SYSC 5701

Operating System Methods for Real-Time Applications

Access Control

Winter 2014

Recall

Resource-Sharing Dependencies

- **A job cannot proceed (is blocked) because of resource-sharing synchronization**
- **Resource-sharing requires mutually exclusive access to the resource**
- **Can cause priority inversions**

- **We looked into the priority ceiling protocol to deal with the priority inversions**
  - **See slides: PCPW14**

Recall

Properties of Basic Priority Ceiling Protocol

- **no deadlock !**
- **job blocks in at most one critical section**
  - **blocking is bounded (at most one)**
  - **no chain blocking → shorter blocking bound than Priority Inheritance Protocol**
- **once acquire first resource, all resources needed will be available when requested**

What about RM Theory?

- Priority Ceiling Protocol bounds delay to the single largest delay of a lower priority job !
  - for each job, include max. delay
- recall RM utilization test:

$$\sum u_i \leq n (2^{1/n} - 1)$$

- in following, assume that if $i < j$
  - then priority τ_i > priority τ_j

Consider Utilization Test for Each Task

consider τ_1 case:

- **B_1 is the worst case time τ_1 spent blocked by lower priority tasks**

$$\frac{C_1}{T_1} + \frac{B_1}{T_1} \leq u(1)$$

Consider Each Task (con't)

- consider τ_2 case:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{B_2}{T_2} \leq u(2)$$

- blocking of C_1 is included in C_2 & B_2 !
  - ensured by protocol!

    (think this through on next slide!)
- do not need to consider an "additional" B

Thinking Through …

- consider τ_1 :

$$\frac{C_1}{T_1} + \frac{B_1}{T_1} \leq u(1)$$

B_1 represents max. blocking by lower priority tasks (lower than τ_1) using resources having a priority ceiling greater than (or equal to) the priority of τ_1

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{B_2}{T_2} \leq u(2)$$

If τ_2 caused B_1 then B_1 is included in C_2 and B_2 is due to blocking by a task with lower priority than τ_2

If τ_2 did not cause B_1 then B_1 = B_2 (same blocking by same lower priority task)

n^{th} case

τ_n cannot be blocked by lower priority tasks since it is the lowest priority task

- **n^{th} task case:**

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} + \cancel{\frac{B_n}{T_n}}^{\,0} \leq u(n)$$

- **if constraints are satisfied for each case, then task set is schedulable, i.e.:**

$$\forall_i \ 1 \leq i \leq n \quad \sum u_i + \frac{B_i}{T_i} \leq i\,(2^{1/i} - 1)$$

represents worst case utilization of τ_i blockage

Example

τ_1 : $T_1 = 30$ $C_1 = 10$ $B_1 = 10$

τ_2 : $T_2 = 80$ $C_2 = 15$ $B_2 = 20$

τ_3 : $T_3 = 100$ $C_3 = 25$ $B_3 = 0$

- want to consider

$$\forall_i \ 1 \le i \le n$$

Case: i = 1

- $\sum u_i + \dfrac{B_i}{T_i} \leq u(1)$

$$\frac{10}{30} + \frac{10}{30} \leq 1$$

- $0.66 \leq 1$ ☺

Case: i = 2

- $\sum u_i + \frac{B_i}{T_i} \le u(2)$

  $\frac{10}{30} + \frac{15}{80} + \frac{20}{80} \le 0.82$

- $0.771 \le 0.828$ ☺

Case: i = 3

- $\sum u_i + \frac{B_i}{T_i} \leq u(3)$

$$\frac{10}{30} + \frac{15}{80} + \frac{25}{100} + \frac{0}{100} \leq 0.779$$

$$0.771 \leq 0.779 \quad ☺$$

Simpler Test?

- can use tighter bound to avoid computing n equations:

$$\sum u_i + \max\left\{ \frac{B_1}{T_1}, \frac{B_2}{T_2}, \ldots, \frac{B_{n-1}}{T_{n-1}} \right\} \le n\,(2^{1/n} - 1)$$

- proof: $\forall_i \; 1 \le i \le n$:
  - $n\,(2^{1/n} - 1) \le i\,(2^{1/i} - 1)$
  - n^{th} case gives tightest bound!

$$\max\left\{ \frac{B_1}{T_1}, \frac{B_2}{T_2}, \ldots, \frac{B_{n-1}}{T_{n-1}} \right\} \ge \frac{B_i}{T_i}$$

  - max is worst case! → conservative

Hmmm …

- tighter bound may fail when individual equations succeed
- previous example:

$$\frac{10}{30} + \frac{15}{80} + \frac{25}{100} + \max\left(\frac{10}{30}, \frac{20}{80}\right) \leq 0.779$$

- $0.804 \leq 0.779$ ☹

Demand Analysis Revisited
(for cases where utilization test fails)

- can formalize utilization at scheduling points:
  - denote set of scheduling points for interval [0, T] as: S(T)
- set of scheduling points for tasks with periods less than <u>or equal to</u> T_i:

$$S_i(T_i) = \{ \; k \cdot T_j \;|\; j = 0..i \; ; \; k = 0 .. \lfloor T_i / T_j \rfloor \}$$

remember: $T_j \leq T_i$ for $j = 0 .. i$!!

