Concurrent Systems

3D4 ⟷ CS2016

Operating Systems

Andrew Butterfield
ORI.G39, Andrew.Butterfield@scss.tcd.ie

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

with thanks to Mike Brady

The xv6 operating system

- Developed by MIT for teaching
  - Keeps it simple, rather than efficient!
  - https://pdos.csail.mit.edu/6.828/2012/xv6.html (or Google 'xv6')
- Open source
  - github.com/mit-pdos/xv6-public.git
- Code Listing
  - https://pdos.csail.mit.edu/6.828/2018/xv6/xv6-rev11.pdf
- Accompanying 'Book'
  - https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf

The xv6 ‘book’

- V. Good introduction to OS design - Chapter 0 is well worth a read!
- Each chapter handles a topic: problem, solution, code commentary

The function `swtch` performs the saves and restores for a thread switch. `swtch` doesn’t directly know about threads; it just saves and restores register sets, called *contexts*. When it is time for a process to give up the CPU, the process’s kernel thread calls `swtch` to save its own context and return to the scheduler context. Each context is represented by a `struct context*`, a pointer to a structure stored on the kernel stack involved. `Swtch` takes two arguments: `struct context **old` and `struct context *new`. It pushes the current registers onto the stack and saves the stack pointer in `*old`. Then `swtch` copies `new` to `%esp`, pops previously saved registers, and returns.

Let’s follow a user process through `swtch` into the scheduler. We saw in Chapter 3 that one possibility at the end of each interrupt is that `trap` calls `yield`. `Yield` in turn calls `sched`, which calls `swtch` to save the current context in `proc->context` and switch to the scheduler context previously saved in `cpu->scheduler` (2822).

The xv6 code-listing

```
2800 // Enter scheduler.  Must hold only ptable.lock
2801 // and have changed proc->state. Saves and restores
2802 // intena because intena is a property of this
2803 // kernel thread, not this CPU. It should
2804 // be proc->intena and proc->ncli, but that would
2805 // break in the few places where a lock is held but
2806 // there's no process.
2807 void
2808 sched(void)
2809 {
2810   int intena;
2811   struct proc *p = myproc();
2812
2813   if(!holding(&ptable.lock))
2814     panic("sched ptable.lock");
2815   if(mycpu()->ncli != 1)
2816     panic("sched locks");
2817   if(p->state == RUNNING)
2818     panic("sched running");
2819   if(readeflags()&FL_IF)
2820     panic("sched interruptible");
2821   intena = mycpu()->intena;
2822   swtch(&p->context, mycpu()->scheduler);
2823   mycpu()->intena = intena;
2824 }
```

- Smart way to do listing:
  - 2-columns of 50 lines, so line-numbers have page number
- Each chapter handles a topic:
  - problem,
  - solution,
  - code commentary

The xv6 scheduler (1)

- Multicore!
  - we show an overview here (inner loop body not shown '…')
- Key variables:
  - Pointer `c` will point to selected process
  - Pointer `p` is used to walk the process table (`ptable`) from start to finish
  - Note the use of locking for the process table!
- Book p63, Listings 2750-2791

```
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{ struct proc *p;
  struct cpu *c = mycpu(); c->proc = 0;
  for(;;){
    sti();   // Enable interrupts on this processor.
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    ...
    }
    release(&ptable.lock);
  }
}
```

The xv6 scheduler (2)

- We skip non-runnable processes
  - We choose the next runnable process
    - Round-Robin scheduler
    - Simple
- Prep. for running it:
  - Mark as running, and set processor mode to 'user' (`switchuvm`)
- Running it: call `swtch`
- When process returns to scheduler:
  - set processor mode to 'kernel'
  - set selected process pointer to zero.

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;

  // Switch to chosen process.  It is the process's job
  // to release ptable.lock and then reacquire it
  // before jumping back to us.
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING;

  swtch(&(c->scheduler), p->context);
  switchkvm();

  // Process is done running for now.
  // It should have changed its p->state before coming back.
  c->proc = 0;
}
```

The xv6 process table (1)

- Table has a ‘spinlock’ used to manage access
- It has an array of `struct proc`
    - contains ‘metadata’ for each process.
- Lots of headers
    - At a guess, perhaps `proc.h` is relevant?
- Listings 2400-2412

```c
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "x86.h"
#include "proc.h"
#include "spinlock.h"

struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

The xv6 process table (II)

- Complicated!
- Most not relevant to scheduling, though.
- Relevant : `sz pid foil cwd` (?)
- Listings 2334-2351

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

```
enum procstate { UNUSED, EMBRYO, SLEEPING
               , RUNNABLE, RUNNING, ZOMBIE };
```

The xv6 process context

- A C struct containing the relevant 6 registers, exactly as they appear on the stack
- Listings 2316-2332

```
// Saved registers for kernel context switches.
// Don't need to save all the segment registers (%cs, etc),
// because they are constant across kernel contexts.
// Don't need to save %eax, %ecx, %edx, because the
// x86 convention is that the caller has saved them.
// Contexts are stored at the bottom of the stack they
// describe; the stack pointer is the address of the context.
// The layout of the context matches the layout of the stack in swtch.S
// at the "Switch stacks" comment. Switch doesn't save eip explicitly,
// but it is on the stack and allocproc() manipulates it.
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

xv6 context switching

- Assembly Language!
  - x86
- Switches out a process or the kernel
- Switches in the kernel or a process
- A C prototype is provided
- Listings 3050-3078

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

xv6 and CS2016/3D4

- Full use of xv6 requires
  - installing an emulator called Qemu, with MIT 'tweaks'
  - being proficient with the gdb debugger
  - installing cross-compilers if not using Unix
  - building familiarity over time
- Our use:
  - we shall abstract out the scheduler
  - provide a test harness program to exercise it
  - so you can improve the simple algorithm