6.S081: Scheduling

Adam belay <abelay@mit.edu>

Agenda today

- Previously: System calls, interrupts, page tables, and locks
- Today's focus: Thread scheduling
  - Uses timer interrupts (discussed before)

Why support multiple tasks?

- Time sharing: Many users/tenants
- Program structure: E.g., prime number sieve from Lab
- Parallel speedup on multicore hardware
- Workload consolidation: improves energy efficiency

The thread abstraction

- Simplifies programming with many tasks
- Each thread is an independent serial execution
- A thread contains a stack, registers, and a PC
- Threads expose concurrency to the OS
    - Across multiple cores, each core runs a thread
    - Kernel switches between threads on a core

Memory sharing w/ threads

- Xv6 kernel: threads share memory -> needs locks
- Xv6 user: one thread per process, no sharing
- Linux: multiple threads can run in a process, each share the same memory

Another approach: Events

- Event-driven programming
  - See libevent for an example
  - See epoll() for how Linux provides event notifications
  - Traditionally requires ugly spaghetti code
  - Modern languages (e.g., rust) make it cleaner
- Event systems are faster than threads on Linux
  - But fundamentally, both have similar performance

Thread design challenges

- How to interleave many threads on few cores
- Interleaving must be transparent
  - Programs shouldn’t be able to tell how when they are sharing cores
- Needs to save and restore thread state
- A “scheduler” decides which thread to run next
  - What is a thread never blocks or yields?

Threading design space

- **Preemptive vs. cooperative**: Does it pause running threads to run other threads?
- **Work conserving**: Does every core stay busy when there is enough work to run?
- **User v.s. kernel**: Are scheduling mechanisms (the scheduler, state saving/restoring, preemption, etc.) implemented in userspace or kernelspace?
  - Hybrid approaches are possible
  - User threads are significantly faster

Threading performance goals

- **Fairness**: Does each thread get an equal share of CPU time?
- **Latency**: How long does a runnable thread get delayed?
- **Tail/max latency**: What is the longest possible delay?
- **Overhead**: How expensive is a context switch?

A rich literature (e.g., queuing theory) offers proof of performance properties for many designs

Modern CPUs are even more interesting

- Schedulers control frequency, clock gating
- Some cores are faster than other cores
- Using fewer cores allows for a higher frequency (i.e., thermal envelope is constant)
- New security bugs restrict which apps can run on which cores at a time
- SMT can add more parallelism within a core

Preemptive scheduling

- Timer hardware on each core fires periodically
- Kernel uses these timer interrupts to grab control from busy (unyielding) threads
- Kernel saves state for running thread, switches to different thread
    - State restored later to resume (transparency)

Scheduler states

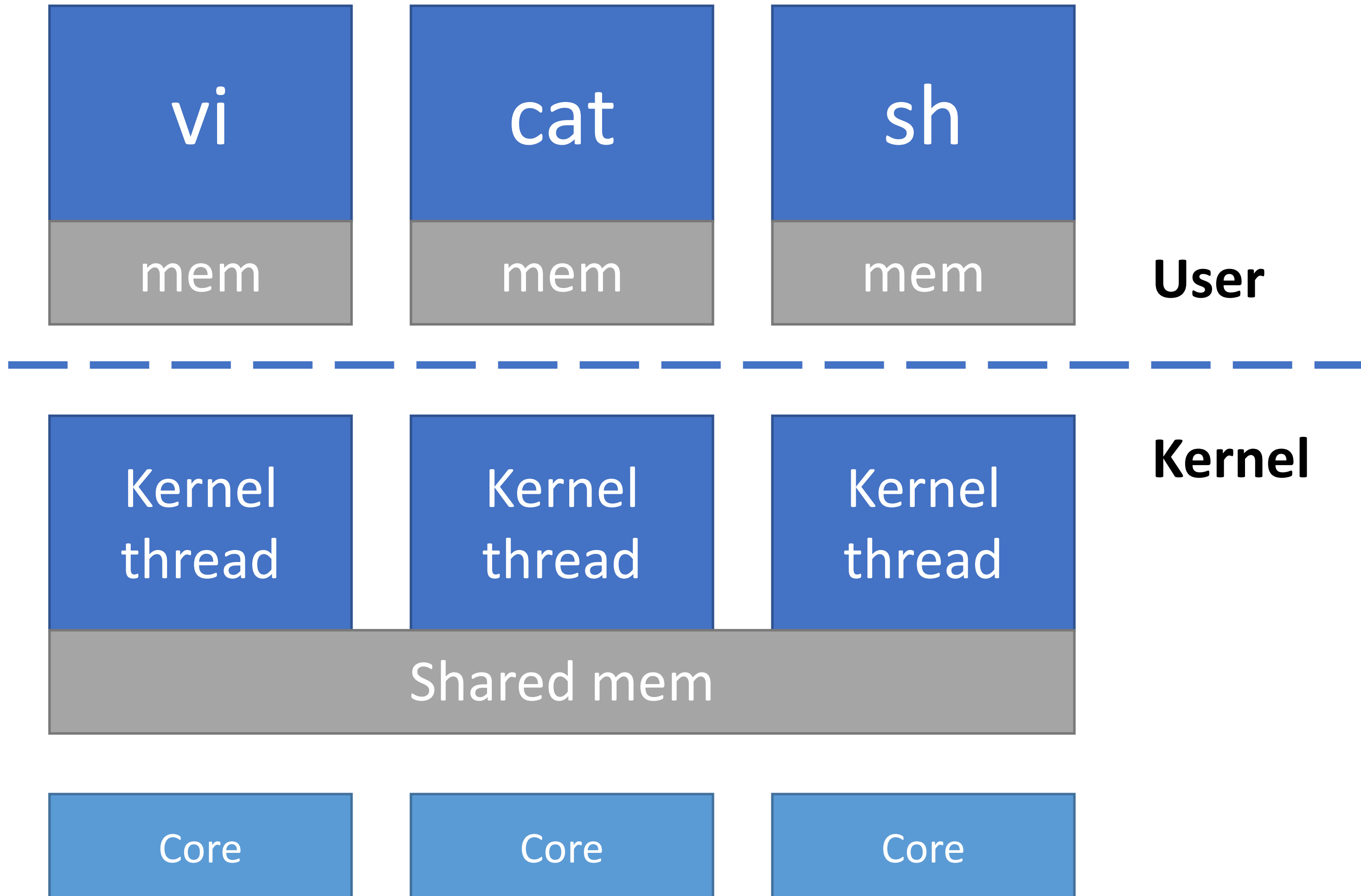
Most schedulers implemented as per-thread state machine:

