorithm if and only if $L \leq 1$.

Proof of Lehoczky, Sha, and Ding (1987)

Proof: Given the worst case phasing was proved by Liu and Layland to be a critical instant and we require schedulability for all task phasings, we can restrict attention to the worst case. Given the assumptions of perfect preemption, no blocking, no overhead and jobs are ready at their initiation times, two conclusions follow. First, jobs of τ_i will be preempted only by higher priority jobs, and they will preempt any lower priority jobs. Thus only $\tau_1, \ldots, \tau_i$ need be considered to determine if τ_i can be scheduled. Second, starting at a critical instant, the processor will not idle before either the first job of τ_i is finished or it misses its deadline, whichever occurs first.

(continued)

Proof of Lehoczky, Sha, and Ding (1987)

Task τ_i completes its computation requirement at time $t \in [0,T_i]$, if and only if all the requests from all the jobs with priority higher than i and C_i, the computation requirement of τ_i, are completed at time t. The total of such requests is given by $W_i(t)$, and they are completed at t if and only if $W_i(t) = t$ and $W_i(s) > s$ for $0 \leq s < t$. Dividing by t, we find equivalently $L_i(t) = 1$. It follows that a necessary and sufficient condition for τ_i to meet its deadline is the existence of a $t \in [0,T_i]$ such that $L_i = 1$. Consequently, a necessary and sufficient condition for τ_i to meet its deadline under all phasings is $L_i \leq 1$. This proves the first assertion.

Example of Lehoczky, Sha, and Ding Test

- $P_1 = 100$, $P_2 = 150$, $P_3 = 210$,
  $C_1 = 20$, $C_2 = 30$, $C_3 = 80$.
- $L_1(t) = C_1 \lceil t/P_1 \rceil / t = 20 \lceil t/100 \rceil / t$
- $L_2(t) = (20 \lceil t/100 \rceil + C_2 \lceil t/P_2 \rceil) / t = (20 \lceil t/100 \rceil + 30 \lceil t/150 \rceil) / t$
- $L_3(t) = (20 \lceil t/100 \rceil + 30 \lceil t/150 \rceil + 80 \lceil t/210 \rceil) / t$

For which values of t do we need to check that $L_3(t) \leq 1$?

LSD Continuous → Discrete

The characterization of schedulability given in Theorem 1 requires a minimization over the continuous variable $t \in [0, T_i]$. In fact, only a finite number of times t need to be checked. The function $L_i(t)$ is piecewise monotonically decreasing, that is $\lceil t/T_j \rceil / t$ is strictly decreasing except at a finite set of values called the rate monotonic scheduling points. When t is a multiple of one of the periods T_j, $1 \le j \le i$, the function has a local minimum, is left continuous and jumps to a higher value to the right. Consequently, to determine if τ_i can meet its deadline one need only search over these local minima, the multiples of $T_j \le T_i$ for $1 \le j \le i$. Specifically, let

$$S_i = \{k \cdot T_j \mid j=1,\ldots,i; k=1,\ldots,\lfloor T_i/T_j \rfloor\}.$$

LSD "Scheduling Points"

$$S_i = \{k \cdot T_j \mid j=1,\ldots,i;\ k=1,\ldots,\lfloor T_i/T_j \rfloor\}.$$

The elements of S_i are the scheduling points for task i, the deadline of task i and the arrival times of tasks of priority higher than i before the deadline of task i assuming a critical instant phasing. For example, suppose we consider three tasks with periods $T_1 = 5$, $T_2 = 14$, and $T_3 = 30$. The scheduling points are given by $S_1 = \{5\}$, $S_2 = \{5, 10, 14\}$ and $S_3 = \{5, 10, 14, 15, 20, 25, 28, 30\}$. It follows that

$$L_i = \min_{\{t \in S_i\}} L_i(t).$$

LSD Summary

THEOREM 2: A set of n independent periodic tasks scheduled by the rate-monotonic algorithm will always meet its deadlines, for all task phasings, if and only if

$$\forall i,\ 1 \le i \le n,\ \min_{(k,l)\in R_i} \sum_{j=1}^{i} C_j \frac{1}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil \le 1$$

where C_j and T_j are the execution time and period of task τ_j respectively and $R_i = \{(k, l) \mid 1 \le k \le i,\ l = 1, \ldots \lfloor T_i/T_k \rfloor\}$.

Generalization of Periodic Tasks

- So far,
  deadline of a periodic task
  = end of period
- Generalization:
  periodic tasks with fixed **deadlines** ***relative*** to start of period
  (deadlines possibly *sooner* than end of period).

Deadline-Monotonic (DM) Scheduling

- Assume ***relative*** deadlines are *constant*.
- Assign **priorities** in order of nearest relative deadline.
- Since the relative deadlines are constant, this is a *static* priority assignment.
- (When deadline = period, this becomes the same as rate-monotonic schedule.)
- DM has been shown ***optimal*** for this more general case (Leung and Whitehead, 1982).

If relative deadlines are ≤ period and not constant

- EDF (dynamic scheduling) always works
- Schedulability can be checked using "processor demand" approach:
  - Processor demand in an interval [0, L] is the computation time required in order for all tasks to complete by their deadlines in that interval.
  - Check that processor demand ≤ length of interval, at selected times (which?).

Summary of Rule Applicability

	Deadline = Period	**Deadline ≤ Period**
Static Priority	Rate Monotonic	Deadline Monotonic
Dynamic Priority	Earliest Deadline First	Earliest Deadline First

Comments on RMS by Lehoczky, Sha, and Ding

In spite of the apparent dominance of the nearest deadline algorithm over the rate monotonic algorithm, the rate monotonic algorithm is of great practical importance [1, 2, 5, 6, 10]. First, it can be used to ensure that the timing requirements of the most important tasks are met when a transient overload occurs. Second, it provides a convenient way to offer fast response times to aperiodic tasks while still meeting the deadlines of the periodic tasks using the deferrable server algorithm [2], the sporadic server algorithm [10] or the extended priority exchange algorithm [8]. Third, it can be modified to handle task synchronization requirements by using the priority ceiling protocol [5]. Fourth, it can be conveniently used to schedule tasks where imprecise computation is permitted [4]. Finally, it is easy to implement in processors, in I/O controllers and in communication media [9, 11]. Consequently, it provides an approach to system-wide timing integration.

References

- C. L. Liu and J. W. Layland. *Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment*. Journal of the Association of Computing Machinery. January 1973. pp. 46-61.

- John Lehoczky, Lui Sha, and Ye Ding. *The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior*. IEEE Real-Time Systems Symposium, 1989. pp.166-171.

- S.K. Baruah, L.E. Rosier, and R.R. Howell. *Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor*. Journal of Real-Time Systems, **2**, 1990.

Resource Access

- Semaphores can be used to control access to resources in a real-time system.
- Some interesting issues associated with priority arise.

Complex Interactions

- Mutual exclusion
- Priority
- Preemption

Priority + Mutual Exclusion

- We assume “binary” semaphores for now.
- A semaphore’s “value” is either 1 or 0.
- If the value is 0, zero or more process can be blocked.
- Normally the **highest-priority** blocked process should be unblocked first.

