Strict Priority Scheduler

In Pintos, each thread has a priority value ranging from 0 (`PRI_MIN`) to 63 (`PRI_MAX`). However, the current scheduler does not respect these priority values. You must modify the scheduler so that higher-priority threads always run before lower-priority threads (i.e. strict priority scheduling).

Note: In the case of a priority tie, you should schedule threads with the same priority value in a round robin fashion. The existing code already preempts threads once they run past their allotted time slice, so you won't have to worry about this part of the round robin implementation. If you would like a closer look to how this is implemented, take a look at `thread_tick`.

You must also modify the 3 Pintos synchronization primitives (lock, semaphore, condition variable), so that these shared resources prefer higher-priority threads over lower-priority threads.

Additionally, you must implement **priority donation** for Pintos locks. When a high-priority thread (A) has to wait to acquire a lock, which is already held by a lower-priority thread (B), we temporarily raise B's priority to A's priority. A scheduler that does not donate priorities is prone to the problem of **priority inversion** whereby a medium-priority thread runs while a high-priority thread (A) waits on a resource held by a low-priority thread (B). A scheduler that supports priority donation would allow B to run first, so that A, which has the highest priority, can be unblocked. Your implementation of priority donation must handle 1) donations from multiple sources, 2) undoing donations when a lock is released, and 3) nested/recursive donation.

A thread may set its own priority by calling `thread_set_priority` and get its own priority by calling `thread_get_priority`.

If a thread no longer has the highest "effective priority" (it called `thread_set_priority` with a low value or it released a lock), it must immediately yield the CPU to the highest-priority thread.

The actual priority scheduler does not require much complexity in and of itself; consider how extant operating systems implement this sort of scheduler if you're confused as to how to approach this in an efficient way.

1. Don't forget to implement `thread_get_priority`, which is the function that returns the current thread's priority. This function should take donations into account. You should return the effective priority of the thread.
2. A thread cannot change another thread's priority, except via donations. The function `thread_set_priority` only acts on the current thread.
3. If a thread no longer has the highest effective priority (e.g. because it released a lock or it called `thread_set_priority` with a lower value), it must immediately yield the CPU. If a lock is released, but the current thread still has the highest effective priority, it should not yield the CPU.

The priority donation component of this task will likely require some thought — it may be helpful to sketch out some scenarios on paper or on a whiteboard to see if your proposed system holds up.

1. You only need to implement priority donation for locks. Do not implement them for other synchronization variables (it doesn't make any sense to do it for semaphores or monitors anyway, since, for example, there is no notion of a unique "owner" of a semaphore). However, you need to implement priority scheduling for locks, semaphores, and condition variables. Priority scheduling is when you unblock the highest priority thread whenever a resource is released or a monitor is signaled.
2. A thread can only donate (directly) to 1 thread at a time, because once it calls `lock_acquire`, the donor thread is blocked.
3. Your implementation must handle nested donation: Consider a high-priority thread H, a medium-priority thread M, and a low-priority thread L. If H must wait on M and M must wait on L, then we should donate H's priority to L.
4. If there are multiple waiters on a lock when you call `lock_release`, then all of those priority donations must apply to the thread that receives the lock next.

You are not required to augment the scheduler for user threads; you can just let the scheduler treat all threads the same way, even if they belong to the same process. As a pathological example, if a user program A has 100 threads, and a user program B has only 1 thread, most of the CPU will be dominated by A's threads, and B's thread will be starved. You are not required to augment the scheduler to make this scenario more fair.

Copyright © 2022 CS 162 staff.