Visualize on blackboard

Scheduling Point (con't)

- **maximum period in set of i + 1 tasks:** T_i
- **set of scheduling points for the set of tasks:** $S_i(T_i)$
- **utilization by tasks at scheduling point** $T_s \in S_i(T_i)$

$$= \frac{\sum_{j=0}^{i} \boxed{\left\lceil \frac{T_s}{T_j} \right\rceil} C_j}{T_s}$$

number of task j jobs released during interval T_s

Carleton UNIVERSITY

Min U at Scheduling Point

- for **i** tasks, minimum utilization over set of scheduling points $S_i(T_i)$: U_{min}

$$= \min \left(\frac{\sum_{j=0}^{i} \left\lceil \frac{T_s}{T_j} \right\rceil C_i}{T_s} \right)_{T_s \in S_{\mathbf{i}}(T_i)}$$

- Task i guaranteed schedulable when: $U_{min} \leq 1$

Carleton UNIVERSITY

Scheduling Point + Blocking

- generalize Scheduling Point solution to include blocking
- consider set of i tasks: T_i is largest period
- utilization for task i must include:

  single execution of job i

  \+ all preemption by higher priority jobs

  \+ worst case blocking by a lower job

U at Scheduling Point

preemption by i – 1 higher priority tasks

execution of job i

worst case blocking by a lower priority job

$$\frac{\sum_{j=0}^{i-1} \left\lceil \frac{T_s}{T_j} \right\rceil C_j}{T_s} + \frac{C_i}{T_s} + \frac{B_i}{T_s}$$

Carleton UNIVERSITY

Schedulability Test

For all i : 0 ≤ i ≤ n , exists $T_s \in S_i(T_i)$

$$\left(\frac{\sum_{j=0}^{i-1} \left\lceil \frac{T_s}{T_j} \right\rceil C_j}{T_s} + \frac{C_i}{T_s} + \frac{B_i}{T_s} \right) \leq \mathbf{1}$$

How to compute B_i ?

1. identify β_i

- set of resources accessed by lower priority tasks (lower than τ_i) and having a priority ceiling greater than (or equal to) the priority of τ_i

  → the resource accesses that might block τ_i !

2. create β_i^*

- subset of β_i created by merging nested critical sections (inner section subsumed by outer section)

3. select B_i = member of β_i^* with longest duration

B_i Selection Example

τ_1 : L(S_1) 1 sec U(S_1)

determines ceiling of S_1

determines ceiling of S_2

τ_2 : L(S_2) 2 sec U(S_2)

τ_3 : L(S_1) 3 sec L(S_2) U(S_2) 1 sec U(S_1)

