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

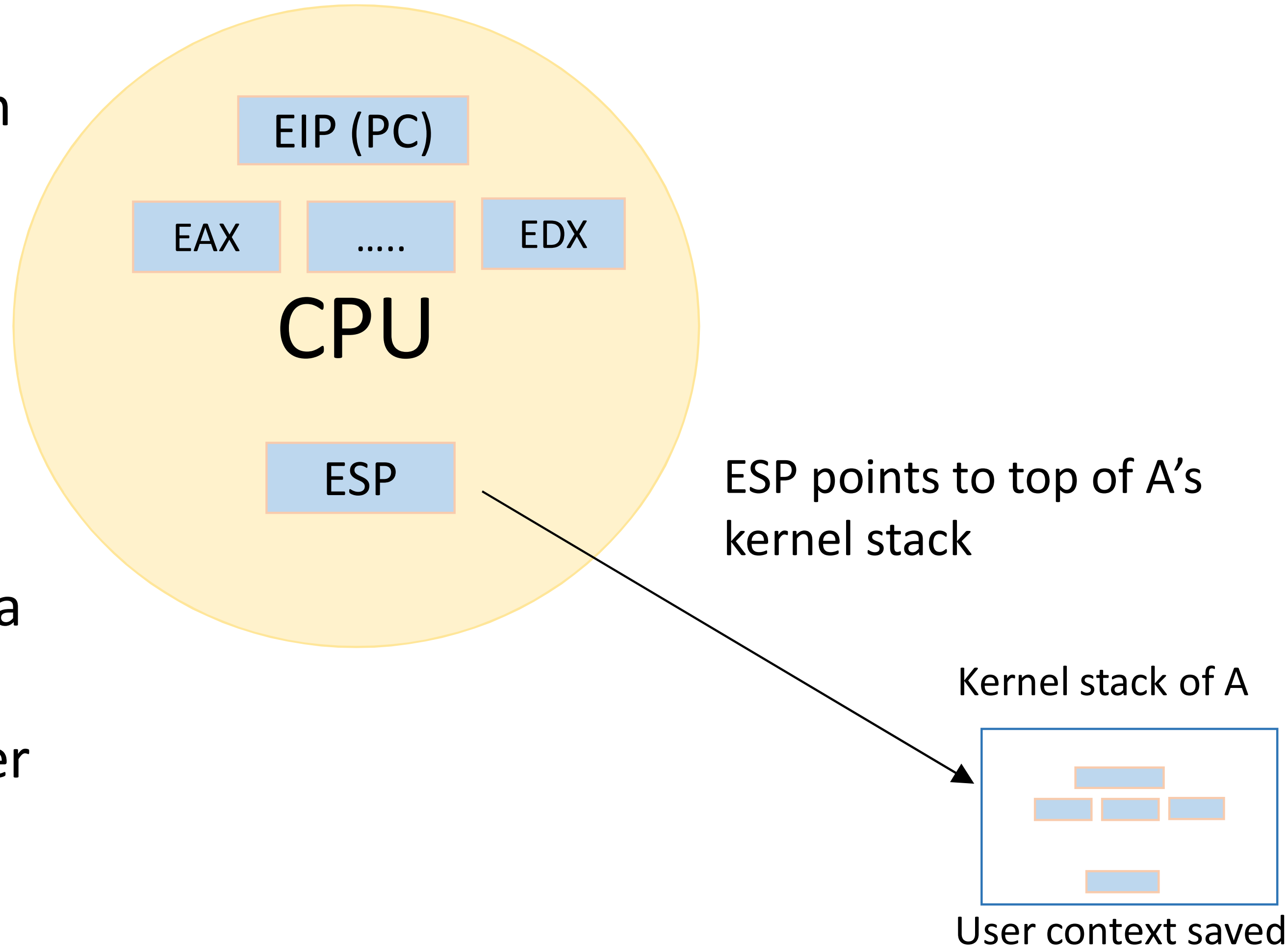
- OS maintains list of all active processes (PCBs) in a data structure
  - Processes added during fork, removed after clean up in wait
- OS scheduler is special code in the OS that periodically loops over this list and picks processes to run
- Basic outline of scheduler code
  - When invoked, save context 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

