CS3210: Coordination (Sleep and Wakeup)

Tim Andersen

Administrivia

- Quiz 1 Handed Out (Pick up if you haven't received yours)
- Lab 4 Part A due March 10th

Today's plan

- Context switching (i.e., `swtch` and `sched`) in detail
- Sequence coordination
  - xv6: sleep & wakeup
- Challenges
  - Lost wakeup problem
  - Signals

Multiplexing

- Sleep and wakeup mechanism switches when a process
  - Waits for a device or pipe I/O to complete
  - Waits for a child to exit
  - Waits in the sleep system call
- xv6 periodically forces a switch
- Creates the illusion that each process has its own CPU

Implementation challenges

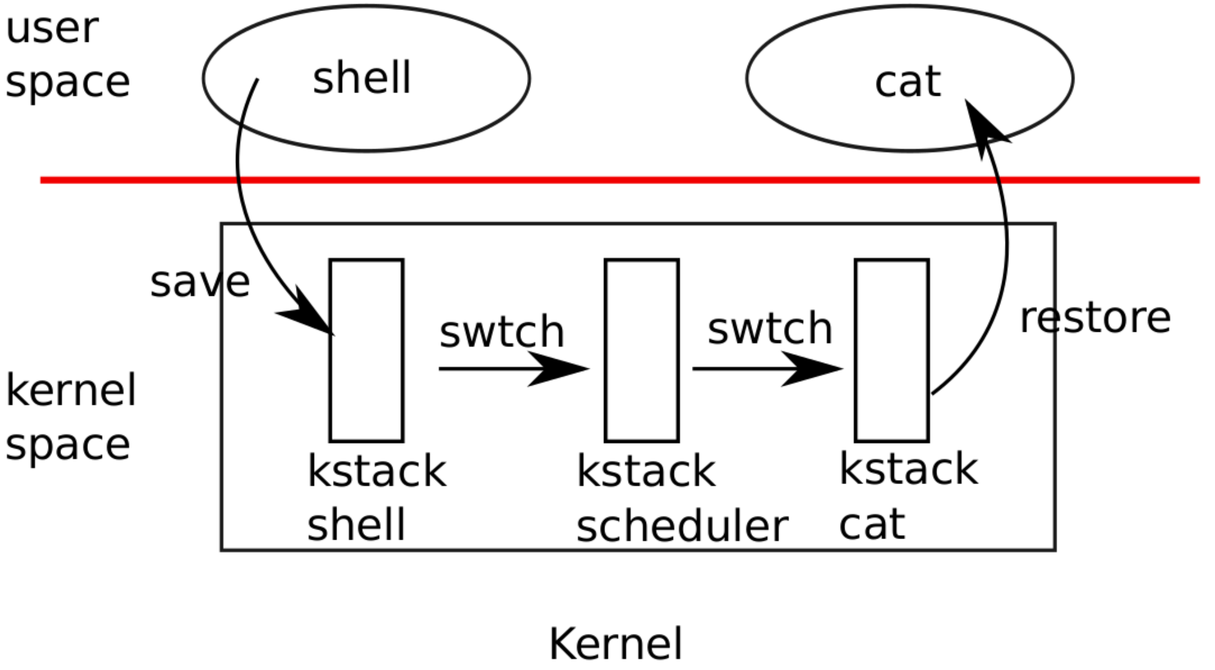
- Q: How to switch from one process to another?
  - A: Context switching
- Q: How to make context switching transparent?
  - A: Timer interrupts
- Q: How to switch among processes running concurrently?
  - A: Locking
- Q: How to coordinate processes?
  - A: Sleep on events (e.g., pipe, child exit)

Two kinds of context switch

1. From a process's kernel thread to CPU scheduler thread
2. From the scheduler thread to a process's kernel thread.

- xv6 never directly switches from user-space to user-space
  - user-kernel transition (system call or interrupt)
  - context switch to scheduler
  - context switch to new process's kernel thread
  - trap return