- **Running**: actively using a core
- **Runnable**: able to run, but not using a core
- **Sleeping**: not able to run, not using a core

What to do with threads that aren’t “running”?

- Set aside state: registers, PC, memory
  - No need to save/restore memory, it won’t go anywhere
  - So in practice, need a save area for registers and PC
- Keep track of scheduler state of each thread
  - E.g., which threads are runnable?

Threading in xv6



Thread switching in xv6

- Switches among threads, interleaving on cores
- **Trapframe**: saved user registers
- **Context**: saved kernel registers
- Separate scheduler thread per core
- *Context switch*: term for switching from one thread to another

Thread switching in xv6

1. *User thread* **->** *kernel thread*: save user registers in trapframe
2. *Kernel thread* **->** *scheduler thread*: save kernel registers in context
3. *Scheduler thread* **->** *kernel thread*: restore kernel registers from context
4. *Kernel thread* **->** *user thread*: restore user registers from trapframe

More about scheduler threads

- One per core, each has stack + context
- Kernel thread switch to the core’s local scheduler thread
  - Which switches to another thread if one is RUNNABLE
  - Could be the same thread too (e.g., yield())
- Why a separate scheduler stack?
  - Makes it easier to handle exit()
  - Gets off kernel stack, allowing another core to run the last thread in parallel
- Policy: Scan process table in order until runnable thread is found

More details

- Each core is either running the scheduler thread, which spins waiting for a runnable thread, or is running exactly one user/kernel thread
- Each thread is either running on exactly one core, or its registers are saved in its context+trapframe
- Threads that aren’t running have a context that will resume from swtch()

Proc struct

- p->trapframe: holds saved user thread's registers
- p->context: holds saved kernel thread's registers
- p->kstack: points to the thread's kernel stack
- p->state: RUNNING, RUNNABLE, SLEEPING
- p->lock: protects state, and other things

Demo: Spin

Scheduler locking strategy

- yield() acquires the process' lock
- scheduler() code looks like normal acquire/release
  - In reality, scheduler acquires, yield() releases
  - then yield() acquires, scheduler releases
  - And so on…
- Very unusual: lock is released by different thread than the one that acquired it

Q: Why hold p->lock across swtch()?

- Could we instead drop p->lock right before swtch()?

Scheduler locking strategy

- p->lock makes multiple steps atomic
  1. p->state marked runnable
  2. Save registers in p->context
  3. Stop using p’s kernel stack

No other scheduler thread can start running p until these steps complete:

Q: Why does schedule() enable interrupts periodically?

Q: What is xv6's scheduling policy?

- i.e., how does xv6 decide what thread to run next?
- Is this a good policy?

Q: Why are locks forbidden to be help before calling yield()?

- Other than p->lock
- i.e., sched() checks that noff==1

Q: Why are locks forbidden to be help before calling yield()?

- Suppose P1 holds L1, then yields CPU
- P2 runs, tries acquire(L1)
- P2 spins waiting, interrupts are turned off so no timer will occur
- DEADLOCK: P2 won’t yield, P1 can’t execute

Could we get rid of separate, per-core scheduler thread?

- Would be faster, avoids one swtch() call

Could we get rid of separate, per-core scheduler thread?

- Would be faster, avoids one swtch() call
- Yes!
  - Scheduling loop could run on thread's kernel stack
  - What if thread is exiting?
  - What if another core wants to run the thread?
  - What if there are fewer threads than cores?
  - All this can be dealt with, but not easy. Give it a try!

Conclusion

- xv6 provides a shared-memory thread model for kernel code, and a single thread per process for user code
- Preemption via timer interrupts
- Transparency: saves and restores registers
- Locking and stacks are a tricky issue to get right
- Next lecture: mechanisms for threads to wait for each other