Scheduling and Context switching in xv6

Mythili Vutukuru

CSE, IIT Bombay

Context switching in xv6

- Every CPU has a scheduler thread (special OS process that runs scheduler code)
- Scheduler goes over list of processes and switches to one of the runnable ones
- The special function “swtch” performs the actual context switch from scheduler thread to user process

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    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;
    }
    release(&ptable.lock);

  }
}
```

Scheduler and sched

- Scheduler switches to user process in "scheduler" function
- User process switches to scheduler thread in the "sched" function
- The function "swtch" called to context switch from user process to special scheduler process
- Scheduler process picks next process and the cycle repeats

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

When does user process call sched?

- Yield: Timer interrupt occurs, process has run enough, gives up CPU
- Exit: Process has called exit, sets itself as zombie, gives up CPU
- Sleep: Process has performed a blocking action, sets itself to sleep, gives up CPU

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

```
  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

```
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();

```

struct context

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

- In both scheduler and sched functions, the function “swtch” switches between two “contexts”
- Context structure: set of registers to be saved / restored when switching from one process to another
  - EIP where the process stopped execution, so that it can resume from same point when it is scheduled again in future
  - And a few more registers (why not all? more later)
- Context is pushed onto kernel stack, and pointer to the structure is stored in struct proc (p->context)

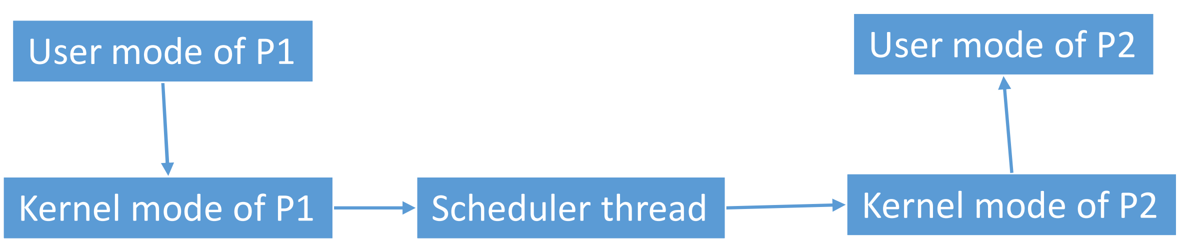
Context structure vs. trap frame in xv6

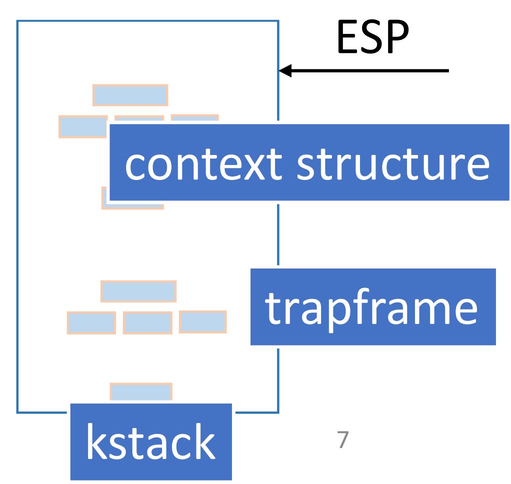
- Struct proc stores pointers to two structures on kernel stack
  - Trapframe is saved when CPU switches to kernel mode (e.g., PC in trapframe is PC value when syscall was made in user code)
  - Context structure is saved when process switches to another process (e.g., PC value when swtch function is invoked)
  - Both reside on kernel stack, struct proc has pointers to both
  - Example: Process has timer interrupt, saves trapframe on kstack, then context switch, saves context structure on kstack

```
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
```

Summary of context switching in xv6

- What happens during context switch from process P1 to P2?
  - P1 goes to kernel mode and gives up CPU (timer interrupt or exit or sleep)
  - P1 switches to CPU scheduler thread
  - Kernel stack of P1 has context structure and trap frame below it
  - Scheduler thread finds runnable process P2 and switches to it
  - P2 had given up CPU after saving context on its kernel stack in the past, so its kernel stack also has context structure and trap frame
  - P2 restores context structure, resumes in kernel mode
  - P2 returns from trap to user mode





swtch function

- Save registers in context structure on kernel stack of old process
- Switches ESP to context structure of new process
- Pops registers from new context structure
- CPU now has context of new process

```
# 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
```

Arguments to swtch function

- Both CPU thread and process maintain a context structure pointer variable (struct context *)
- swtch takes two arguments: address of old context pointer to switch from, new context pointer to switch to
  - When invoked from scheduler: address of scheduler's context pointer, process context pointer

```
2781          swtch(&(c->scheduler), p->context);
```

  - When invoked from sched: address of process context pointer, scheduler context pointer

```
2822    swtch(&p->context, mycpu()->scheduler);
```

  - Understand why the first argument is address and second is not

Why save and restore only some registers?

