Chapter 3a: xv6 Process Internals

Gene Cooperman

Copyright ©2026 Gene Cooperman, gene@ccs.neu.edu
This text may be copied as long as the copyright notice remains and no text is modified.

In this chapter, we will look at the xv6 UNIX operating system, Sixth Edition UNIX (UNIX Version 6). That was first released in May 1975, the first wide release outside Bell Labs. A port that runs in 32-bit mode on a modern Intel CPU was created at MIT as xv6. Locally, there is a version at Northeastern University, which is hyperlinked, making it easier to read.

In the chapter 1-proc-table-file-table ("Linux/UNIX Process and File Management"), we saw the internal concepts and design behind the UNIX/Linux operating system. That chapter introduced three core data structures: the process table, the system-wide open file table, and the filesystem. In that chapter, we also discussed some commonly used system calls, and how they relate to the underlying concepts.

The earlier chapter described the underlying concepts of the operating system, and the user-visible system calls. This chapter resolves the middle portion: the operating system that uses the concepts to implement the user-visible system calls. This exercise deepens the understanding of the underlying concepts, by showing what are the implementational constraints that elaborate on those concepts.

In Part 3a, we look into the code behind the process table and its table entry, `struct proc`. In Part 3b, we trace the control flow of a single system call (`fork`) through that code. In Part 3c, we examine the code behind the table of open files and its table entry, `struct file`. After that, we examine the in-memory inode tables inside a filesystem, and an entry in that table, `struct inode`. (The actual inode table on disk has entries, `struct dinode`. Finally, user-level system calls never take inodes as arguments. They take paths or file descriptors as argument. The missing link is the stat system call family, with an output parameter, `struct stat`.

TO UPDATE: Remove less important data structures of xv6; xv6 CPU: Assume one CPU. The xv6 'struct cpu' is an attempt to allow xv6 to run with multiple cores, using symmetric multiprocessing (SMP: a simplified version of CPU cores)

Contents

3a xv6 Process Internals: Data Structures and Operations in `proc.h` and `proc.c` 1
3a.1 Data Structures in `proc.h` . . . 2
3a.2 Operations in `proc.c` . . . 4
3a.3 Context Switching: The Full Picture . . . 9
3a.4 The Mechanics of `swtch.S` . . . 10

3a xv6 Process Internals: Data Structures and Operations in `proc.h` and `proc.c`

This is an explanation of how xv6 represents processes and CPUs, and how the core process-management functions work. The original UNIX v6 on which it is based supported only a single CPU core, in keeping with the

technology of the 1970s.

When xv6 was created at MIT, they generalized the code to support multiple CPU cores, which meant that multiple processes could simultaneously be "RUNNING". After doing this, they were also forced to introduce locks (spinlocks), so that no two running processes would simultaneously try to access the system-wide process table or open file table.

For simplicity, we will always assume that there is only one CPU core (one instance of `struct cpu` in the language of xv6). We will also assume that we can ignore any global locks associated with the process or file table.

Finally, the original UNIX v6 would have had a single global variable, `proc`, which would always point to the currently running process: `> struct proc *proc;`

In xv6, each CPU core would have its own "proc" pointing to the currently running process on that core. We can also ignore this, and pretend that `proc` is a global variable pointing to the unique running process on a single CPU core.

NOTE: For those who care, `proc` is actually defined in the `proc.h` file of xv6. It's "weird" definition is:
`extern struct proc *proc asm("%gs:4");`
It initializes `proc=asm("%gs:4"). asm("%gs:4")` is an extended C syntax which says that the pointer to the running `struct proc` can be found at address `%gs:4` in memory. `%gs:4` refers to a thread-local address using the x86 `gs` register, but a full description of the details is out of scope for this explanation.

