Overview

- Today: this lecture explains unbounded priority inversion, Priority Inheritance, and Priority Ceiling.

- **To learn more on real-time scheduling**:
- **see chapter 7 on “Hard Real-Time Computing Systems” book from G. Buttazzo** (useful chapters are in the Lab!): unbounded priority inversion, Priority Inheritance.

Handling Hard Aperiodic Requests

- Most hard aperiodic requests are triggered by threshold based warnings; for example, a sensor checks the temperature of a device every 50 ms
  - if the temperature is ok, do nothing;
  - if the temperature is too high, send out a warning until it cools down.

- We do not know when the warning will come. But we know that if the warning comes:
  - each warning is separated by some minimum interval, 50 ms in this case
  - the handling time of warning must be bounded, say 5 ms.
  - so, we can just tailor a sporadic server with period 50 and budget 5 ms.

- Question: what is the advantage of sporadic server versus polling server in this example?

Handling Hard Aperiodic Requests

- Most hard aperiodic requests are triggered by threshold based warnings, for example, a sensor checks the temperature of a device every 50 ms
  - if the temperature is ok, do nothing
  - if the temperature is too high, send out a warning until it cools down,

- We do not know when the warning will come. But we know that if the warning comes:
  - each warning is separated by some minimum interval, 50 ms in this case
  - the handling time of warning must be bounded, say 5 ms.
  - so, we can just tailor a Sporadic Server with period 50 and budget 5 ms.

- Question: what is the advantage of sporadic server vs polling in this example?
- Answer: SS has much better response time; a polling server with period 50ms cannot guarantee a relative deadline D=50ms.

Handling Soft Real Time Requests

- How can we estimate the avg. response time of soft aperiodic requests?
- The basic tools here are 1) queueing theory and 2) simulation. We will limit ourselves to the simplest form of queueing theory model in this class.

- M/M/1 queue:
  - Poisson arrival with average arrival rate λ, say 10 arrivals on average per sec.
  - Exponential service time, $1/\mu$, say 0.01 sec on average (avg. execution time of aperiodic task).
  - The CPU_workload is average arrival rate times service time: λ / μ
  - The server_bandwidth is $U_s = C_s / P_s$.
  - The server_workload ρ, is equal to CPU_workload / server_bandwidth

  - The average response time (queue waiting time + service time) is:

$$w = (1/\mu)/(1 - \rho)$$

Handling Soft Real Time Requests

- Sporadic Server (SS) Design guideline: Give it as high priority as possible and as much “budget” as possible, without causing periodic tasks to miss their deadlines.

- For getting a rough initial estimation of avg. response time, we may use queuing theory formula M/M/1, provided that the sporadic server executes at a high priority.

- The M/M/1 approximation for sporadic server is:

$$W = \frac{1/\mu}{1-\rho}$$

To estimate the average response time accurately, you should use simulations.

Sample Problem: estimate response time of aperiodics

- Example:
- an aperiodic task with average execution time 1 msec and average inter-arrival time 100 msec creates a ___ average CPU_workload.

- Server budget is C = 5 and server period is P = 100 ➔ server_bandwidth = ___.

- The server_workload ρ is (CPU_workload / server_bandwidth) = ____.

- The M/M/1 approximation of avg response time when using the sporadic server is:

$$W = \frac{1/\mu}{1-\rho} = \underline{\qquad\qquad}$$ Average response time of aperiodics!

Sample Problem: estimate response time of aperiodics

- Example:
- an aperiodic task with average execution time 1 msec and average inter-arrival time 100 msec creates a 1% average CPU_workload.
- Server budget is C = 5 and server period is P = 100 ➔ server_bandwidth = 5%.
- The server_workload ρ is (CPU_workload / server_bandwidth) = 0.01/0.05 = 0.2.
- NOTE: We should NOT use 0.01 as server_workload, since we have only a fraction of the CPU (5%) assigned to the sporadic server.
- The M/M/1 approximation for sporadic server is:

$$W = \frac{1/\mu}{1-\rho} = \frac{1}{1-0.2} = 1.25ms \quad \Big\} \text{ Average response time of aperiodics!}$$

Unbounded Priority Inversion

- When a high priority task is delayed by lower priority tasks, it is said that priority inversion has occurred and the high priority task is blocked by the lower priority task.

- Priority inversion occurs during synchronization.

- When tasks synchronize, we expect delays due to the use of mutual exclusion.

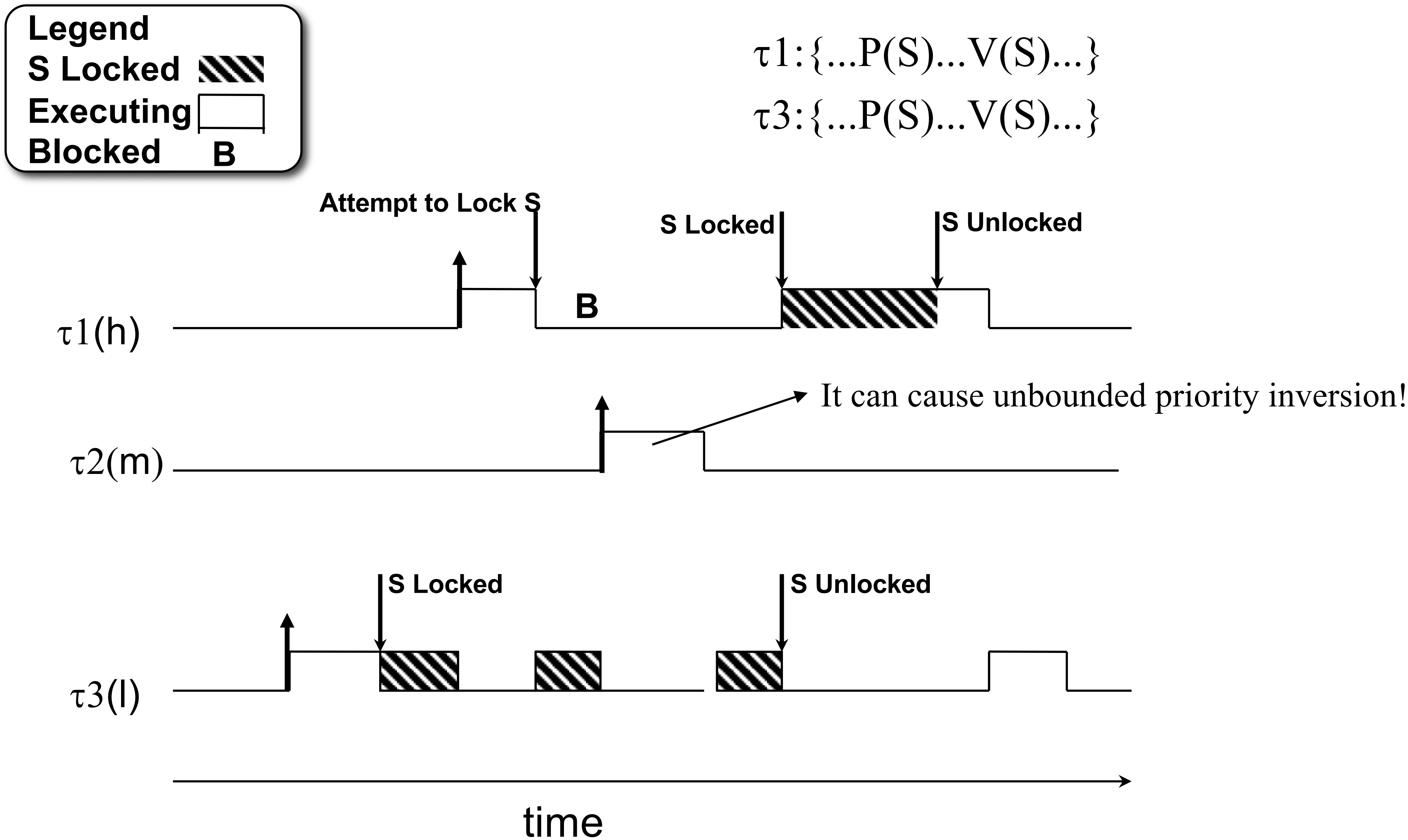
- And we expect that the delay due to mutual exclusion is a function of the duration of the critical sections.

- When the duration of priority inversion <u>is not bounded</u> by a function of the duration of critical sections, <u>unbounded priority inversion</u> is said to occur.

