Some Notes on Thread Scheduling

COMP 321

Dave Johnson

1

A Thread Scheduler

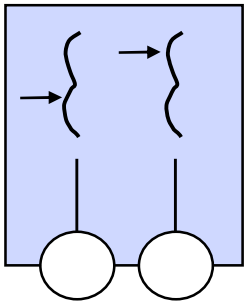
A thread scheduler has many things to manage and keep track of

- Giving each thread an (approximately) "fair" share of the CPU time
  - Thread context switching to divide the CPU time that the kernel gives you among the user-level threads for that process
- Managing threads that are blocked on I/O
  - Knowing which threads should get a share of CPU time
- Keeping track of and joining threads when requested
  - Know which threads have completed and not yet joined
  - Managing a join request for some thread
- Reclaiming thread resources after a threads complete
  - Example: free the thread's stack (it was automatically allocated at create)

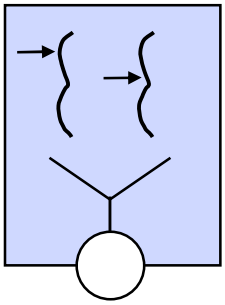
2

Kernel-Level Threads vs. User-Level Threads

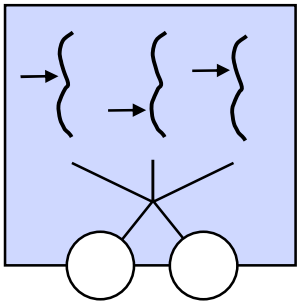
Can multiplex user-level threads on kernel-level level threads in many ways



One kernel-level thread per user-level thread



One kernel-level thread Multiple user-level threads



Many user-level threads on many kernel-level threads

Kernel-Level Threads vs. User-Level Threads

Can multiplex user-level threads on kernel-level level threads in many ways

- ***1:1 (kernel-level threading)*** – each user-level thread is mapped to a separate thread in the kernel
  - Kernel is in control of scheduling each thread, but expensive
- ***N:1 (user-level threading)*** – all user-level threads are mapped to a single thread in the kernel thread
  - Kernel doesn't even need to know that threading is being used
  - Scheduling of threads is done at the user level, much less expensive
- ***N:M (hybrid threading)*** – N user-level threads mapped to M kernel threads
  - Flexible dynamic user-level scheduling between user and kernel scheduling
  - Simple example: M could be the number of CPU cores

The Effect of a Kernel-Level Thread Blocking

A kernel-level thread blocks when doing something that the kernel blocks it for

- Example: a user-level thread starts some blocking I/O operation
  - The kernel-level thread currently running it is blocked by the kernel
  - So that kernel-level thread no longer available to run any user-level threads
- The other kernel-level threads handling that process's threads can still run
- But if there's only one kernel-level thread for that process's threads
  - Then all user-level threads in that process are effectively blocked
- Or if now all kernel-level threads for that process are blocked
  - Then again, all user-level threads in that process are effectively blocked

Blocking for I/O in a User-Thread is Very Bad

This kind of blocking can be avoided and needs to be avoided

- This would block the current kernel-level thread
  - No longer available to service other user-level threads for this process
  - And could (depending on setup) be the last kernel-level thread for it
- ***It is thus important to provide I/O multiplexing and non-blocking I/O within the threads library***
  - To make sure user-level threads don't block on I/O
  - And to enable the thread scheduler to wake up appropriate threads when they have pending I/O that is ready

Be Careful About Signals and Thread Scheduling

As we've seen, a signal can happen anytime, asynchronously

- As always, if you have a signal handler for that signal
  - Be careful what you do in that signal handler
  - Be careful when you should block and can unblock that signal
- This issue can particularly impact thread scheduling
  - A signal may relate to something that directly impacts the scheduler
  - A signal handler may need to read and/or modify some state used by the scheduler
  - For example, SIGPROF can be used to tell the scheduler to switch to running some other thread now
  - But that could occur as the scheduler is switching for some other reason, or as some new thread is being created or an existing thread terminated