Ceiling (S_1) = priority (τ_1) **Ceiling(S_2) = priority (τ_2)**

Determining B_i's

- **Ceiling (S_1) = 1   Ceiling(S_2) = 2**
- **for τ_1:**

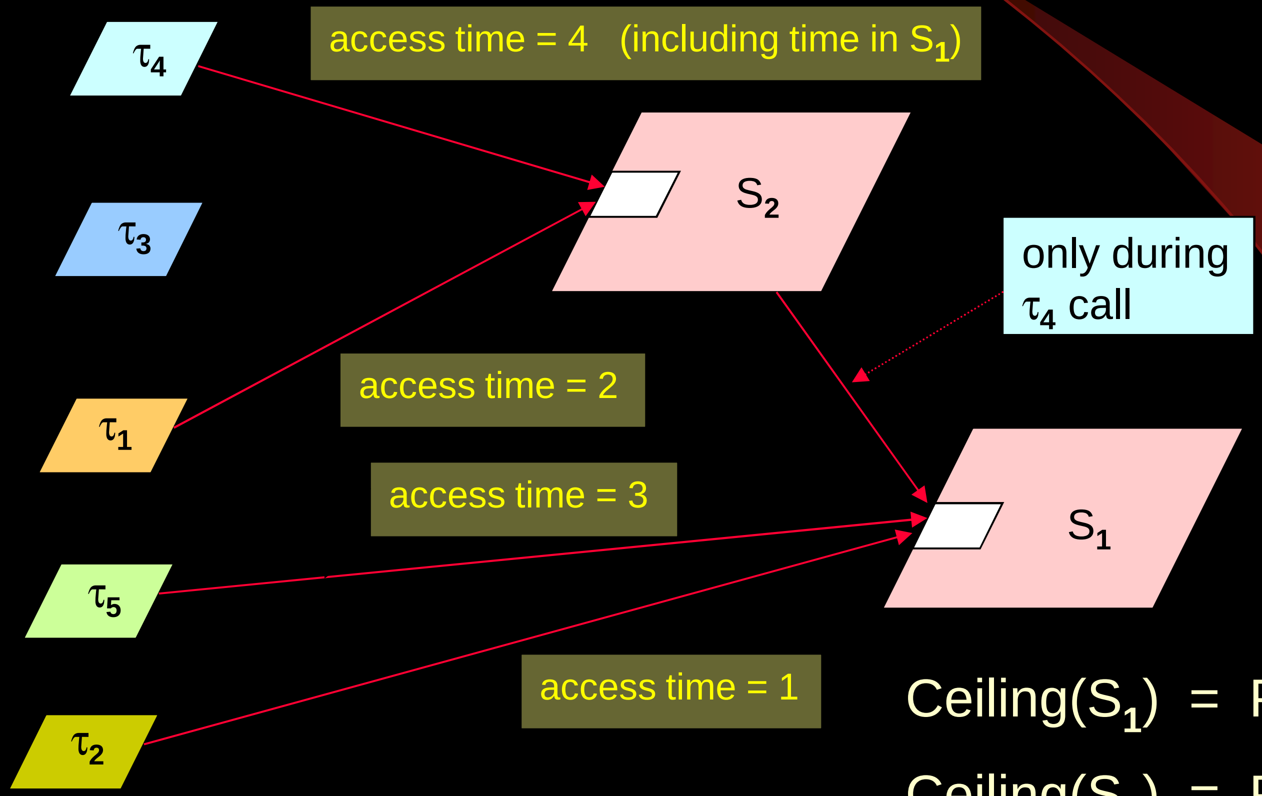
  $$\beta_1^* = \{ \tau_3\overbrace{[L(S_1)\ \ U(S_1)]}^{\text{3 sec}} \}$$

  $$\therefore B_1 = 3 \text{ sec}$$

- **for τ_2:** $$\beta_2^* = \{ \tau_3\overbrace{[L(S_1)\ \ U(S_1)]}^{\text{3 sec}} \}$$

  $$\therefore B_2 = 3 \text{ sec}$$

Recall Client / Server Example



$\text{Ceiling}(S_1) = \text{Priority}(\tau_2)$

$\text{Ceiling}(S_2) = \text{Priority}(\tau_1)$

Determining B_i's

Consider β_i :

- **β_5 : τ_5 is the lowest priority task $\therefore$ $B_5 = 0$**
- **β_4 : τ_4 blocked by τ_5 access to S_1 $\therefore$ $B_4 = 3$**
- **β_3 : τ_3 blocked by τ_5 access to S_1 and by τ_4 access to S_2 $\therefore$ $B_3 = 4$**
  - **N.B. τ_3 blocked indirectly even though it does not access servers!**

Determining B_i's (con't)

- **β_2: τ_2 blocked by τ_5 access to S_1**
  and by τ_4 access to S_2 $\therefore$ $B_2 = 4$

- **β_1: τ_1 blocked by τ_4 access to S_2 $\therefore$ $B_1 = 4$**
  - **N.B. τ_1 not blocked for access to S_1 since ceiling of S_1 = Priority(τ_2)**

Resource Access
Purpose vs. Technical Detail

- so far, only case of access dependency considered has been nested access
- may have alternate dependencies
  - e.g. read *x*, modify value, then write *x*
  - involves multiple accesses (read, write)
  - really want to lock *x* for duration of
    read → modify → write

