

Context switching

Mythili Vutukuru
CSE, IIT Bombay

Why switch between processes?

- Sometimes when OS is in kernel mode, it cannot return back to the same process that was running in user mode before
 - Process has exited or must be terminated (e.g., segfault)
 - Process has made a blocking system call
- Sometimes, the OS does not want to return back to the same process
 - The process has run for too long
 - Must timeshare CPU with other processes
- In such cases, OS performs a context switch from one process to another
 - Switch from kernel mode of one process to kernel mode of another
 - OS scheduler decides which process to run next and switches to it

OS scheduler

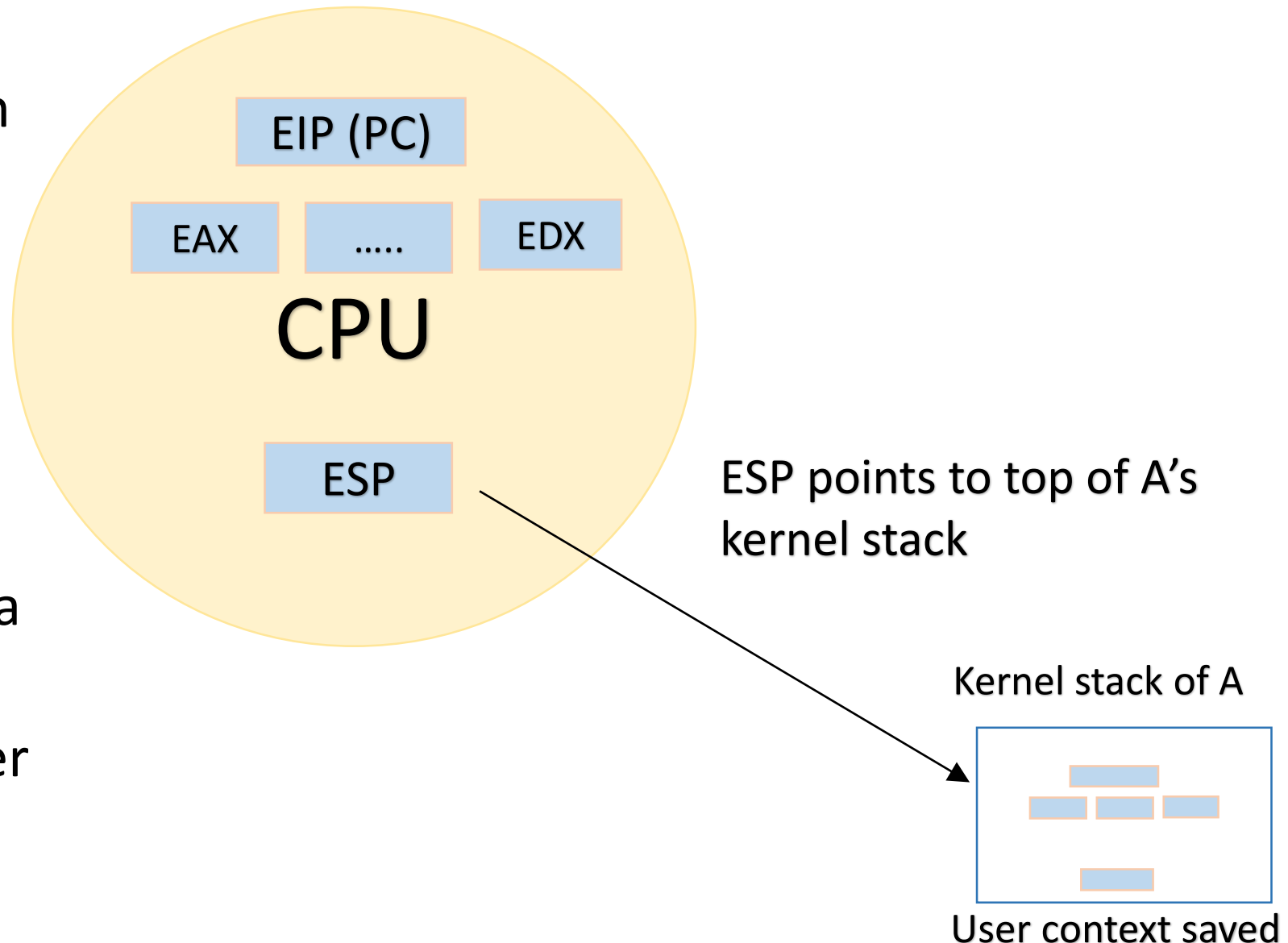
- OS maintains list of all active processes (PCBs) in a data structure
 - Processes added during fork, removed after clean up in wait
- OS **scheduler** is special code in the OS that periodically loops over this list and picks processes to run
- Basic outline of scheduler code
 - When invoked, **save context** of currently running process in its PCB
 - Loop over all **ready/runnable** processes and identify a process to run next
 - **Restore context** of new process from PCB and get it to run on CPU
 - Repeat this process as long as system is running