Big picture: switching



Context switching

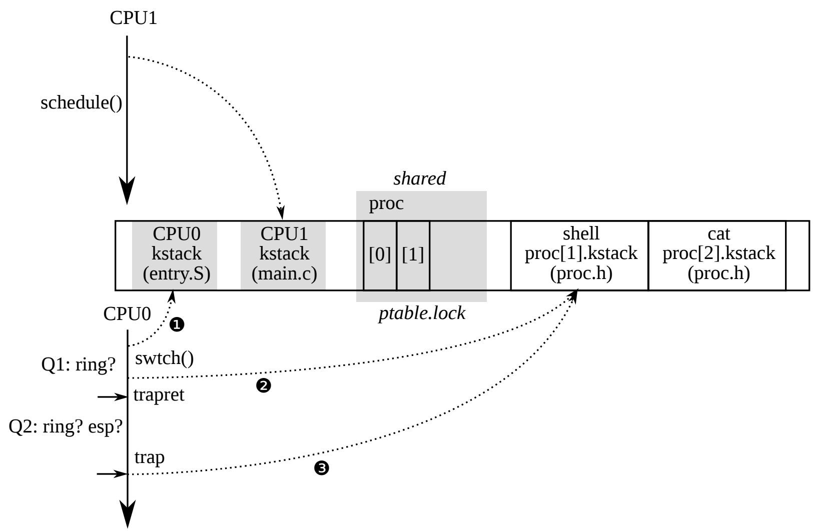
- Every xv6 process has its own kernel stack and register set
- Every CPU has its own scheduler thread
- Switching from one thread to another
  - Save and load CPU registers, including %esp, %eip.

`swtch`

```
void swtch(struct context**, struct context*);
```

- Doesn't know about threads just saves and loads sets of registers called contexts
- When time to give up CPU, kernel thread calls `swtch` to save itself and return to scheduler context
- Context is a `struct context*`, stored on the kernel stack
- CPU pushed onto stack and saves stack pointer to `*old`
- Copies `new` to `%esp`, pops previous registers, and returns

Switching overview: CPU perspective



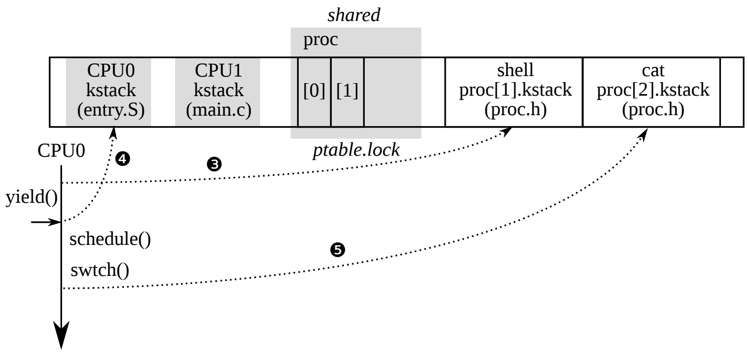
`yield`

- At the end of each interrupt, trap can call `yield`
- `yield` -> `sched` -> `swtch`
- Switches from `proc=>context` to `cpu=>scheduler`

`swtch`: Detailed Look

- Loads its arguments off the stack into `%eax` and `%edx` before it loses its arguments when it changes `%esp`
- Only callee-save registers saved
  - `%ebp`, `%ebx`, `%esi`, `%ebp`, and `%esp`
  - First four pushed, `%esp` saved as `*old`
- `%eip` was saved by the `call` instruction and is just above `%ebp`
- Moves pointer to new context into `%esp`
- Inverts sequence of steps to load context

Switching overview: CPU perspective



Show xv6 Code

- `swtch()`, `scheduler()`, `sched()`
- about `ptable.lock`

Scheduling

- Process giving up the CPU must
  - aquire `ptable.lock`
  - release any other locks
  - update its own state (e.g., RUNNABLE, SLEEPING)
  - call `sched`
- `yield`, `sleep`, and `exit` all do these steps

DEMO: sched

```
br sched
commands
p cpus[cpunum()].proc.pid
c
end
```

`ptable.lock`

- Held across calls to `swtch`
- Caller holding lock passes control to switched to code
- Needed because process state and context must be kept invariant across `swtch`
- Without lock, a different CPU might try to run a process after RUNNABLE but before kernel stack switch.
  - Result is two CPUs with same stack.