3a.1 Data Structures in `proc.h`

3a.1.1 `struct proc` – the process control block

```
struct proc {
  uint sz;                     // size of process memory (bytes)
  pde_t *pgdir;                // page directory (virtual address space)
  char *kstack;                // bottom of kernel stack for this process
  enum procstate state;        // process state
  int pid;                     // process ID
  struct proc *parent;         // parent process
  struct trapframe *tf;        // trapframe for current syscall/interrupt
  struct context *context;     // saved kernel context (for swtch())
  void *chan;                  // if SLEEPING: the channel being waited on
  int killed;                 // if non-zero, process has been killed
  struct file *ofile[NOFILE]; // open files
  struct inode *cwd;           // current working directory
  char name[16];              // process name (for debugging)
};
```

Key fields grouped by purpose:

Memory management:
- `sz` – the size of the user address space. Grows via `sbrk()` / `growproc()`.
- `pgdir` – the page directory. Each process has its own, giving it a private virtual address space.
- `kstack` – each process has a dedicated kernel stack, used when it traps into the kernel.

Process relationships:
- `pid` – unique process identifier.
- `parent` – pointer to the parent process. Used by `exit()` and `wait()` for process cleanup.

- `killed` – a flag set by `kill()`. The process isn't immediately destroyed; instead, it is marked and will check this flag at the next safe point (e.g., returning to user space from a trap) and call `exit()`.

File system:
- `ofile[NOFILE]` – array of pointers to open `struct file` entries. The file descriptor `fd` is an index into this `ofile[]` array.
- `cwd` – pointer to the inode of the current working directory.

Debugging:
- `name` – a human-readable name, typically set to the name of the executable by `exec()`. (In modern Linux, they use `execvp()` or a variant, instead of `exec()`.

Scheduling and context switching:
- `state` – the current lifecycle state.
- `context` – pointer to saved callee-saved registers on the kernel stack, used by `swtch()`.
- `tf` – pointer to the trapframe on the kernel stack, used by `trapret` to return to user space.
- `chan` – when a process is `SLEEPING`, this records *what* it's sleeping on (an arbitrary address used as a key by `sleep()`/`wakeup()`).

Next, we look more closely at the states of a process life cycle, and the context (the saved registers when this process is not running) and

3a.1.2 `enum procstate` – states in the life cycle of a process

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

State	Meaning
`UNUSED`	Slot in process table is free
`EMBRYO`	Process is being initialized
`RUNNABLE`	Ready to run, waiting for scheduler
`RUNNING`	Currently executing on a CPU
`SLEEPING`	Blocked, waiting for an event
`ZOMBIE`	Exited, waiting for parent to call `wait()`

3a.1.3 `struct context` – saved registers for context switching

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

This stores the **callee-saved registers** plus the instruction pointer. It is the minimal state needed to suspend and resume a kernel thread. Note that this is *not* the full user-space register state – that is saved in the trapframe. `struct context` is used only for switching between kernel execution contexts: from a process's kernel thread to the scheduler, or vice versa.

The layout matches what `swtch()` (in `swtch.S`) pushes and pops on the kernel stack.

3a.1.4 `ptable` – the global process table

Defined in `proc.c`:

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

All processes in the system live in this fixed-size array (typically `NPROC = 64`). So, this early UNIX allowed at most 64 processes at one time. The spinlock protects concurrent access – any code that reads or modifies process states must hold `ptable.lock`. The process table is the central data structure for process management: `allocproc()` scans it for free slots, the `scheduler()` scans it for runnable processes, `wait()` scans it for zombie children, and so on.

As discussed previously, we will assume a single CPU core, and therefore a single running process. The lock in `ptable` would have been needed only if there were multiple CPU cores with multiple running processes, each simultaneously accessing the process table.