Unbounded Priority Inversion

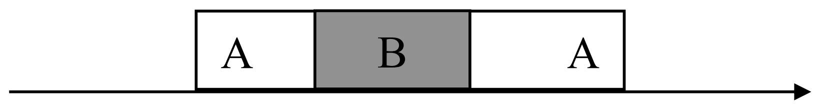
$\tau1$:{...P(S)...V(S)...}

$\tau3$:{...P(S)...V(S)...}



Basic Priority Inheritance Protocol

- We shall assume that:
  1) a job does not self-suspend inside a critical section;
  2) if nested semaphores are used, they will be properly nested.



  3) Critical sections are guarded by binary semaphores. This means that only one job at a time can be within the critical section corresponding to a particular semaphore S_k.
  4) Jobs J1, J2,…,Jn are listed in descending order of nominal priority, with J1 having the highest nominal priority.

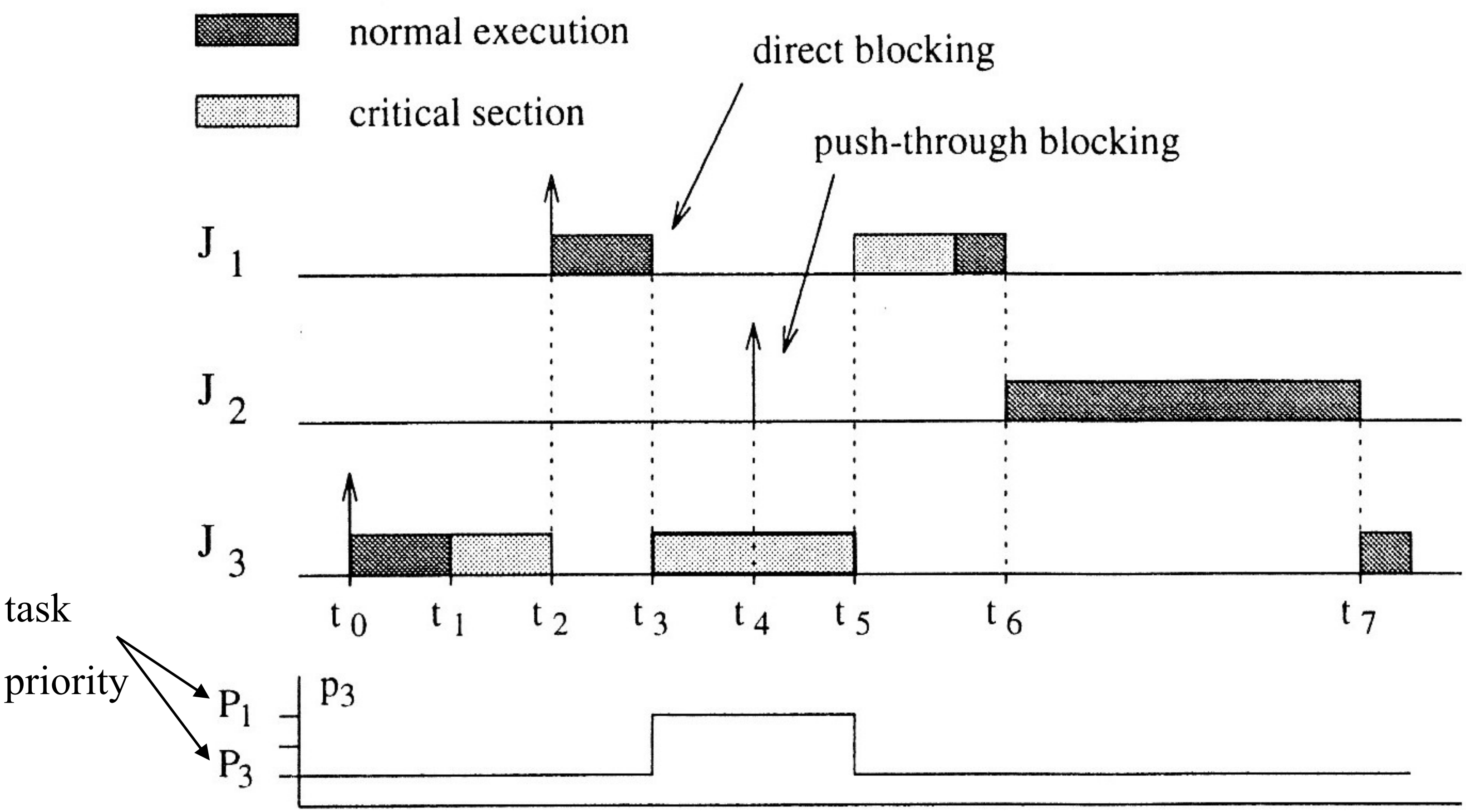
Basic Priority Inheritance Protocol

- Rule 1: When a lower priority task blocks higher priority tasks during its critical section, it uses the highest priority of all the blocked tasks.

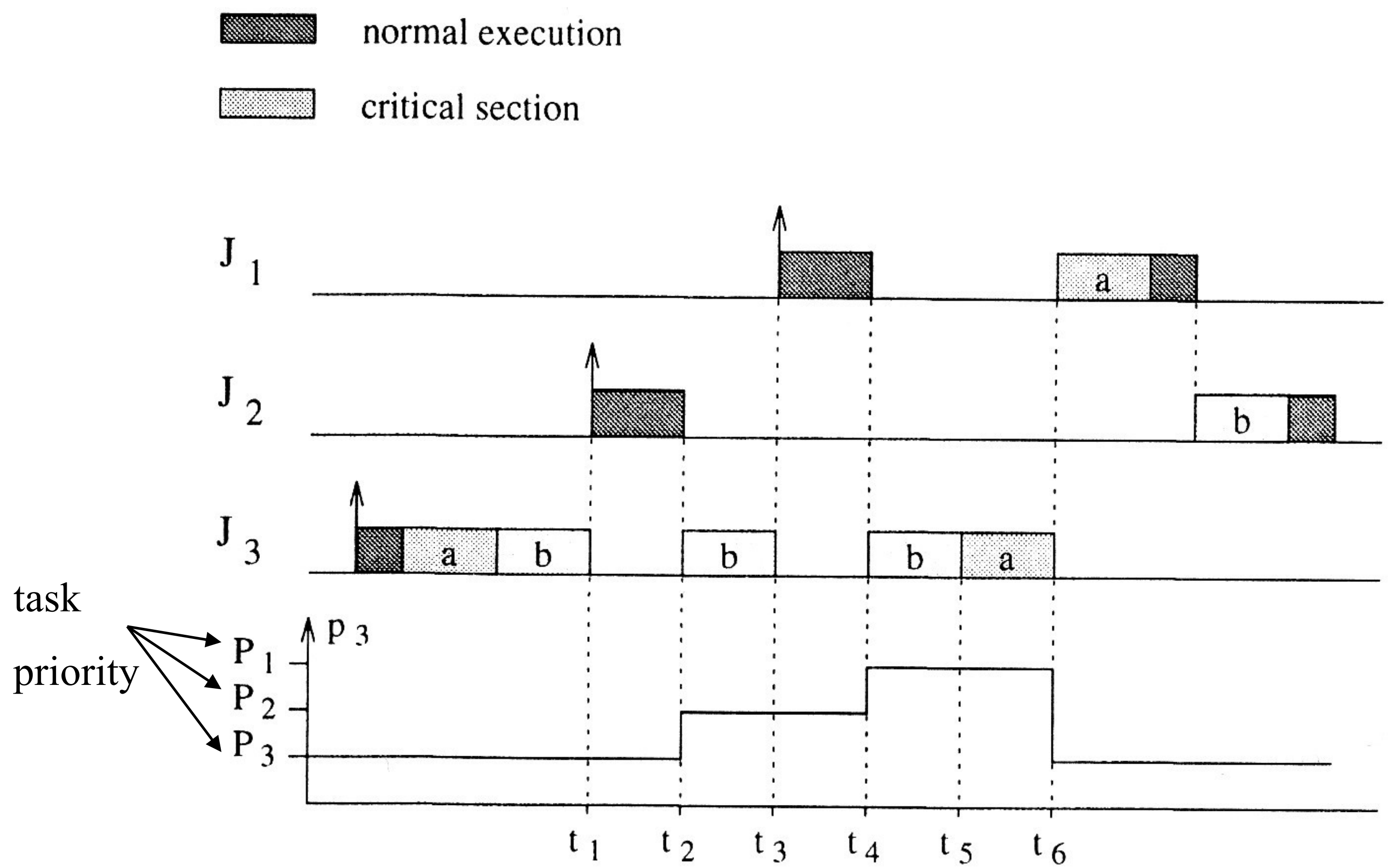
- Rule 2: When a task exits its critical section, it returns to its normally assigned priority

- Rule 3: Priority inheritance is transitive; that is, if a job J3 blocks a job J2, and J2 blocks a job J1, then J3 inherits the priority of J1 via J2.

