Real-Time Computing

Definition of Real-Time Computing

- Real-time computing is computing in which the **time plays an essential role** in the result.

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

Obligatory Inspiring Example

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

- July 20, 1969, landing module 10,000 feet above the Moon:
  - Houston: “Eagle, you’re go for a landing.”
  - Houston: “One minute [of fuel left]”.
    ...
  - Lander: 100 feet, 3 1/2 down, 9 forward.”
  - Houston: “30 Seconds.”
  - Lander: “OK, engine stop.”

Example of Real-Time

(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."

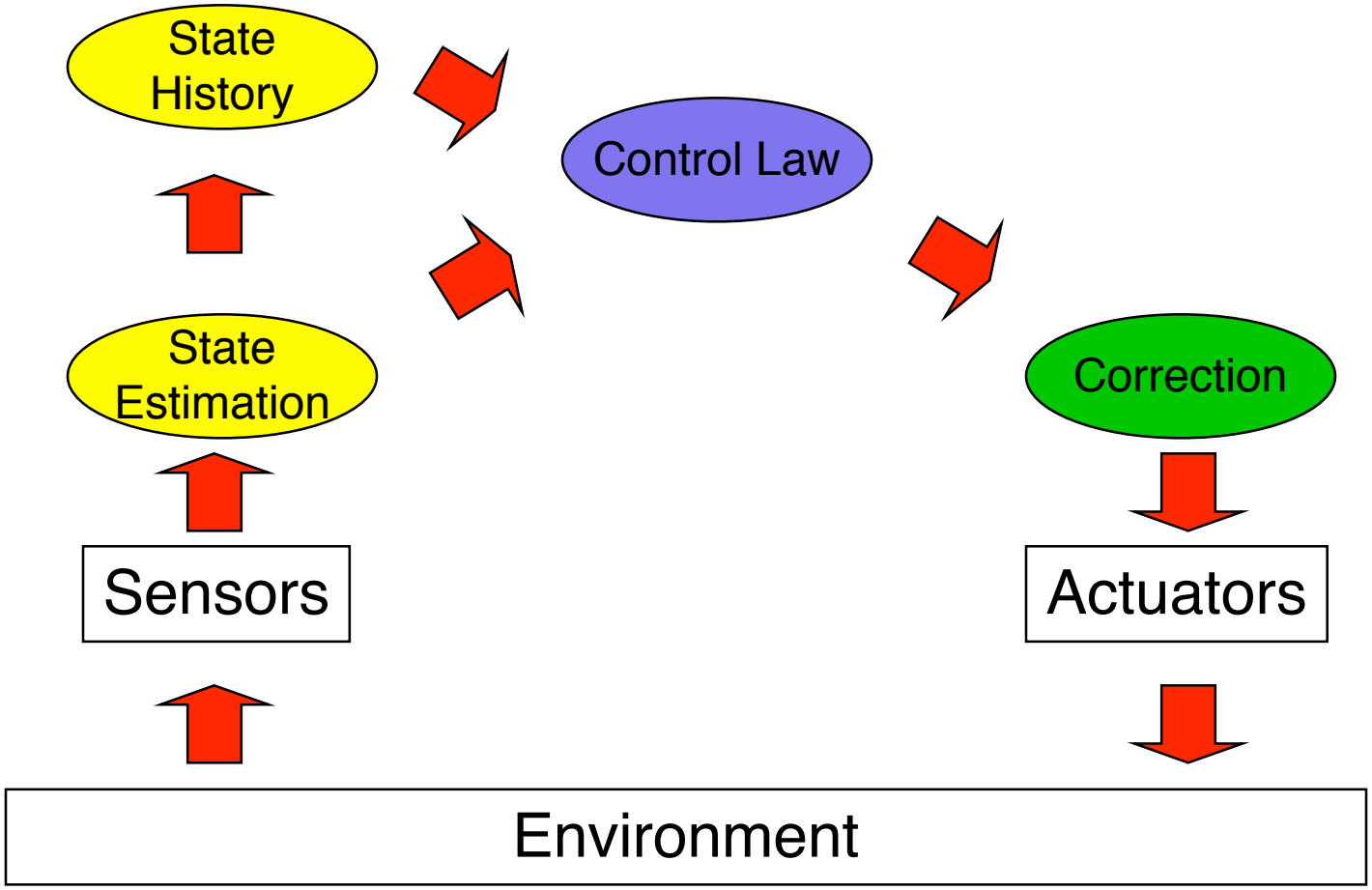
Control Loops

- 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}*

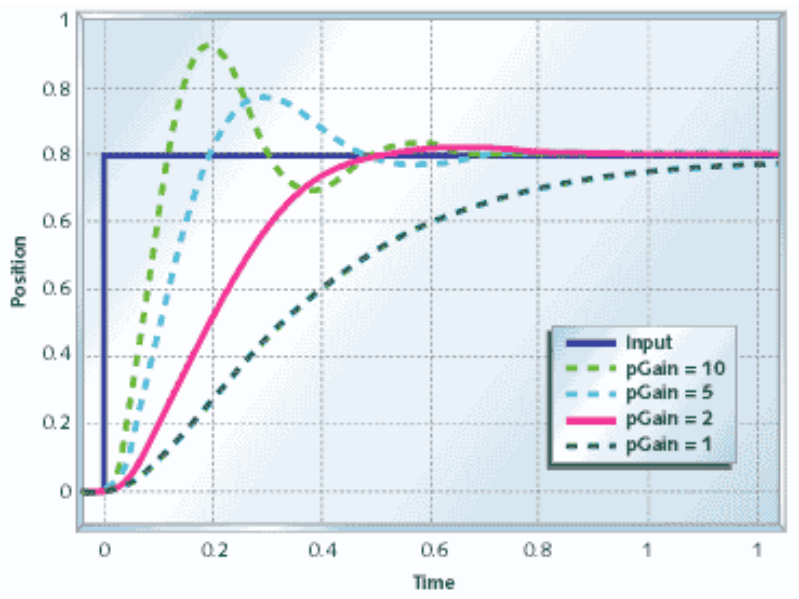
- PI controller (proportional-integral)

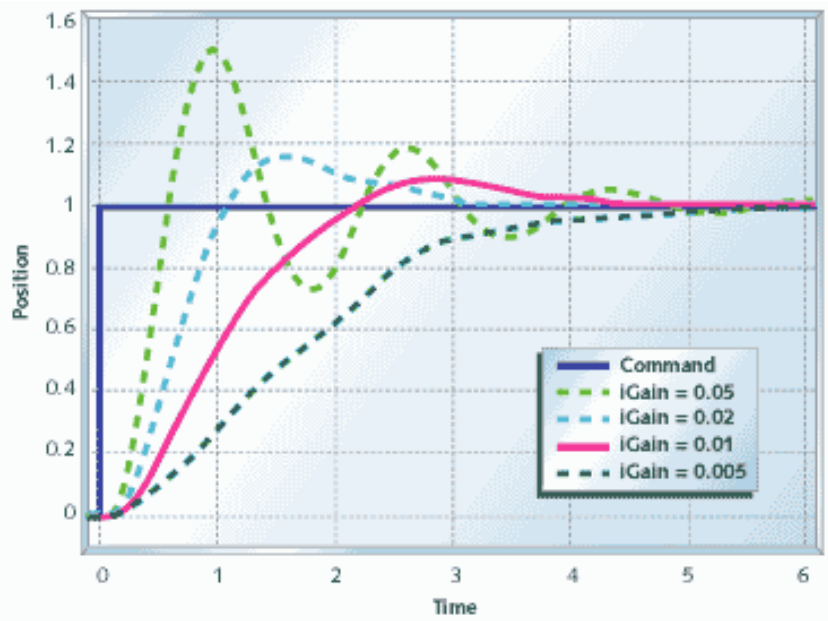
  {Measure error;
  Open the valve in proportion to integral of error}*

- PID controller (proportional-integral-derivative)

  Similar to PI, with “kicker” based on derivative of output

P vs. PI Controller Response

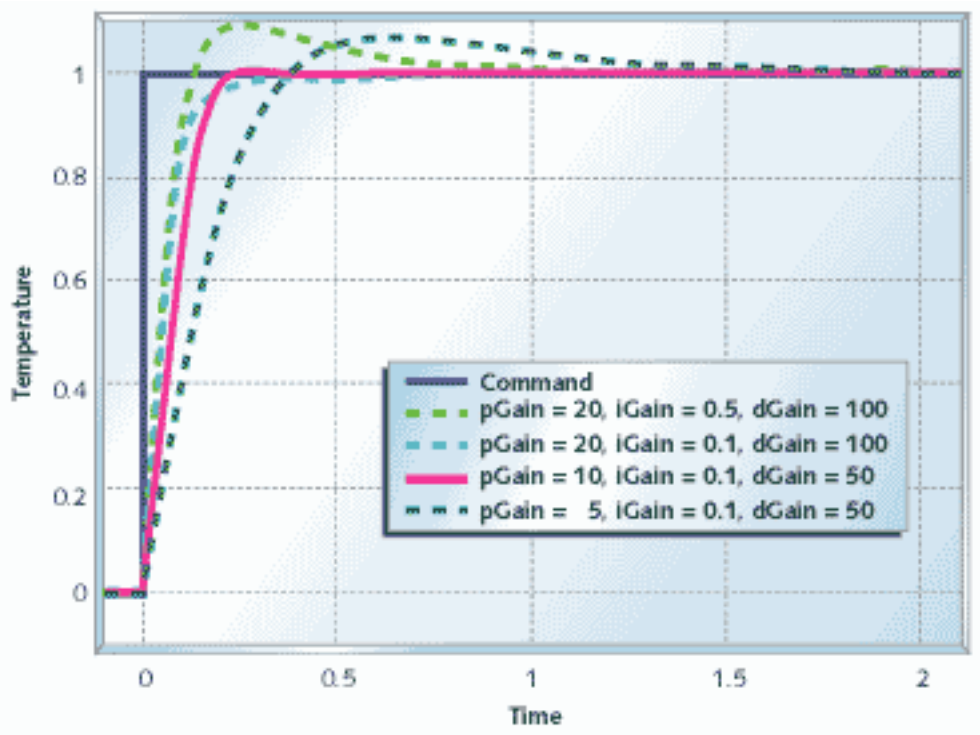




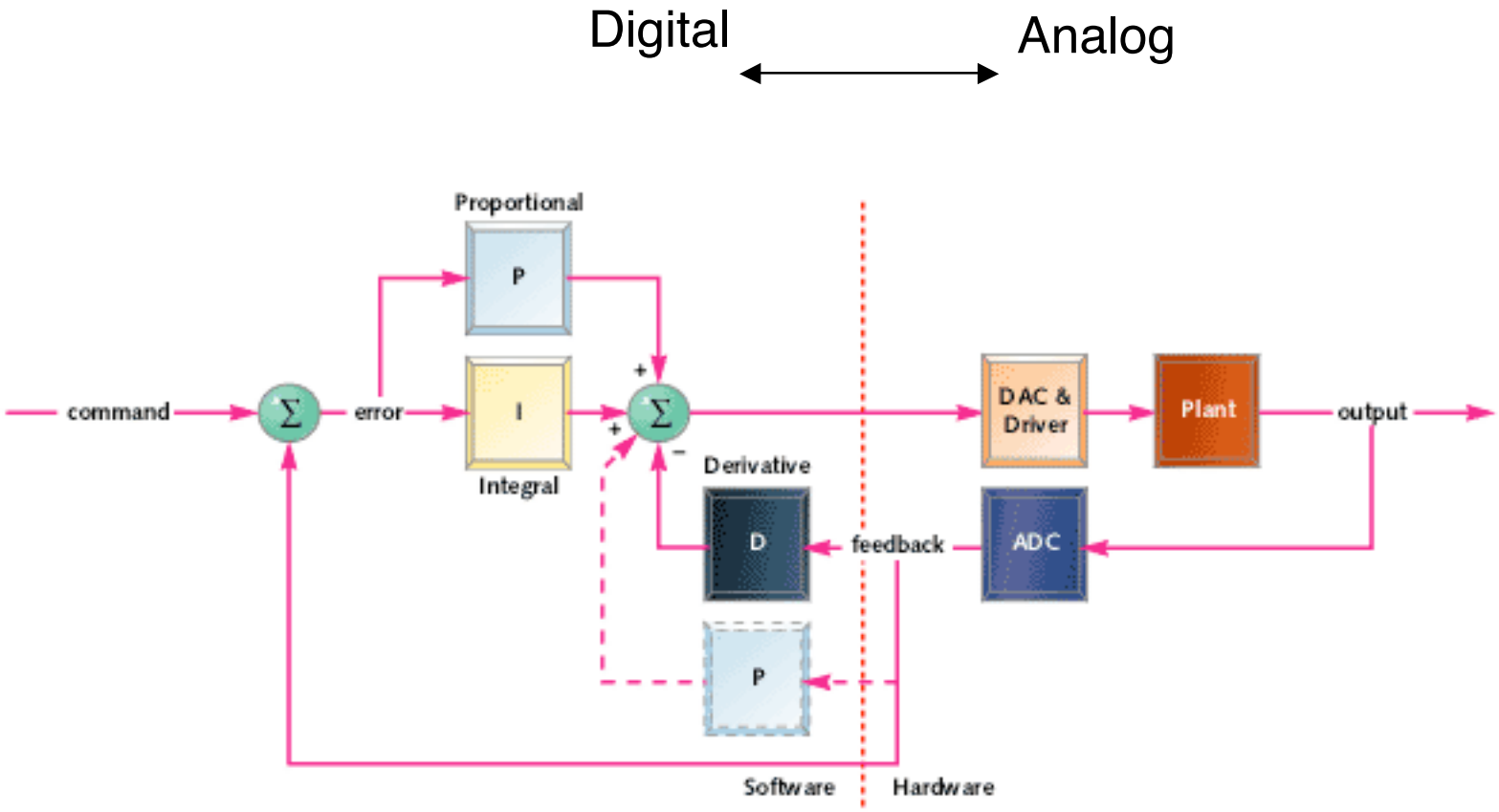
Figures from:

`http://www.embedded.com/2000/0010/0010feat3.htm`

PID Response



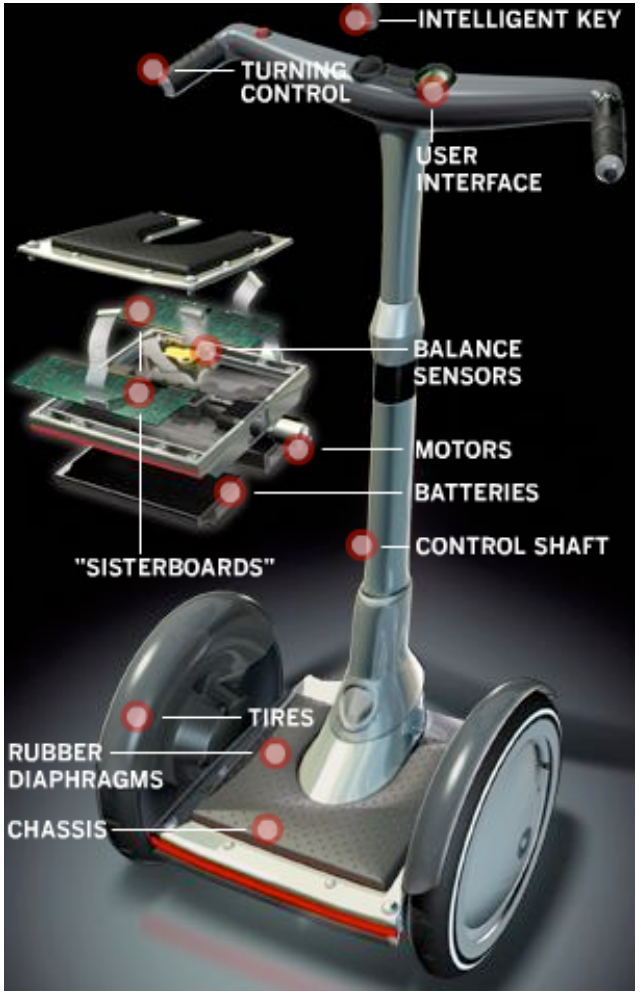
PID Controller



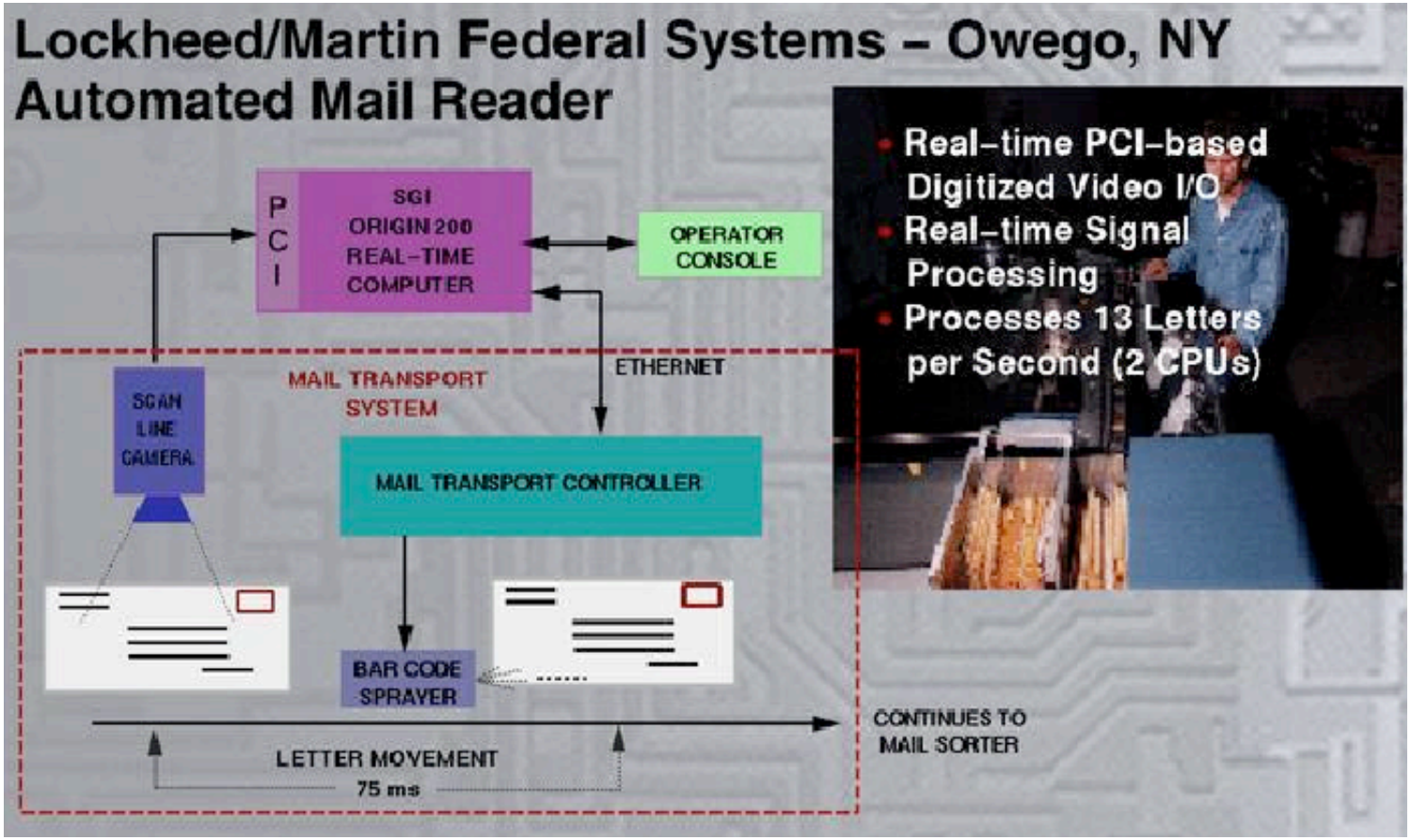
Sample PID Controller Code

```
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;
  }
```

Sample Controller Application



Real-Time Application

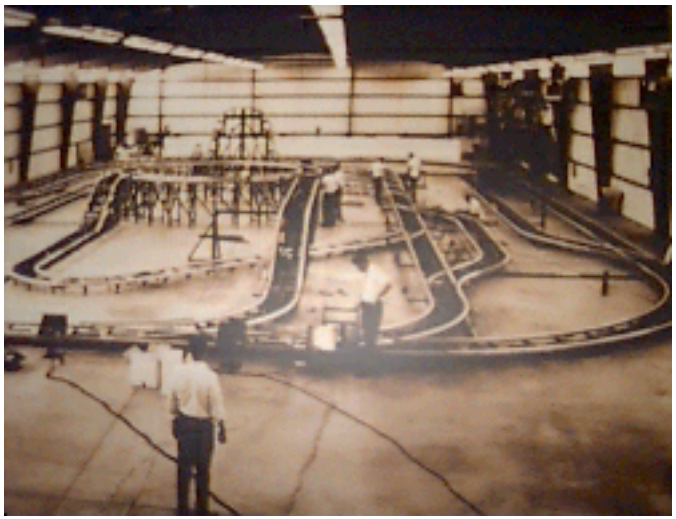


Real-Time Application

Lockheed Martin: Distributed Mission Training

Replacing flight hours with simulator hours

Real-Time Application
New Denver Airport Baggage Handler



BAE Automated Systems, Inc.

New Denver Airport

- 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 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

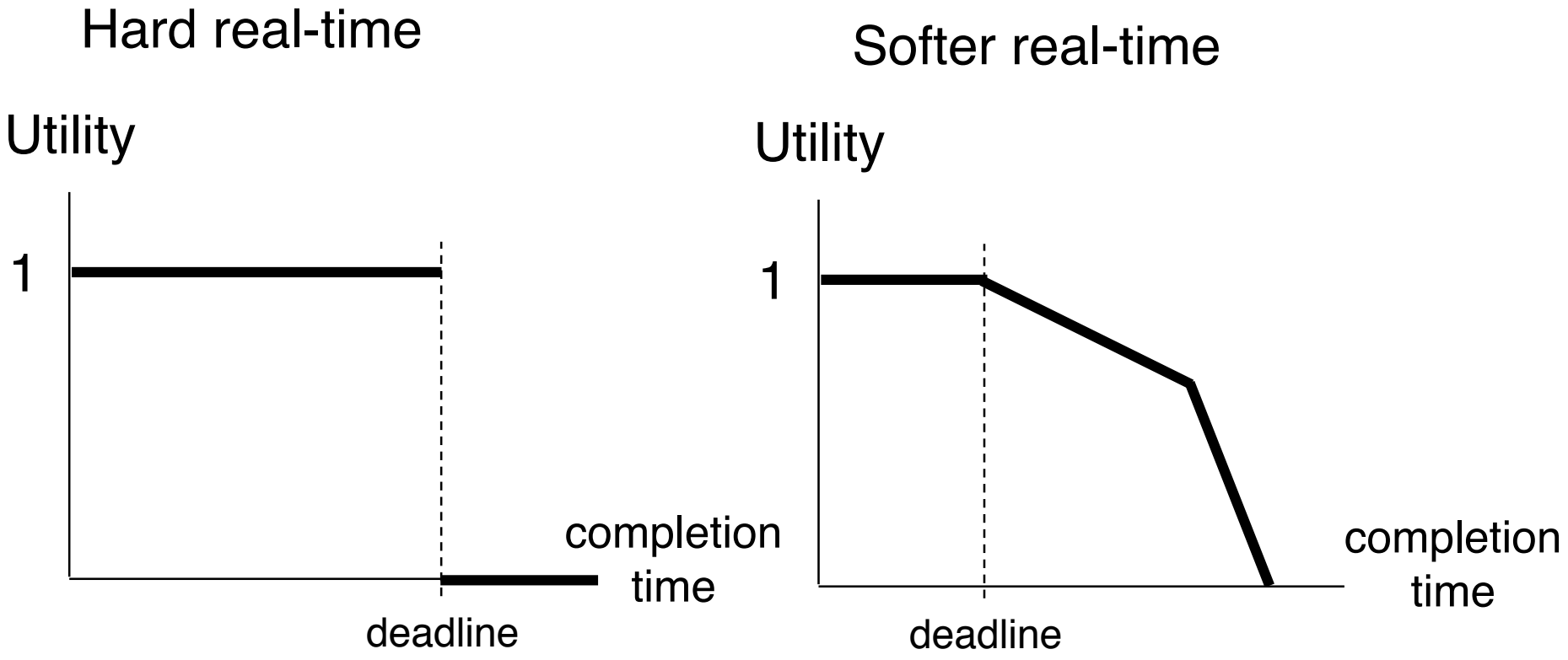
Distinctions

- “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 be correct.
- **Isochronous** real-time: Tasks should finish neither too late nor too early.