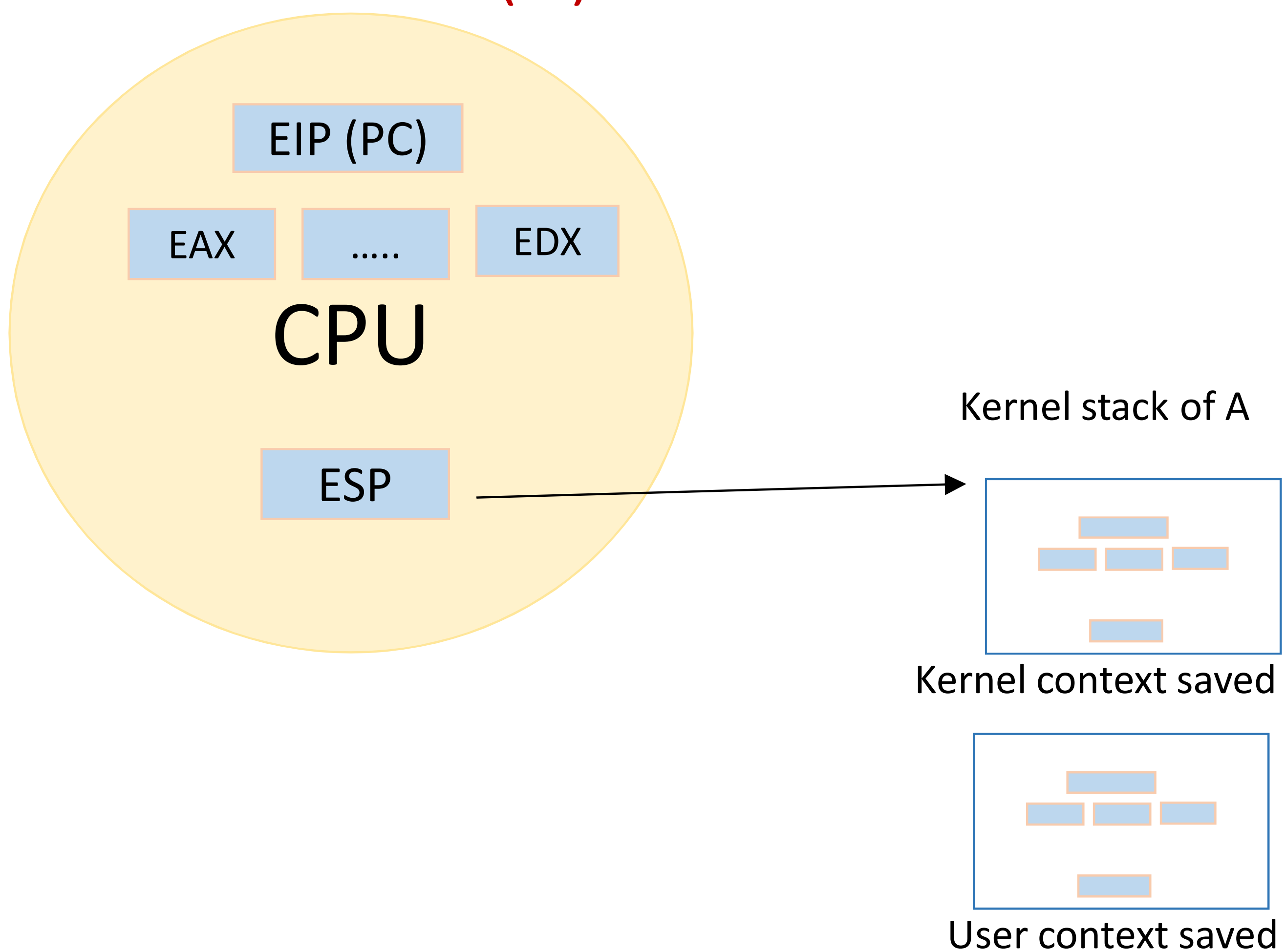
Mechanism of context switch (1)