Preemption in Critical Section

- Can a task be preempted while in its critical section?
- If the critical section is brief, then possibly no need to preempt.
- If the critical section is long, then preemption by a higher priority process might be needed to avoid missing deadlines.

Priority Contention

- Consider two tasks, H high priority, L low priority, that **share a resource** protected by a critical section.
- Suppose L has locked the semaphore, but now H wants to lock it as well.

  H is (temporarily) **blocked** by L.
- This form of contention is usually allowed as a necessary part of operations.

Priority Inversion

Continuing the previous example . . .

- L, with the resource locked, could be preempted by a **medium priority task M** that doesn't necessarily use the resource.
- M could go on **indefinitely**, since its priority is higher than that of L, **indirectly** blocking H through L.
- So we have an **anomaly** of a lower-priority M blocking H, i.e. the priorities are effectively **inverted**.

Possible Resolutions of Priority Inversion

- **Abort** the low-priority task:
  - Messy if this leaves the system in an **inconsistent** state.
  - Require **roll-back** of the low-priority one.
- Use Priority Inheritance Protocol (PIP)
- Use Priority Ceiling Protocol (PCP)

Priority Inheritance Protocol (PIP)

- During the time L is in its critical section, **and** while H is blocked because of L,

  L **inherits** the (higher) priority H.

- Thus L cannot be preempted by M.

- So there is no inversion of M's priority over H's.

Possible Resolution of Blocking

- **Priority Inheritance Protocol**:
  - More generally: A task locking a shared resource **inherits** the priority of the **highest-priority** task blocked on the resource.
  - The locking task's priority is thus ***recomputed*** whenever:
    - Other tasks request or release the shared resource, or
    - The locking task leaves the critical section, in which case the highest-priority blocked task is awakened.

3-Task Example without PIP

Legend: Solid fill is executing, **Hatched is waiting for something**

Time	T1 (nominal priority 1)	T1.5 (nominal priority 1.5)	T2 (nominal priority 2)	Comment
1			Normal	T2 released
2			Holds Sema S	T2 requests S, granted
3	Normal		Still Holding S	T1 released, T2 preempted
4	Waiting on S		Still Holding S	T1 requests S, blocks
5		Normal	Still Holding S	T1.5 released, T2 preempted
6				**T1.5 is effectively blocking T1**
7				

Inversion

2-Task PIP Example

Legend: Solid fill is executing, **Hatched is waiting for something**

Time	T1 (nominal priority 1)	T2 (nominal priority 2)	Comment
1		Normal	T2 released
2		Holds Sema S	T2 requests S, granted
3	Normal	Still Holding S	T1 released, T2 preempted
4	Waiting on S	Still Holding S	T1 requests S, blocks, T2 **priority elevated to 1**
5	Holds Sema S	Preempted	T2 releases S
6	Normal		T1 releases S
7		Normal	T1 finishes, T2 resumes

3-Task Example with PIP

Legend: Solid fill is executing, **Hatched is waiting for something**

Time	T1 (nominal priority 1)	T1.5 (nominal priority 1.5)	T2 (nominal priority 2)	Comment
1			Normal	T2 released
2			Holds Sema S	T2 requests S, granted
3	Normal		Still Holding S	T1 released, T2 preempted
4	Waiting on S		Still Holding S	T1 requests S, blocks, **T2 priority elevated to 1**
5			Still Holding S	T1.5 released, T2 not preempted
6	Holds Sema S			T2 releases S, priority lowered, preempted by T1
7		Normal		T1 releases S and finishes

No Inversion

Transitivity Complication

- **Priority Inheritance must also be made Transitive**:
  - If T_1 blocks T_2, and T_2 blocks T_3, then T_1 should inherit the priority of T_3.
- Note that transitive inheritance is needed only in the case of **nested** critical sections, using different semaphores.

3-Task Transitive Example with PIP

Solid fill is executing, Hatched is waiting for something

Time	T1 (nominal priority 1)	T2 (nominal priority 2)	T3 (nominal priority 3)	Comment
1			Normal	T3 released
2			Holds Sema S2	T3 requests S2, granted
		Normal	Still Holding S2	T2 released, T3 preempted
3		Holds S1	Still Holding S2	T2 requests S2, granted
4	Normal	Still Holds S1	Still Holding S2	T1 released, T2 preempted
5	Waiting on S1	Holding S1	Still Holding S2	T1 requests S1, blocks, **priority of T2 elevated to T1**
6	Waiting on S1	Holding S1, Waiting on S2	Holds S2	T2 requests S2, blocks, **priority of T3 elevated to T1** (**trans**.)
7	Waiting on S1	Holding S1 and S2		T3 releases S2 and finishes, T2 acquires S2
8	Holds S1			T2 releases S2, then S1, T1 acquires S1

PIP Summary

- Jobs are scheduled based on their ***active priorities***.
- When a job J_i tries to ***enter a critical section*** and the resource is blocked by a lower priority job, the job J_i is blocked. Otherwise it enters the critical section.
- When a job J_i is ***blocked***, it transmits its active priority to the job J_k that holds the semaphore. J_k resumes and executes the rest of its critical section with a priority $p_k = p_i$ (it ***inherits*** the priority of the highest priority of the jobs blocked by it).
- When J_k exits a critical section, it ***unlocks*** the semaphore and the highest priority job blocked on that semaphore is awakened. If no other jobs are blocked by J_k, then p_k is set to *its original priority*, otherwise it is set to the highest priority of the jobs now blocked by J_k.
- Priority inheritance is ***transitive***, i.e. if 1 is blocked by 2 and 2 is blocked by 3, then 3 inherits the priority of 1 via 2.

Semaphore Implementation

- With PIP, there is some added overhead:
  - A locked semaphore must remember the process that locked it (so that it can change its priority, if necessary).
  - Additional work in handling transitivity

Mars Pathfinder Experience

"But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data.

Mars Pathfinder used VxWorks

"VxWorks provides preemptive priority scheduling of threads. Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks."

"Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft."

- A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes)."

Mars Pathfinder Thread Priorities

- The meteorological data gathering task ran as an infrequent, low priority thread, … When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex.
- The spacecraft also contained a communications task that ran with medium priority."

☞

High priority: retrieval of data from shared memory
Medium priority: communications task
Low priority: thread collecting meteorological data

Mars Pathfinder Priority Inversion

"Most of the time this combination worked fine. However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running. After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset. This scenario is a classic case of priority inversion."

Mars Pathfinder Problem Solved

Priority inheritance also solved the Mars Pathfinder problem: the VxWorks operating system used in the pathfinder implements a flag for the calls to mutex primitives. This flag allows priority inheritance to be set to "on". When the software was shipped, it was set to "off".

The problem on Mars was corrected by using the debugging facilities of VxWorks to change the flag to "on", while the Pathfinder was already on the Mars [Jones, 1997].

