***In God we trust***

# Os lab EX3

Mani Hosseini 810102552
Shayan Maleki 810102515
Hamid Mahmoudi 810102549

---

## *Questions :*

*We Used Gemini LLM to help with answering the questions you can see the chat history in the link below:*

chat history

# 3.

**1 & 2 )**
In this question we are to find the PCB inside the xv6 which is located in the proc.c and proc.h file. With further examinations we came across the struct named proc with many attributes:

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  int priority;                // process priority
  int qticks;                  // ticks used in current time slice (for RR quantum)
  uint creation_time;         // Time when process was created
  int home_core;               // which CPU this process is assigned to at first

  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

- **`proc.state`: Mapping Conceptual States to xv6 Implementation**

  The process state diagram from the textbook provides a high-level, conceptual model of a process's lifecycle. In contrast, the `enum procstate` found in xv6's `proc.h` offers a more granular and concrete implementation view. The names used in xv6 are not arbitrary; they are chosen to be more descriptive of what is *actually happening* within the kernel's code.

- **From New to EMBRYO**

The **New** state corresponds to the **`EMBRYO`** state in xv6. The name `EMBRYO` is intentionally more descriptive. While “New” simply means a process is being created, `EMBRYO` pinpoints a specific phase in that creation.

- **From Ready to RUNNABLE**

  The conceptual **Ready** state is implemented as **`RUNNABLE`** in xv6. This is primarily a semantic difference.

- **Running**

  The **Running** state is named **`RUNNING`** in xv6. There is no difference here.

- **From Waiting to SLEEPING**

  The diagram’s **Waiting** state is called **`SLEEPING`** in xv6. This is standard terminology in Unix-like systems. `SLEEPING` is more specific than “Waiting.” It implies that the process has voluntarily relinquished the CPU by calling the `sleep()` function, usually to wait for a specific event like I/O completion. A `SLEEPING` process is not in contention for the CPU and can only be woken up by a corresponding `wakeup()` call on the specific channel (`chan`) it is waiting on. The term implies a well-defined, voluntary pause.

- **From Terminated to ZOMBIE**

  The most significant and descriptive difference is the mapping of the **Terminated** state to the **`ZOMBIE`** state. “Terminated” implies a final, complete end. However, in xv6 (and other Unix-like systems), when a process calls `exit()`, it cannot be immediately erased from the system. The kernel must preserve its process control block to store its final exit status, which the parent process needs to collect via the `wait()` system call. Therefore, a `ZOMBIE` process is one that is “dead” (it has ceased execution) but is also “undead” (its PCB still occupies a slot in the process table). It performs no computation but cannot be fully reaped until its parent acknowledges its death.

- **The UNUSED State: An Implementation Detail**

  Finally, xv6 includes an **`UNUSED`** state that has no equivalent in the conceptual diagram. The diagram is an abstract model and does not concern itself with the low-level management of the PCB list. xv6, however, uses a fixed-size array (`ptable.proc[NPROC]`) to store all PCBs. The `UNUSED` state is a purely administrative flag used by `allocproc()` to mark a slot in this array as empty and available for a new process. It is a practical necessity for the kernel’s resource management.

- 

`int pid;`

- **Conceptual Role:** The **Process ID (PID)** is a unique integer that serves as the official identifier for a process throughout the system. While humans might refer to processes by name (e.g., `sh`, `ls`), the kernel uses the PID for all unambiguous operations like termination, waiting, or debugging.

- **Implementation in xv6:**

  A new, unique PID is assigned in `allocproc()` by incrementing a global counter: `p->pid = nextpid++;` (`proc.c`, line 193).

  It is used by the `kill(int pid)` system call to find the target process by iterating through the process table (`ptable`) and comparing `p->pid` with the requested `pid` (`proc.c`, line 515).

  The `fork()` system call returns the child's `pid` to the parent process, allowing the parent to manage its child.

`struct proc *parent;`

- **Conceptual Role:** This pointer creates the process hierarchy (tree structure). Every process (except the very first one) has a parent. This relationship is critical for:

1. **Cleanup:** A parent process is expected to clean up its "zombie" children using the `wait()` system call.
2. **Signaling/Communication:** It defines which processes can wait for which other processes to terminate.

`int killed;`

- **Conceptual Role:** This field acts as a flag for **asynchronous termination**. When `kill(pid)` is called, the target process might be in the middle of a critical kernel operation or sleeping. Terminating it instantly could corrupt kernel data structures. The `killed` flag is a "request" for the process to terminate itself at the next safe opportunity.

`int priority;` **(Lab-specific extension)**

- **Conceptual Role:** This field is used by the scheduler to implement **priority scheduling**. Processes with higher priority (in this lab, a lower number like `PRI_HIGH = 0`) should be chosen to run before processes with lower priority.

`int home_core;`

- **Conceptual Role:** This field implements **processor affinity**, assigning a process to a default or "home" CPU core. In a multi-core system, especially a heterogeneous one with different types of cores (Performance vs. Energy-efficient), this is crucial for load balancing and scheduling decisions.

---

**3)**

The creation of the first user process in xv6 begins during the kernel boot sequence. After initializing various hardware subsystems (memory, interrupts, locks), the `main` function calls `userinit()`. This function is responsible for manually crafting the first process (`initcode`) without using the `fork()` system call, which serves as the template for all future processes.

```
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  picinit();       // disable pic
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}
```

The journey begins when `userinit` calls `allocproc` to obtain a slot in the process table. This function is responsible for the initial state transition from “non-existent” to “under construction.”

**The Search for Space:**

Inside `allocproc`, the kernel acquires the `ptable.lock` to ensure atomicity. It iterates through the fixed-size `ptable.proc` array looking for a slot with the state **`UNUSED`**.

```c
// proc.c: line 190
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  if (p->state == UNUSED)
    goto found;
```

**Transition 1: UNUSED →\rightarrow→ EMBRYO**

Once a free slot is found, the state is immediately changed to **`EMBRYO`**. This is the first critical state transition.

- **Reasoning:** The process now has an identity (`pid`) and a slot in the table, but it is not yet ready to run. It lacks a memory map and valid register context. It is effectively in a "gestational" phase.

```
// proc.c: line 196
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
```

**Resource Initialization:**

While in the `EMBRYO` state, `allocproc` allocates the kernel stack (`kalloc()`) and sets up the `context` pointer. This context is prepared so that when the process eventually runs, it starts execution at `forkret`. Finally, `allocproc` returns the pointer `p` back to `userinit`.

## The Initialization Phase: `userinit()`

Back in `userinit`, the process pointer `p` is currently in the `EMBRYO` state. The function now proceeds to "flesh out" the process by assigning it memory and execution context.