- 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



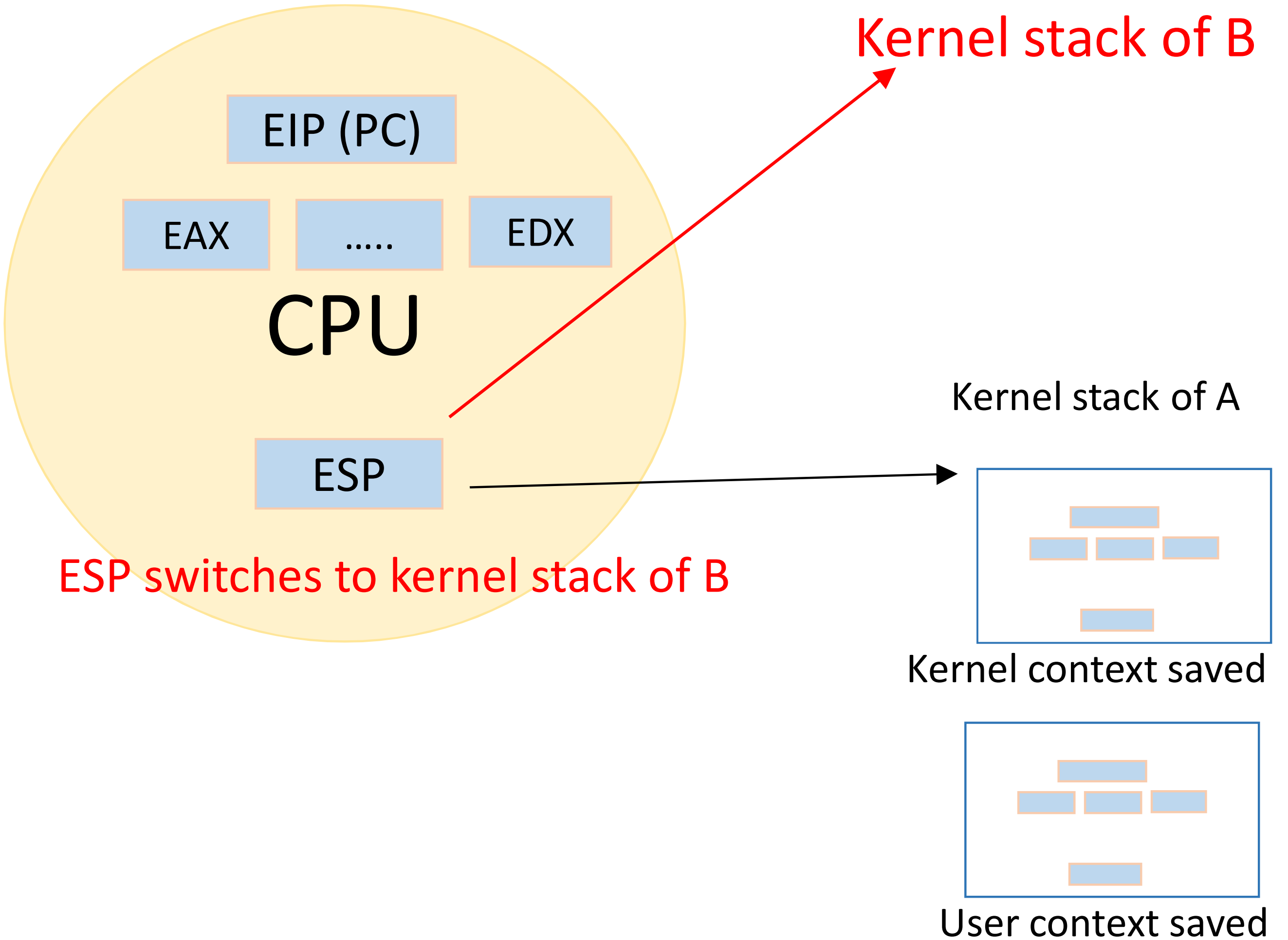
Mechanism of context switch (2)