Therefore, `proc.h` has a global array:
`extern struct cpu cpus[NCPU];`
But for us, `NCPU=1` (one CPU). And `proc.h` also defines the `struct cpu` (simplified here):

```
struct cpu {
  struct context *scheduler;  // saved context for the scheduler thread
  ...
  struct proc *proc;          // the process currently running on this CPU, or NULL
};
```

For the two fields that we care about:

- **`scheduler`** stores the saved register state for the scheduler process itself, just as each process table entry has a field to save the process's context. When a process yields the CPU, `swtch()` saves the process's context and restores this one, returning control to `scheduler()`.
- **`proc`** points to whichever process of the process table is currently `RUNNING`. Since we are assuming a single CPU, there is just one, unique running process.

3a.2 Operations in `proc.c`

3a.2.1 `pinit()` – initialize the process table (line 24)

```
Called once at boot (from main.c).
Initializes ptable.lock.
```

This is straightforward: just `initlock(&ptable.lock, "ptable")`. After this, the process table is ready for use.

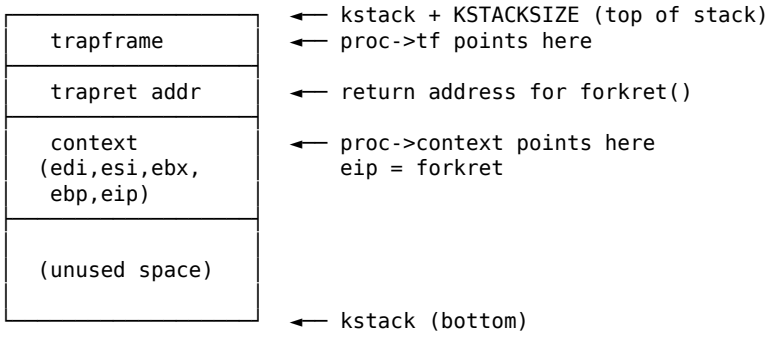
3a.2.2 `allocproc()` – allocate a new process slot (called internally)

`allocproc()` is not listed in the file reference because it is `static` (internal to `proc.c`), but it is the foundation for both `userinit()` and `fork()`. It:

1. Acquires `ptable.lock`
2. Scans `ptable.proc[]` for a slot with `state == UNUSED`
3. If found, sets `state = EMBRYO` and assigns `pid = nextpid++`

4. Releases `ptable.lock`
5. Allocates a kernel stack (`KSTACKSIZE` bytes via `kalloc()`)
6. Sets up the bottom of the kernel stack with:
   - A **trapframe** – space for saving user registers on trap entry
   - A **context** – initialized so that `swtch()` will jump to `forkret()`, which then falls into `trapret`

The kernel stack layout after `allocproc()`:



This synthetic stack setup is the mechanism that allows a newly created process to be "switched to" by the scheduler as if it were a process that had previously been switched away from. The first `swtch()` into this process restores the context, "returns" to `forkret()`, which "returns" to `trapret`, which pops the trapframe and enters user space.

3a.2.3 `userinit()` – create the first user process (line 79)

```
Called once at boot (from main.c), after pinit().
Creates the init process (pid 1).
```

Since there is no parent process to `fork()` from, `userinit()` manually constructs the first process:

1. Calls `allocproc()` to get a new process slot and kernel stack
2. Calls `setupkvm()` to create a page table with kernel mappings
3. Copies a small piece of code (`initcode.S`) into the process's user memory at address 0 – this code will call `exec("/init", ...)`
4. Sets up the trapframe: `eip` points to address 0 (the initcode), `esp` points to `PGSIZE` (one page of stack), segment registers set for user mode
5. Sets `cwd` to the root directory inode
6. Sets `state = RUNNABLE`

When the scheduler first picks this process, it enters user space running `initcode.S`, which immediately `exec()`s `/init`, which in turn `fork()`s a shell.