**Memory and Registers:**

1. **Page Directory:** `setupkvm()` creates the kernel page table.
2. **Code Loading:** `inituvm()` loads the binary machine code (`initcode`) into the process's memory.
3. **Trapframe Setup:** The Trapframe (`tf`) is manually configured with User Mode segments (`SEG_UCODE`, `SEG_UDATA`) and the instruction pointer (`eip`) is set to 0 (the entry point of the init code).

**Transition 2: EMBRYO →\rightarrow→ RUNNABLE**

Once the memory is allocated and registers are configured, the process is fully formed. `userinit` acquires the `ptable.lock` one last time to perform the final state change.

```
// proc.c: line 280
acquire(&ptable.lock);

p->state = RUNNABLE;
p->priority = PRI_NORMAL;
rq_push(&cpus[p->home_core], p);
```

## Transition Analysis

In response to the specific question regarding the state transitions of a process during its creation in xv6 (Admitted 2.3, Question 3), the analysis of `proc.c` reveals the following specific chain of states:

UNUSED⟶EMBRYO⟶RUNNABLE

1. **UNUSED:** The initial state of the process table slot.

2. **EMBRYO:** The intermediate state assigned in `allocproc()` while the kernel stack and context are being allocated.

3. **RUNNABLE:** The final state assigned in `userinit()` (or `fork`) after the virtual memory and trapframe are fully initialized, marking the process as ready for the scheduler.

**4)**

## 1. Maximum Process Limit in xv6

The first part of the question asks for the maximum number of processes possible in xv6. This limit is defined by the constant `NPROC` in `param.h`, which is typically set to 64.

This limit is **system-wide**, not per-CPU. The core data structure is the global process table,

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC]; // Global array of all process structures
} ptable;
```

The second part of the question asks what happens when this limit is exceeded. To analyze this, we trace the execution path of a `fork()` system call when the process table is already full.

**The User Program's Request**

The journey begins in a user-level program. This program has already created 64 processes (or there are 64 processes total in the system), filling the `ptable`. It then executes the `fork()` system call one more time, intending to create a 65th process.

```
// In a user program, e.g., my_test.c
pid = fork(); // Attempt to create the 65th process
```

This triggers a system call trap, causing the CPU to switch from User Mode to Kernel Mode and begin executing the kernel's `fork()` function.

**Entering the Kernel's `fork()` Function**

The kernel's `fork()` function (`proc.c`, line 351) has one primary initial task: to allocate a new process structure for the child. It does this by calling `allocproc()`.

```
int fork(void)
{
  // ...
  // Allocate process.
  if ((np = allocproc()) == 0) { // <-- The critical check
    return -1;
  }
  // ... (code to copy memory, etc. is never reached)
}
```

**The Failure Inside `allocproc()`**

`allocproc()` is responsible for finding an empty slot in the global `ptable`. It acquires a lock and iterates through the entire `proc` array, searching for a structure with the state `UNUSED`.

```
// proc.c: line 190
static struct proc *
allocproc(void)
{
  // ...
  acquire(&ptable.lock);

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if (p->state == UNUSED)
      goto found; // <-- This jump is never taken

  // The loop finishes without finding an UNUSED slot.
  release(&ptable.lock);
  return 0; // <-- KERNEL REACTION: Return a null pointer (0)
}
```

**Kernel Reaction:** In our scenario, the loop runs from the first process to the 64th and finds that none have the state `UNUSED`. The loop completes, and execution falls through to the lines after it. The kernel does not panic or crash. Its defined behavior is to release the lock and **return `0` (a null pointer)**, signaling that allocation has failed.

**The `fork()` Function's Response**

Execution returns from `allocproc()` back to the `fork()` function. The `if` condition now evaluates to `true` because `allocproc()` returned `0`.

```
// proc.c: line 355
if ((np = allocproc()) == 0) { // This is now if (0 == 0), which is true
  return -1; // <-- USER FEEDBACK: fork() returns -1
}
```

The `fork()` function immediately executes the code inside the `if` block, which is to `return -1`. This `-1` is the return value of the entire system call.

**The Final Feedback to the User**

The system call is complete. The kernel returns control to the user-level program. The `pid` variable in the user program, which was assigned the result of `fork()`, now holds the value `-1`.