Scheduling and context switching

- OS scheduling involves two tasks
 - Policy to pick which process to run (next lecture)
 - Mechanism to switch to that process (this lecture)
- Non preemptive (cooperative) schedulers are polite
 - Switch only if process blocked or terminated
- Preemptive (non-cooperative) schedulers can switch even when process is ready to continue
 - CPU generates periodic timer interrupt
 - After servicing interrupt, OS checks if the current process has run for too long

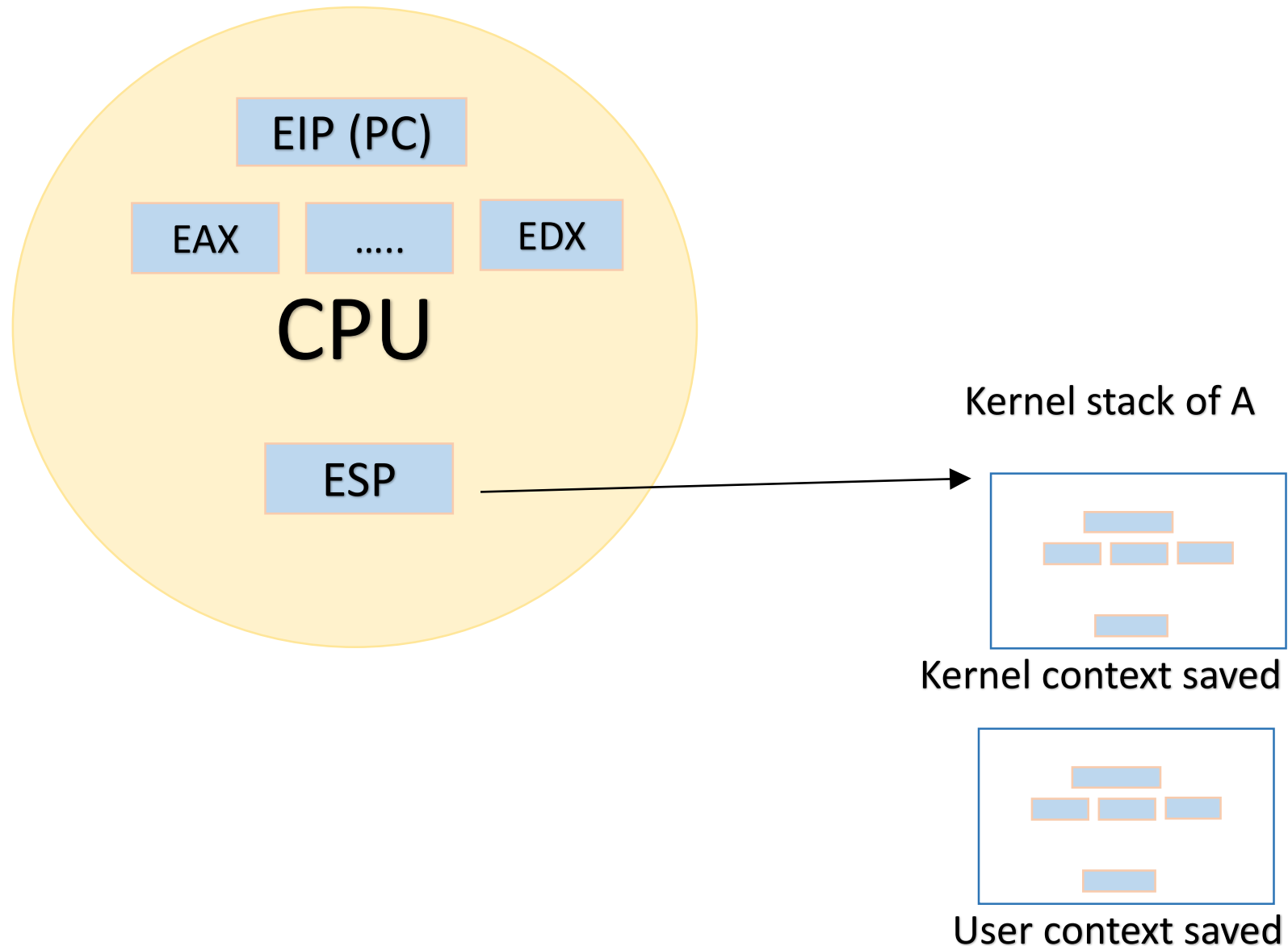
Mechanism of context switch (1)

