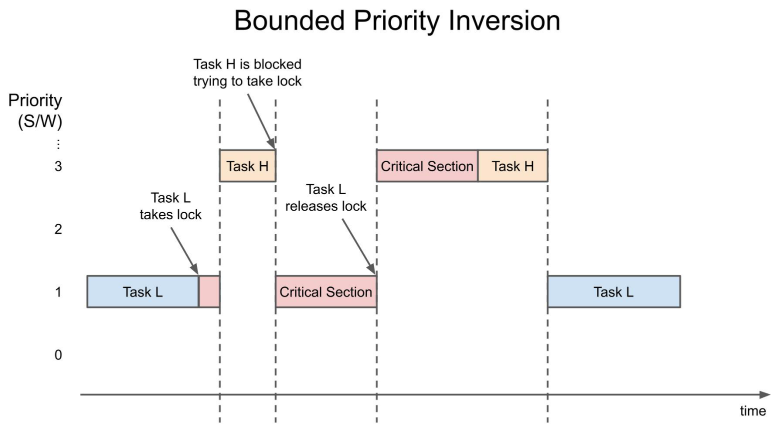
PART II: PRIORITY INVERSION

→ a bug when a high priority task is indirectly preempted by a low priority task

↳ eg: when a low priority task holds a mutex that a high priority task must wait for when executing

- Bounded priority inversion

  ↳ the length of time of inversion is bounded by how long the priority task is in the critical section (holding the lock)



Task H is blocked by lower priority task →

↓

priority of task H is directly inverted

- Unbounded priority inversion

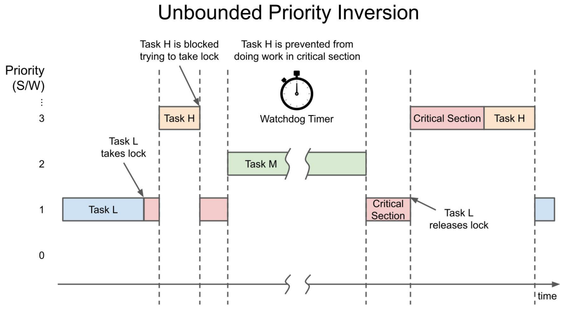
  ↳ medium priority task interrupts high priority task while it holds the lock

Task M can block task L for ANY amount of time

↓

because task L holds the lock

↳ hence it also blocks task H



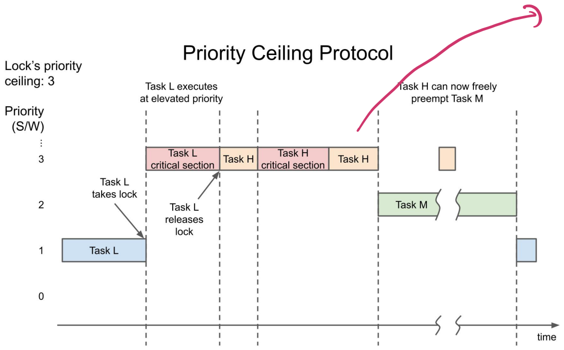
SOLUTION

① PRIORITY CEILING PROTOCOL

↳ when a task takes a lock, it's priority level is automatically boosted to that of the priority ceiling

→ priority ceiling is determined by the maximum priority of any tasks that needs to use the resource or lock

→ this means L&H runs together

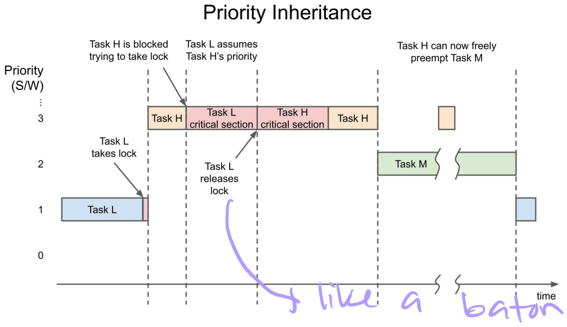


- Priority ceiling is 3
- Task L's priority is now the same as Task H's
  ↳ prevents Task M from running until H&L are done

② PRIORITY INHERITANCE

↳ boosting priority of a task holding a lock to that of the task trying to take the lock

Priority Inheritance



→ like a baton pass

- Task L's priority is only boosted when L tries to take the lock
  ↳ ∴ Task M cannot interrupt it

priority of task L drops once it releases the lock

This only solves unbounded priority inversion

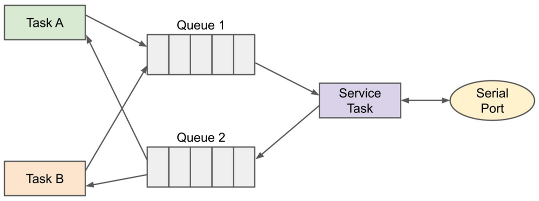
Bounded priority inversion → solved through good programming practise

→ keep critical section short

→ avoid using critical section or locking mechanism that can block a high priority task

→ use one task to control a shared resource
eg. using queues to receive messages from a task that handles the serial port

Using a Service Task



- this avoids using critical section for the serial port