```
// In a user program, e.g., my_test.c
pid = fork(); // This call returns -1

if (pid < 0) {
  printf("fork failed: out of processes!\n"); // A well-written program handles the error
}
```

---

**5)**

## The Multi-Core System (Preventing Parallel Race Conditions)

In a multi-core system, each CPU core runs its own instance of the `scheduler()` function when it has no process to run. They all look at the same shared data structure: the global `ptable` and the per-CPU ready queues.

**The Problem:**

Without a lock, the following disastrous scenario could happen:

1. **Core A** enters its `scheduler()` loop and starts scanning the ready queue. It finds a `RUNNABLE` process, `P`.
2. **Simultaneously**, **Core B** enters its `scheduler()` loop and also starts scanning its ready queue. Let's imagine process `P` was moved to Core B's queue by the load balancer. It also finds the same `RUNNABLE` process `P`.
3. **Core A** decides to run `P`. It sets `P->state = RUNNING` and performs a context switch to `P`.
4. **Core B**, unaware of Core A's actions, *also* decides to run `P`. It also sets `P->state = RUNNING` and performs a context switch to `P`.

**The Consequence:**

The system would immediately crash. You would have two different CPU cores trying to execute the same process, using the same kernel stack and memory space. This would lead to complete data corruption, a "double free" situation when the process exits, and a kernel panic.

**The Solution (`ptable.lock`):**

The `acquire(&ptable.lock)` call ensures **mutual exclusion**. It guarantees that only one CPU core can be inspecting and modifying the process states and ready queues at any given moment. The entire operation of "finding a runnable process, removing it from the queue, and marking it as running" becomes **atomic**.

When Core A acquires the lock, Core B must wait. By the time Core A is done and releases the lock, process `P` is no longer `RUNNABLE` and is no longer in any ready queue, so Core B will simply skip it and look for a different process.

## As for the single core :

**No, the lock is not redundant; it is still absolutely necessary on a single-core system.** The lock is required to prevent race conditions between the scheduler's code and interrupt handler code that can run on the same core.

**Scenario 1: Without the `ptable.lock` (Incorrect Behavior)**

Imagine the scheduler is scanning the process table to find a runnable process. It has already checked array indices 0 through 19 and is now at index 20. At this exact moment, a disk interrupt occurs for a process that happens to be at index 10.

1. The CPU stops running the scheduler and jumps to the interrupt handler.
2. The handler executes the `wakeup()` function, which finds the process at index 10 and changes its state from `SLEEPING` to `RUNNABLE`.
3. The interrupt handler finishes. The CPU resumes the scheduler code exactly where it left off, at index 20.
4. The scheduler continues its scan from index 20 to the end of the table.

**The result is a bug:** The scheduler has already passed index 10, so it completely misses the fact that the process there is now ready to run. This "lost wakeup" causes the process to wait unnecessarily for the scheduler to complete its entire current cycle and start a new one from the beginning, introducing significant latency.

**Scenario 2: With the `ptable.lock` (Correct Behavior)**

The scheduler first acquires the `ptable.lock`, which also disables interrupts on the core.

1. The scheduler holds the lock and begins scanning the process table.
2. The disk interrupt signal arrives, but since interrupts are disabled, the CPU ignores it and the interrupt remains "pending." The scheduler's execution is not interrupted.
3. The scheduler completes its entire scan of the process table on a stable, unchanged set of data.
4. After its scan, the scheduler releases the `ptable.lock`. This action re-enables interrupts.
5. The CPU immediately handles the pending interrupt, and the process at index 10 is safely set to `RUNNABLE`.

**The result is correct behavior:** The system state remains consistent. On its very next scan, the scheduler is guaranteed to see the `RUNNABLE` process at index 10 and schedule it to run. The lock ensures the scheduler's operation is **atomic** and cannot be corrupted by an untimely interrupt.

---

**6)**

**Scenario Setup:**

> `CPU_A` is idle and is executing the `scheduler()` function. It has acquired `ptable.lock` and is in the middle of its `for` loop (Step 3).
>
> `CPU_B` is busy. It finishes an I/O operation and an interrupt handler on `CPU_B` calls `wakeup()`. This `wakeup()` call will make a process, let's call it `Proc_P`, `RUNNABLE`.

The key is that for `CPU_B` to change `Proc_P`'s state, it *also* needs to acquire `ptable.lock`. Since `CPU_A` already holds the lock, `CPU_B` must wait (spin). `CPU_B` can only make `Proc_P` runnable *after* `CPU_A` releases the lock. This happens when `CPU_A` completes one full scan of the process table.

Let's consider the two cases.

---

## Case 1: The "Unlucky" Case (Next Iteration)

This happens if the newly ready process is located at an index *before* `CPU_A`'s current position in the table.

1. **`CPU_A` scans:** `CPU_A` holds the lock and is scanning the process table. Let's say it has already scanned up to index `i=30` and `Proc_P` is at index `j=10`.
2. **`CPU_A` continues:** Since `CPU_A` has already passed index 10, it will not see `Proc_P` in this iteration. It continues scanning from index 31 to the end (`NPROC-1`). It finds no runnable processes.
3. **`CPU_A` releases lock:** Having completed its full scan, `CPU_A` reaches the end of its `for` loop and executes `release(&ptable.lock)` (Step 5).
4. **`CPU_B` gets lock:** `CPU_B`, which was waiting, now successfully acquires `ptable.lock`. It finds `Proc_P` (at index 10), changes its state from `SLEEPING` to `RUNNABLE`, and then releases the lock. `CPU_B` then returns from its interrupt.
5. **`CPU_A` starts over:** `CPU_A` goes back to the top of its infinite `for(;;)` loop (Step 1). It re-acquires `ptable.lock` (Step 2) and starts a **brand new scan** from the beginning of the table (index 0).

6. **CPU_A finds the process:** As CPU_A scans, it reaches index 10, sees that Proc_P->state is now RUNNABLE, and schedules it for execution.

**Conclusion for Case 1:** If the awakened process is located at an array index that the scheduler has already passed, it will only be found and scheduled in the **next iteration**.

---

## Case 2: The "Lucky" Case (Same Iteration)

This is a more subtle case that is theoretically possible, though less likely. It requires the scheduler on one CPU to be preempted by an interrupt.

1. **CPU_A scans:** CPU_A is scanning the process table. It is at index i=10. The process that will be awakened, Proc_P, is at a future index, j=30.
2. **Interrupt on CPU_A:** An interrupt occurs *on CPU_A*. CPU_A pauses its scheduler scan, saves its state, and runs the interrupt handler.
3. **Handler wakes process:** This interrupt handler's execution path leads to wakeup() being called. Since the code is running on CPU_A which already holds ptable.lock, it can proceed without waiting. It changes the state of Proc_P (at index 30) to RUNNABLE.
4. **Handler finishes:** The interrupt handler completes. CPU_A restores its saved state and resumes executing the scheduler function *exactly where it left off*—still inside the for loop, holding the lock, about to check index 11.
5. **CPU_A finds the process:** CPU_A continues its scan. When its loop variable p points to the process at index 30, it will see that p->state == RUNNABLE and will schedule it for execution.

**Conclusion for Case 2:** If a process is made RUNNABLE by an interrupt on the *same CPU* that is currently scheduling, and that process is located at an index *after* the scheduler's current scan position, it can be found and scheduled in the **same iteration**.

---

7)

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

As we can see if we go back to the proc struct we would see a context attribute which store the:
**`eip` (Instruction Pointer): The Program Counter**

It saves the memory address of the very next instruction the process was about to run. It is the program's "bookmark."

**`ebp` (Base Pointer): The Stack Frame Anchor**

It saves a stable pointer to the base of the current function's workspace on the stack. This is crucial for finding local variables and for correctly returning from a chain of function calls.

**`ebx` (Base Register): A General-Purpose Work Register**

It saves a general-purpose value (like a counter or a pointer) that the interrupted code was using. The operating system promises to preserve it so the program can resume without losing its data.

**`esi` (Source Index): The Source Pointer**

It saves a pointer that typically points to the **source** of data for an operation (like copying a string from one place to another).

**`edi` (Destination Index): The Destination Pointer**

It saves a pointer that typically points to the **destination** for a data operation (where the string is being copied *to*).

**8)**

To fully answer Question  regarding the role of the Program Counter (PC) in `struct context`, the execution flow must be traced from the moment a user process is interrupted until it resumes. This process involves two distinct Program Counters: the User PC and the Kernel PC.

**1. Execution and Interrupt (Saving the User PC)**

The process begins by executing a user-level program on the CPU. Suppose the execution has reached line 5 of the assembly code, meaning the physical Program Counter (EIP) is holding the value `5`. At this exact moment, a hardware interrupt (such as a timer tick) occurs. The CPU hardware immediately responds by stopping the user code and switching to kernel mode. During this transition, the hardware automatically takes the current value of the EIP (`5`) and pushes it onto the process's kernel stack. This saved value is stored in the Trapframe structure (`p->tf->eip`). At this stage, the User PC is safely preserved, and the CPU begins executing the kernel's interrupt handler code.

**2. The Decision to Switch (Saving the Kernel PC)**

The kernel interrupt handler executes and determines that the current process should yield the CPU. The handler calls the `yield()` function, which subsequently calls `sched()`.

```
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;
}
```

Inside `sched()`, the kernel prepares to put the process to sleep and switch to the scheduler. The function `swtch(&p->context, ...)` is called. At this precise moment, the `swtch` function takes the address of the *next* instruction (the return address inside `sched`) and saves it into the Process Control Block's context structure (`p->context->eip`). This saved address

represents the Kernel PC. It marks exactly where the kernel thread stopped executing so it can later finish the `sched` function.

**3. Restoration of the Kernel Context**

After some time, when the process becomes `RUNNABLE` again, the Scheduler loop selects it for execution.