- Process A has moved from user to kernel mode
- Kernel stack of A already has user register context
- After running for some time in kernel mode, A cannot run anymore (e.g., disk read initiated but data takes time to arrive)
- OS scheduler picks another process B to run next



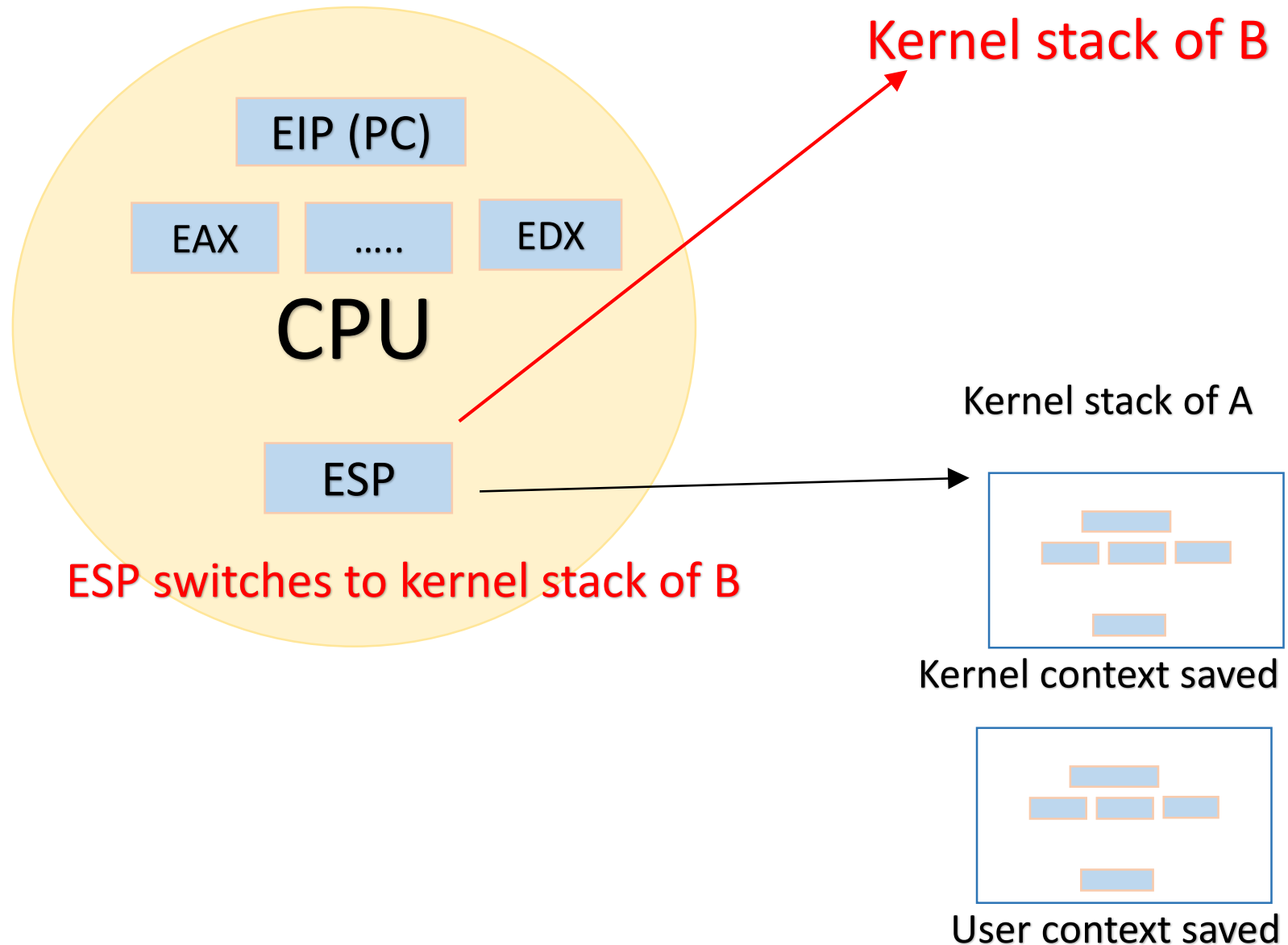
Mechanism of context switch (2)