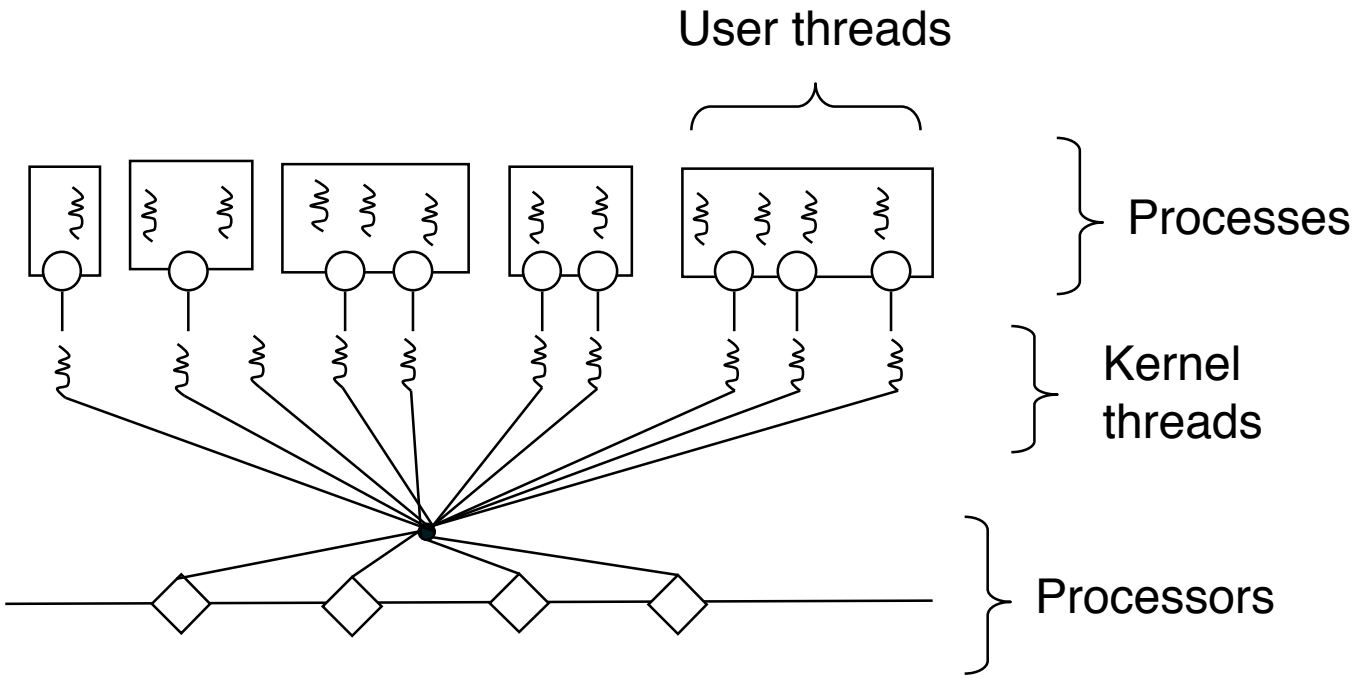
Computing Blocking Time

- If the computation time (without blocking) within critical sections is known, then the blocking time due to priority inheritance can be computed.
- This can be used in determining schedulability of a system of tasks with shared resources, such as a rate-monotonic scheduler.

Pragmatic Example: Solaris Operating System

- Solaris kernel schedules based on LWP's (Light-Weight Processes, aka "Kernel Threads").
- Regard LWP's as schedulable units of processor time ("slots").
- A new LWP can be created as an optional aspect of creating a new thread.

Solaris LWP's vs. Processes



Example: Solaris Operating System

- Each LWP has a priority, in one of three coarse classes:
  - RT (real-time) is highest.
  - System class is middle (not used by user processes).
  - TS (time-share) is lowest.

Example: Solaris Operating System

- For the time-sharing class, the dispatch priority is calculated from
  - amount of CPU used since last I/O (less ⇒ higher-priority)
  - its Unix ***nice*** level (set by the user)
- For the real-time class,
  - the highest-priority LWP runs until it blocks, terminates, reaches end of time-slice, or is preempted.

nice (Unix)

NAME

nice - run a program with modified scheduling priority

SYNOPSIS

nice [OPTION] [COMMAND [ARG]...]

DESCRIPTION

Run COMMAND with an adjusted niceness, which affects process scheduling.
With no COMMAND, print the current niceness.
Nicenesses range from -20 (most favorable scheduling) to 19 (least favorable).

-n, --adjustment=N
add integer N to the niceness (default 10)

--help display this help and exit

--version
output version information and exit

NOTE: your shell may have its own version of nice, which usually supersedes the version described here. Please refer to your shell's documentation for details about the options it supports.

Example: Solaris Operating System

- When a process is created, its LWP gets the scheduling class and priority of the parent process.
- A **thread** can be either:
  - ***bound*** to a specific LWP
  - ***unbound*** (multiplexed among various LWP's)
- All unbound threads in a process have the same class and priority.
- Bound threads have the class and priority of the LWP to which they are bound.

Example: Solaris Operating System Priority Inheritance

- Solaris implements **“basic” priority inheritance protocol**:
  - When a higher-priority thread is blocked, its priority is given to the lower-priority thread blocking it.
  - When the lower-priority thread ceases to block a higher priority one, its priority is set back to the original priority.
- Source: Ben Catanzaro, *Multiprocessor System Architectures*, Sun Microsystems, 1994.

Commentary on the Solaris Protocol from VxWorks FAQ

"The priority inheritance protocol also accounts for the ownership of more than one mutual exclusion semaphore at a given time. A task in such a situation will execute at the priority of the highest priority task blocked on any of the owned resources. The task will return to its normal, or standard, priority only after relinquishing **all** of the mutual exclusion semaphores with the inversion safety option enabled.

If you use nested mutex semaphores with priority inheritance turned on, then when a task inherits a high priority due to some inner semaphore it owns, **it doesn't lose that priority until it relinquishes all the semaphores** it holds. **This doesn't quite follow the rules for priority inheritance** (to the extent that there really are any rules) in that normally, **a task's inherited priority should decrease as it releases** each nested semaphore to whatever the priority ceiling is for the semaphores it still holds. Getting this incremental priority reduction to work right in practice, though, is extremely difficult (**some of the SUN papers on the Solaris real time scheduling indicate that this was one of the hardest things for SUN to get right in their OS upgrades**)."