```
if (p)
{
  c->proc = p;
  switchuvm(p);
  p->qticks = 0;
  p->state = RUNNING;

  if (SCHED_DEBUG)
  {
    cprintf("SCHED: cpu %d core_type %s running pid %d (ct=%d, home=%d)\n",
            id,
            (c->core_type == CORE_E ? "E" : "P"),
            p->pid,
            p->creation_time,
            p->home_core);
  }

  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);
```

The scheduler calls `swtch(&c->scheduler, p->context)`. This function loads the saved Kernel PC from `p->context->eip` back into the CPU's actual EIP register. Consequently, the CPU immediately jumps back to the code inside the `sched()` function, exactly where it left off in Step 2. The kernel thread is now running again, allowing the function calls to return in sequence (`sched` returns to `yield`, which returns to the `trap` handler).

**4. Return to User Mode (Restoring the User PC)**

Finally, the kernel finishes handling the interrupt and prepares to return to user space. The trap handler executes the return code (`trapret`), which looks at the Trapframe saved on the stack in Step 1. It extracts the original User PC value (5) from `p->tf->eip` and places it into the CPU's EIP register while switching back to user mode. The process resumes execution at line 5 of the user program, completely unaware that it was ever interrupted.

# Interrupt 5.3 :

This section analyzes the sequence of events within the `trap()` function in `trap.c` when a timer interrupt (`T_IRQ0 + IRQ_TIMER`) occurs for a process in the `RUNNING` state. As described in the project documentation, this event triggers two primary actions: incrementing the system time and potentially yielding the CPU.

**1. System Time Increment**

The first action is the increment of the global system clock. The code responsible for this is located within the `switch` statement that handles different trap numbers.

```
switch (tf->trapno)
{
case T_IRQ0 + IRQ_TIMER:
  if (cpuid() == 0)
  {
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }
  lapiceoi();
  break;
```

**Global Time (`ticks`)**: The `ticks` variable is a global counter that represents the system's uptime, measured in clock ticks. It is a shared resource for the entire operating system.
**Single Timekeeper (`cpuid() == 0`)**: In a multi-core system, every CPU receives its own timer interrupts. If every core were to increment the global `ticks` variable, the system clock would advance incorrectly (i.e., `NCPU` times too fast). To prevent this and to avoid contention on the `tickslock`, the system designates a single core—by convention, CPU 0—as the sole

timekeeper. Only when the interrupt occurs on CPU 0 is the `ticks` counter incremented. Other CPUs receiving the timer interrupt will skip this block and simply acknowledge the interrupt .

**2. Second event :**

The second action is to check if the currently running process has exhausted its time slice and should be preempted. This logic is executed after the initial handling of the interrupt.

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

  struct proc *p = myproc();
  int id = cpuid();

  if (id % 2 == 0)
  {
    // E-core: 30 ms quantum (3 ticks)
    p->qticks++;

    // ...

    if (p->qticks >= QUANTUM_TICKS)
    {
      p->qticks = 0;
      yield();
    }
  }
```

The code first ensures that the conditions for preemption are met:

1. `myproc()`: A process is currently assigned to and running on this CPU.
2. `myproc()->state == RUNNING`: The process is actively executing instructions, not `SLEEPING` or in another state.
3. `tf->trapno == T_IRQ0 + IRQ_TIMER`: The interrupt that caused this trap was indeed the timer, which is the trigger for preemptive scheduling.

If these conditions are true and the process is on an E-core (`id % 2 == 0`), its time slice counter (`p->qticks`) is incremented. If this counter reaches the predefined quantum (`QUANTUM_TICKS`), it signifies the process has used its allotted time. The counter is reset, and

the `yield()` function is called. The `yield()` function is responsible for changing the process's state from `RUNNING` to `RUNNABLE` and invoking the scheduler to select a new process to run. This mechanism enforces the time-slicing policy for preemptive multitasking.

**9)**

Now lets answer the question 9 :

At first we need to understand what sti is and what it does:

## Understanding `sti` and Interrupts

**The Interrupt Flag (IF):** Every CPU has a special register called `EFLAGS` which contains several control bits. One of these bits is the **Interrupt Flag (IF)**.

When the **IF bit is set to 1**, the CPU will respond to external hardware interrupts (like the timer, keyboard, disk controller, etc.).

When the **IF bit is cleared to 0**, the CPU will ignore (mask) all of those interrupts.

**The Instructions:**

`sti` (Set Interrupt Flag): This assembly instruction sets the `IF` bit to 1. It's like telling the CPU, "You are now allowed to be interrupted by hardware."

`cli` (Clear Interrupt Flag): This instruction sets the `IF` bit to 0. It tells the CPU, "Do not get distracted by any hardware interrupts until I say so."

The `sti()` instruction at the beginning of the `scheduler`'s main loop is fundamentally important for the operating system to function. If it were removed, the system could easily deadlock and freeze completely.

```
void scheduler(void)
{
  struct cpu *c = mycpu();
  c->proc = 0;

  int id = c - cpus;
  c->core_type = (id % 2 == 0) ? CORE_E : CORE_P;

  static int last[NCPU] = {0};

  for (;;)
  {
    // Enable interrupts on this processor.
    sti();
```

To understand the problem, we must consider the case where the scheduler's ready queue is empty.

Imagine a scenario where all processes in the system are in the `SLEEPING` state, waiting for an event like disk I/O to complete. There are no processes in the `RUNNABLE` state.

**Scheduler's Job:** The CPU, having no process to run, will be executing the `scheduler()` function. The scheduler's job is to continuously loop, looking for a process that becomes `RUNNABLE`.

**How Do Processes Become Runnable?** A sleeping process becomes `RUNNABLE` when the event it's waiting for occurs. This notification almost always comes from a **hardware interrupt**. For example:

- A disk finishes reading data and sends an **IDE interrupt**. The handler `ideintr()` runs and calls `wakeup()` on the sleeping process.

**The Deadlock without `sti()`:**

When the kernel enters the `scheduler`, interrupts are typically disabled (from a previous `cli` call in `sched()` or `sleep()`).

If the `sti()` instruction is **removed** from the scheduler loop, the CPU's Interrupt Flag will remain **0**.

The scheduler will loop forever, searching for a `RUNNABLE` process, but it will never find one.

When the disk I/O completes and the hardware sends an interrupt signal, the CPU will **completely ignore it** because interrupts are disabled.

The interrupt handler will never run. The `wakeup()` call will never happen. The sleeping processes will never be marked as `RUNNABLE`.

The CPU is now stuck in an infinite loop inside the scheduler, and the processes that could give it work to do will never wake up. The entire system is frozen.