Basic Priority Inheritance Protocol



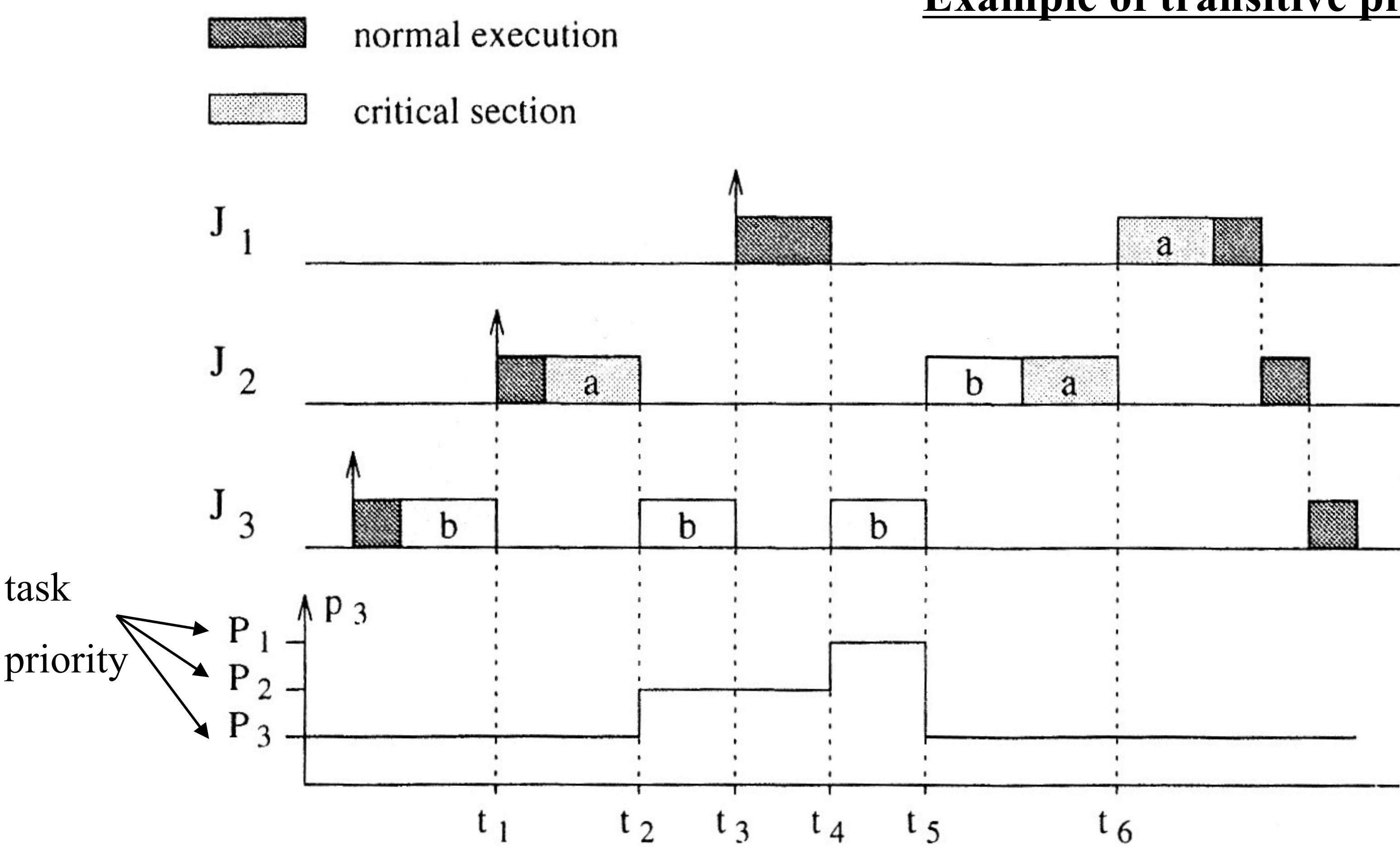
Basic Priority Inheritance Protocol



Nested critical sections

Basic Priority Inheritance Protocol

<u>Example of transitive priority inheritance</u>



Transitive priority inheritance can occur only in the presence of nested critical sections.

Schedulability analysis

- Class exercise: check the schedulability of the following task set:

	C	p	B
T_1	1	2	1
T_2	1	4	1
T_3	2	8	0

Schedulability analysis

- Class exercise: check the schedulability of the following task set:

	C	p	B
T_1	1	2	1
T_2	1	4	1
T_3	2	8	0

- Notice that periods are harmonic, so the utilization bound is 1!

$$T_1 \rightarrow \frac{c_1}{p_1} + \frac{B_1}{p_1} = 1$$

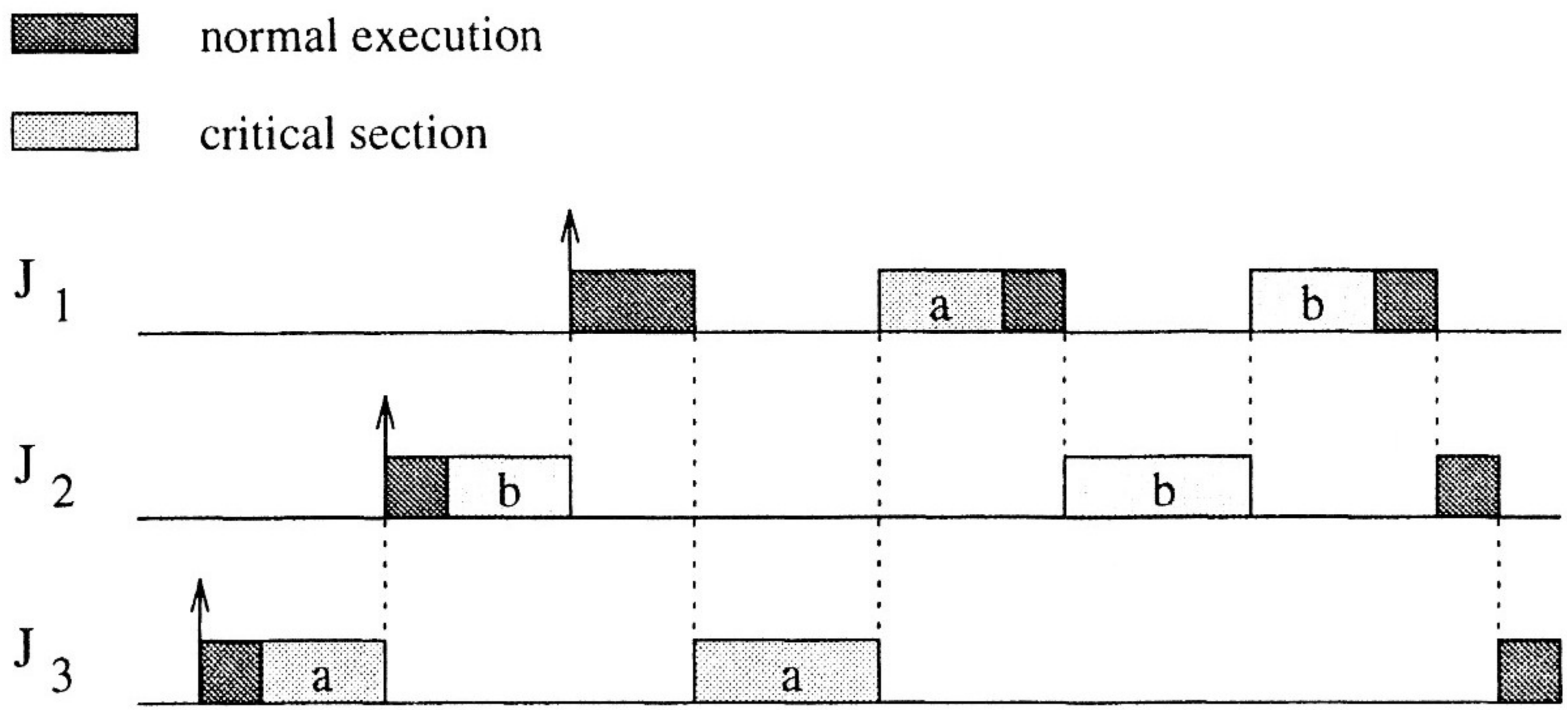
$$T_2 \rightarrow \frac{c_1}{p_1} + \frac{c_2}{p_2} + \frac{B_2}{p_2} = 1$$

The task set is schedulable!

$$T_3 \rightarrow \frac{c_1}{p_1} + \frac{c_2}{p_2} + \frac{c_3}{p_3} + \frac{B_3}{p_3} = 1$$