PIP Drawbacks

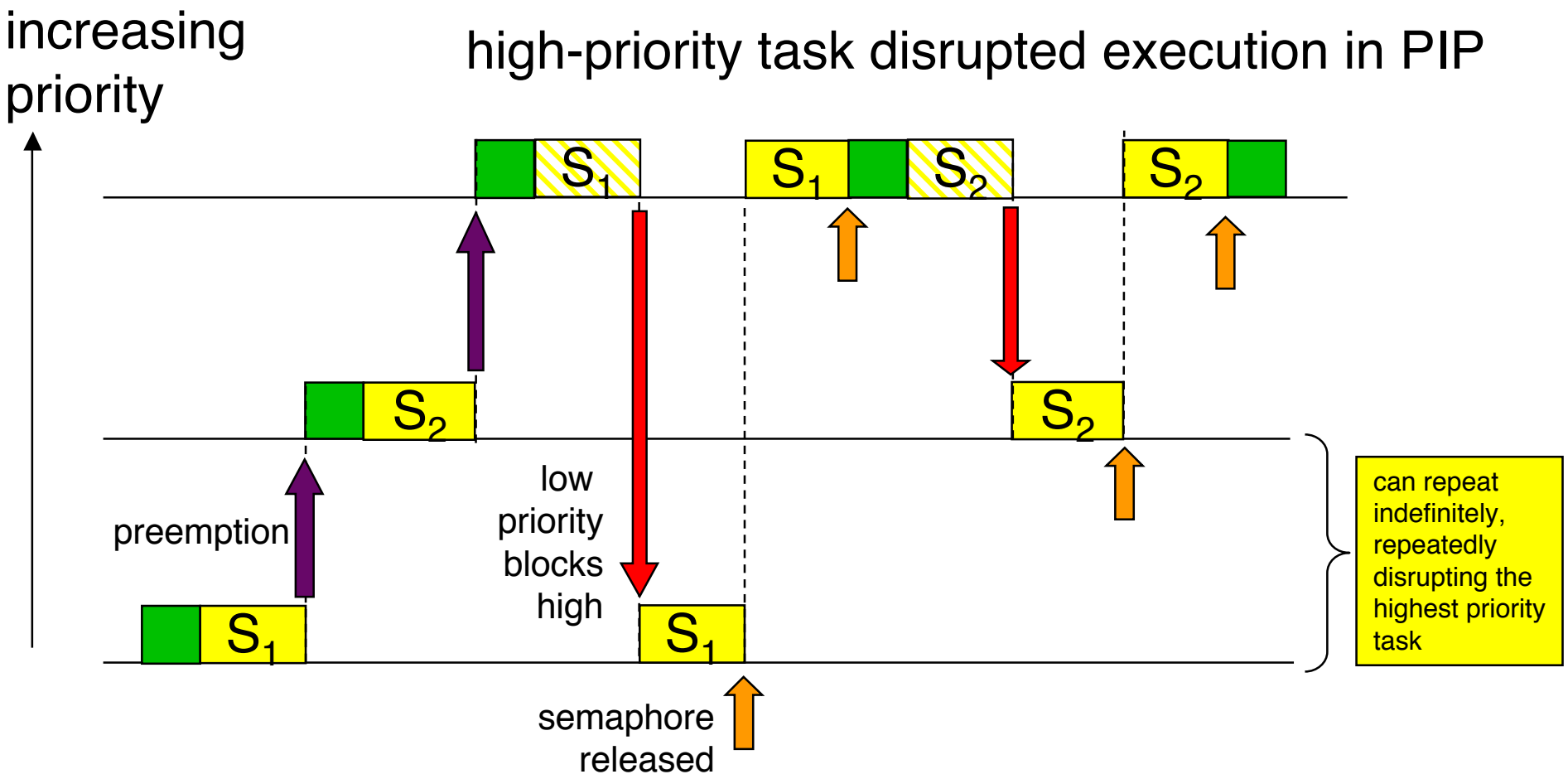
- The PIP does not prevent a high priority task from being blocked **multiple times** by lower priority ones, extending the blocking time **indefinitely** (“livelock”).
- No deadlock prevention measures are included in the protocol, but PIP does not particularly help with this.

Chained Blocking

- A deficiency of the PIP:

  Once a task gains entry to a critical section, it could still *subsequently* be blocked by a *lower* priority task.

Chained Blocking in PIP



Priority Ceiling Protocol (PCP)

- Each **semaphore** S has a (static) **priority ceiling** C(S) equal to the priority of the **highest-priority task that could possibly lock it** (**must establish in advance!!**).

- A task is allowed to enter a critical section only if its priority is ***higher*** **than the priority** ***ceilings*** of **all** semaphores ***currently*** **locked** by other tasks. Otherwise the task is "**blocked**" on the semaphore (even if it hasn't really **locked** it yet).

- If a task is **blocked** on a semaphore in the sense above, the task **locking** that semaphore **inherits** the priority of the blocked task (as in PIP).

Priority Ceiling Protocol (PCP)

- When a task **locking** a semaphore **unlocks it,** its priority is set to ceiling of the highest priority resource that it **now** holds (or to its base priority, if none).

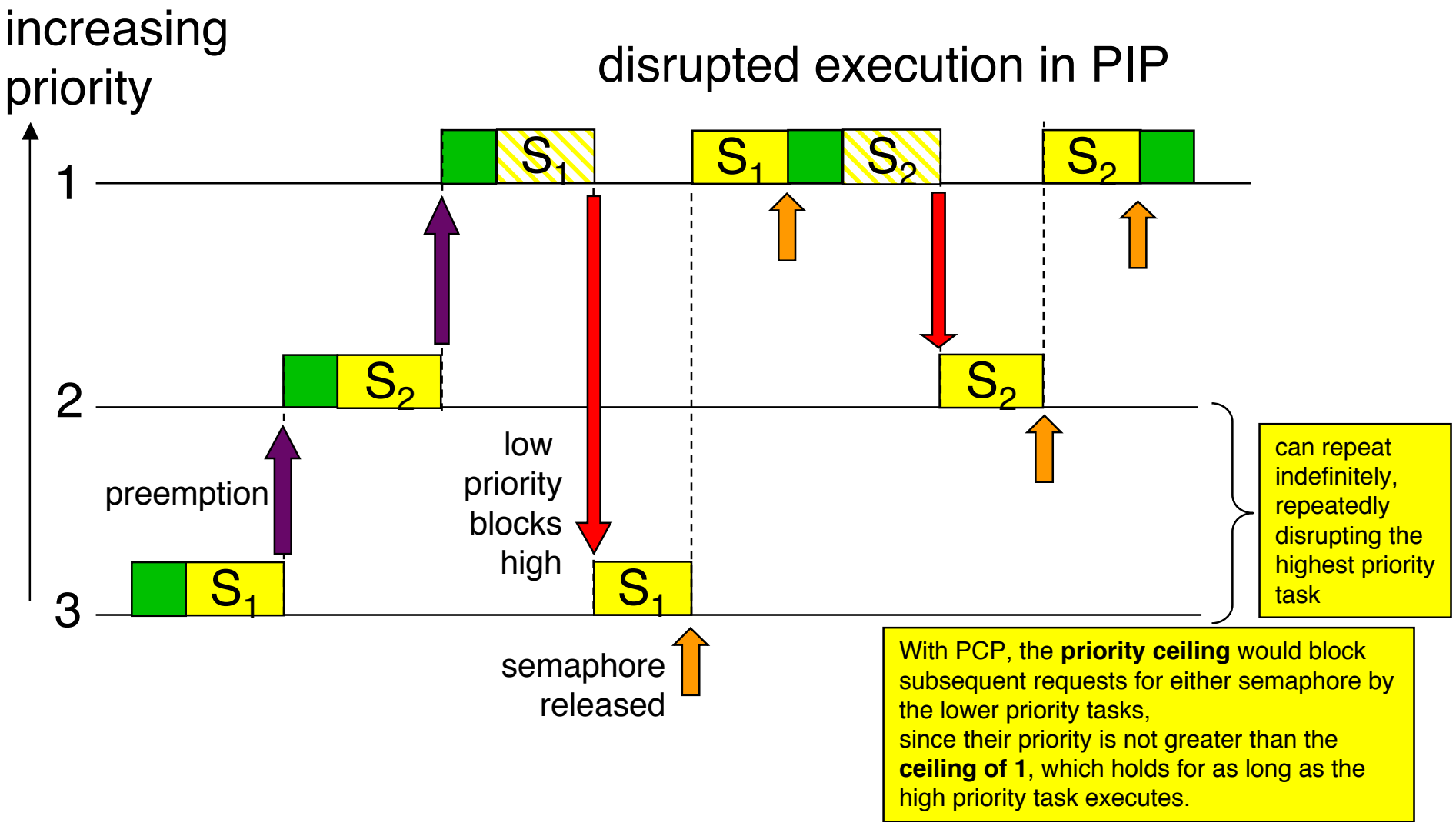
- Priority inheritance is constrained to be **transitive**, as with the PIP.