`sched` and `scheduler`

- Kernel thread always gives up in `sched` and switches to same location in scheduler
- Almost always switches to a process in `sched`.
- Thread switches follow a simple pattern between sched and scheduler
  - Coroutines
- Exception is `forkret` when process is first scheduled

Scheduler

- Loops over process table looking for RUNNABLE processes
- Finding one, sets current per-CPU process to proc
- Switches page table with `switchuvm`, marks as RUNNING, and calls `swtch`

Sequence coordination

- How to arrange for threads to wait for each other to do
  - e.g., wait for disk interrupt to complete
  - e.g., wait for pipe readers to make space in pipe
  - e.g., wait for child to exit
  - e.g., wait for block to use

Producer and Consumer Queue

- Queue allows one process to send a nonzero pointer to another process
- For only one sender and one receiver on different CPUs.
- Send loops until queue is empty then puts pointer p in the queue
- Recv loops until the queue is non-empty and takes the pointer out
- Both modify `q=>ptr`, but send only writes the pointer when zero
- Recv only writes when nonzero, so no lost updates

Strawman solution: spin

```
struct q { void *ptr; };

void* send(struct q *q, void *p) {
  while(q=>ptr != 0)
    ;
  q=>ptr = p;
}

void* recv(struct q *q) {
  void *p;
  while((p = q=>ptr) == 0)
    ;
  q=>ptr = 0;
  return p;
}
```

Strawman solution: spin

- Q: cpu0 send(), cpu1 recv()?
- Q: cpu0 send(), cpu1 send()?
- Q: cpu0 recv(), cpu1 send()?
- Q: cpu0 recv(), cpu1 recv()?
- Q: problem?

Better solution: primitives for coordination

- Sleep & wakeup (xv6)
- Condition variables (e.g., `pthread_cond`)
- Barriers (next tutorial)

Sleep & wakeup

- `sleep(chan)`
  - sleeps on a "channel", an address to name the condition we are sleeping on
- `wakeup(chan)`
  - wakeup wakes up all threads sleeping on `chan`
  - this may wake up more than one thread

Attempt 1: sleep & wakeup

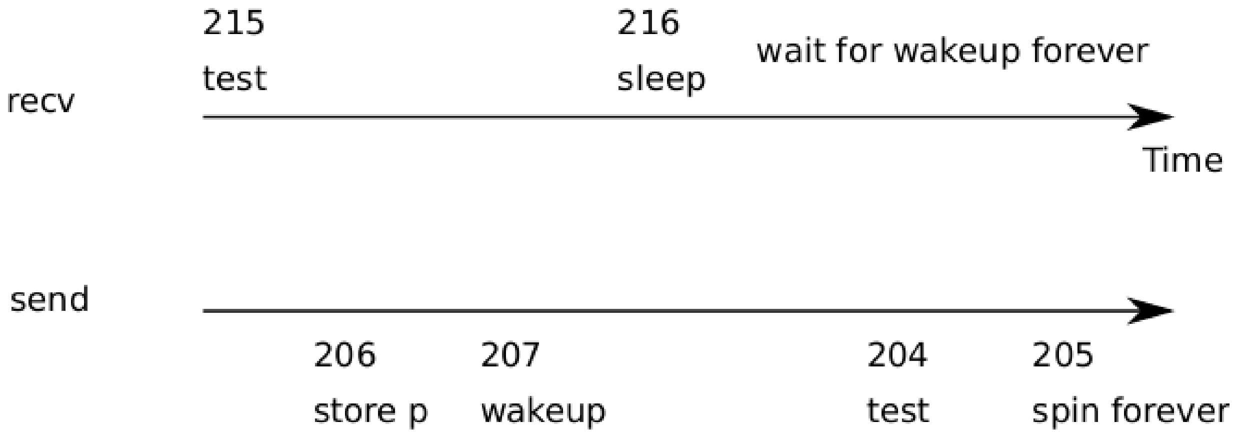
```
void* send(struct q *q, void *p) {
  while(q=>ptr != 0)
    ;
  q=>ptr = p;
  wakeup(q);  /* Q? */
}

void* recv(struct q *q) {
  void *p;
  while((p = q=>ptr) == 0)
    sleep(q);  /* Q? */
  q=>ptr = 0;
  return p;
}
```