Chained Blocking under PIP

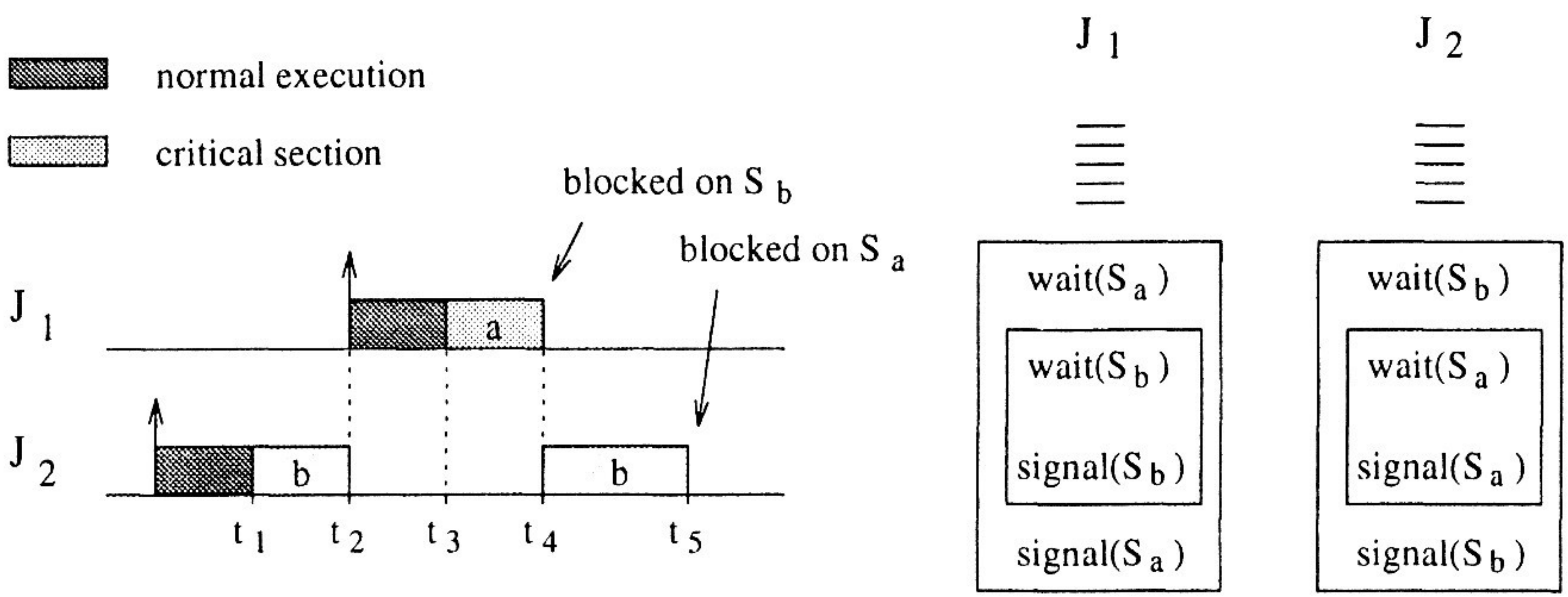
- J_1 is blocked for the duration of two critical sections, once to wait J_3 to release S_a and then to wait J_2 to release S_b. This is called a *chained blocking*.



Deadlock Under PIP

- Priority inheritance does not prevent a deadlock. Notice, however, that the deadlock does not depend on the PI protocol but it is caused by an erroneous use of semaphores.

- The deadlock can be solved by imposing a total ordering on the semaphore accesses.



PI: blocking time computation I

- So far, we analyzed the task set schedulability by assuming the knowledge of the blocking time B_i for each task T_i.
- How do we compute the blocking time of each task?
- The evaluation of the maximum blocking time for each task can be computed based on the following property:

> When using the PI protocol, a job J can be blocked for at most the duration of min(n, m) critical sections, where n is the number of lower priority jobs that could block J and m is the number of distinct semaphores that can be used to block J.

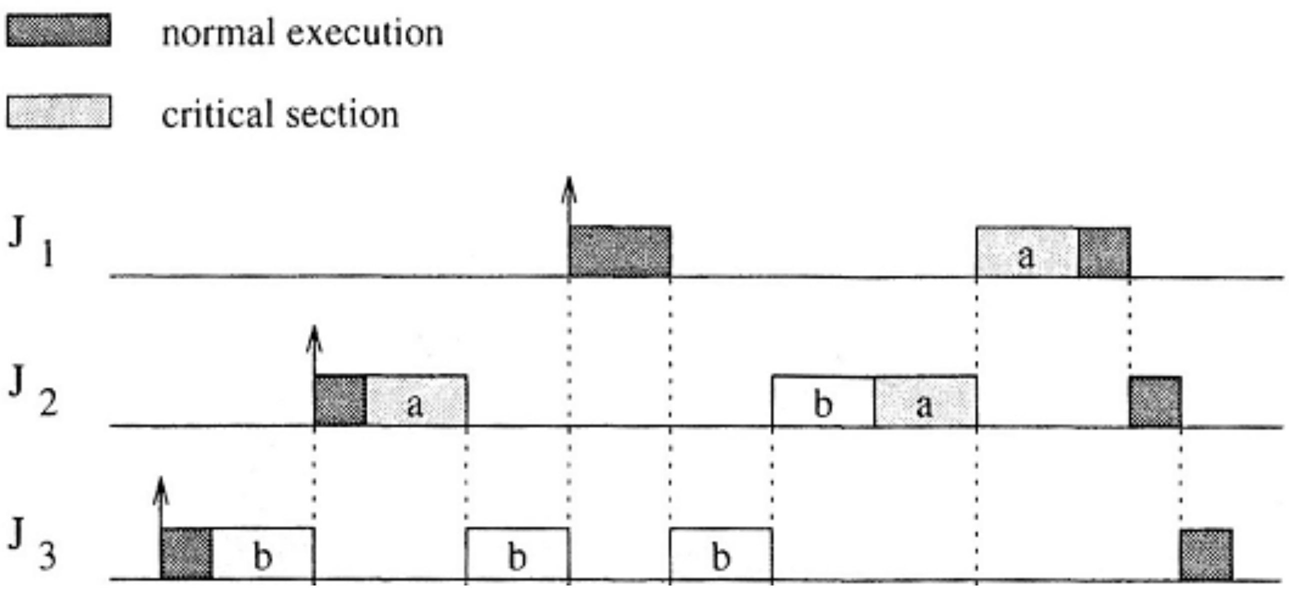
- Notice that a precise evaluation of B_i is quite complex. A simplified algorithm is obtained if nested critical sections are forbidden (it avoids transitive inheritance).

PI: blocking time computation II

- Let's define a very important concept (you will find it again in PC protocol):

- **the ceiling $C(S_k)$ of a semaphore S_k is the priority of the highest priority task that may use S_k:**

$$C(S_k) = \max_i (prio_i \mid task \quad T_i \quad uses \quad S_k)$$



$C(S_A)=$

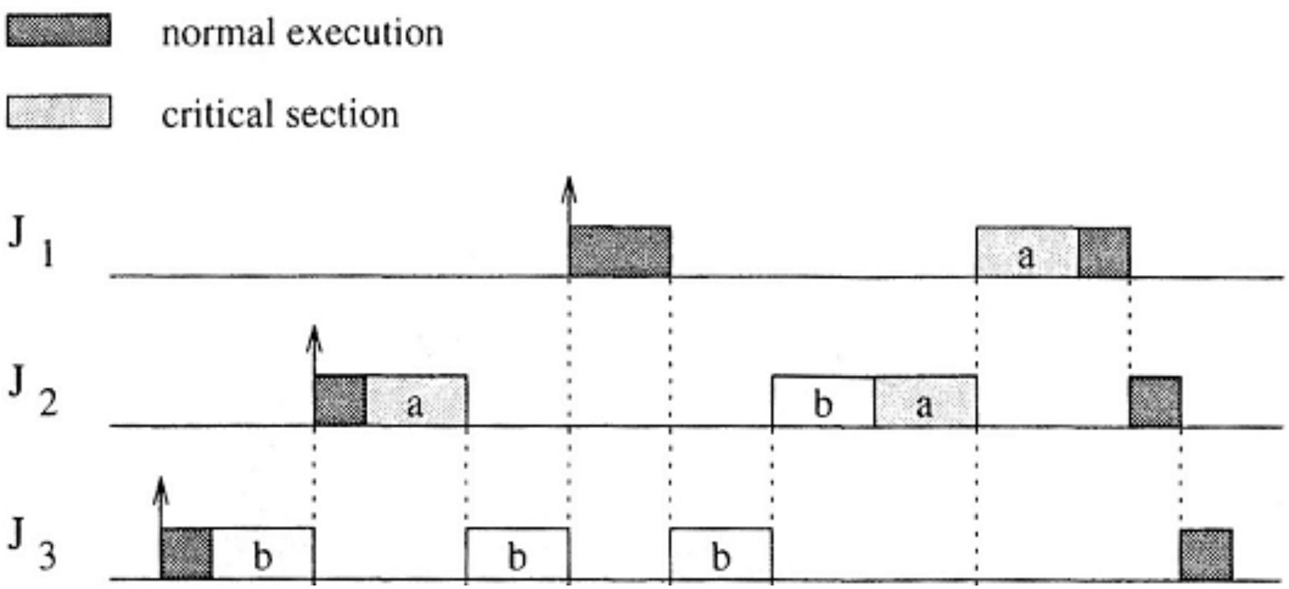
$C(S_B)=$

PI: blocking time computation II

- Let's define a very important concept (you will find it again in PC protocol):