Priority Ceiling Protocol (PCP)

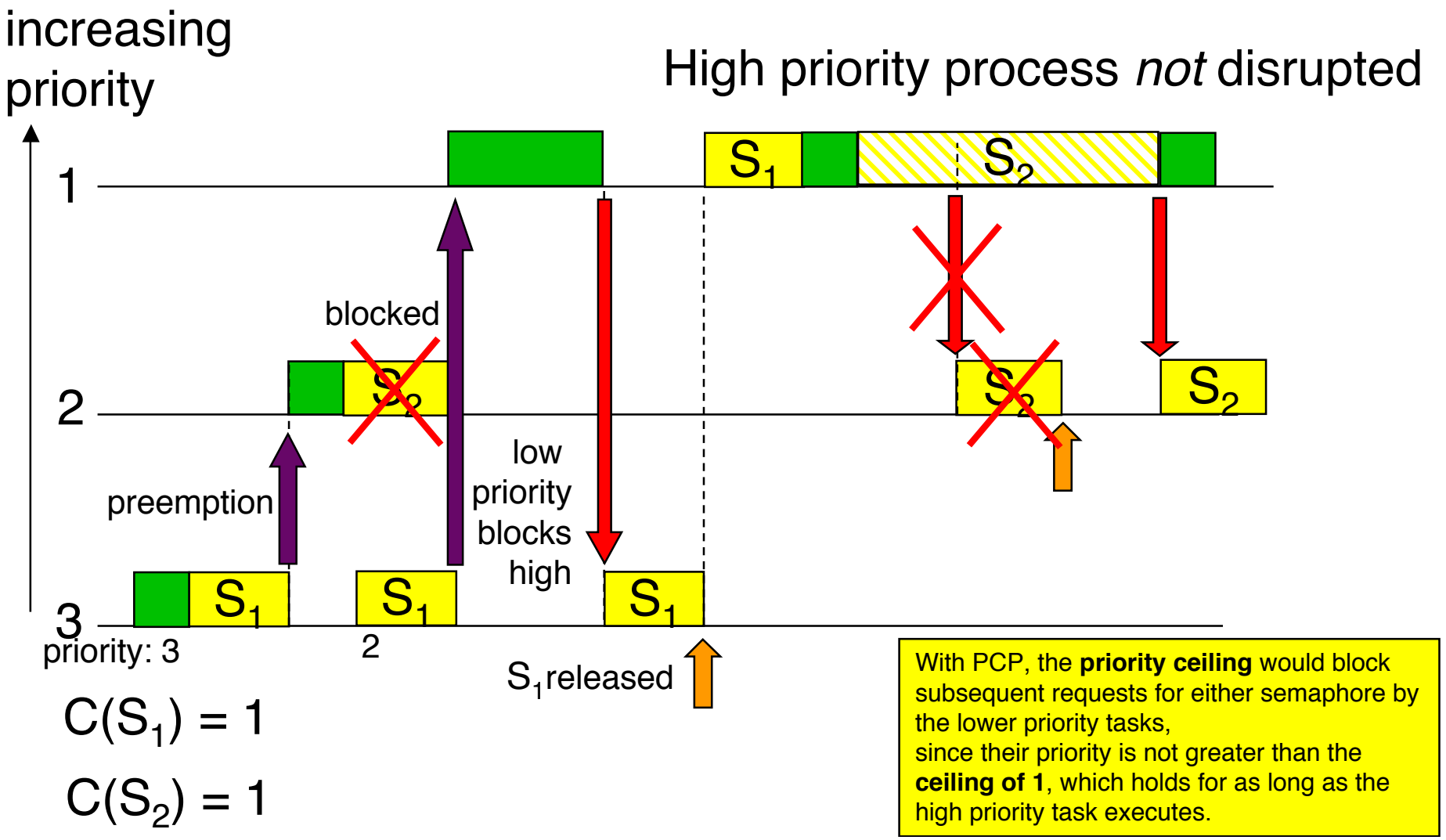
- The PCP **prevents the "chained blocking" deficiency** of the PIP:

  Once a task gains entry to a critical section, it could *subsequently* be blocked by a *lower* priority task.

Chained Blocking in PIP



Chained Blocking Avoided in PCP



Comparison: PIP vs. PCP

- PCP has **fewer context switches** at run-time, in that a high priority task cannot be blocked more than once at the same level.
- PCP is more demanding, in that it requires a **thorough analysis** of a task's behavior (in terms of which semaphores in might request).
- PCP **de facto prevents deadlocks**, since it induces an **ordering** on the way that resources can be requested.