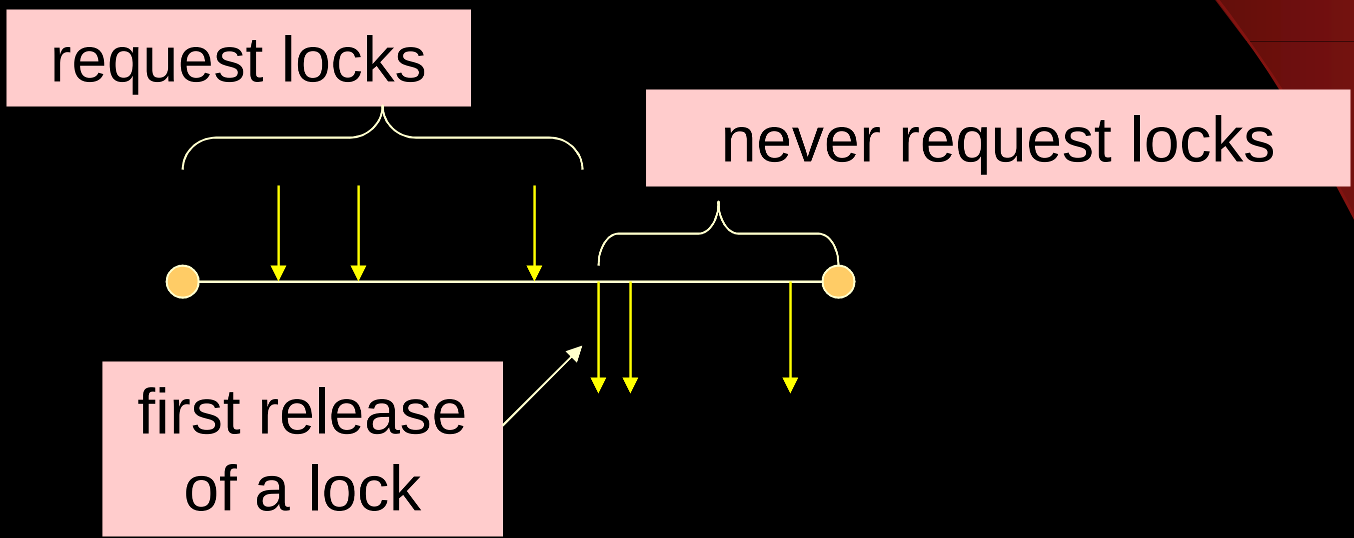
Serializability

- for operations involving multiple accesses to a resource
- concurrent operations by job 1 and job 2 have potential for interleaving of accesses
- want net behaviour to be serialization of accesses → result is same as if job1 followed by job2, or vice versa

Carleton UNIVERSITY

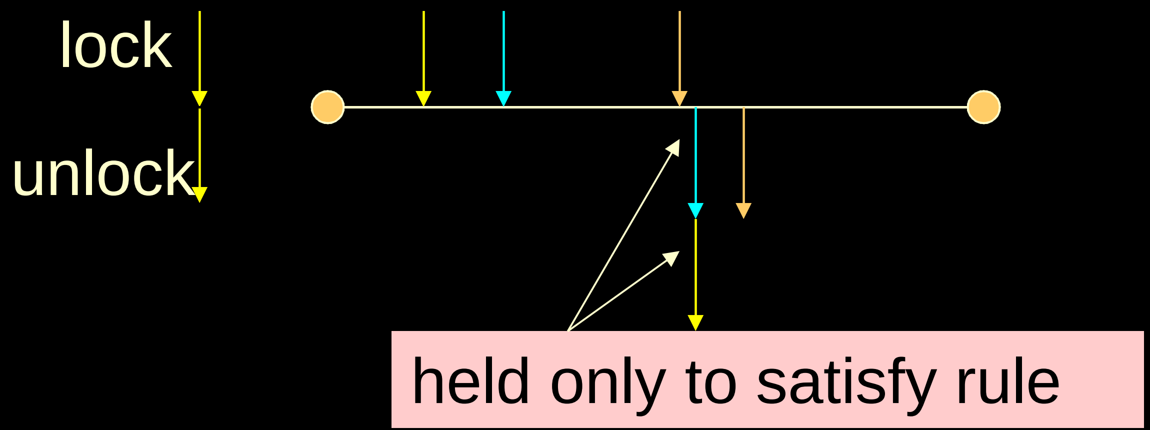
2 Phase Locking (2PL)

- **ensures serializability**
- **Rule: a job never requests any lock after its first release of any lock**



Augment with 2PL

- augment Priority Ceiling Protocol with 2PL
  - **Advantages**: serializable, no deadlock, no chain blocking
  - **Penalty**: may have prolonged unnecessary blocking



Alternative: Convex-Ceiling Protocol

- **reduces duration of blocking (over 2PL)**
- **for each job, maintain remainder priority ceiling function:**
- **RP(J, t) = highest priority ceiling of all resources J requires *after* time t**
- **when job released: RP(J, 0) = highest priority ceiling of all resources J requires**

Remainder Priority Ceiling Function (con't)

- when no resources required, RP(J, t) = Ω
- when job is finished with resource, sends notification to scheduler
  - if ceiling for remaining required resources is lower, scheduler adjusts RP down to value of remainder ceiling
- RP is a strictly decreasing function !