**Conclusion for Part 1:** Without `sti()`, the scheduler becomes a "black hole" for an idle CPU. The CPU spins uselessly and is deaf to the external events that are necessary to make new work available.

**Can Any Interrupt Be Executed?**

- **With `sti()` (Normal Operation):** Yes. The entire point of calling `sti()` is to allow the CPU to be interrupted while it is searching for a task. This allows a timer interrupt to preempt a long-running process or an I/O interrupt to wake up a sleeping process, making the system responsive.

- **Without `sti()` (The Problem Scenario):** No. In the state where the scheduler is looping with interrupts disabled, no maskable hardware interrupts (timer, disk, keyboard) can be executed. The CPU will not transfer control to their respective handlers. This is precisely what causes the deadlock.

**10)**

The frequency at which the timer interrupt fires is determined by the configuration of the Local APIC (LAPIC) registers in the `lapicinit` function. Three specific lines of code control this behavior by setting the initial count, the clock speed divider, and the operating mode.

**1. Setting the Initial Count (`TICR`)**

The timer operates as a countdown mechanism. The first step is setting the value from which it starts counting down.

lapicw(TICR, 10000000);

This line sets the **Timer Initial Count Register** to 10,000,000. The hardware will start at this number and decrement it by one at every clock tick.

**2. Setting the Speed / Divider (`TDCR`)**

The speed at which the countdown happens depends on the system's bus frequency. The hardware allows us to slow this down by dividing the bus clock.

lapicw(TDCR, X1);

This line configures the **Timer Divide Configuration Register**. The value `X1` specifies a "divide by 1" operation. This means the timer will count down at the **full speed** of the system bus frequency, without being slowed down.

Note that we can set the X1 to a larger number this would increase the speed of the counter.

But since we want the normal speed of the processor we set it to 1.

**3. Setting the Mode (`PERIODIC`)**

Finally, we must tell the timer what to do when the countdown reaches zero.

lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));

This line does two critical things:

1. It sets the timer to **PERIODIC** mode. This means that immediately after the counter reaches 0, it automatically reloads the initial value (10,000,000) and begins counting down again. This creates a repeating cycle.

2. It maps the timer to the `IRQ_TIMER` interrupt vector, ensuring the CPU is interrupted every time the cycle completes.

**Conclusion on Frequency:**

The timer interrupt is triggered every time 10,000,000 bus cycles pass. The frequency in Hertz (Hz) is calculated as:

$$\text{Interrupt Frequency} = \frac{\text{Bus Frequency}}{\text{Initial Count}}$$

For example, if the system bus is running at 1 GHz (1,000,000,000 Hz), the calculation is:

$$\frac{1,000,000,000}{10,000,000} = 100 \text{ Hz}$$

In this scenario, the timer interrupt triggers 100 times per second (once every 10ms).

**Relation to `ticks`:**

The global variable `ticks` is the system's way of counting these hardware interrupts. In `trap.c`, the interrupt handler catches the timer signal:

```
case T_IRQ0 + IRQ_TIMER:
  if (cpuid() == 0)
  {
    acquire(&tickslock);
    ticks++;  // <--- The hardware interrupt increments this software counter
    wakeup(&ticks);
    release(&tickslock);
  }
```

Every time the hardware timer completes a cycle (e.g., 100 times a second), the `trap` function runs, and CPU 0 increments the `ticks` variable. Thus, `ticks` represents the total time elapsed since boot, measured in these timer intervals.

---

**11) first we dig into the yield and sched mechanism then will proceed to the question :**

## Mechanism of `yield()` and `sched()` in Context Switching

This section details how a running process voluntarily (or forcefully via timer) gives up the CPU to allow other processes to run. This sequence is the core of the **Round-Robin** scheduling mechanism.

### 1. The Trigger: `yield()`

When the timer interrupt occurs in `trap.c` and the specific quantum logic determines it is time to switch, the function `yield()` is called. Its primary purpose is to move the process from `RUNNING` back to `RUNNABLE` so it can be scheduled again later.

**Key Operations in `yield()`:**

1. **Acquire Lock:** It acquires `ptable.lock`. This is critical because we are about to change the process state and the ready queue, which are shared resources.
2. **Change State:** It changes the process state from `RUNNING` to `RUNNABLE`.
3. **Queue:** It adds the process back to the ready queue (using `rq_push` in this specific implementation) so it doesn't get lost.
4. **Call Scheduler:** It calls `sched()` to perform the actual context switch.

```
void yield(void)
{
  acquire(&ptable.lock);  // 1. Lock the process table
  myproc()->state = RUNNABLE; // 2. Change state so scheduler knows it's paused

  // 3. Add back to ready queue (specific to this lab's multi-queue implementation)
  rq_push(&cpus[myproc()->home_core], myproc());

  sched(); // 4. Call the lower-level switch function
  release(&ptable.lock);  // 5. Release lock ONLY after we return (are rescheduled)
}
```

### 2. The Switcher: `sched()`

The `sched()` function is the gateway to the context switch. It is a safety wrapper around the assembly `swtch` function. It ensures that the system is in the correct state before switching stacks.

**Key Operations in `sched()`:**

1. **Sanity Checks:** It verifies that the process is holding the lock (so no one else messes with it during the switch) and that interrupts are disabled (to prevent corruption during the delicate stack switch).

```
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");
```

2. **The Context Switch:** It calls `swtch`. This is the magic moment.

   It saves the **Current Process's registers** into `p->context`.

   It loads the **CPU's Scheduler registers** from `c->scheduler`.

```
swtch(&p->context, mycpu()->scheduler);
```

3. **The "Pause":** The CPU jumps to the scheduler's code. The execution flow of the *current process* effectively stops at the `swtch` line.

**3. The Return (Resuming Execution)**

It is crucial to understand that `yield()` calls `sched()`, and `sched()` calls `swtch()`.

When the process is picked again by the scheduler in the future:

1. The scheduler calls `swtch`.
2. Registers are restored from `p->context`.
3. The instruction pointer (`eip`) is restored to the line inside `sched()` right after the *previous* `swtch` call.
4. `sched()` finishes and returns to `yield()`.
5. `yield()` releases the lock (`release(&ptable.lock)`).
6. The process continues running its user code exactly where it left off.

**now to answer the question 11 we can safely say that the yield does change the process state from running to runnable which in the diagram 1 is noted as interrupt.**

---

**12)**

The time slice is **30 milliseconds**.

