CS 347M (Operating Systems Minor)

Spring 2022

Lecture 5: Process Scheduling

Mythili Vutukuru

CSE, IIT Bombay

OS scheduler

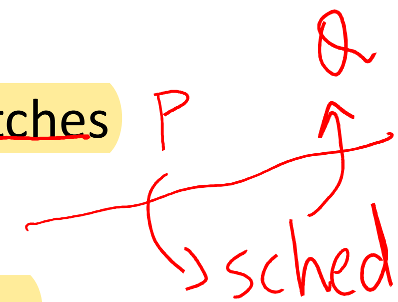
P5 P6 . . .

P1 P2 P3 P4

- OS scheduler schedules process on CPU cores
  - One process at a time per CPU core
  - Multiple processes can run in parallel on multiple cores
- Scheduling policy: which one of the ready/runnable processes should be run next on a given CPU core?
  - Mechanism of context switching (save context of old process in its kernel stack/PCB and restore context of new process) is independent of policy
- Simple scheduling policies have good theoretical guarantees, but not practical for real operating systems
  - Real-life schedulers are very complex, involve many heuristics

Preemptive vs. non preemptive schedulers

- When is the OS scheduler invoked to trigger a context switch?
  - Only when a process is in kernel mode for a trap, but not on every trap
- Non-preemptive scheduler performs only voluntary context switches
  - Process makes blocking system call
  - Process has exited or has been terminated



- Preemptive scheduler performs involuntary context switches also
  - Process can be context switched out even if process is still runnable/ready
  - OS can ensure that no process runs for too long on CPU, starving others
- Timer interrupts: special interrupts that go off periodically to trap to OS
  - Used by OS to get back control, trigger involuntary context switches
- Modern systems use preemptive schedulers
  - Process can be context switched out any time in its execution

Goals of CPU scheduling policy

start end

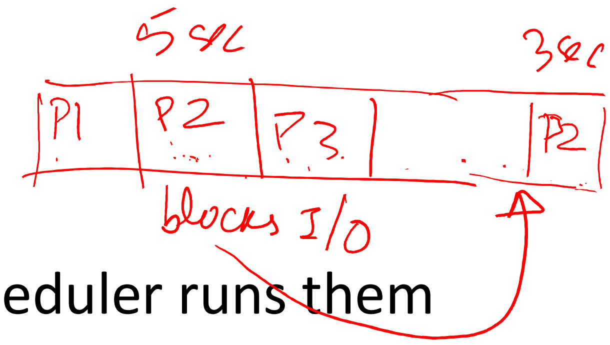
- Maximize utilization: efficient use of CPU hardware
- Minimize completion time / turnaround time of a process (time from process creation to completion)
- Minimize response time of a process (time from process creation to first time it is run)
  - Important for interactive processes

start first scheduled

- Fairness: all processes get a fair share of CPU
  - Can account for priorities also
- Low overhead of scheduling policy
  - Scheduler does not take too long to make a decision (even with large #processes)
  - Scheduler does not cause too many context switches (~1 microsecond to switch)