More PCP Examples : Deadlock Prevention

J1 = $\{..., P(S_0) ..., V(S_0),...\}$

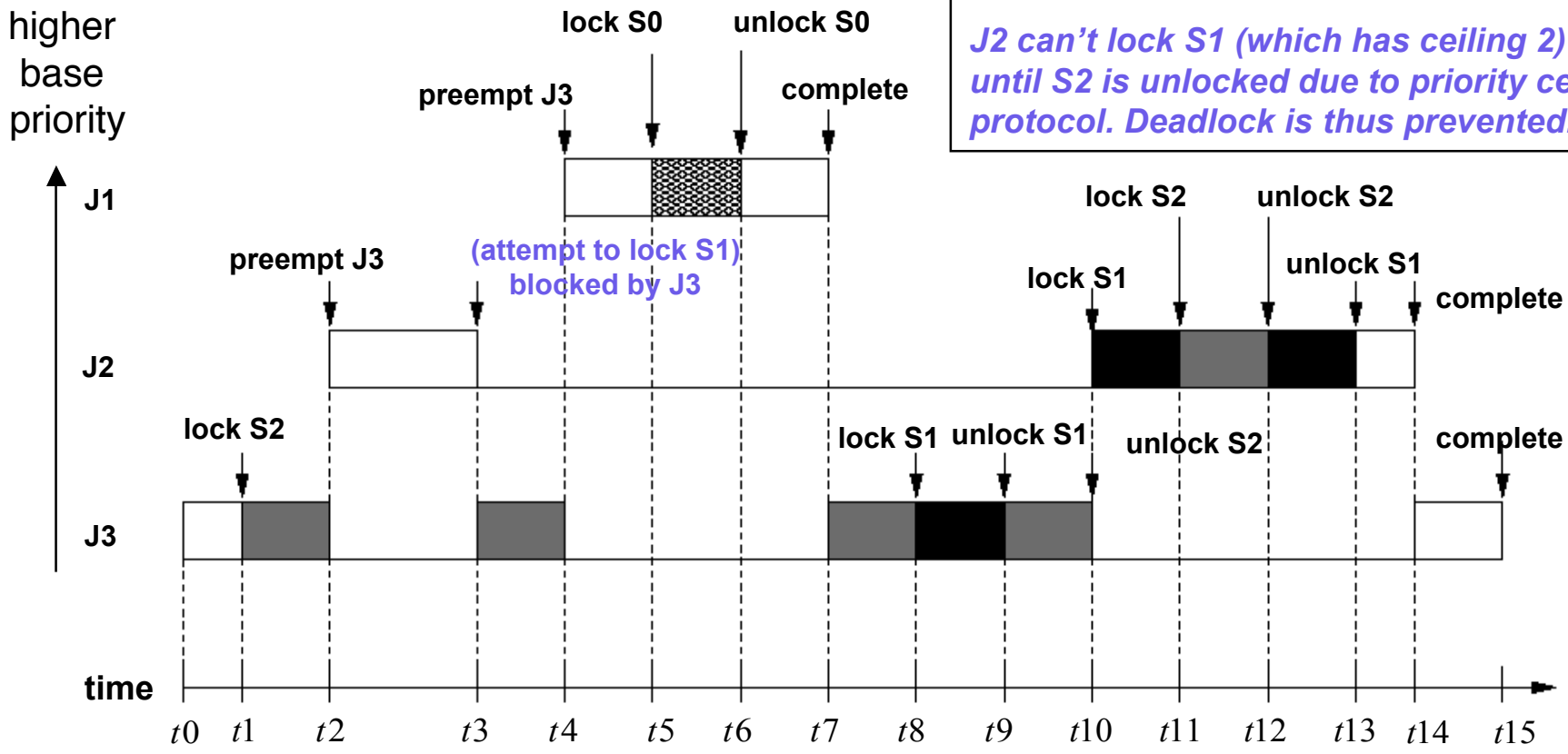
J2 = $\{..., P(S_1) ..., P(S_2),..., V(S_2) ..., V(S_1),...\}$

J3 = $\{..., P(S_2) ..., P(S_1),..., V(S_1) ..., V(S_2),...\}$

nested crossing semaphores invite deadlock

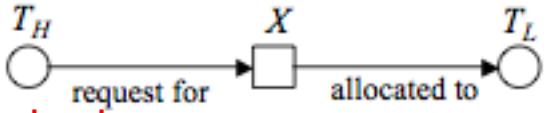
c(S0) = 1, c(S1) = c(S2) = 2

J2 can't lock S1 (which has ceiling 2) until S2 is unlocked due to priority ceiling protocol. Deadlock is thus prevented.



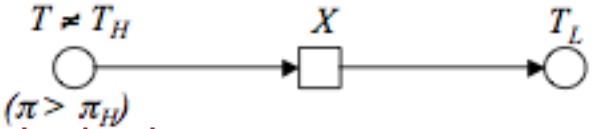
Summary of Basic Blocking Types (not including transitivity)

- Blocking: A higher-priority task waits for a lower-priority task.
- A task T_H can be blocked by a lower-priority task T_L in three ways:
  - directly, i.e.



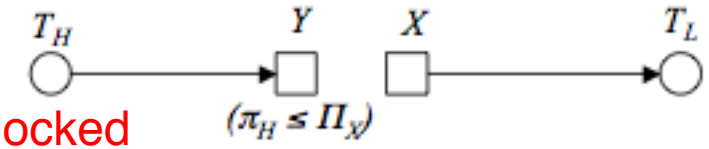
blocked

  - when T_L inherits a priority higher than the priority π_H of T_H.



blocked

  - When T_H requests a resource the priority ceiling of resources held by T_L is equal to or higher than π_H:

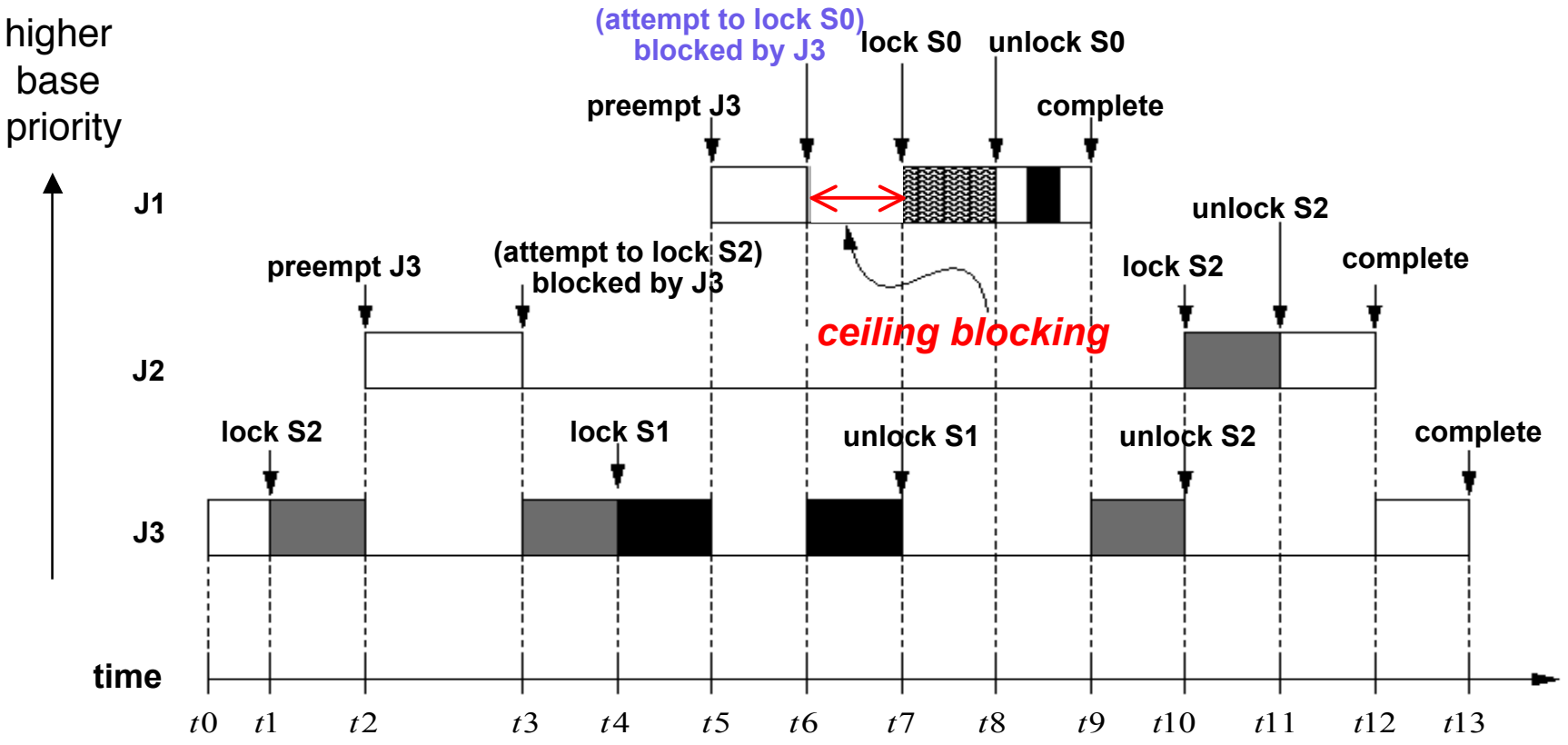


blocked

Ceiling Blocking (unavoidable)

J1 = {..., $P(S_0)$, ..., $V(S_0)$, ..., $P(S_1)$, ..., $V(S_1)$, ...}
J2 = {..., $P(S_2)$, ..., $V(S_2)$, ...}
J3 = {..., $P(S_2)$, ..., $P(S_1)$, ..., $V(S_1)$, ..., $V(S_2)$, ...}

C(S0) is 1.
J1 would need higher priority still in order to lock S0.