- **the ceiling C(S_k) of a semaphore S_k is the priority of the highest priority task that may use S_k:**

$$C(S_k) = \max_i (prio_i \mid task \quad T_i \quad uses \quad S_k)$$



$C(S_A)$= $prio_1$

$C(S_B)$= $prio_2$

PI: blocking time computation III

In the absence of nested critical sections, a critical section $z_{j,k}$ of task T_j guarded by semaphore S_k can block task T_i only if $prio_j < prio_i \leq C(S_k)$.

- The maximum blocking time B_i for each task T_i can be determined as follows:
  - Determine the value of $B_i^{n\_blocking\_task}$

Duration of k-th critical section of task T_j

$$B_i^{n\_blocking\_task} = \sum_{j=i+1}^{n} \underbrace{\max_k [D_{j,k} \mid C(S_k) \geq prio_i]}_{\text{Longest critical section of } T_j \text{ whose semaphore } S_k \text{ can block task } T_i}$$

Lower-priority tasks

PI: blocking time computation IV

- Determine the value of $B_i^{m\_blocking\_section}$

Duration of k-th critical section of task T_j

$$B_i^{m\_blocking\_section} = \sum_{k=1}^{m} \max_j [\underbrace{D_{j,k} \mid C(S_k) \geq prio_i \wedge (j > i)}]$$

All distinct semaphores

Longest critical section guarded by S_k belonging to a lower priority task T_j and such that semaphore S_k can block task T_i

- Select $B_i = \min(B_i^{m\_blocking\_section}, B_i^{n\_blocking\_task})$

PI: blocking time computation V

- Notice that this algorithm provides an upper bound for the blocking factors B_i; however such a bound is not tight as $B_i^{n\_blocking\_task}$ may be computed by considering two or more critical sections guarded by the same semaphore.

- In fact, if two critical sections of different tasks are guarded by the same semaphore, they cannot both block at the same time

- Similarly, $B_i^{m\_blocking\_section}$ may be computed by considering two or more critical sections belonging to the same task

PI: class exercise

- Consider four tasks sharing three semaphores:
- Compute the blocking time of each task

	S_A	S_B	S_c
T_1	1	2	
T_2		9	3
T_3	8	7	
T_4	6	5	4

Duration of the critical section

$$B_i^{n\_blocking\_task} = \sum_{j=i+1}^{n} \max_k [D_{j,k} \mid C(S_k) \geq prio_i]$$

$$B_i^{m\_blocking\_section} = \sum_{k=1}^{m} \max_j [D_{j,k} \mid C(S_k) \geq prio_i \wedge (j > i)]$$

PI: class exercise

- Consider four tasks sharing three semaphores:

	S_A	S_B	S_c
T_1	1	2	
T_2		9	3
T_3	8	7	
T_4	6	5	4

Duration of the critical section

- Compute the blocking time of each task

- Semaphore ceilings: $C(S_A)$= $prio_1$, $C(S_B)$= $prio_1$, $C(S_C)$= $prio_2$

$$B_1^{n\_blocking\_task} = 9 + 8 + 6 = 23$$

$$B_2^{n\_blocking\_task} = 8 + 6 = 14$$

$$B_3^{n\_blocking\_task} = 6$$

$$B_4^{n\_blocking\_task} = 0$$

$$B_i^{n\_blocking\_task} = \sum_{j=i+1}^{n} \max_k [D_{j,k} \mid C(S_k) \geq prio_i]$$

$$B_i^{m\_blocking\_section} = \sum_{k=1}^{m} \max_j [D_{j,k} \mid C(S_k) \geq prio_i \wedge (j > i)]$$

PI: class exercise

$$B_1^{m\_blocking\_section} = 9 + 8 = 17$$

$$B_2^{m\_blocking\_section} = 8 + 7 + 4 = 19$$

$$B_3^{m\_blocking\_section} = 6 + 5 + 4 = 15$$

$$B_4^{m\_blocking\_section} = 0$$

	S_A	S_B	S_c
T_1	1	2	
T_2		9	3
T_3	8	7	
T_4	6	5	4

$$B_1 = 17$$

$$B_2 = 14$$

$$B_3 = 6$$

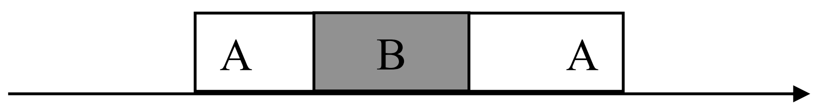
$$B_4 = 0$$

$$B_i^{n\_blocking\_task} = \sum_{j=i+1}^{n} \max_k [D_{j,k} \mid C(S_k) \geq prio_i]$$

$$B_i^{m\_blocking\_section} = \sum_{k=1}^{m} \max_j [D_{j,k} \mid C(S_k) \geq prio_i \wedge (j > i)]$$

Assumptions of Priority Ceiling Protocol

- We shall assume that:
  1) a job does not self-suspend inside a critical section;
  2) if nested semaphores are used, they will be properly nested.



  3) Critical sections are guarded by binary semaphores. This means that only one job at a time can be within the critical section corresponding to a particular semaphore S_k.
  4) Jobs J1, J2,…,Jn are listed in descending order of nominal priority, with J1 having the highest nominal priority.

Priority Ceiling Protocol

- It extends the Priority Inheritance protocol
  - It avoids multiple blocking (chained blocking)
  - It prevents the formation of deadlocks

- A job is not allowed to access a critical section if there are locked semaphores that could block it.

- As a consequence, once a job enters its first critical section, it can never be blocked by lower-priority jobs until its completion.

- To realize this idea, each semaphore is assigned a priority ceiling equal to the priority of the highest-priority job that can lock it.

Priority Ceiling Protocol

- To realize the Priority Ceiling idea, each semaphore is assigned a priority ceiling equal to the priority of the highest-priority job that can lock it.

- **the ceiling $C(S_k)$ of a semaphore S_k is the priority of the highest priority task that may use S_k:**

$$C(S_k) = \max_i (prio_i \mid task \quad T_i \quad uses \quad S_k)$$

- **Key idea**: *a job J is allowed to enter a critical section only if its priority is higher than all priority ceilings of the semaphores currently locked by jobs other than J*

Priority Ceiling Protocol: rules

- Rule 1: Each semaphore S_k is assigned a static priority ceiling $C(S_k)$; notice that it can be computed off-line.
- Rule 2: the highest priority job J (among all jobs ready to run) is assigned the processor.
- Rule 3: let S* be the semaphore with the highest-priority ceiling among all the semaphores currently locked by jobs other than J and let C(S*) be its ceiling.
- Rule 4: to enter a critical section guarded by semaphore S_k, J must have a priority higher than C(S*). If $prio(J) \le C(S^*)$, the lock on S_k is denied and J is said to be blocked on semaphore S* by the job that holds the lock on S*.

Priority Ceiling Protocol: rules

- Rule 5: when a job J is blocked on a semaphore S*, J transmits its priority to the job J* that holds the semaphore S*. So, J* inherits the priority of J. In general, a task inherits the highest priority of the jobs blocked by it.

- Rule 6:when a resource is freed by J*, its priority is updated. J* inherits the highest priority of the jobs (if any) blocked by it; otherwise, it returns to its nominal priority.