3a.2.4 `fork()` – create a child process (line 129)

This is described in detail in the *"xv6 System Call Flow: fork()"* document. In brief (assuming that we have already entered kernel mode):

1. `allocproc()` – get a new slot and kernel stack
2. `copyuvm()` – duplicate the parent's entire address space
3. Copy the trapframe, but set `eax = 0` (child's return value)
4. `filedup()` each open file descriptor (shared, not copied)
5. `idup()` the current working directory

6. Copy the process name
7. Set `state = RUNNABLE` and return the child's pid

3a.2.5 `growproc(n)` – grow or shrink the address space (line 108)

```
Called by sys_sbrk().
Adjusts proc->sz by n bytes (positive = grow, negative = shrink).
```

Calls `allocuvm()` (to grow) or `deallocuvm()` (to shrink) to adjust the page table, then reloads the page table into the hardware via `switchuvm()`. Returns 0 on success, -1 on failure (e.g., out of memory).

3a.2.6 `exit()` – terminate the current process (line 173)

When a process calls `exit()`:

1. Closes all open files (`fileclose()` on each `ofile[]` entry)
2. Releases the current working directory inode (`iput()`)
3. Acquires `ptable.lock`
4. Wakes up the parent (who may be blocked in `wait()`)
5. **Reparents orphaned children**: any children of the exiting process are reassigned to the `init` process (pid 1). If any of those children are zombies, `init` is woken up so it can reap them.
6. Sets `state = ZOMBIE`
7. Calls `sched()` to switch back to the scheduler – **this process never runs again**

The process's kernel stack, page table, and `ptable` entry are *not* freed here. That is the parent's responsibility, in `wait()`.

3a.2.7 `wait()` – wait for a child to exit (line 217)

1. Acquires `ptable.lock`
2. Scans the entire `ptable` for a child process (any process whose `parent == proc`) in state `ZOMBIE`
3. If a zombie child is found:
   - Frees the child's kernel stack (`kfree()`)
   - Frees the child's page table (`freevm()`)
   - Sets the child's state to `UNUSED` (the slot is now free for reuse)
   - Returns the child's pid
4. If children exist but none are zombies yet, calls `sleep()` to block until a child exits
5. If no children exist at all, returns -1

This is the other half of process cleanup. `exit()` makes the process a zombie; `wait()` reaps it and frees its resources.

3a.2.8 `scheduler()` – the per-CPU scheduling loop (line 266)

Each CPU runs an independent `scheduler()` loop (entered from `main.c` after boot, and never returns). The algorithm is simple round-robin:

```
loop forever:
    enable interrupts (so this CPU isn't deaf to devices)
    acquire ptable.lock
    for each process p in ptable:
```

```
        if p->state == RUNNABLE:
            switch to p:
                set cpu->proc = p
                switchuvm(p)           // load p's page table
                p->state = RUNNING
                swtch(&cpu->scheduler, p->context)  // context switch!
                // ... execution returns here when p yields ...
                switchkvm()            // switch back to kernel page table
                cpu->proc = NULL
    release ptable.lock
```

The `swtch()` call saves the scheduler's registers and loads the process's saved context. The scheduler doesn't run again until the process calls `sched()` (via `yield()`, `sleep()`, or `exit()`).

3a.2.9 `sched()` – switch from a process back to the scheduler (line 301)

`sched()` is the counterpart to the `swtch()` call in `scheduler()`. It performs sanity checks and then calls:

`swtch(&proc->context, cpu->scheduler);`

This saves the current process's kernel registers and restores the scheduler's context, resuming the `scheduler()` loop right after the `swtch()` call that originally entered this process.

`sched()` requires that `ptable.lock` is held (to protect the state transition) and that interrupts are disabled.