PCP Implementation

- Individual semaphore queues are no longer necessary: Any task blocked by the PCP can be kept in the ready queue.

- A list of currently-locked semaphores, ordered by priority ceiling, can be maintained to simplify blocking criterion.

Reference

- L. Sha, R. Rajkumar, J. Lehoczky, *Priority Inheritance Protocols*, IEEE Trans. Computers, **20**, 9, pp. 1175-1185, Sep. 1990.

Example Implementation of PCP (Posix Threads)

pthread_mutex_setprioceiling()

set the priority ceiling of a mutex

SYNOPSIS

#include <pthread.h>
int pthread_mutex_setprioceiling(pthread_mutex_t *mutex, int ceiling, int *oldceiling);

DESCRIPTION

The pthread_mutex_setprioceiling() function either locks the mutex if it is unlocked, or blocks until it can successfully lock the mutex, then it changes the mutex's priority ceiling and releases the mutex. When the change is successful, the previous value of the priority ceiling is returned in oldceiling. The process of locking the mutex need not adhere to the priority ceiling protocol.

Ada 9X Priority Ceiling

- Task definitions allow for a pragma Priority as follows:
  pragma Priority(expression)
- Task priorities:
  - ***base priority***: priority defined at task creation, or dynamically set with Dynamic_Priority.Set_Priority() method.
  - ***active priority***: base priority or priority inherited from other sources (activation, rendez-vous, protected objects).
- **Priority-Ceiling** Locking:
  - Every protected object has a *ceiling priority*: Upper bound on active priority a task can have when it calls a protected operation on objects.
- While task executes a protected action, it **inherits the ceiling priority** of the corresponding protected object.
- When a task calls a protected operation, a check is made that its **active priority is not higher than the ceiling of the corresponding protected object**. A Program Error is raised if this check fails.

SRP: Stack Resource Policy
(T.P. Baker)

- Extends PCP.
- Permits dynamic priority assignment.
- Accommodates multi-unit resources (instead of just {0, 1}.
- Allows sharing of runtime stack-based resources.

SRP vs. PCP

- PCP: Task is blocked when it wants to **lock** resource

  vs.

- SRP: Task is blocked when it attempts to **preempt** another task.

Consequences

- A task will **never block** on a resource once it is started.
- Reduces the number of context switches compared to PCP.

Preemption Level

- In addition to a (possibly dynamic) priority, each task has a ***static* preemption level π,** determined by the **resources it will require**.
- Priority is used when tasks are running **without** using resources.
- One task can preempt another iff its **preemption level** is higher.
- **Requirement: If** task T_a is **released *after*** T_b and T_a has **higher priority** than T_b, **then** T_a **must also** have a **higher preemption level** than T_b.

Assigning Preemption Level
One Method: Relative Deadline

For the specific priority assignments mentioned in this paper, the preemption level of a job will be inversely proportional to the *relative deadline* of the job. The relative deadline of a job J is a fixed value, $D(J)$, such that if a request for execution of J arrives at time t, that execution must be completed by time $t + D(J)$. In other words, the relative deadline of a job is the size of the scheduling "window" in which each execution of the job must fit.

From T.P. Baker's original paper

Assigning Higher Preemption Level to Smaller Relative Deadline

Suppose there are two jobs, J and J', with relative deadlines $D(J) = D$ and $D(J') = D'$, respectively. Suppose $\mathcal{J}$ is a job execution request of J such that $Arrival(\mathcal{J}) = t$, and $\mathcal{J}'$ is a request of J' such that that $Arrival(\mathcal{J}') = t'$. In order for $\mathcal{J}'$ to preempt $\mathcal{J}$, we must have:

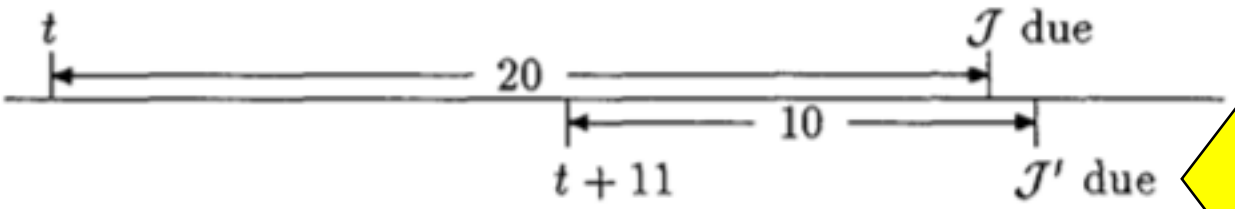
i. $t < t'$ (so $\mathcal{J}$ can get started);

ii. $p(J) < p(J')$ (so $\mathcal{J}'$ can preempt).

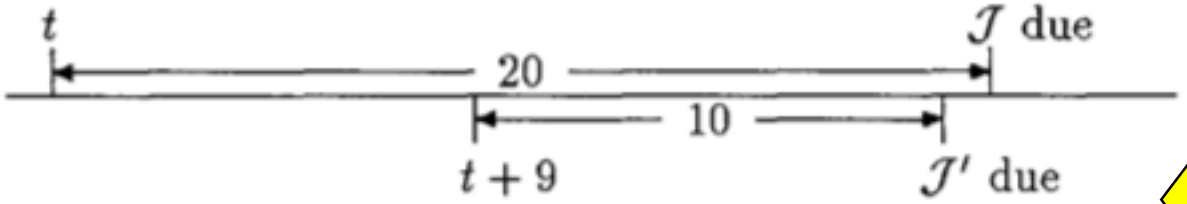
With EDF scheduling, $p(J) < p(J')$ iff $t' + D' < t + D$, so it follows that $D' < D$. It is therefore consistent to define preemption levels so that $\pi(J) < \pi(J')$ iff $D(J) > D(J')$.