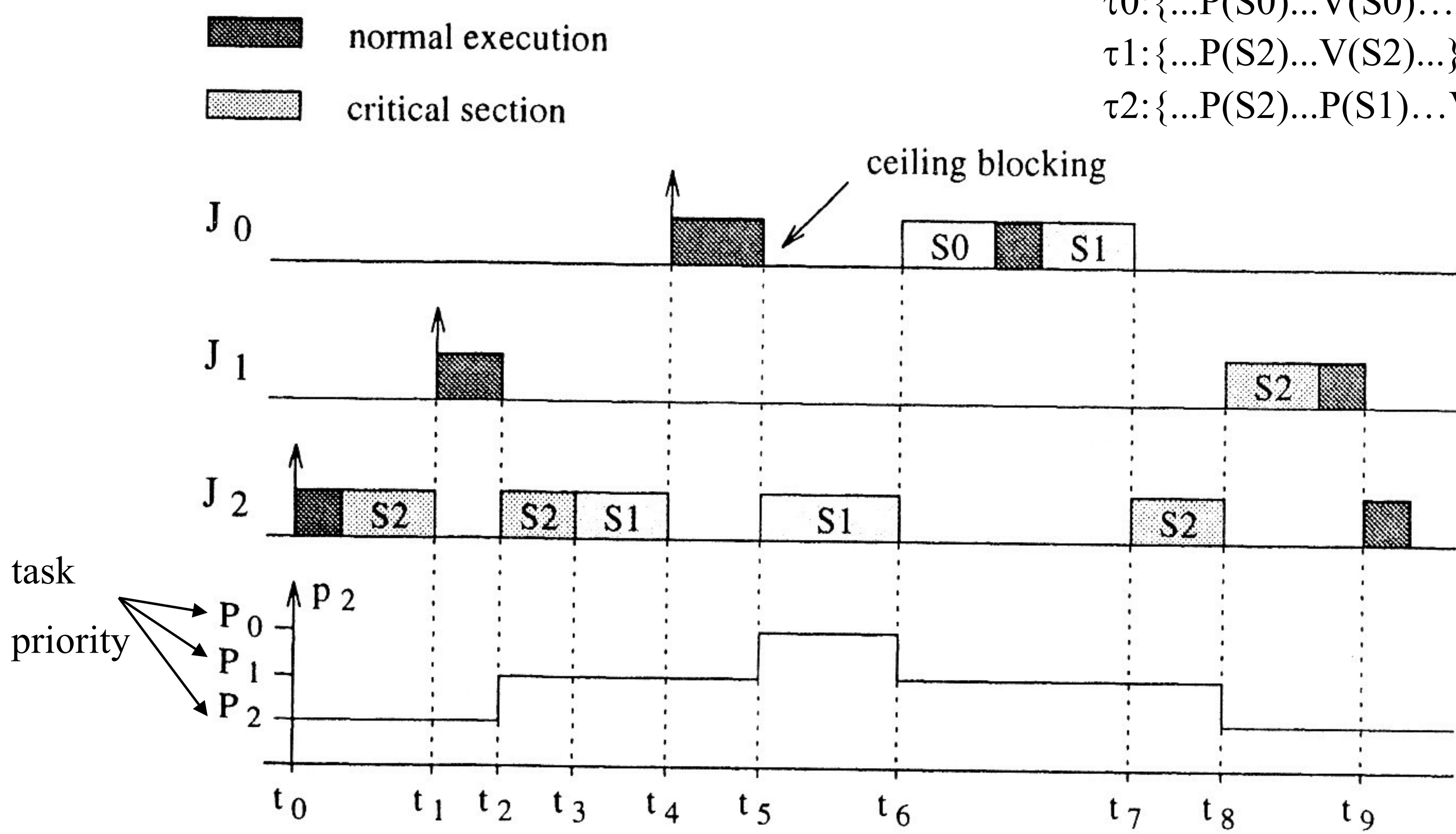
Priority Ceiling Protocol: example

$C(S_0)$= $prio_0$ $C(S_1)$= $prio_0$ $C(S_2)$= $prio_1$

τ0:{...P(S0)...V(S0)…P(S1)...V(S1)}

τ1:{...P(S2)...V(S2)...}

τ2:{...P(S2)...P(S1)…V(S1)...V(S2)}...}

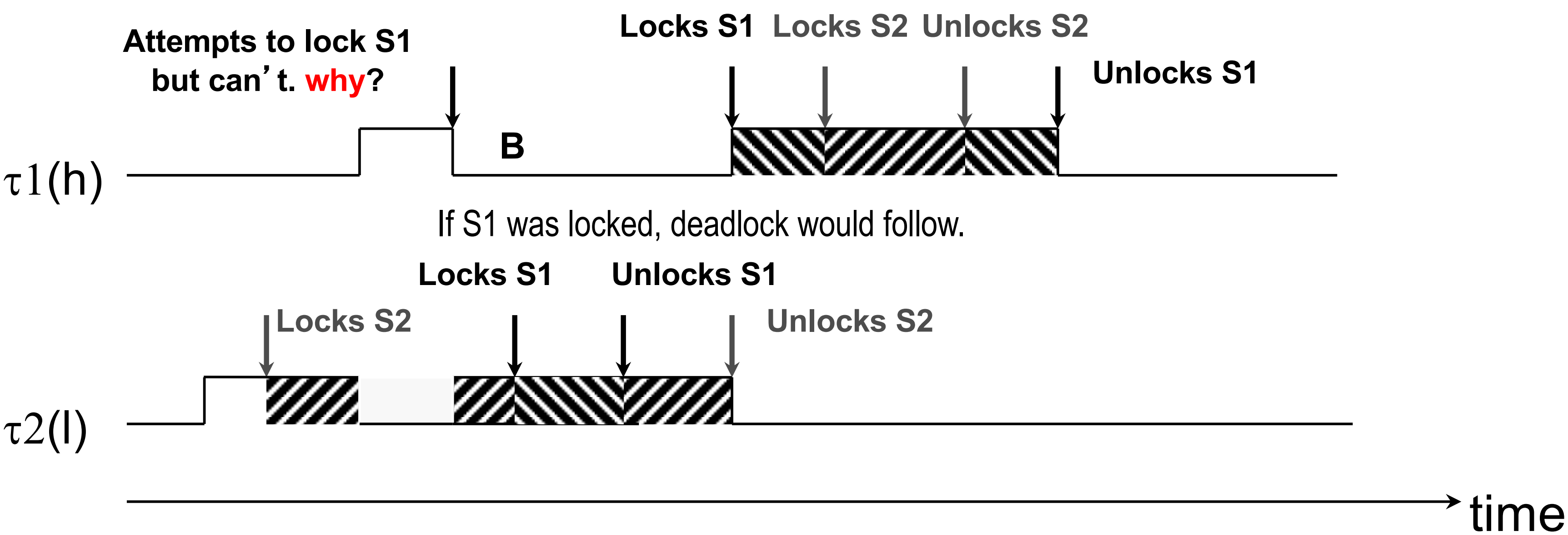


Deadlock Avoidance: Using PCP

Legend
S1 Locked
S2 Locked
Executing
Blocked B

$\tau 1$:{...P(S1)...P(S2)...V(S2)...V(S1)...}

$\tau 2$:{...P(S2)...P(S1)...V(S1)...V(S2)...}

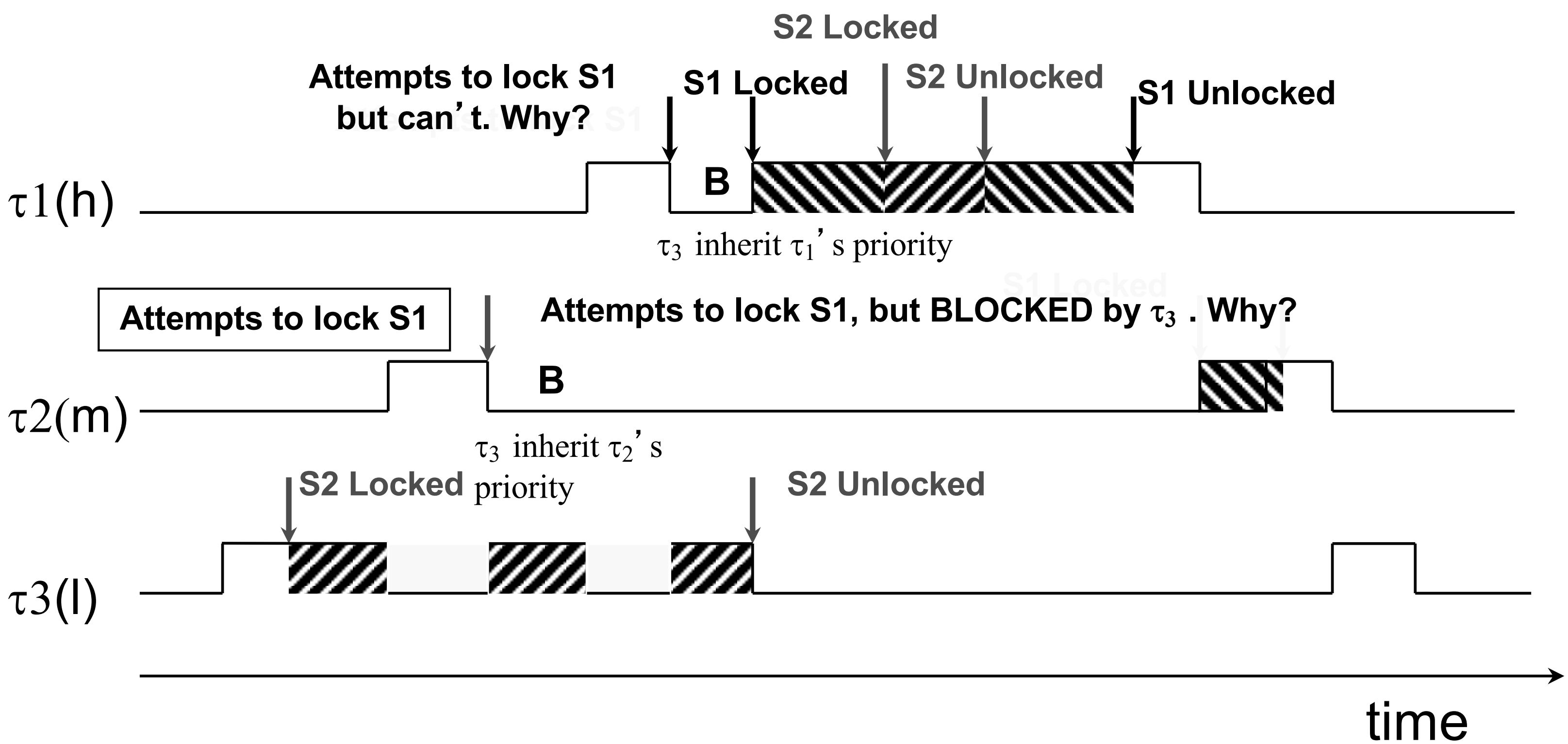


Note: Task T2 can still lock S1 since it owns the S2 lock, S1 is not locked by OTHER tasks

Blocked at Most Once (PCP)

Legend
S1 Locked
S2 Locked
Executing
Blocked **B**

$\tau 1$:{...P(S1)...P(S2)...V(S2)...V(S1)...}
$\tau 2$:{...P(S1)...V(S1)...}
$\tau 3$:{...P(S2)...V(S2)...}



Properties of Priority Ceiling Protocol

Under the Priority Ceiling Protocol, a job J_i can be blocked for at most the duration of the longest outermost critical section among those that can block J_i.

The priority Ceiling Protocol prevents deadlocks

Blocking time computation I

- So far, we analyzed the task set schedulability by assuming the knowledge of the blocking time B_i for each task T_i.
- How do we compute the blocking time of each task?
- The evaluation of the maximum blocking time for each task can be computed based on the following property of PCP:

A job J_i can be blocked for at most the duration of the longest outermost critical section among those that can block J_i.

Blocking time computation II

Under PCP, a critical section $z_{j,k}$ of task T_j guarded by semaphore S_k can block task T_i only if $prio_j < prio_i$ and $C(S_k) \geq prio_i$.

Duration of k-th critical section of task T_j