- What is on the kernel stack when a process/thread has just invoked the swtch? Caller save registers and return address (EIP)
- What does swtch do?
  - Push remaining (callee save) registers on old kernel stack
  - Save pointer to this context in old process PCB
  - Switch ESP from old kernel stack to new kernel stack
  - ESP now points to saved context of new process
  - Pop callee-save registers from new stack
  - Return from function call (pops return address, caller save registers)

swtch function code explanation

- When swtch function call is made, old kernel stack has return address (eip) and arguments to swtch (address of old context pointer, new context pointer)
- Store address of old context pointer into eax
- Store value of new context pointer into edx
- Push callee save registers on kernel stack of old process
- Top of stack esp now points to complete context structure of old process
- Update old context pointer (eax) to point to updated context
- Switch stacks: Copy new context pointer from edx to esp
- Pop registers from new context structure
- Return from swtch in new process

What about new processes?

- The context switching code in xv6 restores context from kernel stack of a process and resumes execution where process stopped earlier
- But what if a process has never run before? Where will newly forked process resume execution when it is switched in by scheduler?
- Kernel stack of new processes (artificially created context structure and trap frame) setup in such a way that
  - EIP of function where it has to start is saved in context structure, so that it appears that process was switched out at the location where we want it to resume in kernel mode
  - Trap frame copied from parent, so it resumes in user mode just after fork
  - Process resumes execution in kernel mode, returns from trap to user space

xv6: fork system call implementation

```c
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

allocproc (1)

- Find unused entry in ptable, mark is as embryo
  - Marked as runnable after process creation completes
- New PID allocated
- New memory allocated for kernel stack, stack pointer points to bottom of stack

```
// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and initialize
// state required to run in the kernel.
// Otherwise return 0.
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  release(&ptable.lock);

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

```

allocproc (2)

- Leave space for trapframe (copied from parent)
- Push return address of “trapret”
- Push context structure, with eip pointing to function “forkret”
- Why? When new process scheduled, begins execution at forkret, then returns to trapret, then returns from trap to userspace
- Hand-crafted kernel stack to make it look like process had a trap and context switch
  - Scheduler can switch this process in like others

```
  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;

  return p;
}
```

Forking new processes: summary

- Fork creates new process (PCB, PID, kernel stack) via allocproc
- Parent memory and file descriptors copied
- Trap frame of child copied from that of parent
  - Result: child returns from trap to exact line of code as parent
  - Only return value of system call in eax is changed, so parent and child have different return values from fork
- State of new child set to runnable, so scheduler thread will context switch to child process sometime in future
- Parent returns normally from trap/system call
- Child runs later when scheduled (forkret, trapret) and returns to user space like parent process

Init process creation

- Init process: first process created by xv6 after boot up
  - This init process forks shell process, which in turn forks other processes to run user commands
  - The init process is the ancestor of all processes in Unix-like systems
- After init, every other process is created by the fork system call, where a parent forks/spawns a child process
- The function “allocproc” called during both init process creation and in fork system call
  - Allocates new process structure, PID etc
  - Sets up the kernel stack of process so that it is ready to be context switched in by scheduler

Init process creation

- Alloc proc creates new process
  - When scheduled, it runs function forkret, then trapret
- Trapframe of process set to make process return to first instruction of init code (initcode.S) in userspace
- The code “initcode.S” simply performs “exec” system call to run the init user program

```c
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();

  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  // this assignment to p->state lets other cores
  // run this process. the acquire forces the above
  // writes to be visible, and the lock is also needed
  // because the assignment might not be atomic.
  acquire(&ptable.lock);

  p->state = RUNNABLE;

  release(&ptable.lock);
}
```

Init user program

- Init program opens STDIN, STDOUT, STDERR files
  - Inherited by all subsequent processes as child inherits parent's files
- Forks a child, execs shell executable in the child, waits for child to die
- Reaps dead children (its own or other orphan descendants)

```
// init: The initial user-level program

#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

char *argv[] = { "sh", 0 };

int
main(void)
{
  int pid, wpid;

  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0);  // stdout
  dup(0);  // stderr

  for(;;){
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
      printf(1, "zombie!\n");
  }
}
```