Priority Ceiling Function

- also maintain **priority ceiling function** for job: $\Pi(J , t)$
- when job released, $\Pi(J, 0) = \Omega$
- each time a resource locked by job, if ceiling of resource is higher than Π, then adjust Π to ceiling of resource
- Π increases until it meets initial RP value

Priority Ceiling Function (con't)

- when resource released, adjust RP first
- if new value of RP < Π , then adjust Π down to RP
- RP and Π decrease in unison

Integrate into a Protocol?

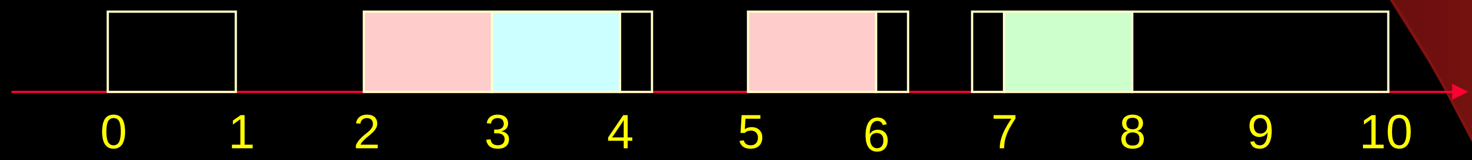
- **current priority ceiling:** $\hat{\Pi}(t)$

  = maximum priority ceiling function Π(J , t)