$$B_i = \max_{j,k}[\underbrace{D_{j,k} \mid prio_j < prio_i, \quad C(S_k) \geq prio_i}_{\text{Longest critical section among } T_j \text{ whose semaphore } S_k \text{ can block task } T_i}]$$

Class exercise

- Consider four tasks sharing three semaphores:

- Compute the blocking time of each task

	S_A	S_B	S_c
T_1		2	
T_2	4		
T_3		7	2
T_4	3	5	4

Duration of the critical section

$$B_i = \max_{j,k}[D_{j,k} \mid prio_j < prio_i, \quad C(S_k) \geq prio_i]$$

Class exercise

- Consider four tasks sharing three semaphores:

	S_A	S_B	S_c
T_1		2	
T_2	4		
T_3		7	2
T_4	3	5	4

Duration of the critical section

- Compute the blocking time of each task

- Semaphore ceilings: $C(S_A)= prio_2$ $C(S_B)= prio_1$ $C(S_C)= prio_3$

$$B_1 = \max(7,5) = 7$$

$$B_2 = \max(3,5,7) = 7$$

$$B_3 = \max(3,5,4) = 5$$

$$B_4 = 0$$

$$B_i = \max_{j,k}[D_{j,k} \mid prio_j < prio_i, \quad C(S_k) \geq prio_i]$$

Class exercise

- Consider four tasks sharing three semaphores:

- Compute the blocking time of each task

	S_A	S_B	S_c
T_1	1	2	
T_2		9	3
T_3	8	7	
T_4	6	5	4

Duration of the critical section

$$B_i = \max_{j,k}[D_{j,k} \mid prio_j < prio_i, \quad C(S_k) \geq prio_i]$$

Class exercise

- Consider four tasks sharing three semaphores:

	S_A	S_B	S_c
T_1	1	2	
T_2		9	3
T_3	8	7	
T_4	6	5	4

Duration of the critical section

- Compute the blocking time of each task

- Semaphore ceilings: $C(S_A)=prio_1$ $C(S_B)=prio_1$ $C(S_C)=prio_2$

$$B_1 = \max(8,6,9,7,5) = 9$$

$$B_2 = \max(8,6,7,5,4) = 8$$

$$B_3 = \max(6,5,4) = 6$$

$$B_4 = 0$$

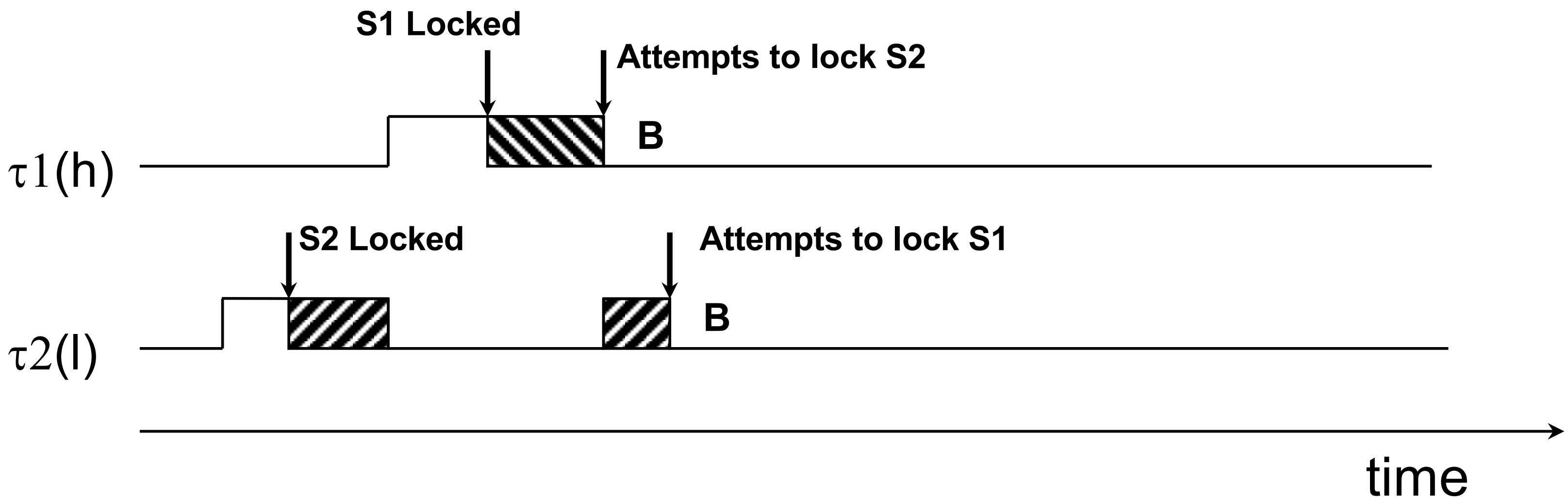
$$B_i = \max_{j,k}[D_{j,k} \mid prio_j < prio_i, \quad C(S_k) \geq prio_i]$$

Appendix: understanding PCP

- What is the key that makes PCP having such nice properties:
  - Blocked at most once
  - Deadlock avoidance

- Let's try to have a more intuitive understanding of PCP……..