We can see that the timer interrupt happens here:

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


struct proc *p = myproc();
int id = cpuid();

if (id % 2 == 0)
{
  // E-core: 30 ms quantum (3 ticks)
  p->qticks++;

  if (SCHED_DEBUG)
    cprintf("TIMER: ticks %d cpu %d pid %d qticks %d\n",
            ticks, id, p->pid, p->qticks);

  if (p->qticks >= QUANTUM_TICKS)
  {
    // don't strictly need to zero here (scheduler does it),
    p->qticks = 0;
    yield();
  }
}
```

First it checks to see if the process is running(sanity check) and also has to see if the trap is the timer interrupt then it looks if the cpu core type is E then increments the qtick which is set there just to count the ticks a process is being run for each E core cpu .
Then when the q tick reaches the Quantum Ticks it calls the yield thus starting the context switch process.

In `trap.c`, the constant `QUANTUM_TICKS` is defined as **3**.

The timer interrupt frequency in xv6 is configured such that one interrupt (tick) occurs every **10ms**.

Therefore, 3 ticks×10 ms = 30ms

This is confirmed by the comment in `trap.c` (Line 128) which states: `// E-core: 30 ms quantum (3 ticks)`.

---

# Wait 3-6 :

# Undrestanding `wait()` System Call before answering the questions:

**1. The Call: A Parent Decides to Wait**

The journey begins in user space. A parent process, after forking a child, reaches a point where it needs to wait for that child to terminate. It invokes the `wait()` system call.

```
// User program (e.g., in sh.c)
pid = fork();
if (pid == 0) {
  // Child process executes a command...
  exit();
} else {
  // Parent process waits for the child to finish.
  wait(); // <-- Our journey starts here.
}
```

This call triggers a trap into the kernel, and the system redirects control to the `sys_wait` function, which in turn calls the `wait()` implementation in `proc.c`.

### 2. Entering the `wait()` Chamber: Preparation and Locking

Once in the kernel, the `wait()` function begins. Its first job is to prepare for a safe search of the process table.

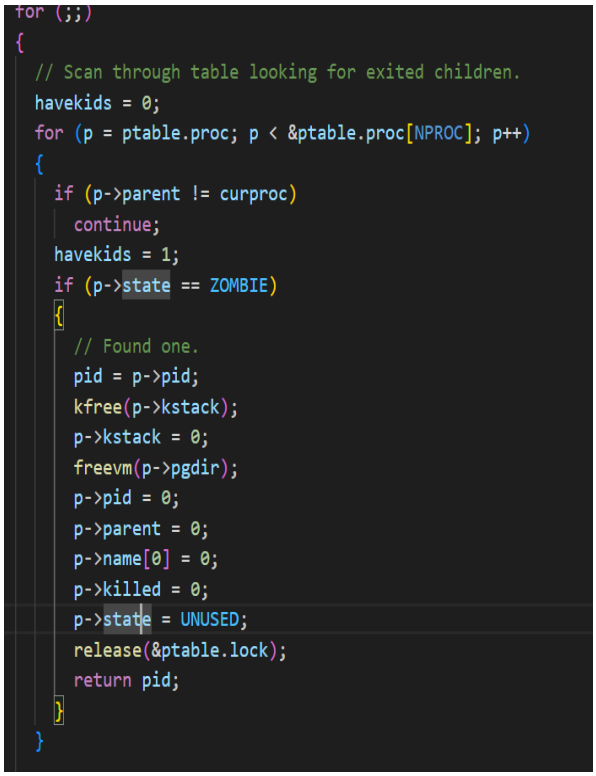
1. **Identify Self:** It gets a pointer to its own process structure via `myproc()`.
2. **Acquire Lock:** It acquires the global `ptable.lock`. This is absolutely critical. It freezes the state of the entire process table, ensuring that no other CPU can modify process states (e.g., another process exiting) while the parent is in the middle of its search.
3. **Enter the Loop:** It enters an infinite `for(;;)` loop. This loop represents the core logic: "Keep checking for exited children, and if none are found, go to sleep and try again later."

```
int wait(void)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc(); // 1. Identify self

  acquire(&ptable.lock); // 2. Lock the process table
  for (;;) // 3. Start the "search-or-sleep" loop
  {
```

### 3. The Search: Looking for a Zombie Child

Inside the infinite loop, the parent now scans the entire process table, looking for two things:

1. Does it have any children at all? (`havekids`)
2. If so, has one of them terminated? (`p->state == ZOMBIE`)



```
for (;;)
{
  // Scan through table looking for exited children.
  havekids = 0;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->parent != curproc)
      continue;
    havekids = 1;
    if (p->state == ZOMBIE)
    {
      // Found one.
      pid = p->pid;
      kfree(p->kstack);
      p->kstack = 0;
      freevm(p->pgdir);
      p->pid = 0;
      p->parent = 0;
      p->name[0] = 0;
      p->killed = 0;
      p->state = UNUSED;
      release(&ptable.lock);
      return pid;
    }
  }
```

If a `ZOMBIE` child is found, the parent "reaps" it by cleaning up all its remaining kernel resources (`kstack`, `pgdir`) and setting its state to `UNUSED`. The `wait()` call is successful, and it returns the child's PID.

**4. The Wait: Going to Sleep**

What if the search loop completes, but no `ZOMBIE` child was found so The code now checks two conditions.

**Case A: No Children to Wait For**

If the loop finishes and `havekids` is still `0`, it means the parent has no children. Waiting is pointless. It releases the lock and returns `-1` to signal an error.

```
if (!havekids || curproc->killed)
{
  release(&ptable.lock);
  return -1;
}
```

**Case B: Children Exist, but Are Still Running**

This is the most interesting case. `havekids` is `1`, but no `ZOMBIE` was found. The parent must now pause its own execution and wait. It does this by calling `sleep()`.

he `sleep(chan, lk)` function is the heart of the waiting mechanism:

1. It sets the parent's state to `SLEEPING`.
2. It specifies a `chan` (channel) to sleep on—in this case, its own process structure pointer (`curproc`).
3. It **atomically** releases the `ptable.lock` and calls the scheduler (`sched()`) to give up the CPU.

**13)**

As mentioned in the descriptions above the `wait` function ultimately uses the **`sleep()`** function to pause its execution and wait for a child to change state.

---

**14)**

`sleep()` is used in `wait()`, but its role is much broader. It is the cornerstone of concurrency management in xv6, used whenever a process must wait for a condition that is fulfilled by an external event, such as:

- **A child process exiting** (in `wait`)
- **Data becoming available in a pipe** (in `piperead`)
- **Space becoming available in a pipe** (in `pipewrite`)
- **A disk I/O operation completing** (in `iderw`)
- **A user typing a key on the keyboard** (in `consoleread`)
- **A certain amount of system time passing** (in `sys_sleep`)

---

**15)**

**Part 1: A Process Begins its Wait**

The journey begins when a process needs to wait for an event that is out of its control, typically an I/O operation. Let's use the example of reading from the disk.

1. **Initiating the Wait:** The process executes a system call like `read()`. The kernel determines that the required data is on the disk. It issues a command to the disk controller hardware via a driver function (e.g., `iderw`).

2. **Calling `sleep()`:** Since the disk is thousands of times slower than the CPU, the process must yield the CPU. It does so by calling the `sleep()` function. The call looks like this:

sleep(b, &idelock);

- The first argument, `b` (a pointer to a buffer structure), is the **wait channel**. This is a unique memory address that identifies the specific event the process is waiting for (in this case, for the disk I/O on buffer `b` to complete).
- Inside `sleep()`, the process's state is set to `SLEEPING`, and the wait channel `b` is stored in its process structure (`p->chan`). The process is now dormant and will not be considered by the scheduler.