- 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



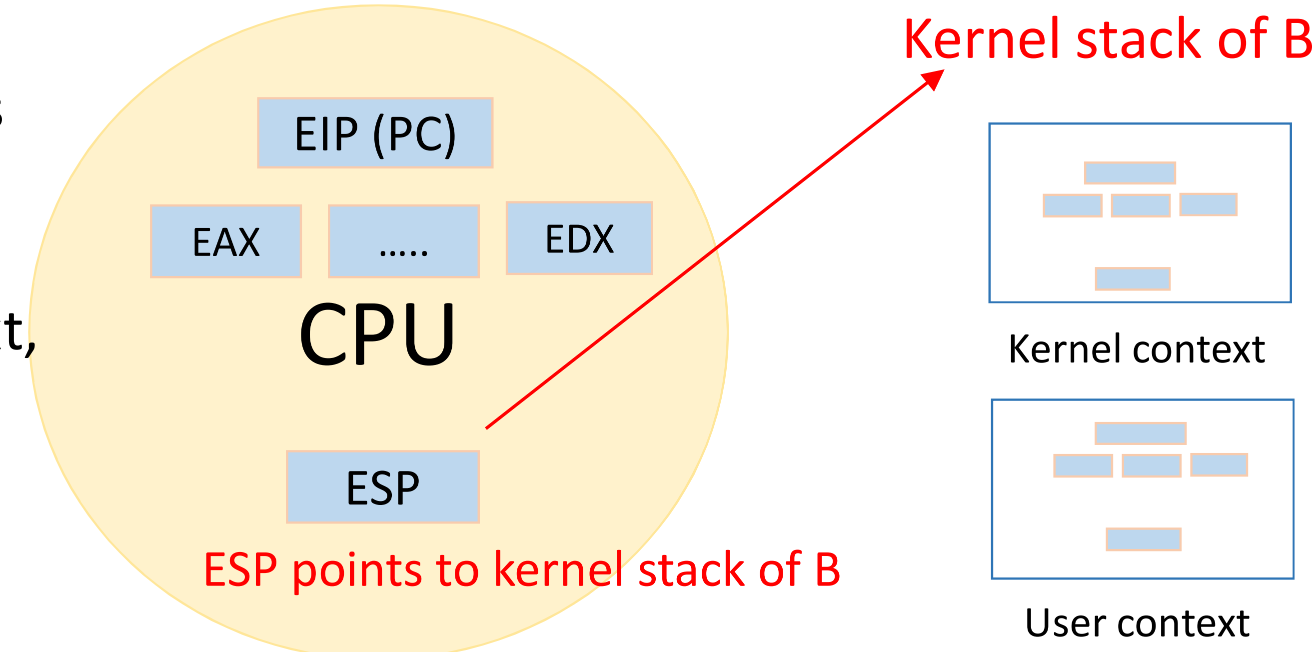
Mechanism of context switch (3)

- 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

```
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

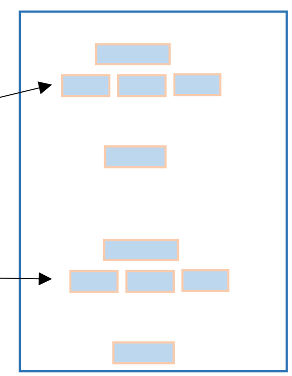
```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

- 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

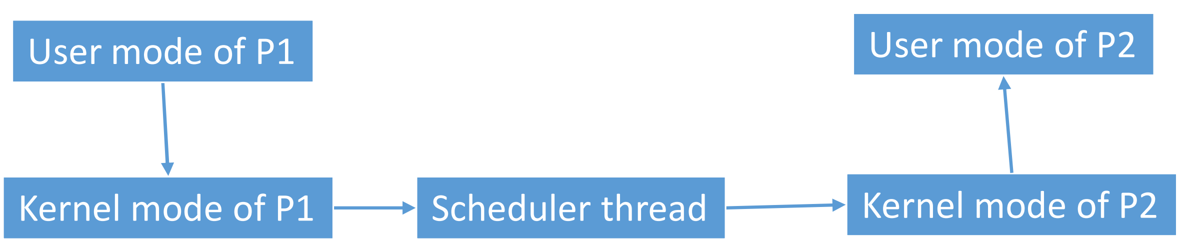
```
int pid;                    // Process ID
struct proc *parent;        // Parent process
struct trapframe *tf;       // Trap frame for current syscall
struct context *context;    // swtch() here to run process
```

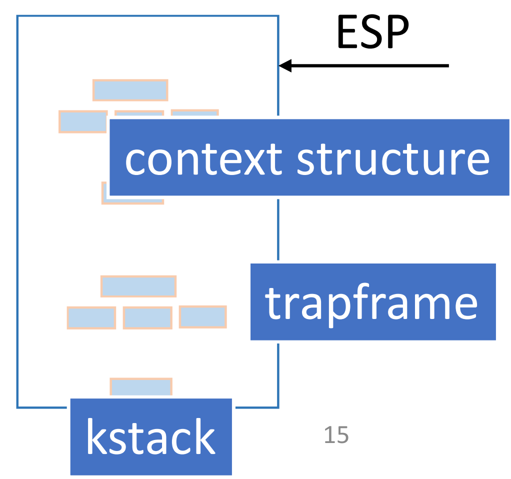


kstack

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

```
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

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