10^{-6}

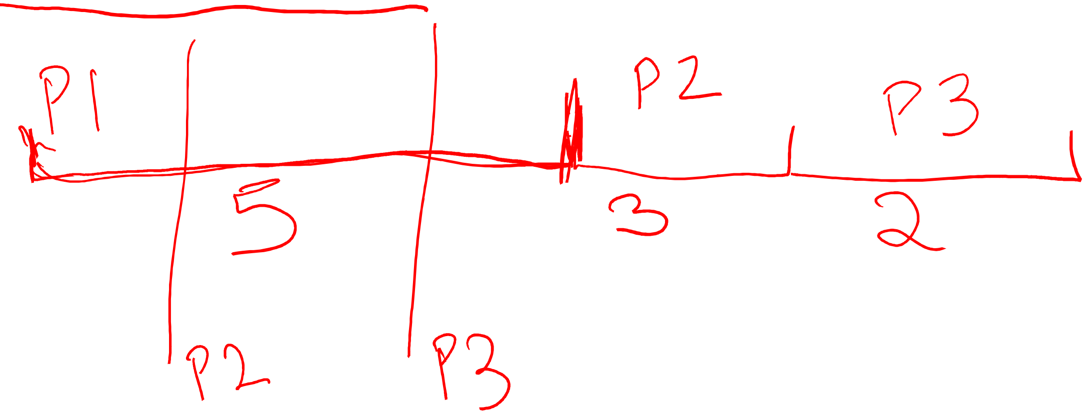
Simplest policy: First In First Out



- Newly created processes are put in a FIFO queue, scheduler runs them one after another from queue
- Non-preemptive: process allowed to run till it terminates or blocks
  - When process unblocks, the next run is separate "job", added to queue again
  - That is, if a process comes back after I/O wait, it counts as a fresh CPU burst (CPU burst = the CPU time used by a process in a continuous stretch)
- Example schedule: P1 (1-5), P2 (6-8), P3 (9 to 10) work out

Given

Process	CPU time needed (units)	Arrives at end of time unit
P1	5	0
P2	3	1
P3	2	3



Problem with FIFO

- Example: three processes arrive at t=0 in the order A,B,C
- Problem: convoy effect (small processes get stuck behind long processes)
- Average turnaround times tend to be high

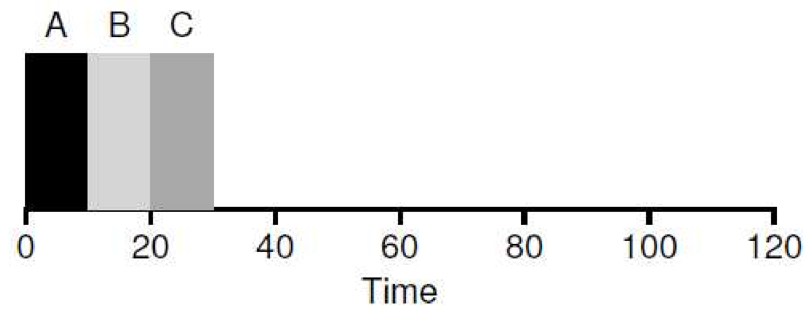
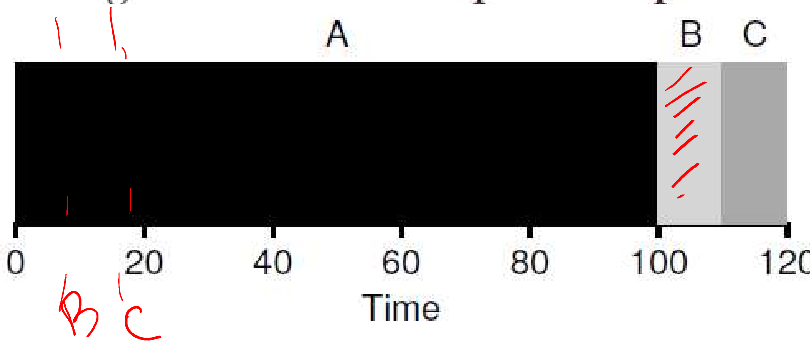


Figure 7.1: **FIFO Simple Example**



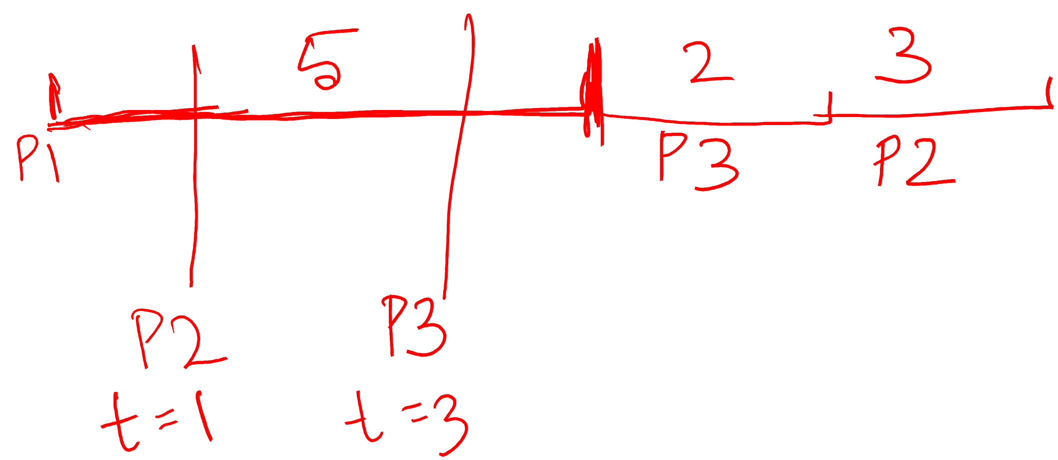
OSTEP

Figure 7.2: **Why FIFO Is Not That Great**

Shortest Job First (SJF)