Hard Real-Time Examples

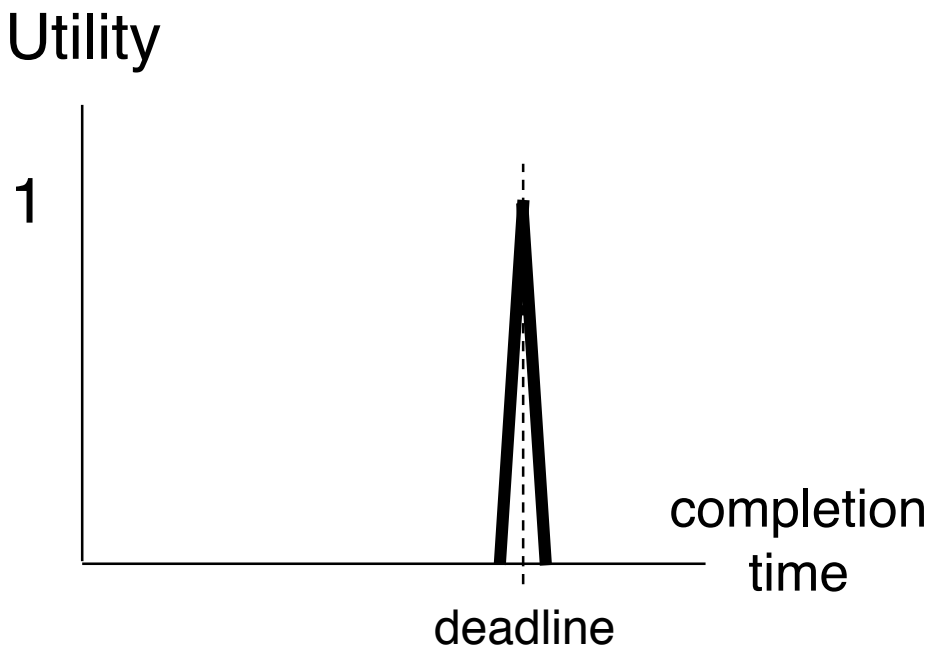
- Vehicular fuel or rendezvous problems
  - Lunar lander
  - Train scheduling
  - Baggage handlers
  - Assembly lines
- Production deadlines
  - Newspaper
  - Live TV show
  - Graduation

Hardness Expressed as a **Utility** Function



Isochronous Real-time Using Utility

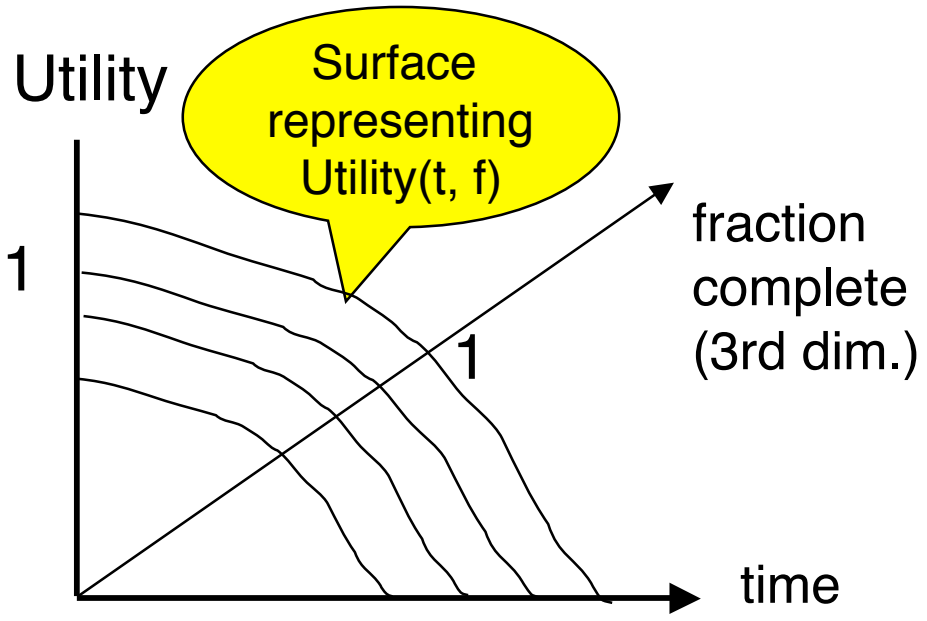
Isochronous real-time



Progressive Utility

- A **generalization** of the preceding concept regards Utility as a function of both **time** *and* **fraction** of the task **completed**.

Utility(t, f) is the utility of completing fraction f of the task by time t.



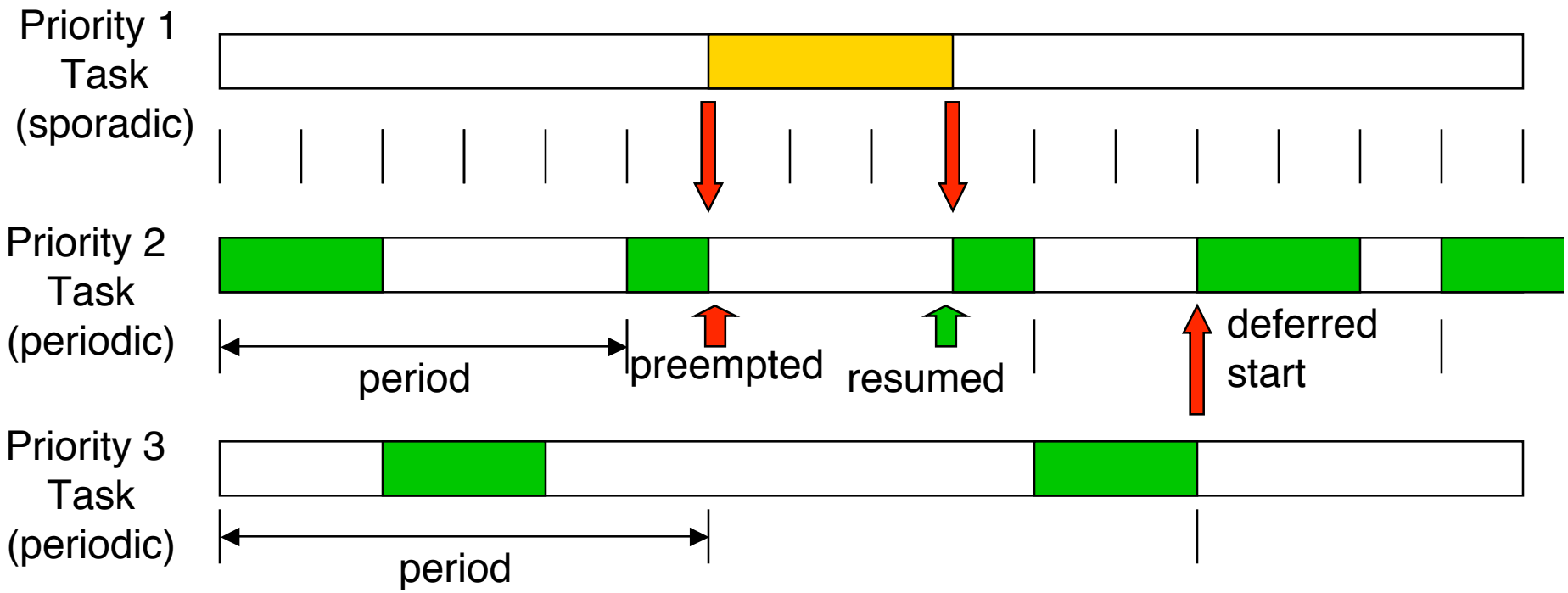
Real-Time may include Communication Considerations

- 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



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

- Ada (see also “Ravenscar profile”)
- PEARL (Process and Experiment Automation Realtime Language)
- Esterel (tick-based)
- Lustre (dataflow)
- Real-Time Java(s)

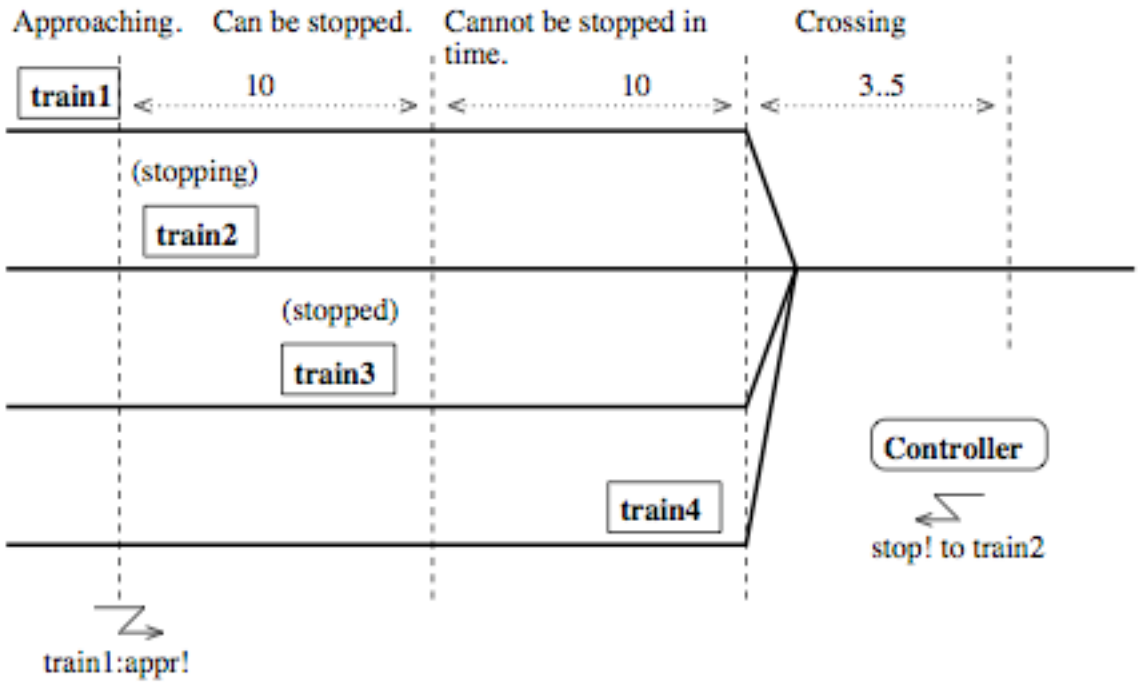
Real-Time Modeling Languages

- 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.
- 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



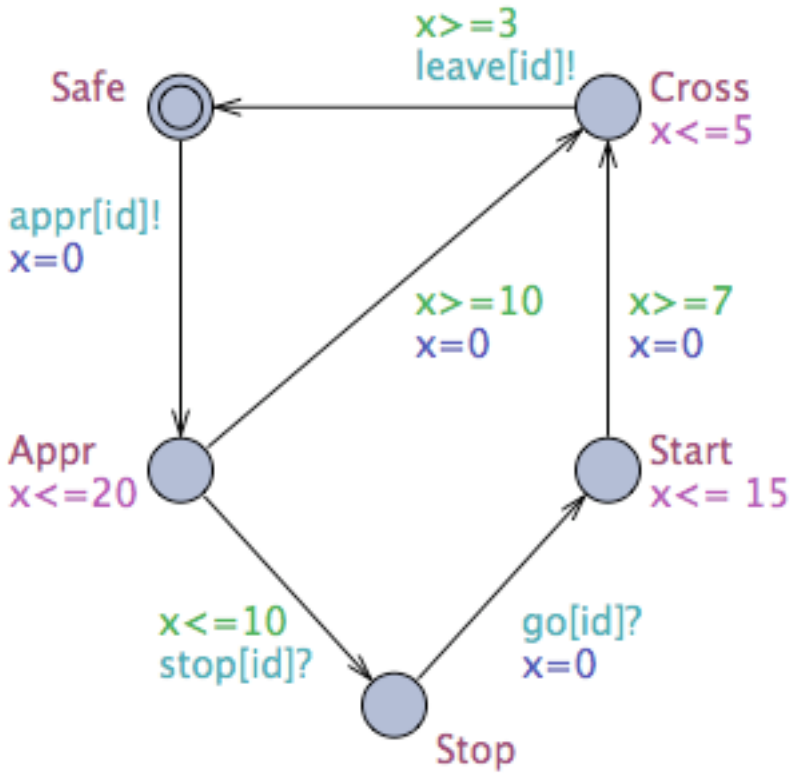
Assumptions:

10 time units are required to stop the train.

