Topic 09: Priority Inversion

Reading: Pages 342-343
Next reading: None

- Priority scheduling assigns a process a priority that reflects its importance. CPU is allocated to high priority processes first, so that a runnable high-priority process never waits for a low-priority one.

- Unfortunately, synchronization mechanisms such as semaphores also affect the order in which processes run, perhaps subverting the scheduling priorities.

- Both are trying to control the order in which processes run.

    - Synchronization mechanisms have the final say. If a process is blocked it cannot run no matter what its priority.

- What happens when a high-priority process tries to enter a critical section occupied by a low-priority process?

    - The high-priority process must wait for the low-priority to leave the critical section.

    - The processes' priorities have no effect.

    - This is called *priority inversion*.

  - Priority inversion can happen with any type of resource (not just the CPU), e.g. a high-priority process waits for a low-priority one to finish with the disk.

- One way to mitigate the problem is to make critical sections short, so that the high-priority processes do not wait long.
  - This isn't guaranteed to work in all situations, however.
- Example #1: processes P1, P2, and P3, of decreasing priority.
  - P3 enters critical section CS.
  - P2 becomes runnable and preempts P3.
  - P1 becomes runnable and preempts P2.
  - P1 tries to enter CS and is blocked.
  - P2 runs until it blocks. Note that P2 is not inside a critical section and may run a very long time. Meanwhile, P1 does not run, even though it is of higher priority and isn't even waiting for P2!
- Even worse, priority inversion can cause a process to busy-wait forever.
- Example #2: processes P1 & P2, of decreasing priority. Critical section CS is protected by a spin-lock.
  - P2 enters CS and acquires lock.
  - P1 becomes runnable and preempts P2.

    - P1 tries to enter CS and busy-waits.
    - P1 spins forever, and all processes of lower priority never run.

- Solution: *priority inheritance* or *priority donation*. When a process is inside a critical section, temporarily boost its priority to at least that of the highest-priority waiting process.
    - When a high-priority process waits on a critical section it temporarily donates its priority to the process in the critical section.
    - Low-priority process runs at high priority for a while.
    - Low-priority process clears critical section quickly.
- Complicates synchronization code, e.g. semaphore implementation must interact with scheduler to adjust priorities.
- In general, how does the system know who is waiting for what?
    - If a high-priority process is waiting on a semaphore, how does the system know which process will eventually V the semaphore so its priority can be boosted?
    - Probably need a more structured synchronization mechanism, e.g. locks, because then it's clear which process is in the critical section and therefore should have its priority boosted.
    - Still doesn't solve general synchronization problem of high-priority process waiting for low-priority to V.