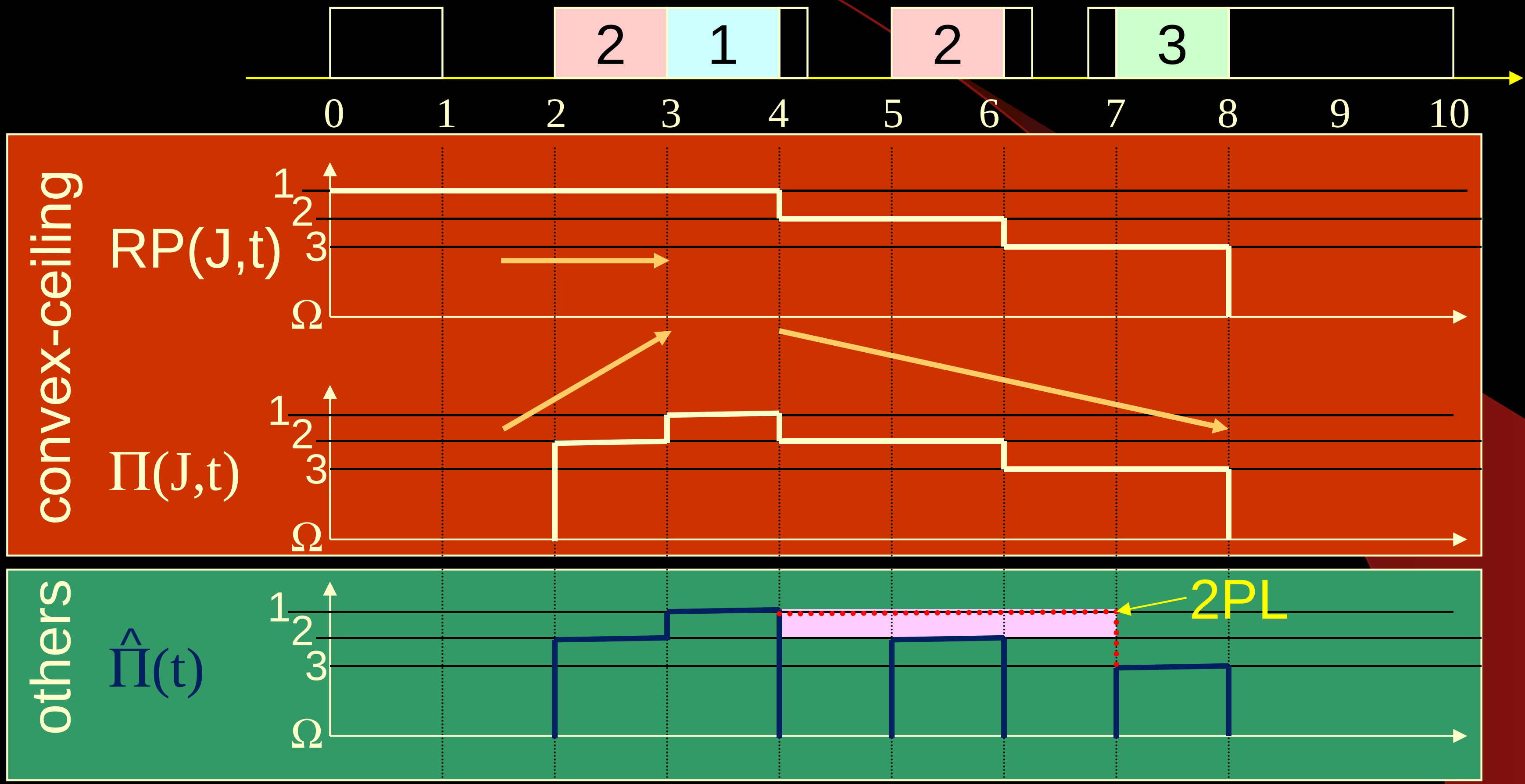
  of all jobs at time t

Example

- suppose 3 resources X, Y, Z
- priority ceilings: X: 1 Y: 2 Z: 3
- job requires Y, then X, then Y then Z



Comparison



2
1
2
3
0
1
2
3
4
5
6
7
8
9
10
convex-ceiling
RP(J,t)
Π(J,t)
1
2
3
Ω
others
$\hat{\Pi}(t)$
2PL

Going Further …

Daniel I. Katcher, Hiroshi Arakawa, and Jay K. Strosnider,

Engineering and Analysis of Fixed Priority Schedulers

IEEE Transactions on Software Engineering, Vol. 19, No. 9, September 1993, pp.920 - 934

Katcher et al.

- research "towards bridging the gap" between real-time scheduling theory and realistic implementations
- analyzes event-driven and timer-driven scheduling
- scheduling costs (overheads)
  - notes dependence on h/w platform!

Katcher et al.

- Assumes all jobs released by interrupts
- looks more closely at interrupts from h/w and o/s perspectives
  - register use
  - independently run ISR vs. ISR “job”
    - schedule, context switch
  - ISR executing kernel code → interaction with scheduler

Katcher et al.

- 6 Definitions of o/s Execution Activities:

C_{int}: time to handle an interrupt → minimal context save & invoke scheduler

C_{sched}: time to execute scheduling code to determine next job to run

C_{resume}: time to resume a job after an interrupt but no context switch

C_{store}: time to save job state and save job in ready-to-run queue

Katcher et al.

C_{load}: time to launch a new active job from front of ready-to-run queue

C_{trap}: time to deal with a completed (current) job and select a new active job

- interrupts: **integrated** vs. **nonintegrated**

Carleton UNIVERSITY

Katcher et al.

Integrated Interrupts: supported in ARM!!

- h/w interrupt priorities are **matched** to s/w job priorities
  - e.g. ISR job at priority 5 has lower priority than job at priority 3 (lower priority cannot interrupt higher!)
  - ISR scheduling integrated with job scheduling
- assumes all jobs triggered by interrupts … oh ☺
  - Offloading to h/w interrupt priority handler!

Katcher et al.

- **when an integrated interrupt occurs:**

$$C_{preempt} = \underbrace{C_{int}}_{\text{interrupt handling}} + \underbrace{C_{sched}}_{\text{scheduling}} + \underbrace{C_{store}}_{\text{store job}} + \underbrace{C_{load}}_{\text{load new } \textit{ISR job}}$$

- **when *ISR job* terminates:**

$$C_{exit} = \underbrace{C_{trap}}_{\text{done } \textit{ISR job}} + \underbrace{C_{load}}_{\text{load new job}}$$

- **worst case o/s overhead** $= C_{preempt} + C_{exit}$

Katcher et al.

- **Recall schedulability test:**

  For all $i : 1 \le i \le n$, exists $T_s \in S_i(T_i)$

$$\left(\frac{\sum_{j=0}^{i-1} \left\lceil \frac{T_s}{T_j} \right\rceil C_j}{T_s} \right) \le \mathbf{1}$$

does not account for blocking by lower priority jobs!