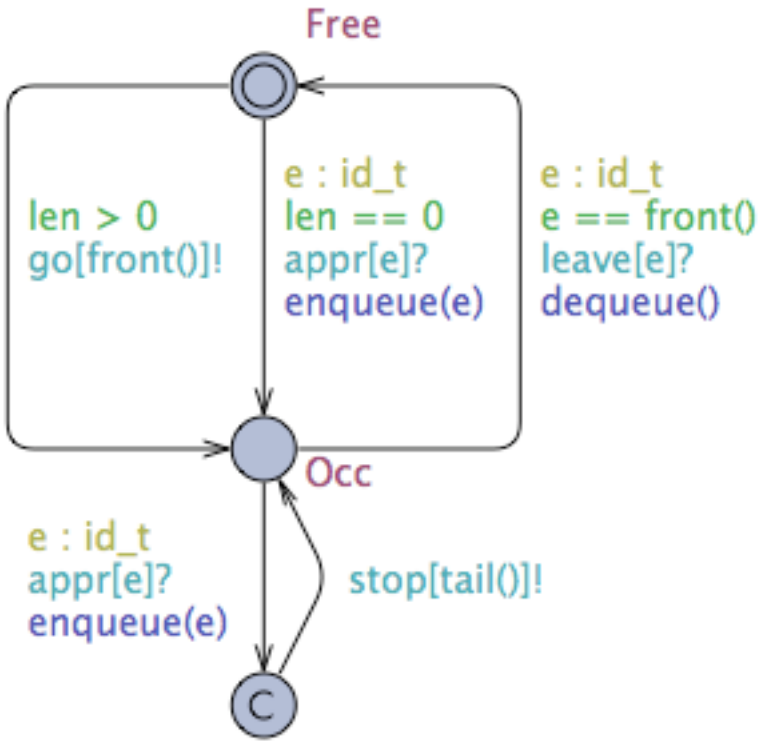
Between 3 and 5 units are required to cross the shared segment.

An Uppaal Model

One Train (x is a clock):



Controller:



appr[] is an array of channels

Scheduling Algorithms

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

Scheduling

A Few Examples

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 decreasing 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 some pair of tasks out-of-order in S' in order of decreasing deadline.
- 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:
  $\max(L_a, L_b)$

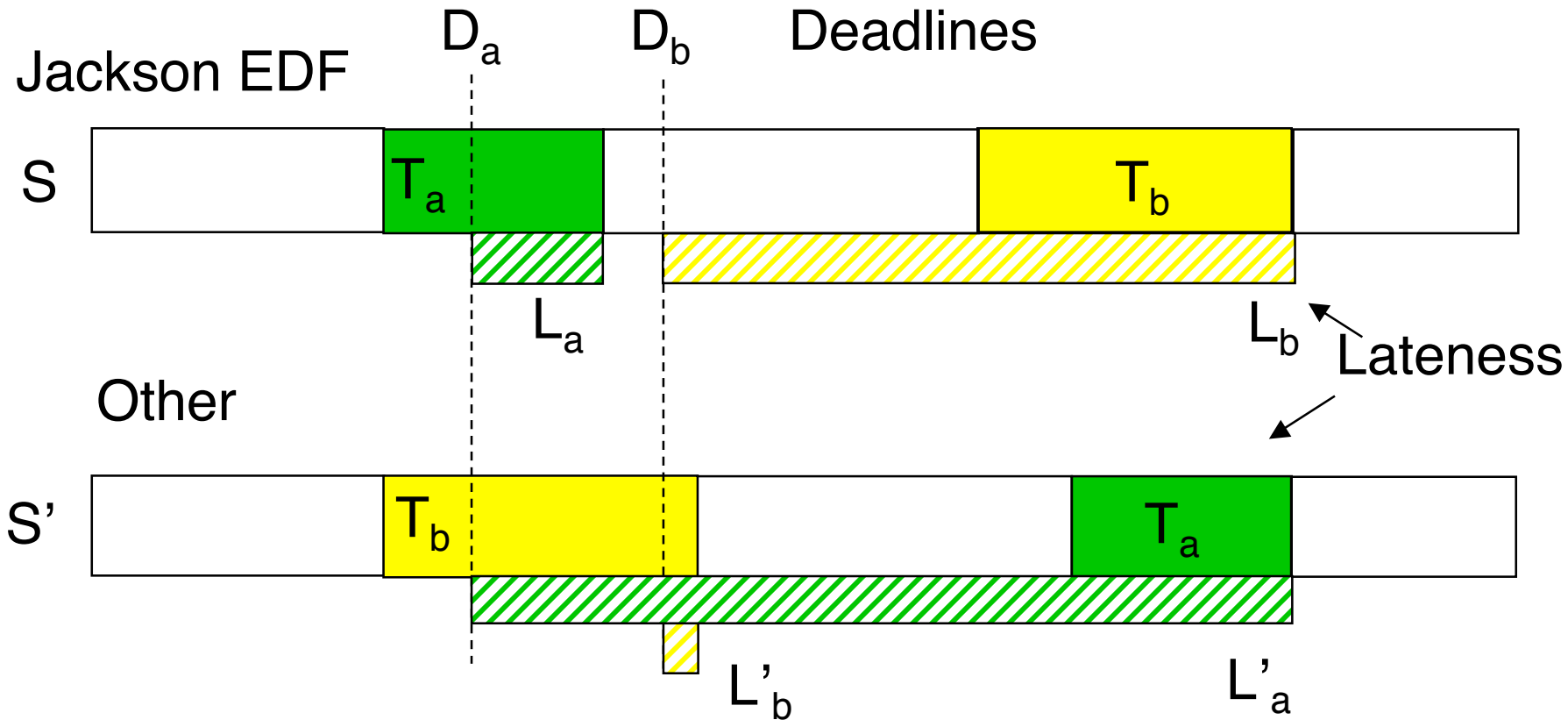
Proof that Jackson's Rule works

- 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 $L_a < L_b$:

$\max(L_a, L_b) = L_b$ — def'n of max, since $L_a < L_b$

$= F_b - D_b$ — def'n of L

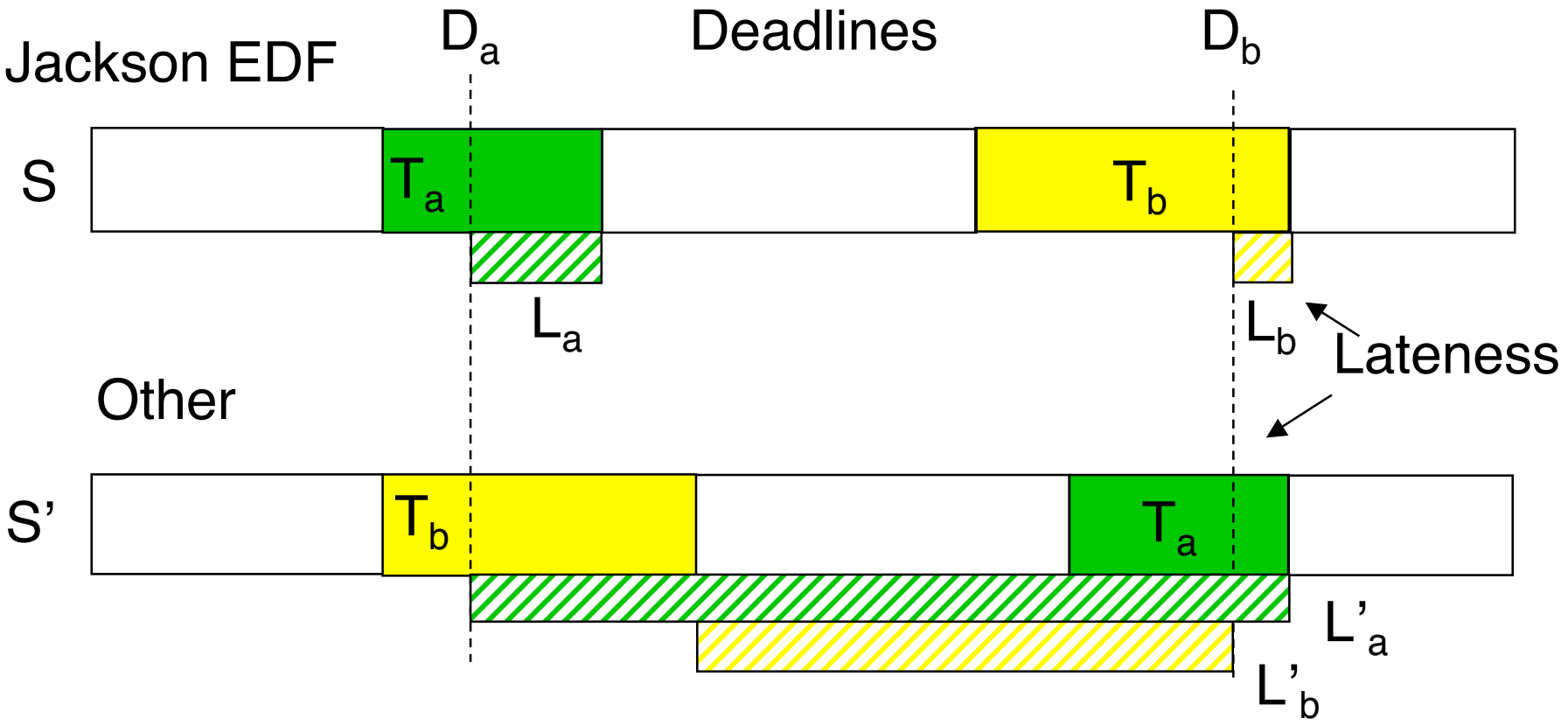
$= F'_a - D_b$ — a and b are flipped in S' vs S

$\leq F'_a - D_a$ — 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 $L_a \geq L_b$:

$\max(L_a, L_b) = L_a$	def'n of max
$= F_a - D_a$	def'n of L
$< F'_a - D_a$	assumption that 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

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

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.

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

Horn's Rule (1974)

- Example where **preemption is essential**:

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

- Horn's rule says: start T_1 at time 0, which would make T_2 wait to time 4, missing its deadline, since it ends at 6. The max lateness would be 1.

- The optimum way without preemption would be do nothing at time 0, start T_2 at time 1, then start T_1 at time 3. The maximum lateness would be 0.

Hard Problem

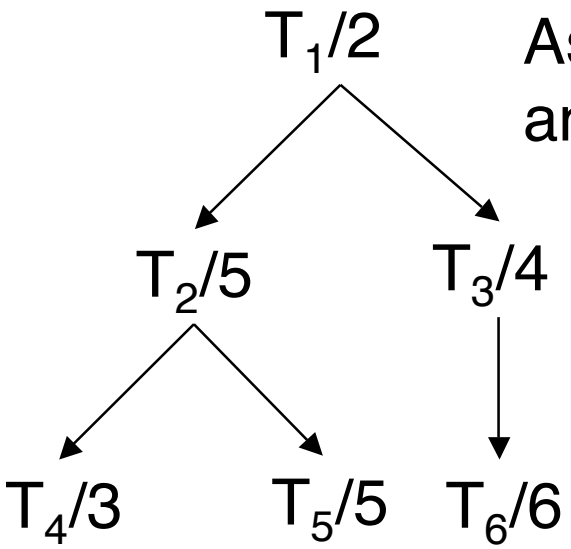
- Finding an **optimal non-preemptive** schedule when arrival times are arbitrary is NP-hard.

- An enumerative, branching, algorithm can be used.

Lawler's Algorithm (1973)

- Schedules a set of simultaneously arriving tasks on **one processor** subject to **precedence constraints**.
- Minimizes maximum lateness for 1 processor, among all **non-preemptive** algorithms.

Example: Lawler's Algorithm



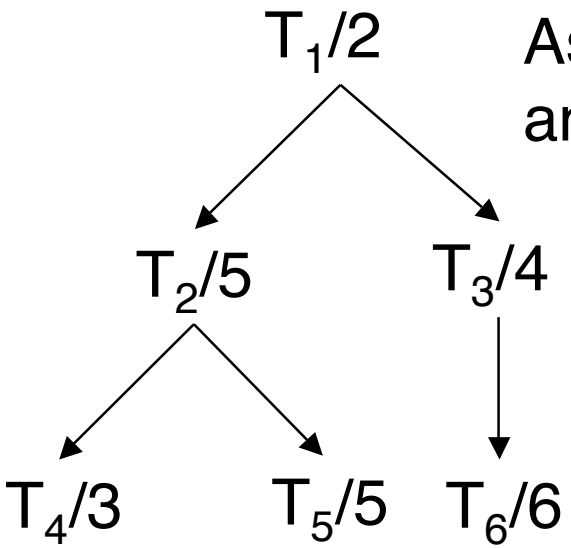
Assume **all computation times are 1** and deadlines are as shown.