Appendix: Mutual Exclusion of Shared Locks - 1

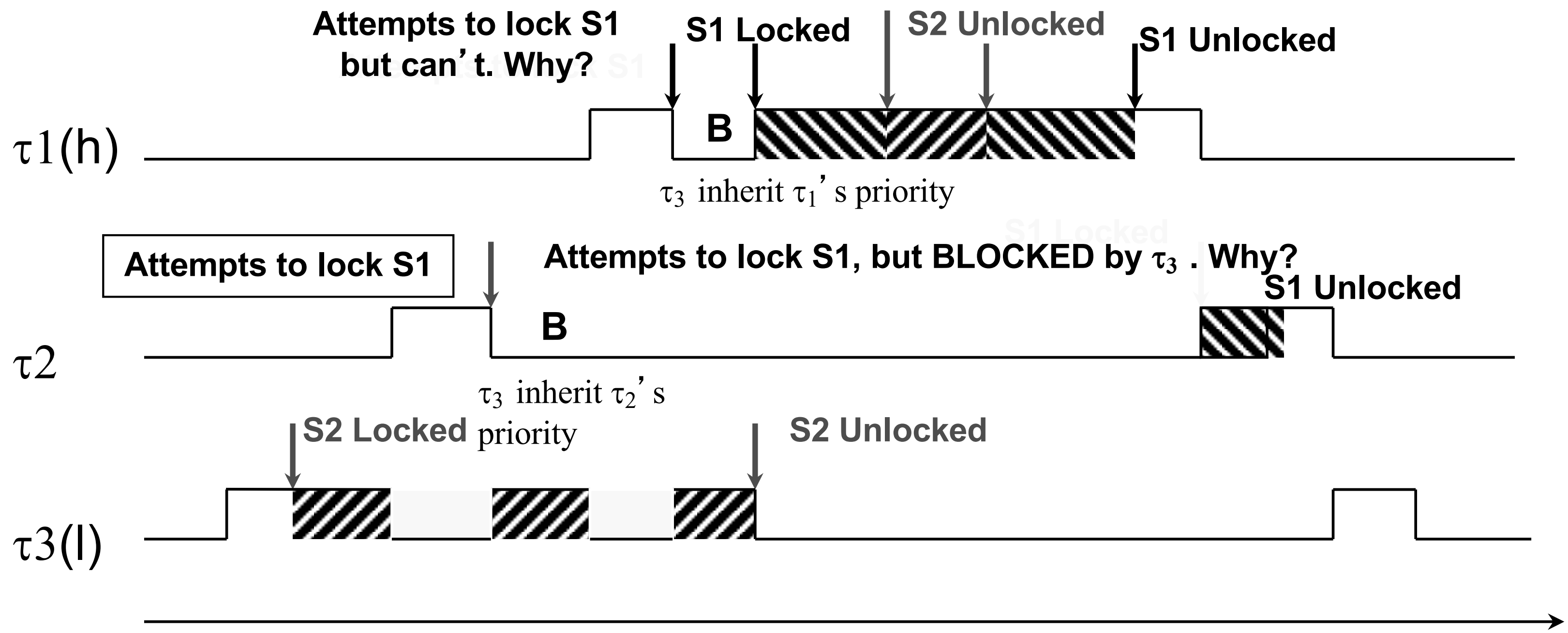


- Look at the deadlock again. For it to happen, the high priority job must hold a lock that low priority job wants and vice versa.

- If a set of jobs J_1, J_2, J_3 shares a set of locks S_1, S_2, …, $S_{n,}$ and anyone, say J_3, gets to hold one of them, than the rest of jobs J1 & J2 are not allowed to touch the locks until J_1 is done. That is, when a job gets one of the shared locks, PCP will guarantee that this job will get ALL the locks that it ever needs… This observation is the foundation of PCP properties

- Ok. Now, what is the key argument in the absence of chained blocking based on the property of mutual exclusion on the shared set of locks?

Appendix: Mutual Exclusion of Shared Locks - 2



- For chained blocking to occur, more than ONE lower priority job has to get the lock before the high priority job starts. Under PCP, when a job gets one lock in a shared set of resources, it prevents other jobs sharing those resources to acquire locks until it is done; hence, <u>it prevents chained blocking</u>.

Appendix: M/M/1

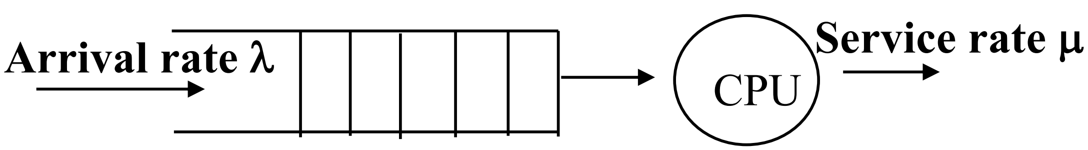
- The M/M/1 queue assumes exponential arrival process and exponential service time.
- In fact, if t^* is the random variable describing the distribution of inter-arrival times between two consecutive arrivals, it follows that:

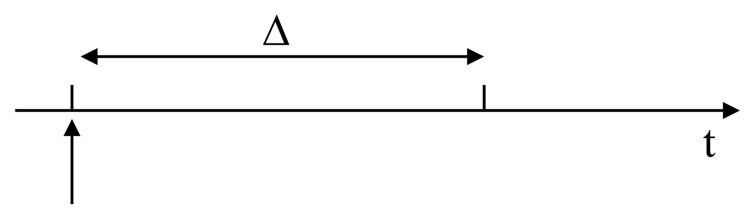
$$A(t) = P\{t^* \le t\} = 1 - e^{-\lambda t}$$

cumulative distribution function (cdf)

- The probability density function (pdf) is:

$$a(t) = \frac{d}{dt} A(t) = \lambda e^{-\lambda t}$$

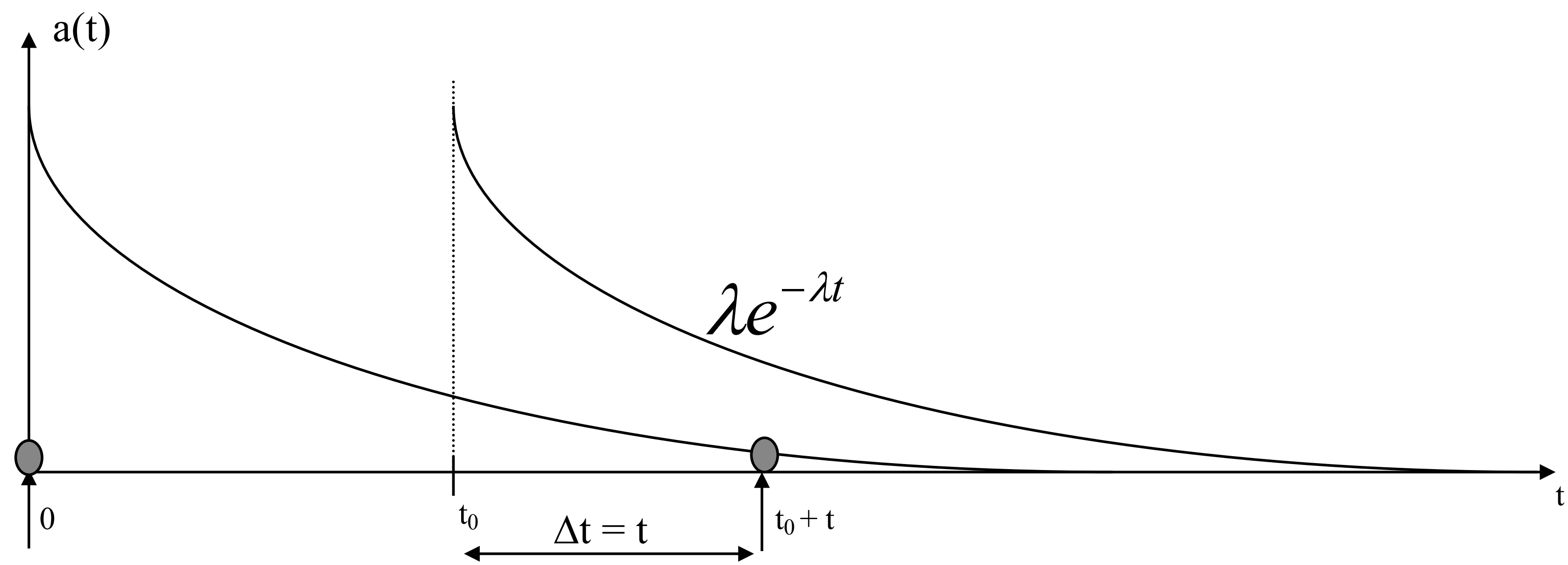




Probability to have a new arrival within Δ is $1-e^{-\lambda \Delta}$

Appendix: M/M/1

- The exponential distribution is <u>**memoryless**</u>



- Suppose you buy a new electronic device and its lifetime probability distribution is exponential. Such a device with exponential lifetime distribution is characterized by the phrase: ***used is as good as new!***