



## LECTURE 9: CONTEXT SWITCHING IN XV6

Context Switching- pg 61-64 of  
xv6Book

# Context switching in xv6

- Every CPU has a scheduler thread (special process that runs scheduler code)
- Scheduler goes over list of processes and switches to one of the runnable ones
- After running for some time, the process switches back to the scheduler thread, when:
  - Process has terminated ✓
  - Process needs to sleep (e.g., blocking read system call) ✓
  - Process yields after running for long (timer interrupt) ✓
- Scheduler thread runs its loop and picks next process to run, and the story repeats
- Context switch only happens when process is already in kernel mode.
  - Example: P1 running, timer interrupt, P1 moves to kernel mode, switches to scheduler thread, scheduler switches to P2, P2 returns to user mode

# Scheduler and sched

- Scheduler switches to user process in “scheduler” function
- User process switches to scheduler thread in the “sched” function (invoked from exit, sleep, yield)

```
2757 void
2758 scheduler(void)
2759 {
2760     struct proc *p;
2761     struct cpu *c = mycpu();
2762     c->proc = 0;
2763
2764     for(;;){
2765         // Enable interrupts on this processor.
2766         sti();
2767
2768         // Loop over process table looking for process to run.
2769         acquire(&ptable.lock);
2770         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2771             if(p->state != RUNNABLE)
2772                 continue;
2773
2774             // Switch to chosen process. It is the process's job
2775             // to release ptable.lock and then reacquire it
2776             // before jumping back to us.
2777             c->proc = p;
2778             switchvm(p);
2779             p->state = RUNNING;
2780
2781             swtch(&(c->scheduler), p->context); switching process
2782             switchkvm();
2783
2784             // Process is done running for now.
2785             // It should have changed its p->state before coming back.
2786             c->proc = 0;
2787         }
2788         release(&ptable.lock);
2789     }
2790 }
2791 }
```

```
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 }
```

*switching to scheduler.*

# Who calls 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

```
2826 // Give up the CPU for one scheduling round.
2827 void
2828 yield(void)
2829 {
2830     acquire(&ptable.lock);
2831     myproc()->state = RUNNABLE;
2832     sched();
2833     release(&ptable.lock);
2834 }
```

```
2862     // be run from main().
2863     first = 0;
2864     iinit(ROOTDEV);
2865     initlog(ROOTDEV);
2866 }
```

```
2894 // Go to sleep.
2895 p->chan = chan;
2896 p->state = SLEEPING;
2897
2898 sched();
2899
```

# struct context

Context  
structure

```
2326 struct context {  
2327     uint edi;  
2328     uint esi;  
2329     uint ebx;  
2330     uint ebp;  
2331     uint eip;  
2332 };
```

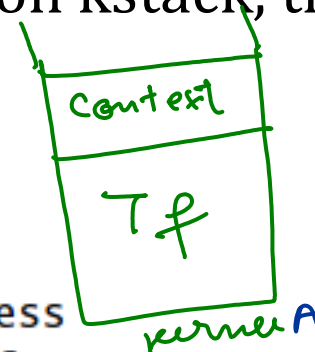
- In both scheduler and sched functions, the function “*switch*” switches between two “*contexts*”
- Context structure: set of registers to be saved when switching from one process to another
  - We must save “*eip*” where the process stopped execution, so that it can resume from same point when it is scheduled again in future
- Context is pushed onto kernel stack, struct proc maintains a pointer to the context structure on the stack (*p->context*)

# Context structure vs. trap frame

- Trapframe (p->tf) also contains a pointer to some register state stored on kernel stack of a process. What is the difference?
  - Trapframe is saved when CPU switches to kernel mode (e.g., eip in trapframe is eip value where syscall was made in user code)
  - Context structure is saved when process switches to another process (e.g., eip value when swtch is called) (imp)
  - Both reside on kernel stack, struct proc has pointers to both
  - Example: P1 has timer interrupt, saves trapframe on kstack, then calls swtch, saves context structure on kstack

```
2342  int pid;  
2343  struct proc *parent;  
2344  struct trapframe *tf;  
2345  struct context *context;
```

```
// Process ID  
// Parent process  
// Trap frame for current syscall  
// swtch() here to run process
```





# swtch function (1)

- 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);

*Handwritten annotations:*  
- Red arrow points to &(c->scheduler) with label "address".  
- Blue arrow points to c->scheduler with label "old context pointer".  
- Blue arrow points to p->context with label "new context pointer".

