OS - CA3

Q1 – PCB structure and states in xv6

In xv6, the Process Control Block (PCB) is the `struct proc` defined in `proc.h`. Its important fields include:

- `sz`: size of process memory.
- `pgdir`: page table pointer.
- `kstack`: pointer to bottom of kernel stack.
- `state`: process state (of type `enum procstate`).
- `pid`: process identifier.
- `parent`: pointer to parent process.
- `tf`: pointer to trapframe (registers at trap/syscall).
- `context`: pointer to saved kernel context (for `swtch`).
- `chan`: wait channel when sleeping.
- `killed`: flag indicating the process should exit.
- `ofile[NOFILE]`: open file table.
- `cwd`: current working directory inode.
- `name[16]`: process name (for debugging).

The states are defined as:

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

Conceptually this matches the textbook PCB figure: it stores PID, state, parent, resources (files, address space), CPU context (trapframe/context), etc. The naming differs, but the information is essentially the same.

Q2 – Mapping xv6 states to textbook states

A reasonable mapping between the textbook process-state diagram and xv6 is:

Textbook state	xv6 state(s)
`new`	`EMBRYO`
`ready`	`RUNNABLE`
`running`	`RUNNING`
`waiting / blocked`	`SLEEPING`
`terminated`	`ZOMBIE`

Q3 – New → Ready transition in xv6

In xv6, the transition from the textbook *new* to *ready* corresponds to changing

`EMBRYO` → `RUNNABLE`.

The relevant functions in `proc.c` are:

- `allocproc()`:
  - Finds an `UNUSED` `struct proc` in `ptable.proc[]`.
  - Sets `p->state = EMBRYO` and assigns a new `pid`.
  - Allocates the kernel stack and initializes `trapframe` and `context`.
- `userinit()`:
  - Creates the very first user process (`initcode`).
  - After initializing its page table and trapframe, it sets

    $$p- > state = RUNNABLE$$

    under `ptable.lock`.
- `fork()`:
  - Calls `allocproc()` to create a child in state `EMBRYO`.
  - Copies address space and registers.
  - Under `ptable.lock`, sets

    $$np- > state = RUNNABLE.$$

Thus, in the xv6 state machine, the *admitted* / *new* to *ready* transition is:

`EMBRYO` → `RUNNABLE`,

implemented in `userinit()` for the first process and in `fork()` for normal children.

Q4 – Maximum number of processes and behavior on overflow

The maximum number of processes is:

```
#define NPROC 64
```

(as defined in `param.h`).

If a user program calls `fork()` repeatedly and all `NPROC` entries in `ptable.proc[]` are in use, then `allocproc()` fails to find an `UNUSED` slot and returns `0`. Consequently, `fork()` returns `-1` to the caller (indicating failure). The kernel does not panic; only the user program sees that `fork()` failed.

Q5 – Why scheduler must lock the process table

In `scheduler()`, each CPU scans `ptable.proc[]` while holding `ptable.lock`. This is required because:

- Multiple CPUs may run `scheduler()`, `fork()`, `exit()`, `sleep()`, `wakeup()`, etc., concurrently and all of these modify `ptable.proc[]`.
- Without mutual exclusion, inconsistent or torn updates could occur (data races), leading to lost wakeups or corrupted states.

Even on a single-CPU system, xv6 still uses the same locking discipline. Strictly speaking, a uniprocessor kernel that never preempts inside the kernel could get away without the lock, but xv6 is written for multiprocessors and allows interrupts inside the kernel, so the lock is needed for correctness.

Q6 – When can a newly runnable process be scheduled?

Suppose CPU 0 is running `scheduler()` and is currently iterating over `ptable.proc[]`. Meanwhile, on CPU 1 some event (e.g., `wakeup`) sets a process `p` to state `RUNNABLE`.

If CPU 0 has already passed over `p` in its current iteration, then `p` will *not* be selected in that iteration. It will only be considered in the *next* iteration, when the schedulers outer `for(;;)` loop restarts and scans the table again.

So: the process is scheduled in the *next* iteration, not the current one.

Q7 – Registers stored in `struct context`

In `proc.h`, `struct context` saves the subset of registers needed for a kernel context switch:

```
edi, esi, ebx, ebp, eip.
```

Segment registers and caller-saved registers (like `eax`, `ecx`, `edx`) are not saved here, consistent with the x86 calling conventions and how swtch is implemented.

Q8 – Program Counter in context and how it is saved/restored

The Program Counter in xv6 is the x86 register:

`eip`.

The comments in `proc.h` explain that `swtch` does not explicitly save `eip` into `context`, but `eip` is on the stack when the switch occurs, and `allocproc()` arranges the initial stack layout so that `context->eip` is set appropriately (to `forkret`). At a context switch:

- The current kernel stack (including a return address that corresponds to the current `eip`) is saved in the old processs `context`.
- `swtch` then loads the new processs `context` (including its saved `eip`).
- When `swtch` returns, execution resumes at the restored `eip` as if returning from a `call`.

Thus, `eip` is effectively saved/restored as part of the context that `swtch` switches to/from.

Q9 – What if interrupts are not enabled in the scheduler?

At the top of `scheduler()`, each CPU calls `sti()` to enable interrupts. If interrupts remained disabled:

- The local APICs timer interrupt would never be delivered to the CPU.
- Therefore, `trap()` would never see `T_IRQ0 + IRQ_TIMER`.
- Consequently, the call to `yield()` at the end of `trap()` (for timer interrupts) would never occur.