- OS saves kernel context (PC, registers, kernel stack pointer) of A on kernel stack
- Why save context again?
- User context captures where execution stopped in user mode
- Kernel context captures where execution stopped in kernel mode



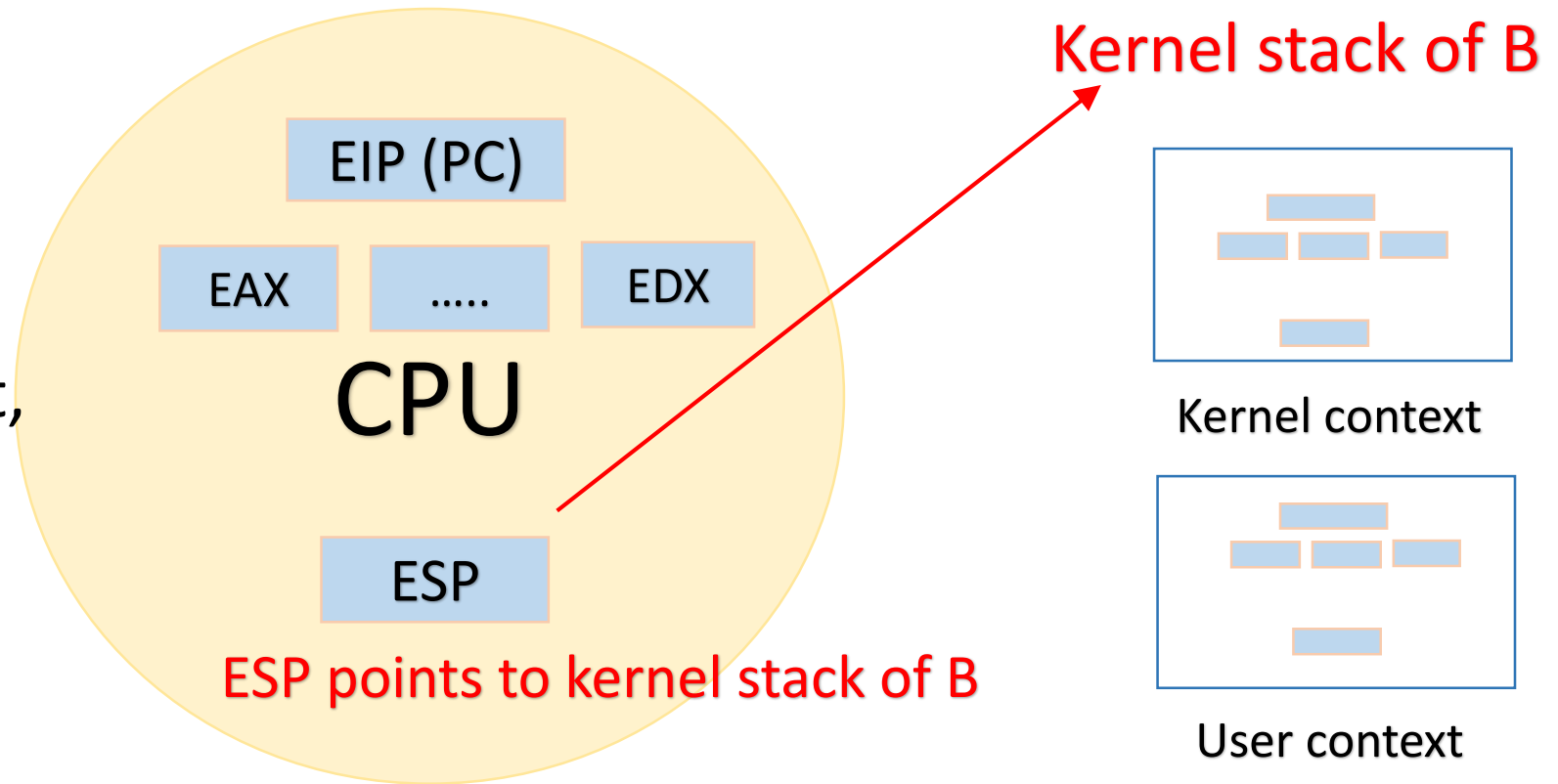
Mechanism of context switch (3)

