Blocking time computation

- We must build a *resource usage table.:*
  - On each row we put a task (decreasing order of priority);
  - On each column we put a resource, in any order;
  - In each cell (i, j) we put
    - the length of the longest critical section of task i on resource Sj ,
    - or 0 if the task does not use the resource.

Blocking time computation

- A task can be blocked only by lower priority tasks:
  - we must consider only the rows below (tasks with lower priority)
- A task can be blocked only on
  - resources that it uses directly,
  - or used by higher priority tasks (*indirect blocking);*
- For each task, we must consider only those columns on which it can be blocked (used by itself or by higher priority tasks).

Example

	Q	R	S	Blocking
A	2	0	0	?
B	0	1	0	?
C	0	0	2	?
D	3	3	1	?
E	1	2	1	?

- Consider A
  - A can be blocked only on Q.
  - Therefore, we must consider only the first column, and take the maximum, which is 3.
- Therefore, $B_A = 3$.

Example

	Q	R	S	Blocking
A	2	0	0	3
B	0	1	0	?
C	0	0	2	?
D	3	3	1	?
E	1	2	1	?

- B can be blocked on Q (*indirect blocking) and on R.*
  - *Therefore, we* must consider the first 2 columns;
  - Consider all cases where two distinct lower priority tasks between 3, 4 and 5 access Q and R,
  - sum the two contributions, and take the maximum;
  - possibilities are:
    - D on Q and E on R: 3 + 2 = 5;
    - D on R and E on Q: 3 + 1 = 4;
- Therefore, $B_B = 5$.

Example

	Q	R	S	Blocking
A	2	0	0	3
B	0	1	0	5
C	0	0	2	?
D	3	3	1	?
E	1	2	1	?

- C can be blocked on Q, R, S
- Work out possible combinations:
  - D on Q and E on R: 5;
  - D on R and E on Q or S: 4;
  - D on S and E on Q: 2;
  - D on S and E on R : 3;
- So blocking for C is 5

Example: final result

	Q	R	S	Blocking
A	2	0	0	3
B	0	1	0	5
C	0	0	2	5
D	3	3	1	2
E	1	2	1	0

Response Time and Blocking

$$R_i = C_i + B_i + I_i$$

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

$$R_i^{(n+1)} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{(n)}}{T_j} \right\rceil C_j$$

Same task as before with an extra term

Maybe pessimistic: A task may not suffer the maximum blocking

Problems with Basic Priority Inheritance

- Multiple blockings
  - A task can be blocked more than once on different semaphores
- Multiple inheritance
  - when considering nested resources, the priority can be inherited multiple times
- Deadlock
  - In case of nested resources, there can be a deadlock

Solution to Problems

- It is possible to avoid this situation by doing an off-line analysis
- Define the concept of resource ceiling
- Anticipate the blocking:
  - a task cannot lock a resource if it can potentially block another higher priority task later.

Ceilings

- The (static) *ceiling* of a resource is the (static) priority of the highest priority task that can access it:

$$\mathit{ceiling}(\mathrm{S}) = \max\{\, \mathit{pri}(\tau) \mid \tau \text{ uses S} \}$$

- The *system ceiling* at a particular time is the maximum of the ceilings of all resources that are locked at that time.

Priority Ceiling Protocols
Constraints:

- Order of locking / unlocking semaphores

  lock S1 .. lock S2 ... unlock S2 .. unlock S1
- Finite period of time in critical section
- Set of semaphores known in advance

Original Ceiling Priority Protocol

- Each task has a static default priority assigned (perhaps by the deadline monotonic scheme)
- Each resource has a static ceiling value defined, this is the maximum priority of the tasks that use it
- A task has a dynamic priority that is the maximum of its own static priority and any it inherits due to it blocking higher-priority tasks.

OCPP

- A task can only lock a resource if its dynamic priority is higher than the ceiling of any currently locked resource (excluding any that it has already locked itself)
- The locking of a first system resource is allowed

Original Ceiling Priority Protocol

- **Scheduling rule** :
  - At release time the current priority of every task is equal to its assigned priority.
  - The task remains at this priority except under the condition stated in priority-inheritance Rule;
  - Every task is scheduled pre-emptively and in a priority driven manner

Original Ceiling Priority Protocol

- **Allocation rule**: whenever a task T requests a resource R at time t one of two conditions occur:
  - R is held by another task. T's request fails and T is blocked
  - R is free
    - if T's priority p is higher than the current system (priority) ceiling p, R is allocated to T
    - otherwise
      - if T is holding the resources whose ceiling is equal to p, then R is allocated to T
      - otherwise T's request is refused and T is blocked.

Original Ceiling Priority Protocol

- **Priority-Inheritance Rule**: When T becomes blocked,
  - The task J that blocks T inherits the current priority p of T.
  - Task J executes at p until it releases every resource whose priority ceiling is equal to or higher than p.
  - At that time J’s priority reverts the value when it gained those resources.

Properties of OCPP

- Theorem
  - *A task can be blocked at most once by any resource or task.*
- Theorem
  - *The Priority Ceiling Protocol prevents deadlock*
    - Therefore, we can nest critical sections safely
- Corollary
  - *The maximum blocking time for a task is at most the length of one critical section*

Original Ceiling Priority Protocol: Problems

- The implementation is very complex, even more than Simple Priority Inheritance
  - Very little known implementations,
  - difficult to prove correctness of implementation
- Causes many context switches

Immediate Ceiling Priority Protocol

- This protocol is also known with the name of Stack Resource Policy
- The basic ideas are the following:
  - We anticipate the blocking even more
  - the task cannot even start executing if it is not guaranteed to take all resources

Immediate Ceiling Priority Protocol

- Properties:
  - Very simple implementation
  - A task blocks at most once before starting execution
  - The execution order is like a “stack”.