Carleton UNIVERSITY

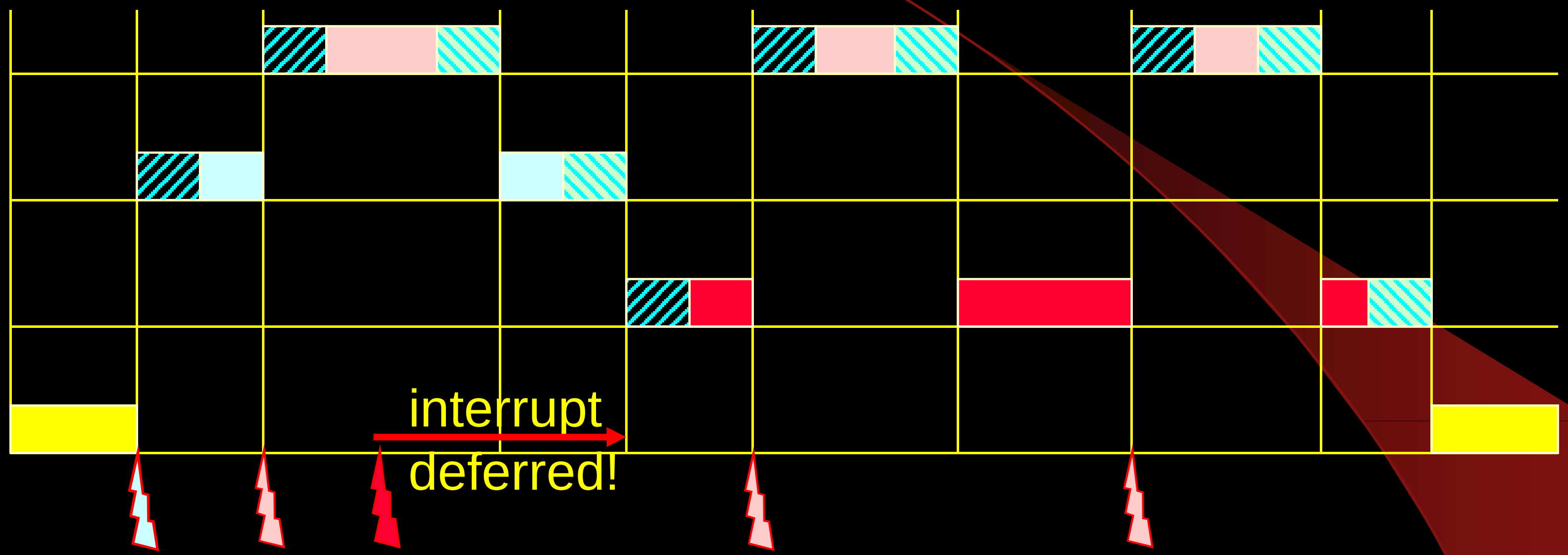
Katcher et al.

schedulability test for integrated interrupts:

For all i : $1 \le i \le n$, exists $T_s \in S_i(T_i)$

$$\left(\sum_{i=1}^{n} \frac{C_i + C_{preempt} + C_{exit}}{T_s} \left\lceil \frac{T_s}{T_i} \right\rceil \right) \le \mathbf{1}$$

Katcher et al.



idle task

$C_{preempt}$

C_{exit}

task 1

task 2

task 3

Katcher et al.

NonIntegrated Interrupts: → **more typical!**

- **h/w interrupt priorities are independent of s/w job priorities**
  - **interrupt always preempts current job**
  - **may introduce preemption blocking!**
- **assumes all jobs triggered by interrupts**

Katcher et al.

- when a nonintegrated interrupt occurs – if *ISR job* should preempt → same as before:

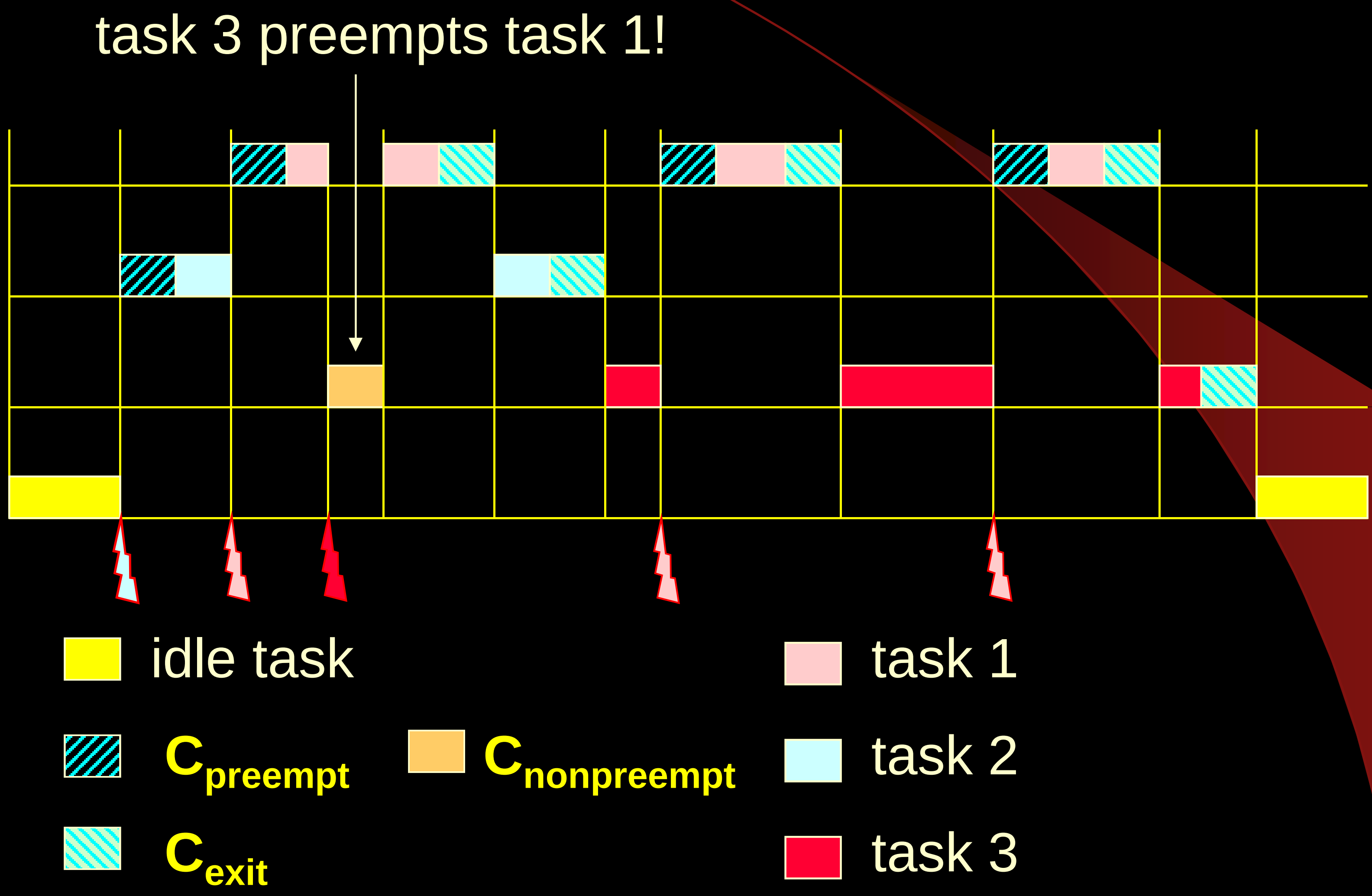
$$C_{preempt} = C_{int} + C_{sched} + C_{store} + C_{load}$$

$$C_{exit} = C_{trap} + C_{load}$$

- if ISR job should not preempt:

$$C_{nonpreempt} = \underbrace{C_{int}}_{\text{interrupt handling}} + \underbrace{C_{sched}}_{\text{scheduling}} + \underbrace{C_{resume}}_{\text{back to job}}$$

Katcher et al.



Katcher et al.

- **schedulability test for nonintegrated interrupts → must include blocking due to lower priority job interrupt preemption!**

$$\sum_{i=1}^{n} \left(\frac{C_i + C_{preempt} + C_{exit}}{T_s} \left\lceil \frac{T_s}{T_i} \right\rceil + \frac{(n-i)\, C_{nonpreempt}}{T_s} \right) \leq \mathbf{1}$$

number of lower priority jobs

Katcher et al.

- results from two case studies: overhead and blocking due to event-driven kernel led to degradation of schedulable utilization by 13% and 18%
- conclude tradeoff exists:

  more blocking → reduce overhead

  decrease blocking → increase overhead

EDF (Very Briefly)

- dynamic priority scheme, optimal
- job with nearest absolute deadline is highest priority
- priority at time t depends on jobs ready at t
- **PRO**: schedulable utilization = 1.0
- **CON**: cannot predict which jobs will miss deadlines & domino effect

EDF Problem

- cannot predict (a priori) which job might miss deadline

  → priority depends on dynamic load
- when job misses deadline, can cause other jobs to miss → domino effect
- need overrun strategy
- rate monotonic does not have this problem
  - lowest priority jobs miss deadlines

EDF Utilization (6.2 in text)

- Sufficient (but pessimistic when $D_k < p_k$):

$$U = \sum_{k=1}^{n} \frac{e_k}{\min(D_k, p_k)} \leq 1$$

- If $D_k \geq p_k$ for all k:
  then $U \leq 1$ is necessary and sufficient

EDF Deferrable Server

- Liu text: Theorem 7.3
- deferrable server:

  period = p_s budget = e_s utilization = u_s
- in system of n tasks and the deferrable server, task with period T_i schedulable when:

$$\sum_{k=1}^{n} \frac{e_k}{\min(D_k, p_k)} + u_s \left(1 + \frac{p_s - e_s}{D_i}\right) \leq \mathbf{1}$$

EDF Blocking

- Liu concludes that priority ceiling protocol better suited to fixed priority scheme than dynamic priority scheme …

  no comparable protocol for EDF

☹