Strawman solution: spin

- Q: cpu0 send(), cpu1 recv()?
- Q: cpu0 send(), cpu1 send()?
- Q: cpu0 recv(), cpu1 send()?
- Q: cpu0 recv(), cpu1 recv()?
- Q: problem? (hint: concurrently run `while` in `send/recv`)

Lost wakeup problem



Attempt1: fixing the lost wakeup problem

- Q: how to atomically run the code (checking and sleeping)?

```
    while((p = q=>ptr) == 0)
      sleep(q);
```

Attempt1: fixing the lost wakeup problem

- Let's use a spinlock

```
struct q {
  struct spinlock lock;
  void *ptr;
};
```

Attempt1: fixing the lost wakeup problem

```
void* send(struct q *q, void *p) {
  acquire(&q=>lock);
  while(q=>ptr != 0)
    ;
  q=>ptr = p;
  wakeup(q);
  release(&q=>lock);
}

void* recv(struct q *q) {
  void *p;
  acquire(&q=>lock);
  while((p = q=>ptr) == 0)
    sleep(q);
  q=>ptr = 0;
  release(&q=>lock);
  return p;
}
```

Problems?

- Q: cpu0 send(), cpu1 recv()?
- Q: cpu0 send(), cpu1 send()?
- Q: cpu0 recv(), cpu1 send()?
- Q: cpu0 recv(), cpu1 recv()?

Attempt2: releasing the lock when sleeping

```
void* send(struct q *q, void *p) {
  acquire(&q=>lock);
  while(q=>ptr != 0)
    ;
  q=>ptr = p;
  wakeup(q);
  release(&q=>lock);
}

void* recv(struct q *q) {
  void *p;
  acquire(&q=>lock);
  while((p = q=>ptr) == 0)
    sleep(q, &q=>lock);
  q=>ptr = 0;
  release(&q=>lock);
  return p;
}
```

Problems?

- Q: cpu0 send(), cpu1 recv()?
- Q: cpu0 send(), cpu1 send()?
- Q: cpu0 recv(), cpu1 send()?
- Q: cpu0 recv(), cpu1 recv()?
- We need a similar treatment for `send()` (i.e., `sleep()`)

Code

- `sleep()`, `wakeup()`
- about: `ptable.lock`

Summary: sleep takes a lock as argument

- Sleeper and wakeup acquires locks for shared data structure
- `sleep()` holds the lock until after it has `ptable.lock`
- Once it has `ptable.lock`, no wakeup can come in before it sets state to sleeping -> no lost wakeup problem
- Requires that sleep takes a lock argument!

Case study: ide (blockio)

- Device I/O is too slow to just spin (wait) for its competition
- `bread(b)` -> `iderw(b)`
  - it waits (sleep) until the requests block is ready
- `trap()` -> `ideintr()`
  - it notifies (wakeup) the waiter

Example: iderw

- code: `iderw()` (sleeper), `ideintr()` (wakeup)
- Q: wakeup cannot get lock until sleeper is already to at sleep, why a loop around sleep?

```
      // Wait for request to finish.
      while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
        sleep(b, &idelock);
      }
```

Another example: pipe

- What is the race if sleep didn't take p->lock as argument?

Many primitives in literature to solve lost-wakeup problem

- Counting wakeup&sleep calls in semaphores
- Pass locks as an extra argument in condition variables (as in sleep)

References

- Intel Manual
- UW CSE 451
- OSPP
- MIT 6.828
- Wikipedia
- The Internet