```
void sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if (p == 0)
    panic("sleep");

  if (lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if (lk != &ptable.lock)
  {                        // DOC: sleeplock0
    acquire(&ptable.lock); // DOC: sleeplock1
    release(lk);
  }
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;
```

As we can see in the sleep function it gets the channel and the lock and it sets the p->chan and change the state to sleeping.

**This part has explained how a process gets put to sleep when its in need of a i/o event to continue.**

**Part 2: The Signal – An Interrupt from the Hardware**

Sometime later, the disk controller finishes reading the data and copying it to the buffer `b` in memory.

1. **Hardware Interrupt:** The disk controller sends an electrical signal to the CPU on a designated Interrupt Request (IRQ) line, in this case, `T_IRQ0 + IRQ_IDE`.

2. **CPU Handoff:** The CPU immediately stops its current task, saves its context in a `trapframe`, and uses the Interrupt Descriptor Table (IDT) to find the address of the kernel's code for handling this specific interrupt.

3. **The `trap()` Dispatcher:** All hardware interrupts are first handled by the generic `trap()` function in `trap.c`. It acts as a grand central station, inspecting the trap number (`tf->trapno`) to determine the cause of the interrupt. For our disk event, it finds a match:

```
case T_IRQ0 + IRQ_IDE:
  ideintr(); // Dispatch to the specific disk interrupt handler
  lapiceoi();
  break;
```

**Part 3: The Awakening – Answering Questions 15(a) and 15(b)**

Control has now been passed to the specific interrupt handler, `ideintr()`. This is where the sleeping process is finally made aware of its event.

## Answering Question 15(a): The Responsible Function

The `ideintr()` function's job is to communicate with the disk controller, confirm the operation is complete, and identify the buffer `b` that is now ready. With the channel identified, it can now wake the sleeping process.

**The function directly responsible for making a sleeping process aware that its event has occurred is `wakeup()`.**

The interrupt handler invokes it with the very same channel the process went to sleep on:

```
// After finding the completed buffer 'b'
wakeup(b);
```

After calling the wake up with its channel in the disk handler we move to wake up function:

```
wakeup1(void *chan)
{
  struct proc *p;

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state == SLEEPING && p->chan == chan)
    {
      p->state = RUNNABLE;
      // enqueue on its home core's ready queue
      rq_push(&cpus[p->home_core], p);
    }
  }
}

// Wake up all processes sleeping on chan.
void wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```

It simply first acquire the  p table lock and then searches for the procces that has the right wait channel and then it puts the state to RUNNABLE and pushes it back into the ready queue.

**The Transition:** When it finds the matching process, the state change occurs. This is the precise moment the process transitions from being inert to being eligible to run. The code executes:

```
p->state = RUNNABLE;
```

This single line of code is the answer to Part B. It moves the process from the **`SLEEPING`** state to the **`RUNNABLE`** state. The process is then placed back onto a CPU's ready queue (`rq_push`), making it available for the scheduler to pick on a future scheduling cycle.

The journey is complete. The process, now `RUNNABLE`, will eventually be scheduled to run again, and its execution will resume right after the `sleep()` call where it left off, now with its I/O data ready to be used.

---

## 15 – C :

The other function that also makes the same transition is the kill()

### The `SLEEPING` to `RUNNABLE` Transition in `kill()`

While `wakeup()` is the standard mechanism for waking processes when an event occurs, there is a second, critical function in the xv6 kernel that triggers the transition from **`SLEEPING`** to **`RUNNABLE`**: the **`kill()`** function.

**1. The Mechanism**

When `kill(pid)` is called, it does not immediately destroy the target process. Instead, it performs two actions atomically (protected by `ptable.lock`):

1. **Sets the Flag:** It sets `p->killed = 1`. This acts as a request or a "signal" to the process that it should terminate.
2. **Forces the Transition:** If the target process is currently in the `SLEEPING` state, `kill()` explicitly changes its state to `RUNNABLE` and adds it to the scheduler's queue.

```
if (p->state == SLEEPING)
{
  p->state = RUNNABLE;  // <--- The Transition
  rq_push(&cpus[p->home_core], p);
}
```

**2. Why force `RUNNABLE` instead of `ZOMBIE`?**

It might seem efficient to simply set the process state directly to `ZOMBIE` inside `kill()`, effectively deleting it immediately. However, this is dangerous and is not done for specific architectural reasons:

- **Holding Locks:** A sleeping process is often deep inside the kernel, in the middle of a complex operation (like file I/O). It may be holding critical **locks** (e.g., an inode lock or a file system log lock). If the process were forced into `ZOMBIE` mode instantly, it would cease execution without ever releasing these locks. This would permanently block other processes and potentially cause the entire operating system to **deadlock** or freeze.
- **Resource Cleanup:** The process needs to "unwind" its stack safely. It needs to close open file descriptors, release memory pages, and ensure data consistency. This cleanup logic is handled by the `exit()` function, not `kill()`.

**3. The Sequence of Termination**

By setting the state to **`RUNNABLE`**, `kill()` ensures the process is given CPU time one last time to clean up after itself. The sequence is as follows:

1. **Wake Up:** The scheduler picks the now `RUNNABLE` process.

2. **Check Flag:** The process resumes execution inside the kernel (usually returning from the `sleep` function). Most kernel loops check the `p->killed` flag immediately after waking up.
3. **Abort:** Upon seeing `p->killed == 1`, the process aborts its system call and attempts to return to user space.
4. **Trap Handling:** Before returning to user mode, the `trap()` function checks the killed flag again.
5. **Self-Destruction:** The `trap()` function calls **`exit()`**. It is strictly the `exit()` function that finally changes the state to **`ZOMBIE`**, releases resources, and wakes up the parent.

**Conclusion:**

The transition from **`SLEEPING`** to **`RUNNABLE`** in `kill()` is a safety mechanism. It ensures that a process cannot "die in its sleep" while holding critical system resources, forcing it to wake up and terminate voluntarily via `exit()`.

---

# 3-7 exit :

In xv6, the termination of a process is not an instantaneous event where a process and its resources are immediately destroyed. Instead, it is a carefully orchestrated two-phase protocol involving both the dying process (the "child") and its parent. This design ensures that no resources are leaked and that the parent process has an opportunity to get information about its child's termination. The central mechanism connecting these two phases is the `ZOMBIE` state.