- The actual moment of the context switch: OS switches ESP from kernel stack of A to kernel stack of next process B
- What will we find on the kernel stack of B?
 - Whatever the OS stored on it when it switched B out in the past



Mechanism of context switch (4)

- Kernel stack of B contains kernel context and user context of B
- OS restores kernel context, resumes execution in kernel mode of B, at the point it gave up CPU
- OS pops user context, resumes execution in user mode of B where it trapped into OS
- Context switch complete!



Understand saving and restoring context

- Context (PC and other CPU registers) saved on the kernel stack in two different scenarios
- When going from user mode to kernel mode, user context (e.g., which instruction of user code you stopped at) is saved on kernel stack by the trap instruction
 - Restored by return-from-trap when process goes to user mode
- During a context switch, kernel context (e.g., where you stopped in the OS code) is saved on the kernel stack by the context switching code
 - Restored when the process is ready to run and switched back in again

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

```
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             swtch(&(c->scheduler), p->context);
2781             switchkvm();
2782
2783             // Process is done running for now.
2784             // It should have changed its p->state before coming back.
2785             c->proc = 0;
2786         }
2787         release(&ptable.lock);
2788     }
2789 }
2790 }
2791 }
```

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

```
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 }
```

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

```
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 }
```

```
2662 // Jump into the scheduler, never to return.
2663 curproc->state = ZOMBIE;
2664 sched();
2665 panic("zombie exit");
2666 }
```

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

struct context

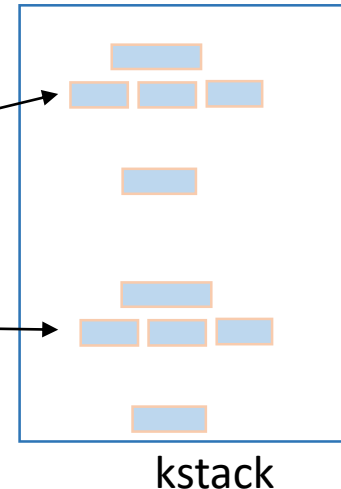
```
2326 struct context {  
2327     uint edi;  
2328     uint esi;  
2329     uint ebx;  
2330     uint ebp;  
2331     uint eip;  
2332 };
```

- In both scheduler and sched functions, the function “switch” switches between two “contexts”
- Context structure: set of registers to be saved / 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 switch 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

```
2342 int pid; // Process ID
2343 struct proc *parent; // Parent process
2344 struct trapframe *tf; // Trap frame for current syscall
2345 struct context *context; // switch() here to run process
```



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

```
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
```

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?

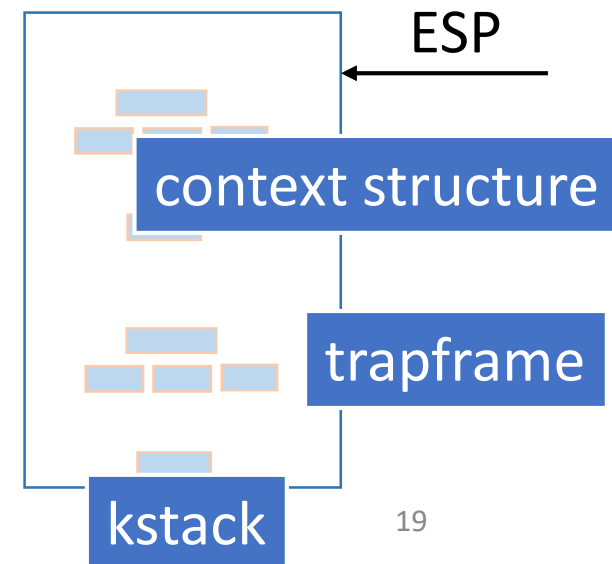
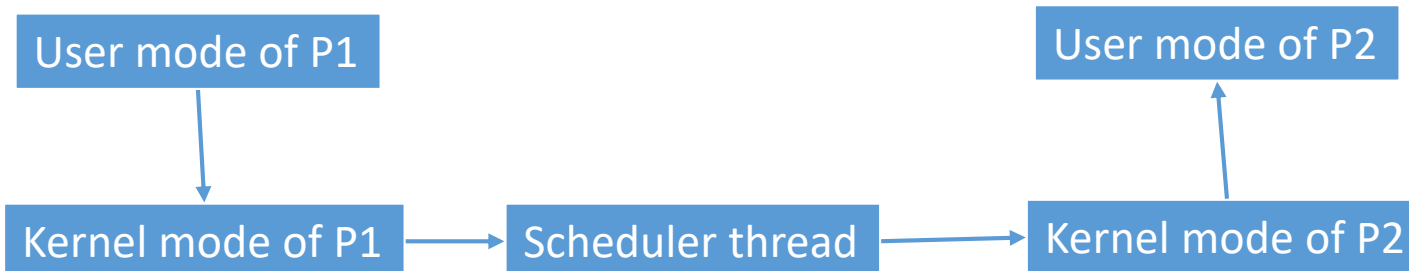
- What is on the kernel stack when a process/thread has just invoked the switch? Caller save registers and return address (EIP)
- What does switch do?
 - Push remaining (callee save) registers on old kernel stack
 - Save pointer to this context in old process PCB
 - Switch ESP from old kernel stack to 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