3a.2.10 `yield()` – voluntarily give up the CPU (line 320)

```
void yield(void) {
  acquire(&ptable.lock);
  proc->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

Called from `trap()` on every timer interrupt. The process is still runnable – it just gives other processes a chance to execute. This is how xv6 implements **preemptive multitasking**: even if a process never voluntarily sleeps, the timer interrupt forces it to yield periodically.

3a.2.11 `sleep(chan, lk)` – block until woken (line 352)

`sleep()` puts the current process to sleep on a **channel** – an arbitrary pointer value used as a key to match sleepers with wakeups:

1. Acquires `ptable.lock` (needed to change state atomically)
2. Releases the caller's lock `lk` (this is critical – see below)
3. Sets `proc->chan = chan` and `proc->state = SLEEPING`
4. Calls `sched()` to switch to the scheduler
5. When eventually woken up: clears `chan`, re-acquires `lk`, and returns

Why release `lk` before sleeping? The caller always holds some lock `lk` that protects the condition it's waiting on (e.g., the pipe lock when waiting for data). If the process slept while still holding `lk`, no other process could ever modify the condition and call `wakeup()` – deadlock. But simply releasing `lk` and then sleeping would create a race: the condition might change between the release and the sleep. The solution is to hold `ptable.lock` across

both the release of `lk` and the state change to `SLEEPING`. Since `wakeup()` also requires `ptable.lock`, no wakeup can be lost.

3a.2.12 `wakeup(chan)` – wake all processes sleeping on a channel (line 401)

```
Acquires ptable.lock.
Scans the entire ptable.
For each process with state == SLEEPING and chan == chan:
    set state = RUNNABLE
Releases ptable.lock.
```

Note that `wakeup()` wakes *all* sleepers on the channel. This is simple but means that woken processes must re-check their condition (typically in a `while` loop), because the condition may no longer be true by the time they actually run – another process may have consumed the event first.

3a.2.13 `kill(pid)` – mark a process for termination (line 412)

`kill()` does **not** immediately destroy the target process. It simply:

1. Scans the `ptable` for a process with the given pid
2. Sets `p->killed = 1`
3. If the process is `SLEEPING`, sets it to `RUNNABLE` (so it wakes up and can notice it's been killed)

The actual termination happens later: when the process returns from a system call or interrupt, `trap()` checks `proc->killed` and calls `exit()`. This deferred approach is necessary because killing a process while it's in the middle of a file system operation (holding locks, partially updating disk structures) could leave the system in an inconsistent state.

3a.2.14 `procdump()` – print process table for debugging (line 436)

Prints the state, pid, and name of each active process, along with a kernel stack backtrace. Triggered by pressing Ctrl+P on the console. Useful for diagnosing deadlocks and hangs.

3a.2.15 Summary of Previous Functions

Function	Line	Purpose
`pinit()`	24	Initialize `ptable.lock` at boot
`allocproc()`	(static)	Find unused slot, set up kernel stack
`userinit()`	79	Create the first user process (init)
`growproc(n)`	108	Grow or shrink process address space
`fork()`	129	Create child process (copy of parent)
`exit()`	173	Terminate current process -> ZOMBIE
`wait()`	217	Reap zombie child -> UNUSED, free resources
`scheduler()`	266	Per-CPU loop: pick RUNNABLE, switch to it
`sched()`	301	Switch from process back to scheduler
`yield()`	320	Set RUNNABLE, call sched() (preemption)
`sleep(chan, lk)`	352	Block on channel, release lock atomically
`wakeup(chan)`	401	Wake all processes sleeping on channel
`kill(pid)`	412	Mark process for deferred termination

Function	Line	Purpose
`procdump()`	436	Debug: print process table to console

Variable	Line	Purpose
`ptable`	10-13	Global process table (lock + array of `NPROC` procs)
`nextpid`	17	Next pid to assign (incremented by `allocproc()`)

3a.3 Context Switching: The Full Picture

The most subtle aspect of `proc.c` is how context switching works. Here is the complete path when a running process gives up the CPU:

```
Process P (running)                    Scheduler (per-CPU)
-------------------                    -------------------

timer interrupt fires
  |
  v
trap() -> yield()
  |
  v
acquire ptable.lock
P->state = RUNNABLE
  |
  v
sched() -> swtch(&P->context, cpu->scheduler)
  |          |
  |     saves P's {edi,esi,ebx,ebp,eip}
  |     restores scheduler's {edi,esi,ebx,ebp,eip}
  |          |
  |          v
  |     scheduler() resumes (after its swtch call)
  |       |
  |       v
  |     switchkvm()
  |     cpu->proc = NULL
  |     release ptable.lock
  |       |
  |       v
  |     (continues scanning ptable for next RUNNABLE process Q)
  |       |
  |       v
  |     acquire ptable.lock
  |     Q->state = RUNNING
  |     switchuvm(Q)
  |       |
```

```
  |       v
  |     swtch(&cpu->scheduler, Q->context)
  |         |
  |     saves scheduler's {edi,esi,ebx,ebp,eip}
  |     restores Q's {edi,esi,ebx,ebp,eip}
  |         |
  |         v
  |     Q resumes in sched(), returns to yield()
  |     release ptable.lock
  |     returns to trap(), returns to user space
  |
  v
(P sits in ptable with state RUNNABLE
 until the scheduler picks it again)
```

The key insight is that **every process switch goes through the scheduler**. A process never switches directly to another process. The path is always: process -> scheduler -> (possibly different) process. The `swtch()` function is called twice per transition – once to enter the scheduler, once to leave it.

3a.4 The Mechanics of `swtch.S`

The `swtch()` function in `swtch.S` is the lowest-level building block of context switching. It is a short piece of assembly – only about 10 instructions – but it is the mechanism that makes multitasking possible.

3a.4.1 The C prototype

From `defs.h`, the declaration is:

```
void swtch(struct context **old, struct context *new);
```

- **`old`** – a pointer to a `struct context *` where the current register state will be saved. This is typically `&proc->context` (saving into the process table) or `&cpu->scheduler` (saving the scheduler's state).
- **`new`** – a pointer to a previously saved `struct context` to restore. This is typically `cpu->scheduler` (switching to the scheduler) or `proc->context` (switching to a process).

3a.4.2 What the assembly does

`swtch.S` performs the following steps:

```
# Step 1: Get arguments from the stack
movl 4(%esp), %eax     # eax = old (pointer to where to save current context)
movl 8(%esp), %edx     # edx = new (pointer to context to restore)

# Step 2: Save current (callee-saved) registers by pushing onto current stack
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi

# Step 3: Save the current stack pointer into *old
```

```
movl %esp, (%eax)      # *old = esp  (i.e., proc->context = esp)

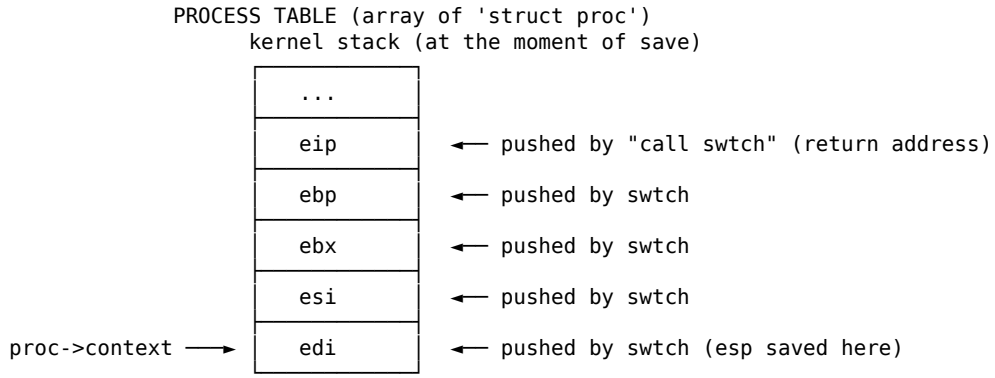
# Step 4: Load the new stack pointer from new
movl %edx, %esp        # esp = new   (i.e., esp = cpu->scheduler, or vice versa)