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.

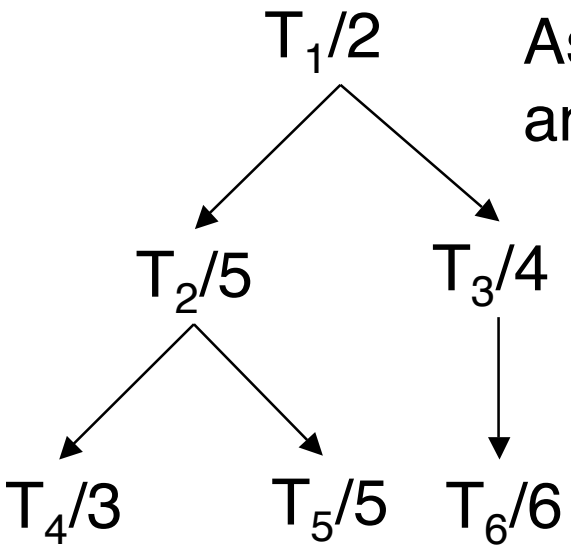
Example: Lawler's Algorithm



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

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.

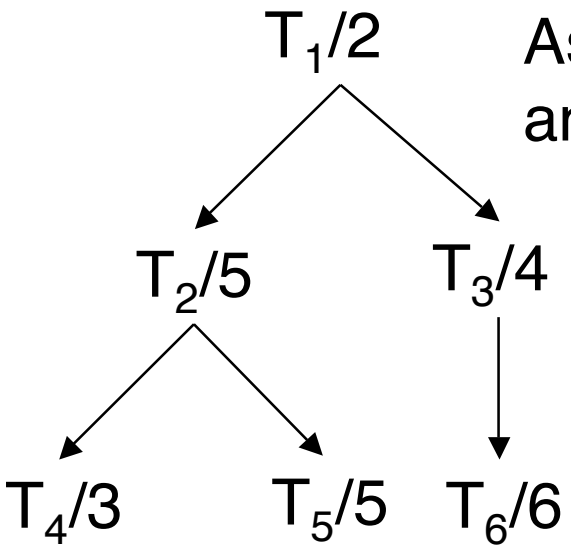
Stacking order (LDF, obeying precedence):
$T_6/6$, $T_5/5$, $T_3/4$, $T_4/3$, $T_2/5$, $T_1/2$

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.

Other Algorithms

- Other scheduling methods may be found in reference:
  - G.C. Buttazo
    Hard Real-Time Computing Systems
    Kluwer Academic Publishers, 1977.

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.
- 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.

“Release Time”

- “phase” is typically not used to describe real-time schedulers, although it has an obvious engineering significance.
- Instead, “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, ...

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.

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)

- 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.

Exercise

- Three periodic tasks:

Task	ComputeTime	Period
T_1	1	3
T_2	1	4
T_3	3	8

- See if you can construct a (1 processor) schedule that schedules these tasks periodically with no deadline missed.

- Remember that tasks can be preempted.

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 nearer deadline.
  - A task with a shorter period but more distant deadline.

Example

T_1

C = 20, P = 100

T_2

C = 30, P = 150

T_3

C = 80, P = 210

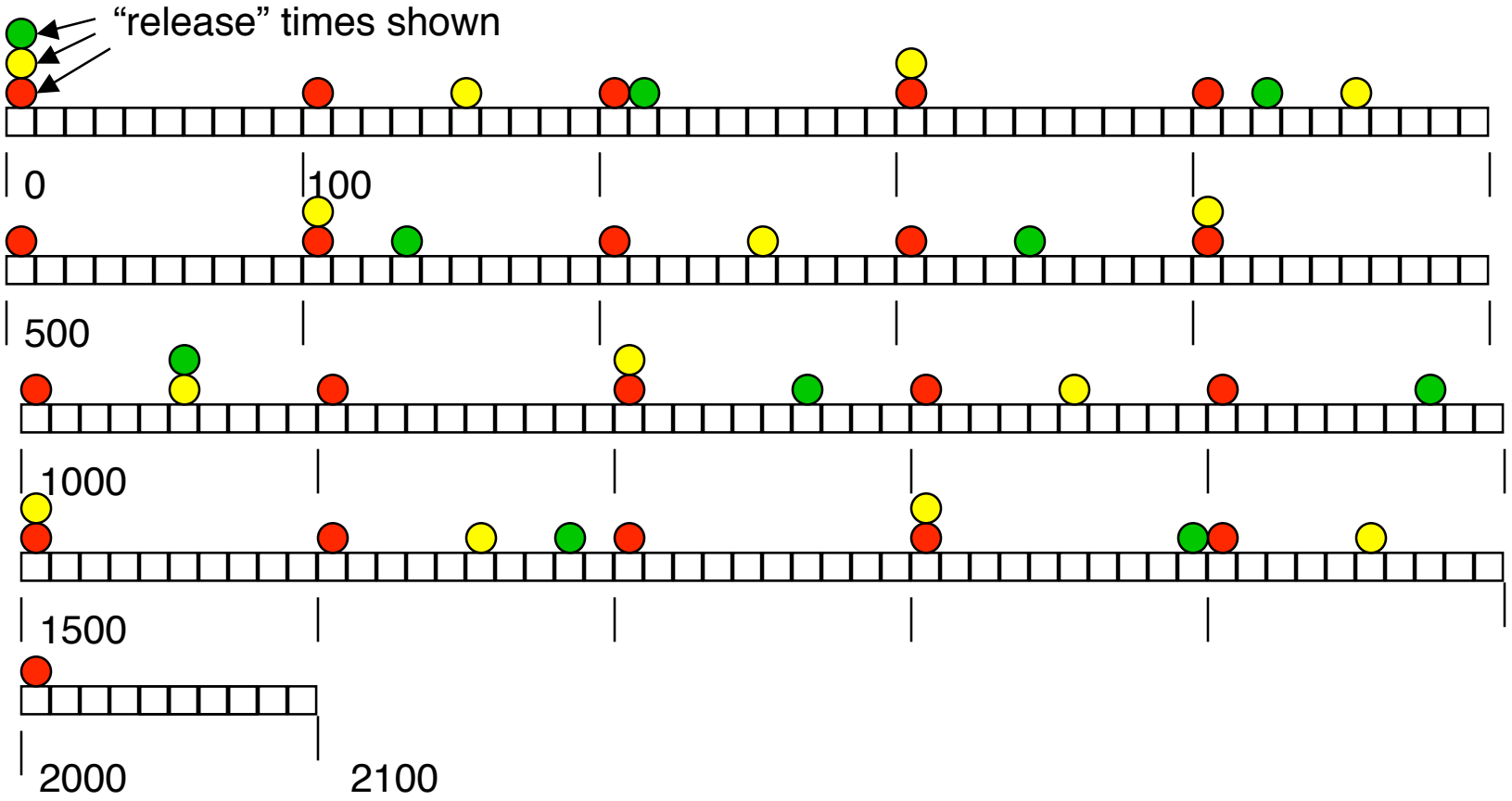
Constructing An EDF Schedule

(each box = 10 time units)

T_1 C = 20, P = 100

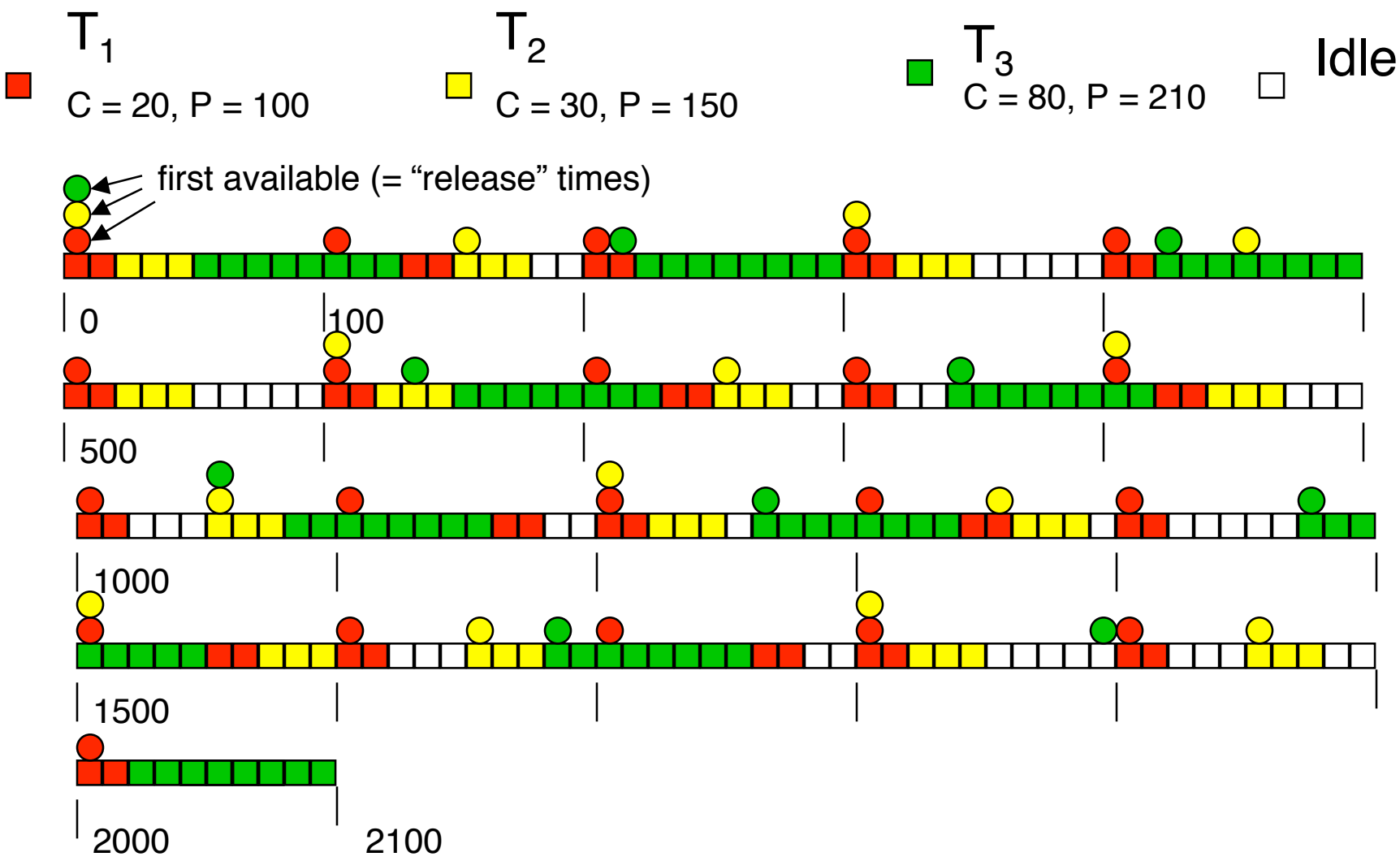
T_2 C = 30, P = 150

T_3 C = 80, P = 210



An EDF Schedule

(each box = 10 time units)



Utilization Bounds for Dynamic Priority Assignment

- For dynamic priority assignment, any system with a **total utilization** ≤ 1 can be scheduled.
- EDF is adequate for constructing such a schedule.

Fixed Priority Systems

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

Rate-Monotonic Assumption (RMA)

- In a fixed priority system, a task that is preemptable by another has ***lower priority***.
- RMA says that tasks with **shorter periods** should generally have **higher priority**.
- Motivation: lower priority tasks can be preempted, then resumed, to allow higher priority tasks to meet their deadlines, which occur with greater frequency.

Note on RMA Assumption

- RMA is a mathematical notion.
- It should **not** be inferred that every set of **user priorities** will agree with RMA.
- However, if priorities don't align this way, it may be worth revisiting the required periodicities.

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)

Exercise

- Revisit your schedule from the previous exercise. Is it rate-monotonic?

RMA Observation

- Case of n tasks, T_1, …, T_n, periods $P_1 \leq \ldots \leq P_n$, times C_1, …, C_n.
  - Consider an interval of time [0, t].
  - During the interval, each T_i must execute $\lfloor t/P_i \rfloor$ times.
  - The total time used for T_i during the interval will be $C_i * \lfloor t/P_i \rfloor$.
  - In order for each T_i to execute the appropriate number of times in the interval, we need $(\forall t' < t)$ $\text{sum}(C_i * \lfloor t/P_i \rfloor) \leq t'$.
  - If we can find a $t \leq \text{lcm}(P_1, \ldots, P_n)$ having this property, then all tasks can be scheduled.