- Assume CPU burst of a process (amount of time a process runs on CPU until termination/blocking) is known apriori unrealistic
- Pick process with smallest CPU burst to run next, non-preemptive
  - Store PCBs in a heap-like data structure, extract process with min CPU burst
- Example schedule: P1 (1-5), P3 (6-7), P2 (8-10)

Process	CPU burst	Arrival time
P1	5	0
P2	3	1
P3	2	3



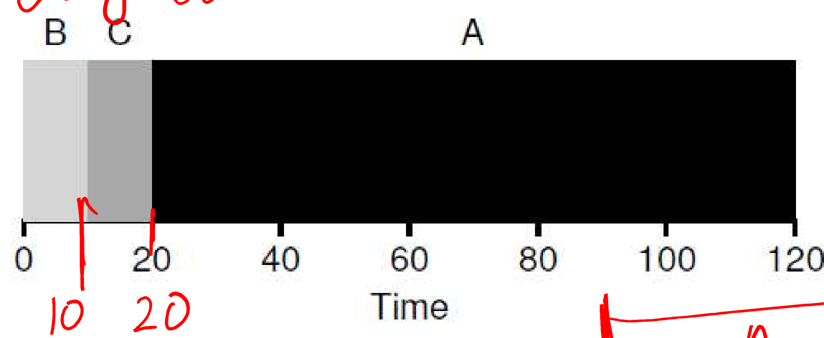
Problem with SJF

A, B, C $t = 0$

avg turnaround

$$\frac{10+20+120}{3} = 50$$

- Provably optimal when all processes arrive together
  - Theoretically guaranteed to have the lowest average turnaround time across all policies (under certain assumptions)
- SJF is non-preemptive, so short jobs can still get stuck behind long ones.



A | B | C

110

Figure 7.3: **SJF Simple Example**

$$\frac{100+110+120}{3} = 110$$

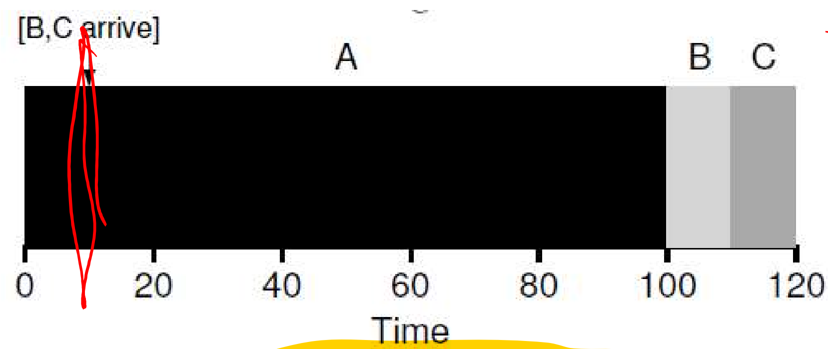
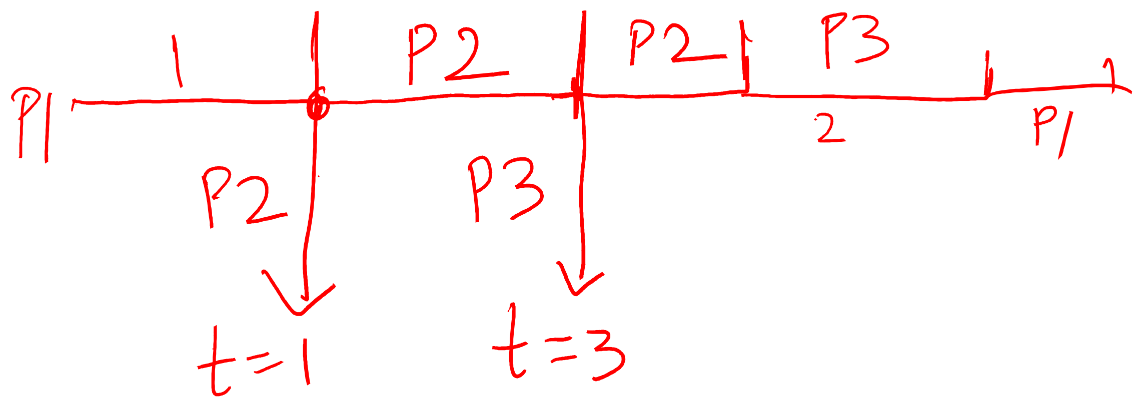