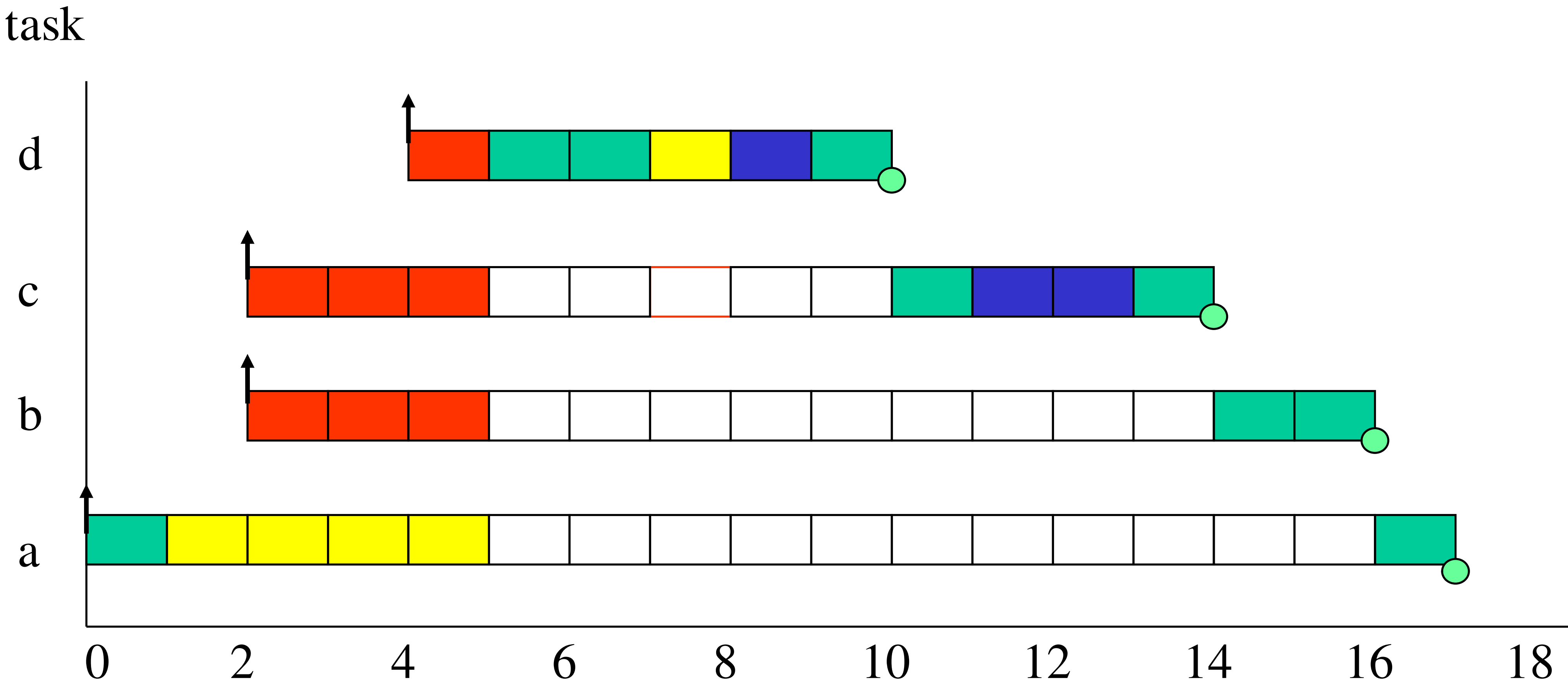
Immediate Ceiling Priority Protocol

- Same model as Ceiling Protocol but different run-time behaviour
- Run-time behaviour:
  - when task P wants to lock resource (semaphore etc) S, the task *immediately* sets its priority to the maximum of its current priority and the ceiling priority of S. When the task finishes with S it resets its priority to what it was before.

ICPP

- Each task has a static default priority assigned (perhaps by the deadline monotonic scheme).
- Each resource has a static ceiling value defined, this is the maximum priority of the tasks that use it.
- A task has a dynamic priority that is the maximum of its own static priority and the ceiling values of any resources it has locked
- Once the task starts actually executing, all the resources it needs must be free; if they were not, then some task would have an equal or higher priority and the task's execution would be postponed

ICPP Inheritance (ex #3)



Commentary

- We don't actually need to lock S because:
  - S can not be locked when P comes to lock it otherwise another task, Q, would be running with at least the same priority as P, and P would not be running
  - If P isn't running it could not try to lock S
- Because the inheritance is immediate P is blocked, if at all *before it starts running*. This is because if a lower priority task holds S it will be running at a priority at least as high as P

OCPP versus ICPP

- ICPP reduces the number of preemptions
- ICPP is very easy to be implemented
  - No need to do inheritance
  - No need to block tasks in semaphore queues
  - It makes it possible for all tasks to share the same stack

OCPP versus ICPP

- ICPP reduces the number of preemptions
- ICPP is very easy to be implemented
  - No need to do inheritance
  - No need to block tasks in semaphore queues
  - It makes it possible for all tasks to share the same stack