Observation

- $(\forall t' < t)\ \text{sum}(C_i * \lfloor t/P_i \rfloor) \leq t'$
  iff
  $(\forall t' < t)$ t' is a time corresponding to the **end** of some task's period $\Rightarrow \text{sum}(C_i * \lfloor t/P_i \rfloor) \leq t'$.

- So it suffices to consider only times that end the period of some task.

Theorem (Liu & Layland, 1973)

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

  total periodic utilization $\leq n*(2^{1/n} - 1)$

- where *total periodic utilization* is defined as
  $\text{sum}(C_i/P_i, i = 1 \text{ to } n)$

- 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

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

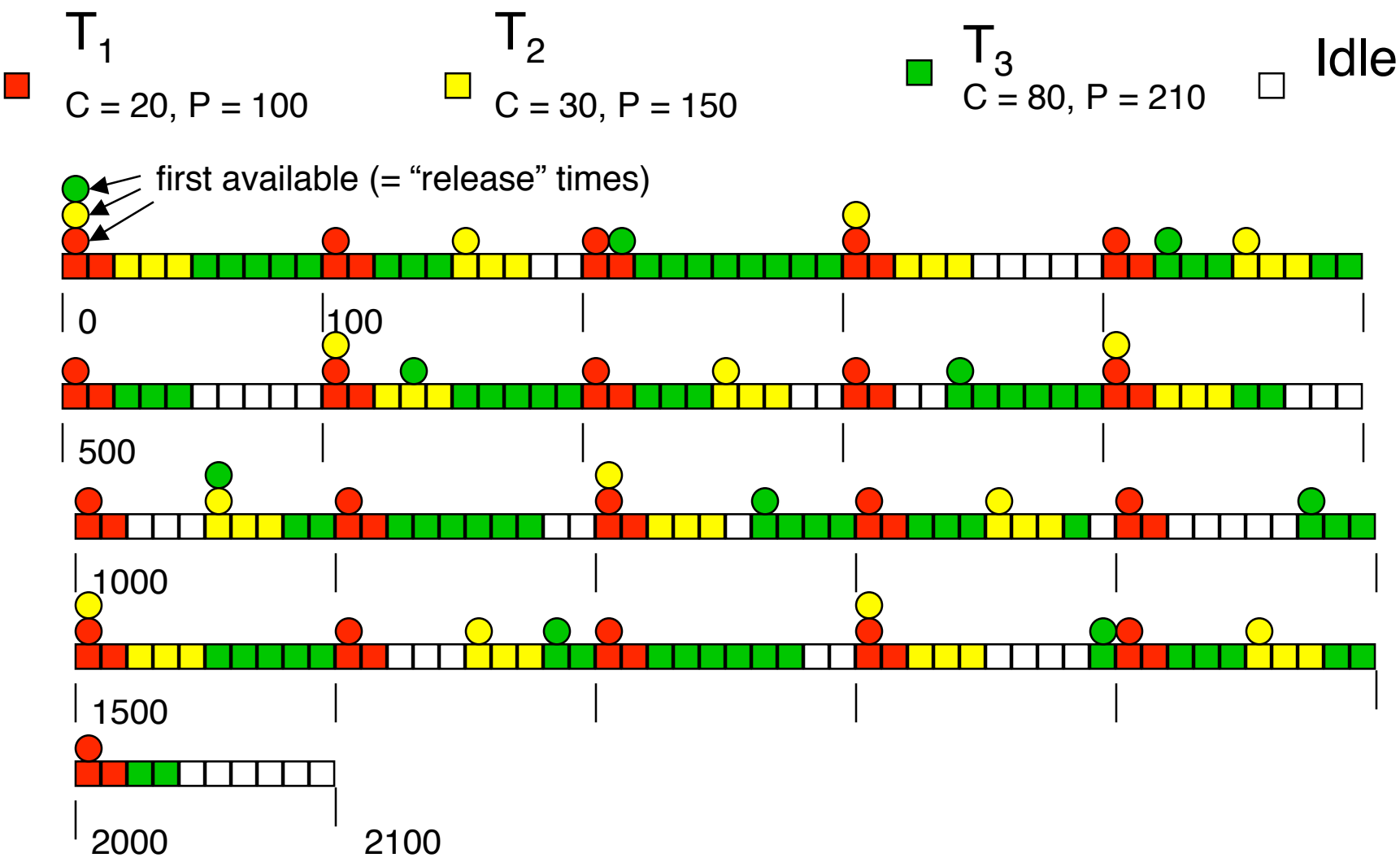
- Consider an interval of time [0, 2100], (2100 is the lcm of 100, 150, and 210) during which T_1 must complete 21 times, T_2 14 times, and T_3 10 times.

- 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)



Lehoczky, Sha, and Ding Theorem (1987)

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

Example

- Consider adding a fourth task T_4:

  $P_1 = 100$, $P_2 = 150$, $P_3 = 210$, $P_4 = 400$,

  $C_1 = 20$, $C_2 = 30$, $C_3 = 80$, $C_4 = 100$.

- Total utilization: 20/100 + 30/150 + 80/210 + 100/400 = 1.03095 > 1, so this set is not realizable.

2nd Theorem of Liu & Layland

- If a set of tasks is schedulable under **any fixed priority** scheme, then it is schedulable using rate-monotonic scheduling.
- In other words, rate-monotonic is **optimal** among *fixed* priority schemes.

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.
- DM has been shown *optimal* for this more general case (Leung and Whitehead, 1982).

If relative deadlines are ≤ period and not constant

- EDF 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.
  - Need only check at release times between 0 and lcm(periods).

Summary of Rule Applicability

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

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.
- The “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 it's 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, since this could leave the system in an **inconsistent** state, or
  - 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.

Possible Resolution of Blocking

- **Priority Inheritance Protocol**:
  - More precisely: 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

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

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

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

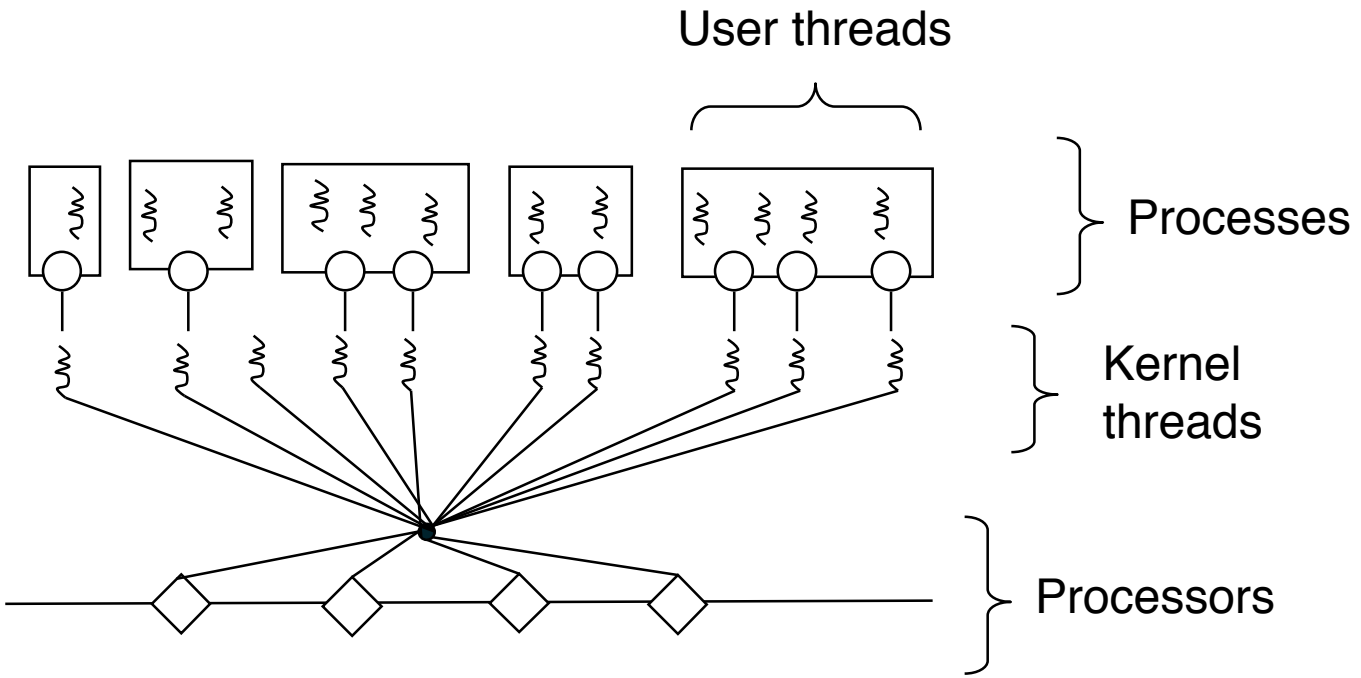
- If the computation time (without blocking) within critical sections is known, then the blocking time due to priority inheritance can be computed.

- 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).
- 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

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

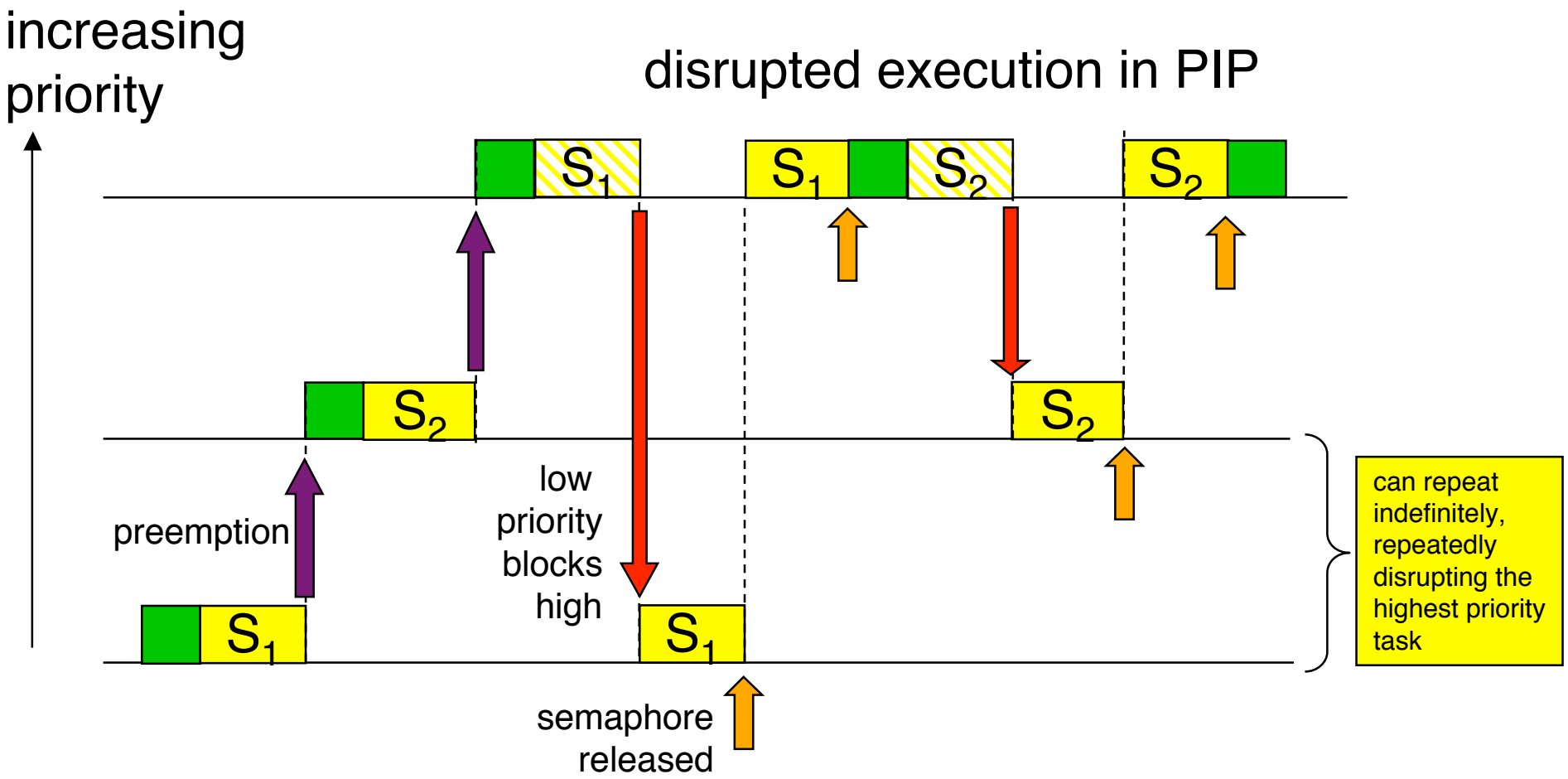
- The PIP does not prevent a high priority task from being blocked **multiple times** by lower priority ones, extending the blocking time **indefinitely** (“livelock”).
- No deadlock prevention measures are included in the protocol, but PIP does not particularly help with this.