# Step 5: Restore (callee-saved) registers by popping from the new stack
popl %edi
popl %esi
popl %ebx
popl %ebp

# Step 6: Return (pops eip from the new stack)
ret
```

3a.4.3 Where the context is saved

Step 3 is the critical save operation: `movl %esp, (%eax)`. Since `old` is `&proc->context`, this writes the current stack pointer into **`proc->context`** – a field in the process's entry in the `ptable`. The four registers just pushed (edi, esi, ebx, ebp) are now sitting on the kernel stack, and `proc->context` points to them. Together with the return address (`eip`) that was pushed by the `call swtch` instruction, these five values constitute the `struct context`:



This is why `struct context` in `proc.h` has exactly these five fields in this order:

```
struct context {
  uint edi;    // top of stack (lowest address) -- popped last on restore
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;    // bottom of saved block -- the return address from "call swtch"
};
```

The layout of `struct context` matches the push/pop order in `swtch.S` exactly. The `proc->context` pointer doesn't point to a separately allocated structure – it points directly into the process's kernel stack, at the location where the registers were pushed.

3a.4.4 Why only callee-saved registers?

`swtch()` is called as a normal C function (from `sched()` or `scheduler()`). The x86 C calling convention requires that `swtch` preserve `ebp`, `ebx`, `esi`, and `edi` – these are the callee-saved registers. The caller-saved registers

(`eax`, `ecx`, `edx`) are the caller's responsibility, and the C compiler has already saved them if needed before calling `swtch()`. So `swtch()` only needs to save and restore the callee-saved registers plus the return address.

This means `swtch()` is not saving the *entire* CPU state – just enough to resume a C function that was suspended at a `call swtch` instruction. The full user-space state (all general-purpose registers, segment registers, flags, etc.) is saved separately in the **trapframe** when the process traps into the kernel.

3a.4.5 The two levels of saved state

This is worth emphasizing, because it is a common source of confusion:

What is saved	Where it's saved	When it's saved	What saves it
User-space registers (all of them)	`struct trapframe` on the kernel stack	On trap entry (syscall, interrupt)	`trapasm.S` (hardware + software)
Kernel-thread registers (callee-saved only)	`struct context` on the kernel stack, pointed to by `proc->context`	On context switch	`swtch.S`

When a process is not running, its full state is captured in these two layers on its kernel stack: the trapframe (holding the user state) and the context (holding the kernel state at the point of the switch). Restoring the process means: `swtch()` restores the kernel context -> the process resumes in `sched()` -> returns through `yield()` or `sleep()` -> returns through `trap()` -> `trapret` restores the trapframe -> `iret` returns to user space.

3a.4.6 A concrete example

When the scheduler switches from process P to process Q:

```
In sched() [process P's kernel thread]:
    swtch(&proc->context, cpu->scheduler)
        -> saves P's {edi,esi,ebx,ebp} on P's kernel stack
        -> stores esp into P's ptable entry:  P->context = esp
        -> loads esp from cpu->scheduler
        -> restores scheduler's {edi,esi,ebx,ebp} from scheduler's stack
        -> ret jumps to scheduler's eip (resumes scheduler loop)

In scheduler() [scheduler thread]:
    swtch(&cpu->scheduler, Q->context)
        -> saves scheduler's {edi,esi,ebx,ebp} on scheduler's stack
        -> stores esp into cpu->scheduler
        -> loads esp from Q's ptable entry:  esp = Q->context
        -> restores Q's {edi,esi,ebx,ebp} from Q's kernel stack
        -> ret jumps to Q's eip (resumes Q in sched(), which returns to yield/sleep)
```

Each `swtch()` call is both an ending and a beginning: it saves one execution context and restores another. The beauty of the design is that `swtch()` itself has no idea what it's switching between – it just saves registers, swaps stack pointers, and restores registers. All the policy (which process to run, when to switch) lives in `scheduler()`, `yield()`, and `sleep()`.

Based on xv6 rev8 (September 1, 2015) – MIT 6.828 teaching operating system.