Figure 7.4: **SJF With Late Arrivals From B and C**

Shortest Remaining Time First (SRTF)

- Preemptive version of SJF
- A newly arrived process can preempt a running process, if CPU burst of new process is shorter than remaining time of running process
  - Avoids problem of short process getting stuck behind long one
- Example schedule: P1 runs for 1 unit, P2 (2-4), P3 (5-6), P1 (7-10)

Process	CPU burst	Arrival time
P1	5 − 1 = ~~4~~	0
P2	~~3~~ − 2 = 1	1
P3	~~2~~	3



Round Robin (RR)

- Every process executes for a fixed quantum slice
  - Slice not too small (to amortize cost of context switch) 1 μs X
  - Slice not too big (to provide good responsiveness) 10 sec X
- Preemptive policy
  - Timer interrupt used to enforce periodic scheduling
- Good for response time and fairness
- Bad for turnaround time

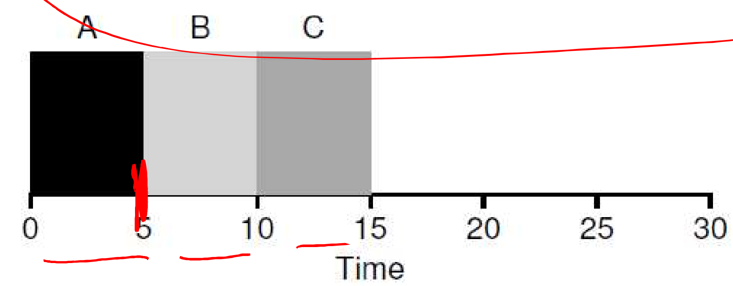


Figure 7.6: SJF Again (Bad for Response Time)

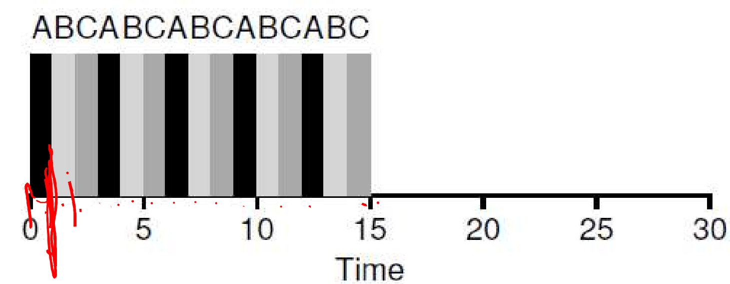


Figure 7.7: Round Robin (Good for Response Time)

Weighted Fair Queueing (WFQ) WRR

- Round robin with ~~different weights~~ or priorities to processes
  - Decided by scheduler or can be set by users
  - Time slice will be in proportion to the weight or priority
- Linux scheduler is a variant of weighted fair queueing
- Real life schedulers may not be able to enforce time slice exactly
  - What if timer interrupt is not exactly aligned with time slice?
  - What if process blocks before its time slice?
- Practical modification: keep track of run time of process, schedule process that has used least fraction of its fair share
  - Compensate excess/deficit running time in future time slices

nice

P1 P2 P3
2 : 1 : 4
20 ms 10 ms 40 ms
11 ms

$\frac{16\,ms}{20}$ $\frac{11\,ms}{10}$ $\frac{33\,ms}{40}$

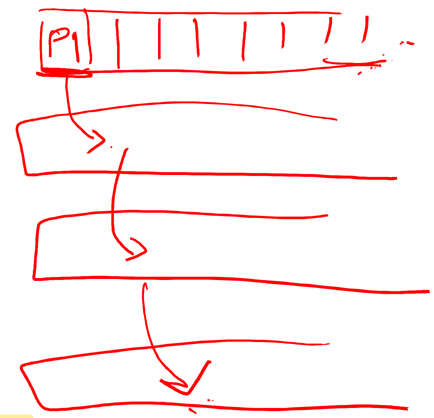
work conserving

Multi-level feedback queue (MLFQ)