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.

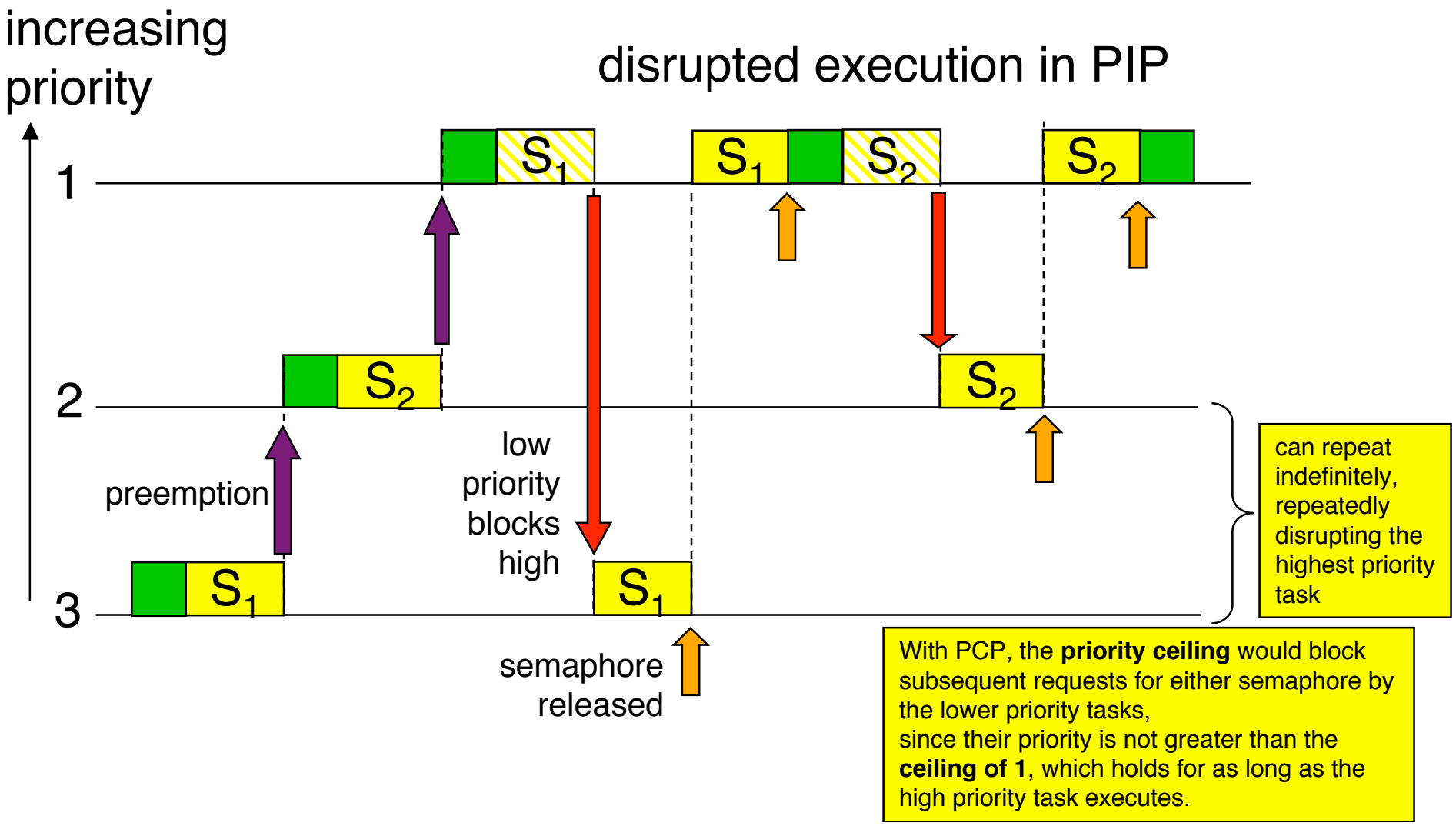
Priority Ceiling Protocol (PCP)

- The PCP **prevents a deficiency** of the PIP:

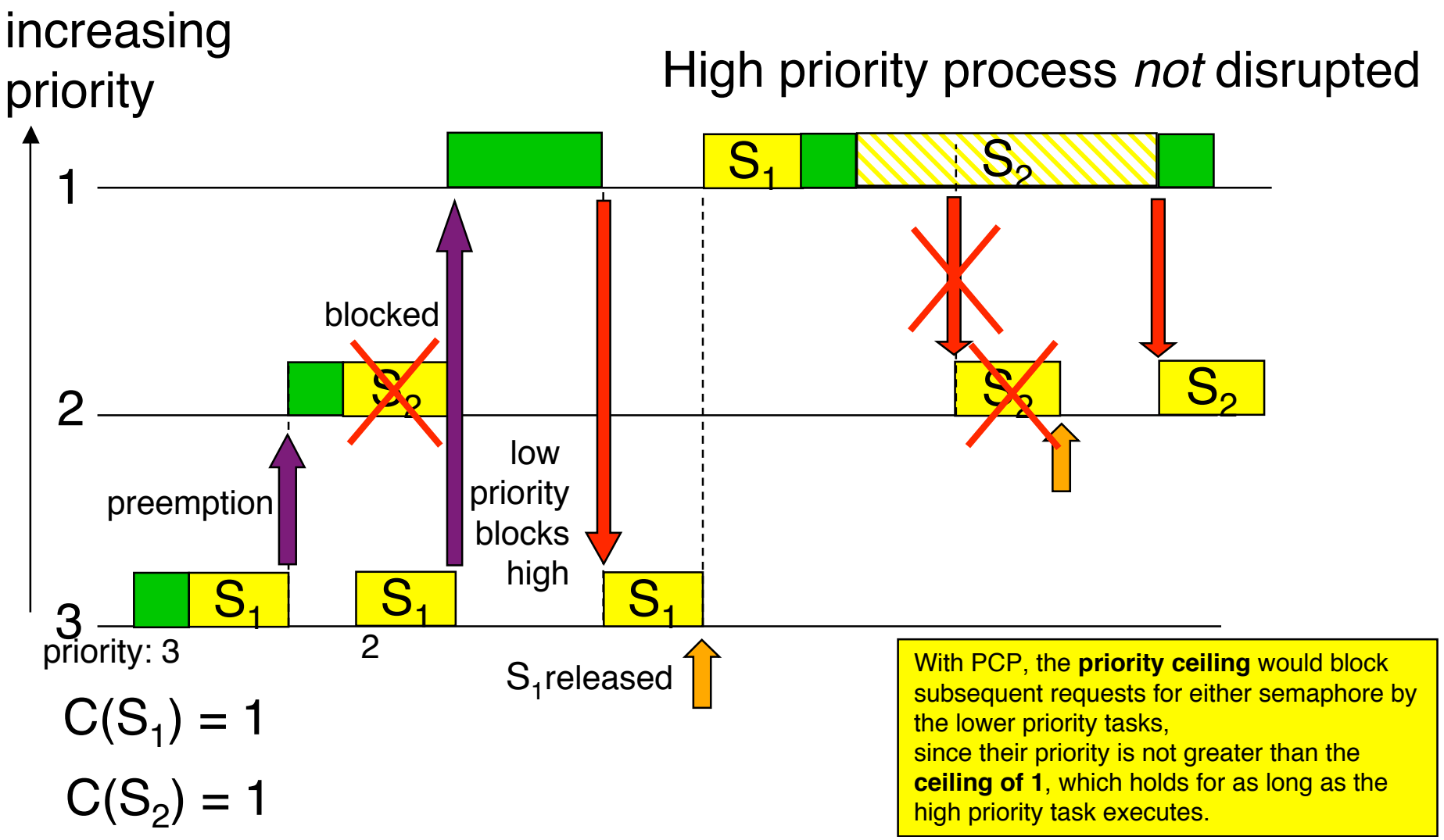
  Once a task gains entry to a critical section, it could *subsequently* be blocked by a *lower* priority task.

  This effect (“chained blocking”) cannot happen with PCP.

Chained Blocking in PIP



Chained Blocking Avoided in PCP



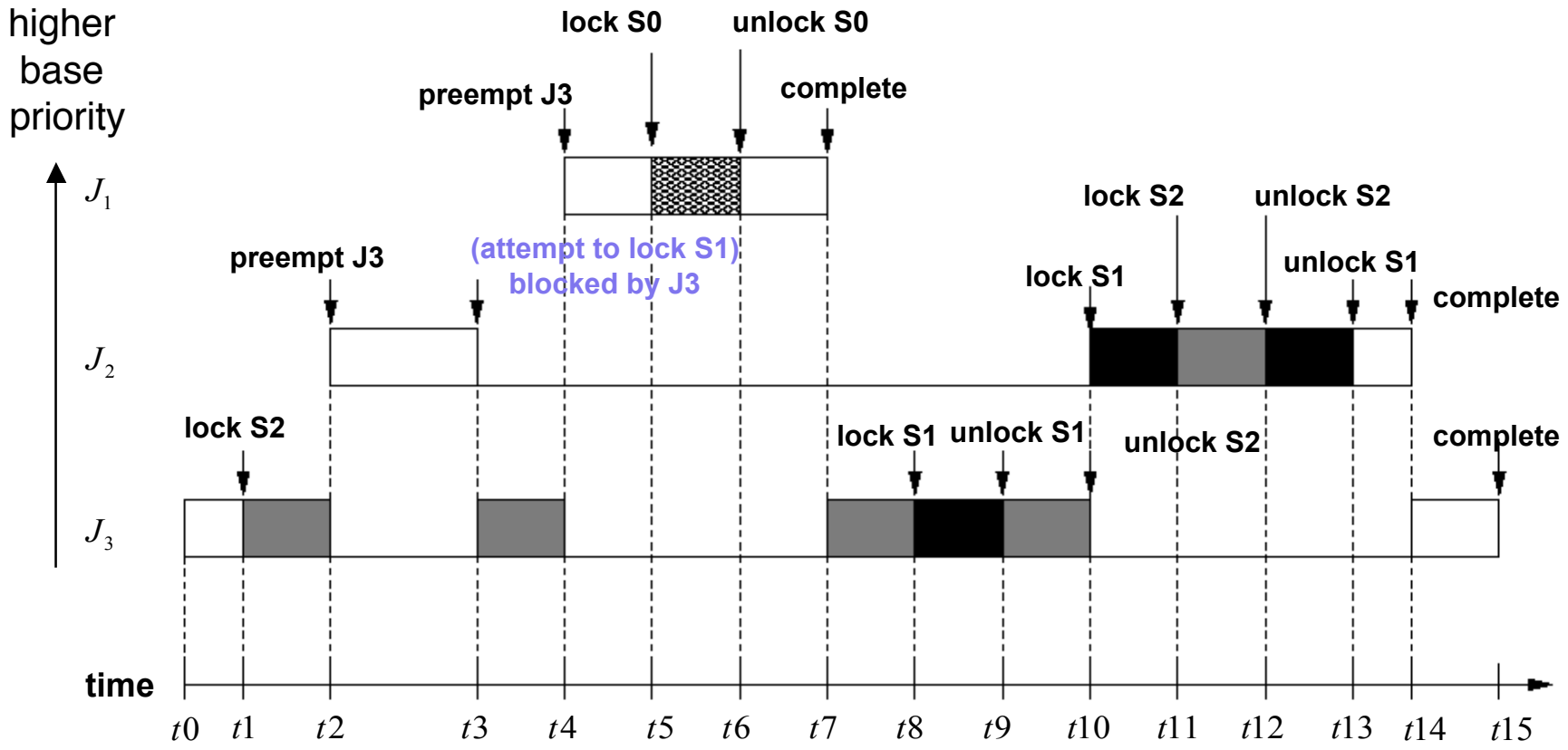
Comparison: PIP vs. PCP

- 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 automatically **prevents deadlocks**, since it induces an **ordering** on the way that resources can be requested.