Result: a running process would never be preempted (unless it explicitly calls `yield()` or blocks), and round-robin scheduling based on the timer interrupt would be effectively disabled. In that state, timer interrupts cannot "run" because they are masked by interrupts being disabled.

Q10 – Timer interrupt period

In `lapic.c`, the local APIC timer is configured as follows:

- `lapicw(TDCR, X1);` (divide by 1)
- `lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));`
- `lapicw(TICR, 10000000);`

The timer counts down from `TICR` at the bus frequency and generates a periodic interrupt. The exact period in milliseconds depends on the emulated hardware/bus frequency. In the standard xv6 setup under QEMU, this configuration corresponds roughly to a timer tick of about

$$\approx 10 \text{ milliseconds per tick.}$$

The lab handout suggests confirming this experimentally by printing time or `ticks` and comparing with wall-clock time.

Q11 – Which function causes the "interrupt" transition in the state diagram?

The hardware interrupt (e.g., timer interrupt) is first handled by the generic `trap(struct trapframe *tf)` function. For the timer interrupt (`T_IRQ0 + IRQ_TIMER`):

- `trap()` increments `ticks` and calls `wakeup(&ticks)` (on CPU 0).
- At the end of `trap()`, if the current process is in state `RUNNING` and the trap was the timer:

```
if (myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER) yie
```

Thus, the key function implementing the "interrupt-driven" transition from `RUNNING` to `RUNNABLE` (and back to scheduler) is `yield()`, which is called from `trap()` in response to the timer interrupt.

Q12 – Time quantum of round-robin scheduling

xv6 uses round-robin scheduling where each process gets one timer tick before being preempted (unless it blocks or voluntarily yields). From Q10, one tick is approximately 10 ms on the reference setup.

Therefore, the scheduling quantum is:

$$\text{Quantum} \approx 10 \text{ ms.}$$

More precisely: one "quantum" is exactly one local APIC timer interrupt interval, whose duration you can measure empirically.

Q13 – Which function does `wait()` ultimately use to sleep?

At the bottom of `wait()`, after scanning the process table for exited children, if none is found but there are children, xv6 calls:

```
sleep(curproc, &ptable.lock);
```

Thus, `wait()` ultimately uses the generic `sleep()` function to block the parent until a child becomes `ZOMBIE`.

Q14 – Other uses of `sleep()`

`sleep()` is the general blocking primitive in xv6. Other typical usages include:

- Waiting for I/O completion in `ide.c` (disk driver).
- Waiting for data/space in `pipe.c` (pipe read/write).
- Waiting for log space in `log.c`.

Any code that wants to block until some condition changes will usually call `sleep(chan, lock)` and be woken up by a matching `wakeup(chan)`.

Q15 – Waking up processes and the corresponding state transition

(a) Kernel function that wakes up waiting processes

The public interface is:

```
void wakeup(void *chan);
```

which, under `ptable.lock`, calls the internal helper:

```
static void wakeup1(void *chan);
```

`wakeup1` scans the process table and sets any `SLEEPING` processes with matching `p->chan == chan` to `RUNNABLE`.

(b) State transition in the textbook diagram

This corresponds to:

$$\texttt{waiting / blocked} \rightarrow \texttt{ready}.$$

In xv6 terms:

$$\texttt{SLEEPING} \rightarrow \texttt{RUNNABLE}.$$

(c) Other functions that cause the same transition

Besides `wakeup`/`wakeup1`, the `kill(int pid)` system call also may change:

$$\texttt{SLEEPING} \rightarrow \texttt{RUNNABLE}$$

for a process that is sleeping: `kill()` sets `p->killed = 1` and, if `p->state == SLEEPING`, directly sets `p->state = RUNNABLE`. Also, `exit()` uses `wakeup1` to wake the parent waiting in `wait()`.

Q16 – Orphan processes in xv6

In `exit()`, xv6 handles children of the exiting process as follows:

- It iterates over `ptable.proc[]`.
- For any process `p` with `p->parent == curproc`, it sets

  $$p- > parent = initproc;$$

  i.e., re-parents all children to the special `init` process.
- If such a child is already in state `ZOMBIE`, it calls `wakeup1(initproc)` so that `init` can `wait()` and clean it up.

Thus, orphan processes are adopted by `init`, which eventually calls `wait()` to reap them, preventing zombies from accumulating.

Q17 – CPU structure in xv6

In `proc.h`, `struct cpu` (per-CPU state) is defined roughly as:

- `uchar apicid;`
  Local APIC ID of this CPU (used to match hardware CPU to `cpus[]` entry).
- `struct context *scheduler;`
  Saved context to switch into the scheduler; `swtch()` jumps here to run `scheduler()`.
- `struct taskstate ts;`
  x86 task state segment; tells hardware which kernel stack to use on interrupts.
- `struct segdesc gdt[NSEGS];`
  Global descriptor table for this CPU (code/data segments, TSS, etc.).
- `volatile uint started;`
  Flag indicating whether the CPU has finished its initialization.
- `int ncli;`
  Nesting depth of `pushcli()` (interrupt-disable) calls on this CPU.
- `int intena;`
  Whether interrupts were enabled before the outermost `pushcli()`; used to restore interrupt state correctly.
- `struct proc *proc;`
  Pointer to the process currently running on this CPU (or `0` if idle in the scheduler).

Together, these fields allow xv6 to manage per-CPU state: they track which process is running where, how to switch into the scheduler, and how to handle interrupts and privilege-level transitions.

AI Chat References

`https://gemini.google.com/share/d53459879b52`
`https://gemini.google.com/share/941a1a80f24d`