high

med ~~low~~

low



- Another practical algorithm, with realistic assumptions
- Multiple queues, one for each priority level
  - Schedule processes from highest priority queue to lowest
  - Use round robin scheduling for processes within same priority level
- Priority set by user or OS, but decays with age
  - Job that uses up its time slice at a priority level goes to lower priority level
  - Why? Ensures short I/O-bound processes that don't use their full slice get priority over long CPU-bound processes that use their fair share all the time

    SJF
  - Ensures short processes get priority over long processes, but without knowing CPU burst of process apriori
- Periodically reset all processes to highest priority level to avoid starvation of low priority or CPU-bound processes

OSTEP

Multicore scheduling

- Scheduling decision needs to be made separately for each CPU core
- Do we bind a process to a particular CPU core always, or do we let a process run on any CPU core that is free?
- Ensuring a process runs on the same core as far as possible is better
  - Avoids coordination overheads across cores, better CPU cache performance
- But, we must be flexible too
  - If CPU core overloaded, some of its processes must move to another core
  - Load balancing across cores to ensure uniform workload distribution

Scheduler in xv6

S P1 P2 sh init

- 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 (round robin)
- The special function "swtch" performs the actual context switch
  - Save context on kernel stack of old process
  - Restore context from kernel stack of new 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);

  }
}
```

save context
restore context

xv6 scheduler and sched functions

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

S

P sleep/exit/yield

```
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);

  }
}
```

old new

PC

```
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;
}
```

give up resume

old new

sp

Who calls sched()?

give up CPU

- 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

timer interrupt

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

ready

exit

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

sleep

blocking syscall

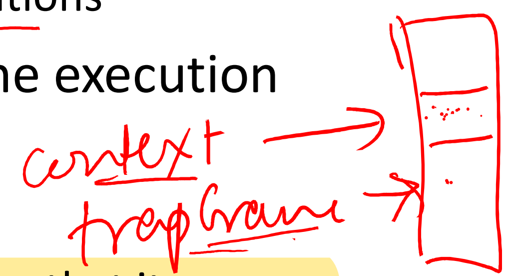
```
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();

```

P → yield → ... → yield

sched() sched()

What about new processes?

- The context switching code in xv6 restores context from kernel stack/PCB of a process and resumes execution where process stopped earlier
  - Recall: "context" structure in PCB stores context during context switch
  - Recall: "trap frame" in PCB stores context during user-kernel transitions
- What if a process has never run before? Where will it resume execution when it is switched in by scheduler?
- Kernel stack of new processes setup in such a way that
  - PC (EIP) of a function where it has to run is saved on kernel stack, so that it appears that process was switched out at the location where we want it to resume
  - Context structure and trap frame suitably created on kernel stack of new process
  - Process resumes execution in kernel mode, returns from trap to user space



Recap: fork system call

- Parent allocates new process in ptable, copies parent state to child
- Child process set to runnable, scheduler runs it at a later time
- Return value in parent is PID of child, return value in child is set to 0

```
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;
```

alloc new PID

```
  *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

usermode

K forkret trapret

- 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
- Go to bottom of stack, leave space for trapframe (copied from parent trap frame in fork)
- Push return address of "trapret"
- Push context structure, with eip pointing to function "forkret"
- Why? When this new process is scheduled, it begins execution at forkret, then returns to trapret, then returns from trap to userspace
- Allocproc has created a hand-crafted kernel stack to make the process look like it had a trap and was context switched out in the past
  - Scheduler can switch this process in like any other

```
// 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;

  // 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;
}
```

PCB

context

tf

mem

kstack

context

forkret

trapret

Forking new process (revisited)

- Fork allocates new process via allocproc
- Parent memory and file descriptors copied (more later)
- Trapframe of child copied from that of parent
  - Result: child returns from trap to exact line of code as parent
  - Only return value 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

```
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;
}
```

fork()

P C

context forkret

tf eip user

kstack

parent tf

context tf

tf P → C

fork