More PCP Examples : Deadlock Prevention

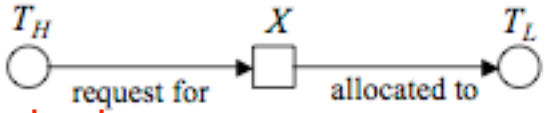
$J_1 = \{..., P(S_0) ..., V(S_0),...\}$
$J_2 = \{..., P(S_1) ..., P(S_2),..., V(S_2) ..., V(S_1),...\}$
$J_3 = \{..., P(S_2) ..., P(S_1),..., V(S_1) ..., V(S_2),...\}$

nested crossing semaphores invite deadlock



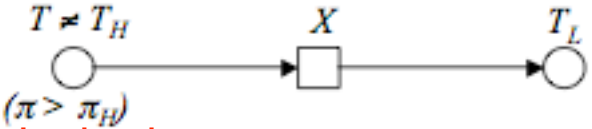
Summary of Basic Blocking Types (not including transitivity)

- 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.



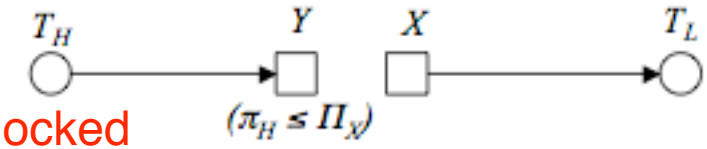
blocked

  - 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:

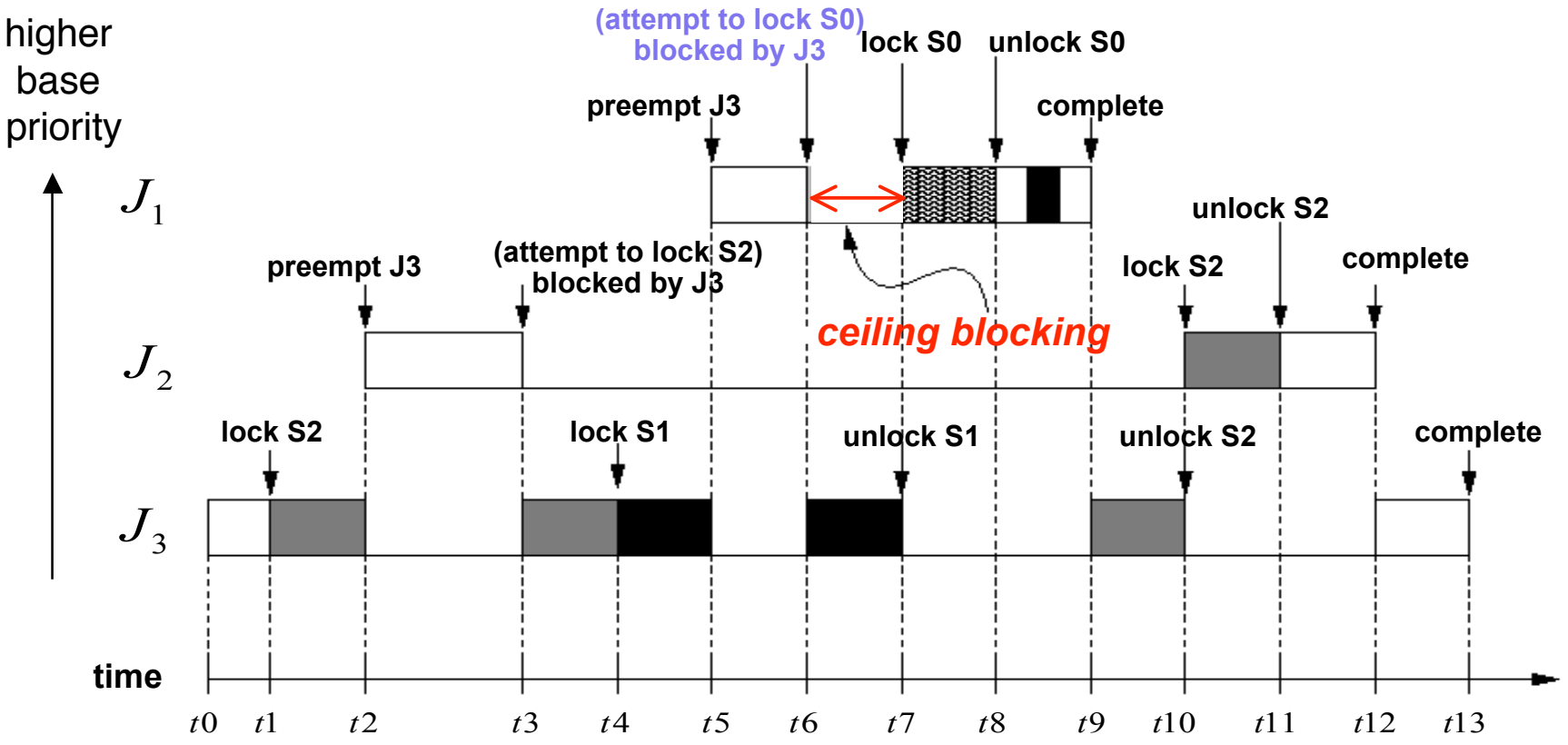


blocked

Ceiling Blocking

$J_1 = \{..., P(S_0), ..., V(S_0), ..., P(S_1), ..., V(S_1), ...\}$
$J_2 = \{..., P(S_2), ..., V(S_2), ...\}$
$J_3 = \{..., P(S_2), ..., P(S_1), ..., V(S_1), ..., V(S_2), ...\}$

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 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.

SRP Properties

- 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.

- **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:

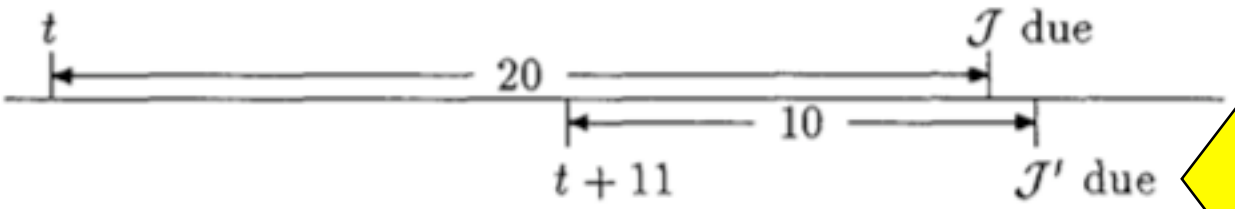
i. $t < t'$ (so $\mathcal{J}$ can get started);

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

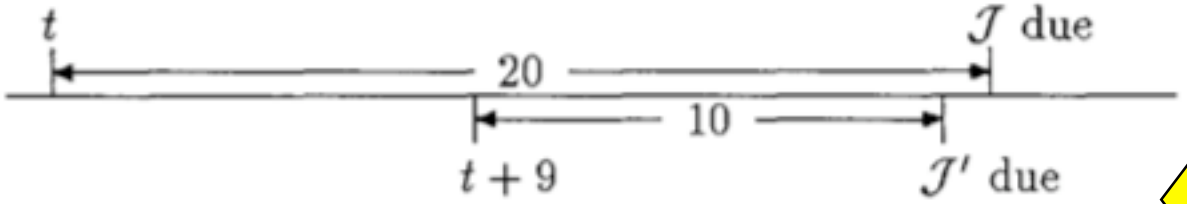
With EDF scheduling, $p(J) < p(J')$ iff $t' + D' < t + D$, so it follows that $D' < D$. It is therefore consistent to define preemption levels so that $\pi(J) < \pi(J')$ iff $D(J) > D(J')$.

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.

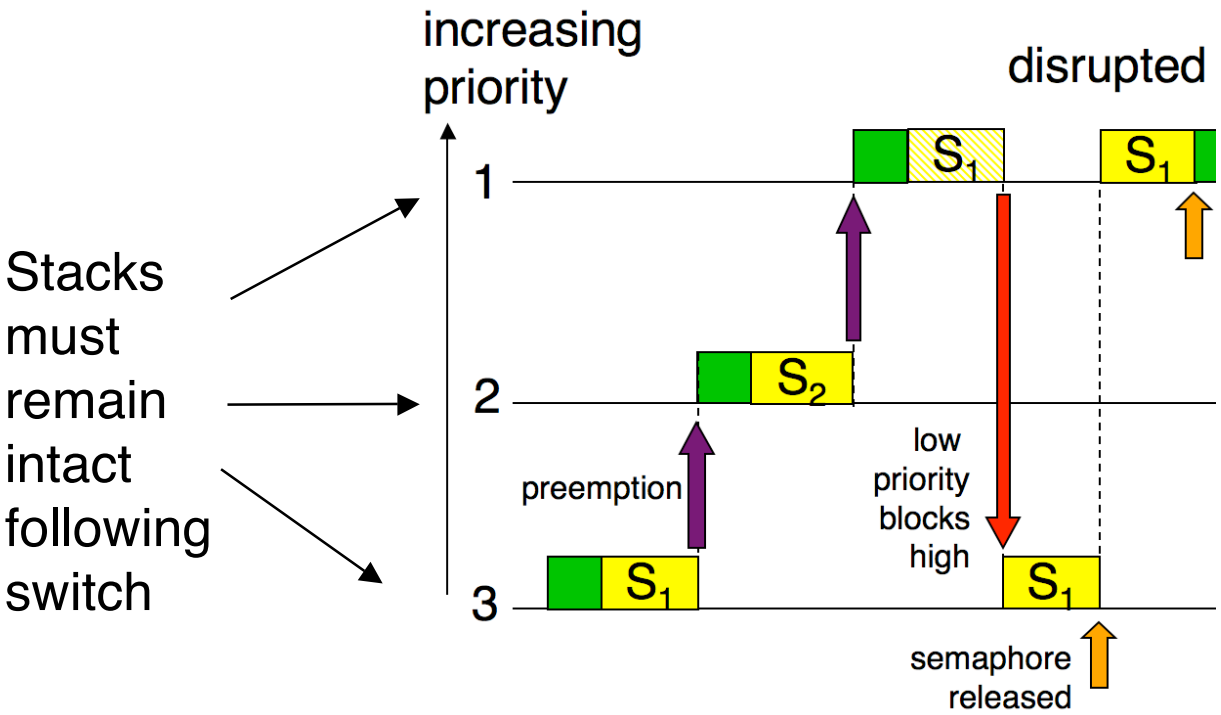
(*with no waiting*)

SRP Impact

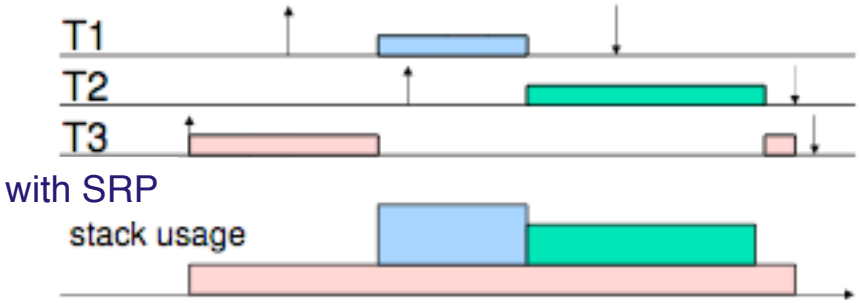
- The Stack-Based Priority-Ceiling Protocol is **deadlock-free**:
  - When a job begins to execute, all the resources it will ever need are free.
  - Otherwise, $\Pi(t)$ would be higher or equal to the priority of the job.
  - Whenever a job is preempted, all the resources needed by the preempting job 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:

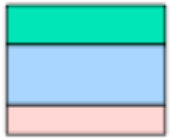


With SRP, Stack is Sharable



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

without SRP



(more space)

Ceiling-Priority Protocol (SRP *without* Stack-Sharing)

- Stack-Based Protocol does not allow for self-suspension
  - Stack is shared resource
- Re-formulation for multiple stacks (no stack-sharing) straightforward:

Ceiling-Priority Protocol

Scheduling Rules:

1. Every job executes at its assigned priority when it does not hold resources.
2. Jobs of the same priority are scheduled on FIFO basis.
3. Priority of jobs holding resources is the highest of the priority ceilings of all resources held by the job.

Allocation Rule:

- Whenever a job requests a resource, it is allocated the resource.