From T.P. Baker's original paper

Preemption Level vs. Priority



Higher Preemption Level (10 < 20)
but Lower Priority (21 > 20)



Higher Preemption Level (10 < 20)
and Higher Priority (19 > 20)

Preemption Level vs. Priority

An example will emphasize the difference between EDF priority and preemption level. Let $\mathcal{P}$ and $\mathcal{P}'$ be two periodic processes, with relative deadlines 20 and 10 (relative to arrival times), respectively. Preemption level 1 is assigned to jobs of $\mathcal{P}$ and preemption level 2 is assigned to jobs of $\mathcal{P}'$, since the relative deadline of $\mathcal{P}'$ is shorter than the relative deadline of $\mathcal{P}$. $\mathcal{P}'$ can never be preempted by $\mathcal{P}$, but this does not mean that job execution requests of $\mathcal{P}'$ always have higher priority than those of $\mathcal{P}$. Suppose a job-request $\mathcal{J}$ of $\mathcal{P}$ arrives at time t, and a job-request $\mathcal{J}'$ of $\mathcal{P}'$ arrives at time $t+11$. Since the absolute deadline of $\mathcal{J}$ is $t+20$ and the absolute deadline of $\mathcal{J}'$ is $t+21$, $\mathcal{J}$ will have higher priority than $\mathcal{J}'$. On the other hand, if $\mathcal{J}'$ had arrived at time $t+9$ its deadline would have been $t+19$ and we would have had $p(\mathcal{J}) < p(\mathcal{J}')$. Thus preemption level is different from priority. This is shown in Figure 3.

From T.P. Baker's original paper

SRP Dynamic Ceiling Computation

$\pi(J)$ **= preemption level of job J**

η_R **= the number of units of R that are currently available**

$\mu_R(J)$ **= the maximum requirement of job J for R**

Current Resource Ceiling (computed as a function of the units of R that are available):

$$C_R(n_R) = \max[\{0\} \cup \{\pi(J) : n_R < \mu_R(J)\}]$$

i.e., the **highest preemption level** of those jobs that could be blocked on R

System Ceiling (the maximum of the current ceilings of all of the resources):

$\Pi_s = \max\{C_{Ri} : i = 1,2,\ldots,m\}$

i.e., the maximum of the Current Resource Ceilings

SRP Rule

- To avoid deadlocks: Once execution begins, make sure that job is not blocked due to resource access.
- Otherwise: Low-priority, preempted, jobs may re-acquire access to CPU, but can not continue due to unavailability of stack space.

- Define: $\Pi(t)$: highest priority ceiling of all resources currently allocated. (*t is time*)
  If no resource allocated, $\Pi(t) = \infty$. (meaning *low* priority)

Protocol:

1. *Update Priority Ceiling*: Whenever all resources are free, $\Pi(t) = \infty$. The value of $\Pi(t)$ is updated whenever resource is allocated or freed.
2. *Scheduling Rule*: After a job is released, it is blocked from starting execution until its assigned priority is higher then $\Pi(t)$. At all times, jobs that are not blocked are scheduled on the processor in a priority-driven, preemptive fashion according to their assigned priorities.
3. *Allocation Rule*: Whenever a job requests a resource, it is allocated the resource.

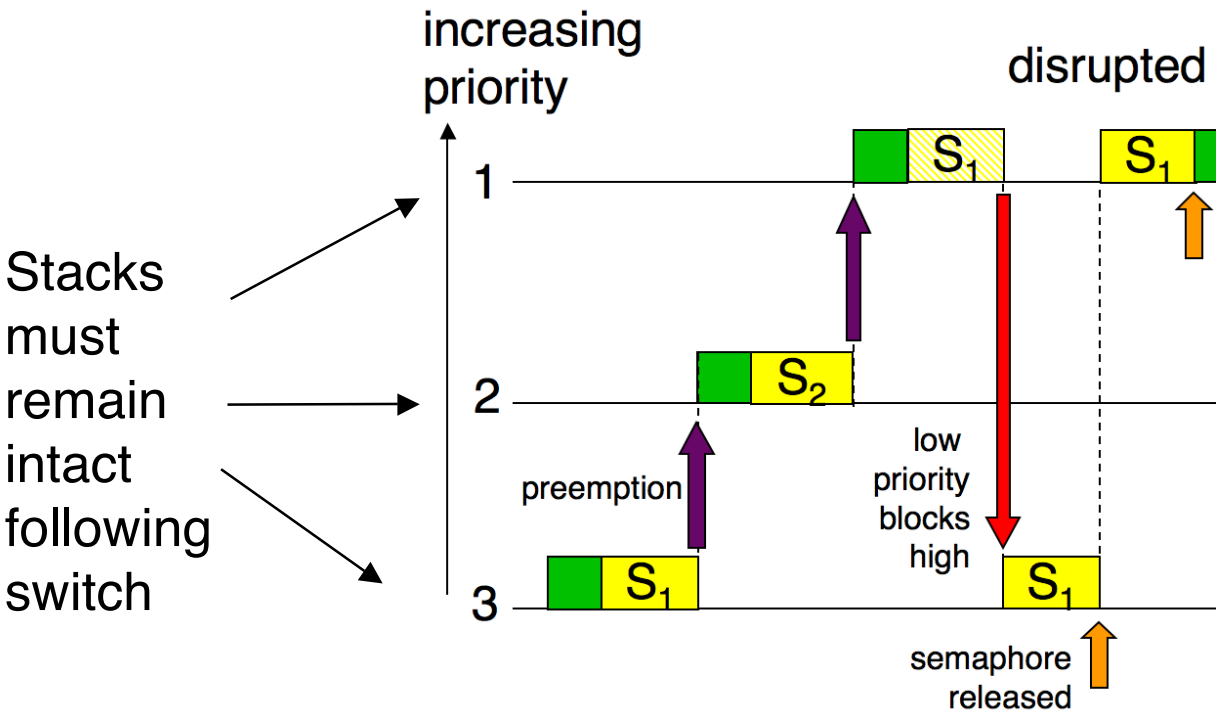
(*with no waiting*)

SRP Impact

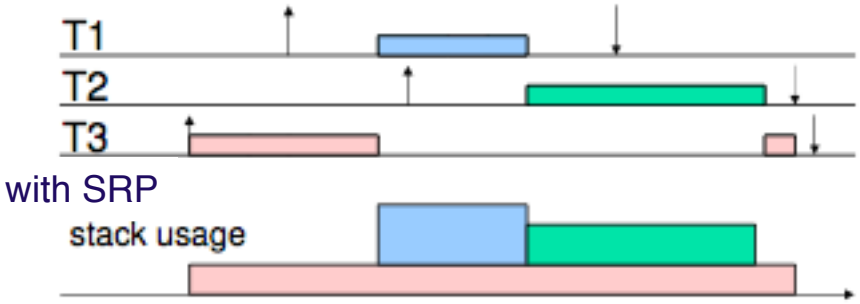
- The Stack-Based Priority-Ceiling Protocol is **deadlock-free**:
  - When a job begins to execute, all the resources it will ever need are free.
  - Otherwise, $\Pi(t)$ would be higher or equal to the priority of the job.
  - Whenever a job is preempted, all the resources needed by the preempting job are free.
  - The preempting job can complete, and the preempted job can resume.
- Worst-case blocking time of Stack-Based Protocol is the same as for Basic Priority Ceiling Protocol.
- Stack-Based Protocol smaller context-switch overhead (2 CS) than Priority Ceiling Protocol (4 CS)
  - Once execution starts, job cannot be blocked.

Stack-Sharing Possibility

- In general, it is not possible for tasks to share the same stack space:

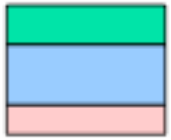


With SRP, Stack is Sharable



(less space,
T1 can't be preempted by T3)

without SRP



(more space)