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



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

```
2579 int
2580 fork(void)
2581 {
2582     int i, pid;
2583     struct proc *np;
2584     struct proc *curproc = myproc();
2585
2586     // Allocate process.
2587     if((np = allocproc()) == 0){
2588         return -1;
2589     }
2590
2591     // Copy process state from proc.
2592     if((np->pgdir = copyvm(curproc->pgdir, curproc->sz)) == 0){
2593         kfree(np->kstack);
2594         np->kstack = 0;
2595         np->state = UNUSED;
2596         return -1;
2597     }
2598     np->sz = curproc->sz;
2599     np->parent = curproc;
```

```
2600     *np->tf = *curproc->tf;
2601
2602     // Clear %eax so that fork returns 0 in the child.
2603     np->tf->eax = 0;
2604
2605     for(i = 0; i < NOFILE; i++)
2606         if(curproc->ofile[i])
2607             np->ofile[i] = filedup(curproc->ofile[i]);
2608     np->cwd = idup(curproc->cwd);
2609
2610     safestrcpy(np->name, curproc->name, sizeof(curproc->name));
2611
2612     pid = np->pid;
2613
2614     acquire(&ptable.lock);
2615
2616     np->state = RUNNABLE;
2617
2618     release(&ptable.lock);
2619
2620     return pid;
2621 }
```

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

```
2468 // Look in the process table for an UNUSED proc.
2469 // If found, change state to EMBRYO and initialize
2470 // state required to run in the kernel.
2471 // Otherwise return 0.
2472 static struct proc*
2473 allocproc(void)
2474 {
2475     struct proc *p;
2476     char *sp;
2477
2478     acquire(&ptable.lock);
2479
2480     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2481         if(p->state == UNUSED)
2482             goto found;
2483
2484     release(&ptable.lock);
2485     return 0;
2486
2487 found:
2488     p->state = EMBRYO;
2489     p->pid = nextpid++;
2490
2491     release(&ptable.lock);
2492
2493     // Allocate kernel stack.
2494     if((p->kstack = kalloc()) == 0){
2495         p->state = UNUSED;
2496         return 0;
2497     }
2498     sp = p->kstack + KSTACKSIZE;
2499
```

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

```
2500 // Leave room for trap frame.
2501 sp -= sizeof *p->tf;
2502 p->tf = (struct trapframe*)sp;
2503
2504 // Set up new context to start executing at forkret,
2505 // which returns to trapret.
2506 sp -= 4;
2507 *(uint*)sp = (uint)trapret;
2508
2509 sp -= sizeof *p->context;
2510 p->context = (struct context*)sp;
2511 memset(p->context, 0, sizeof *p->context);
2512 p->context->eip = (uint)forkret;
2513
2514 return p;
2515 }
```

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

Summary

*As the stack pointer moves to another stack
And the program counter points to another's code
The distraught process begins to wonder
"Am I still alive or am I dead cold?"*

*Little does the process know it will be back soon
For that's the game -- run a bit and get swapped
And when its context is restored into CPU registers
It will resume like it had never been stopped*