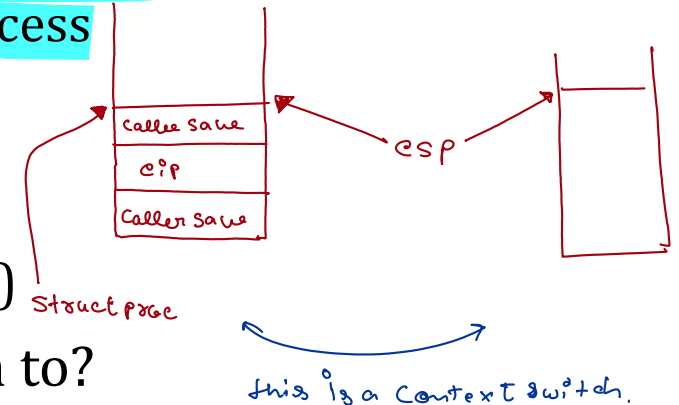
- When invoked from sched: address of process context pointer, scheduler context pointer

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

*Handwritten annotations:*  
- Red arrow points to &p->context with label "scheduler's context pointer".

# swtch function (2)

- What is on the kernel stack when a process/thread has just invoked the swtch?
  - Caller save registers (refer to C calling convention)
  - Return address (eip)
- What does swtch do?
  - Push remaining registers on old kernel stack (only callee save registers need to be saved)
  - Save pointer to this context into context structure pointer of old process
  - 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 *Just like simple function call.*
  - Return from function call (pops return address, caller save registers)
- What will swtch find on new kernel stack? Where does it return to?
  - Whatever was pushed when the new process gave up its CPU in the past
- Result of swtch: we switched kernel stacks from old process to new process, CPU is now executing new process code, resuming where the process gave up its CPU by calling swtch in the past





# swtch function (3)

- When swtch function call is made, **kernel stack of old process already has** (reading from top):  
eip, arguments to swtch (address of old context pointer, new context pointer)
- Store address of old context pointer into eax
  - Address of struct context \* variable in eax
- Store value of new context pointer into edx
  - edx points to new context structure
- Push callee save registers on kernel stack of old process (eip, caller save already present)
- Top of stack esp now points to complete context structure of old process. Go to address saved in eax (old context pointer) and rewrite it to point to updated context of old process
  - struct context \* in struct proc is updated
- Switch stacks: Copy new context pointer stored in edx (top of stack of new process) into esp
  - CPU now on stack of new process
- Pop registers from new context structure, and return from swtch in new process
  - CPU now running new process code

```
3050 # Context switch
3051 #
3052 #   void swtch(struct context **old, struct context *new);
3053 #
3054 # Save the current registers on the stack, creating
3055 # a struct context, and save its address in *old.
3056 # Switch stacks to new and pop previously-saved registers.
3057
3058 .globl swtch
3059 swtch:
3060     movl 4(%esp), %eax
3061     movl 8(%esp), %edx
3062
3063     # Save old callee-saved registers
3064     pushl %ebp
3065     pushl %ebx
3066     pushl %esi
3067     pushl %edi
3068
3069     # Switch stacks
3070     movl %esp, (%eax)
3071     movl %edx, %esp
3072
3073     # Load new callee-saved registers
3074     popl %edi
3075     popl %esi
3076     popl %ebx
3077     popl %ebp
3078     ret
```

# 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)
  - P2 is another process that is ready to run (it had given up CPU after saving context on its kernel stack in the past, but is now ready to run)
  - P1 switches to CPU scheduler thread
  - Scheduler thread finds runnable process P2 and switches to it
  - P2 returns from trap to user mode
- Process of switching from one process/thread to another
  - Save all register state (CPU context) on kernel stack of old process
  - Update context structure pointer of old process to this saved context
  - Switch from old kernel stack to new kernel stack
  - Restore register state (CPU context) from new kernel stack, and resume new process



Thank You