1. **Phase 1 (Child):** The child process initiates its own termination by calling `exit()`. It performs internal cleanup, adopts out its own children, wakes its parent, and transitions itself into the `ZOMBIE` state.
2. **Phase 2 (Parent):** The parent process, in a call to `wait()`, finds the `ZOMBIE` child, performs the final cleanup of the child's core kernel resources, and reclaims its process table entry.

---

## Phase 1: The Child's Self-Termination via `exit()`

When a process decides to terminate (either by finishing its main function, calling the `exit` system call directly, or being killed), the kernel executes the `exit()` function on its behalf. The process is still `RUNNING` at this point and can execute kernel code.

Here is a step-by-step breakdown of the `exit()` function's logic:

```
void exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  if (curproc == initproc)
    panic("init exiting");

  // Close all open files.
  for (fd = 0; fd < NOFILE; fd++)
  {
    if (curproc->ofile[fd])
    {
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);

  // Pass abandoned children to init.
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->parent == curproc)
    {
      p->parent = initproc;
      if (p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

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

1. **Sanity Check:** The code first checks if the current process is `initproc`. `initproc` is the ancestor of all processes and serves as the designated foster parent for orphans. If it were to exit, the system would become unstable. This check prevents that, causing a kernel panic if it ever happens.

2. **Internal Resource Cleanup:** The process, while still running, is responsible for cleaning up its own user-level resources. This includes:

- **Closing Files:** It iterates through its open file table (`ofile`) and calls `fileclose()` on each one. This decrements the reference count on the underlying file structures, allowing them to be freed if no other process is using them.
- **Releasing Current Directory:** It releases its reference to the inode of its current working directory (`cwd`) using `iput()`.
3. **Waking the Parent:** The single most important step for coordinating with the parent. The parent process is likely sleeping inside a `wait()` call, waiting for a child to change state. `wakeup1(curproc->parent)` changes the parent's state from `SLEEPING` to `RUNNABLE`, ensuring the scheduler will give it CPU time soon to handle the child's termination.

4. **Orphan Adoption (Re-parenting):** A process may have children of its own. If it exits, these children become **orphans**. To prevent them from becoming permanent zombies when they eventually exit (since their original parent is gone and cannot call `wait()`), the exiting process re-parents all of its own children to `initproc`. The `initproc` perpetually calls `wait()` in its main loop, guaranteeing it will clean up any adopted children.

5. **The Point of No Return:**

- `curproc->state = ZOMBIE;`: The process officially changes its state to `ZOMBIE`. It is now considered "dead" but not yet "gone." It holds minimal information (like its PID) for the parent to collect but cannot execute any more code.
- `sched();`: The process voluntarily gives up the CPU for the last time by calling the scheduler. This call saves its context and switches to the scheduler's context. Since the process state is `ZOMBIE`, the scheduler will never choose to run it again.
- `panic("zombie exit");`: This line should never be reached. If `sched()` were to ever return to a zombie process, it would indicate a critical kernel bug, and the system panics.

At the end of Phase 1, the child process exists only as a `ZOMBIE` entry in the process table. Its user memory and kernel stack still exist but are inaccessible.

Then when the parent wakes up and wants to follow-up with the wait there is a if statement for the zombie processes:

```
if (p->state == ZOMBIE)
{
  // Found one. Perform the final cleanup.
  pid = p->pid;
  kfree(p->kstack);          // Free the child's kernel stack
  p->kstack = 0;
  freevm(p->pgdir);          // Free the child's page directo
  p->pid = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->killed = 0;
  p->state = UNUSED;         // Mark the process table slot a
  release(&ptable.lock);
  return pid;                // Return the dead child's PID.
```

This is the part where it cleans up everything and sets the state to to UNUSED to officially terminated the process.

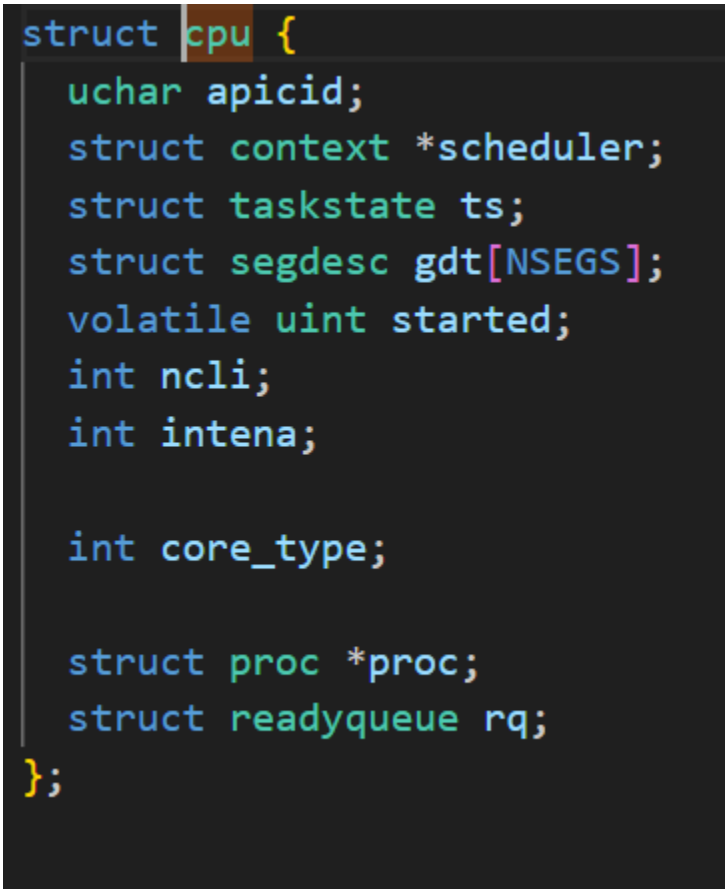
**16)**

So the answer of this question would be in the reparenting section of the explanations above in which after waking up the parent it changes the zombie childs parent to init proc:

```
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
{
  if (p->parent == curproc)
  {
    p->parent = initproc;
    if (p->state == ZOMBIE)
      wakeup1(initproc);
  }
}
```

**17)**

With all of these thorough examination of the xv6 kernel source code now its easy to answer this part . by reaching to the cpu struct we can see:



```
struct cpu {
  uchar apicid;
  struct context *scheduler;
  struct taskstate ts;
  struct segdesc gdt[NSEGS];
  volatile uint started;
  int ncli;
  int intena;

  int core_type;

  struct proc *proc;
  struct readyqueue rq;
};
```

## uchar apicid;

This field holds the unique hardware ID for the core's Local Advanced Programmable Interrupt Controller (LAPIC). The operating system relies on this ID to identify the specific physical core it is currently executing on. This identification is fundamental for routing interrupts correctly to the target core and for performing per-CPU initialization tasks during the boot process.

## struct taskstate ts;

This is a Task State Segment (TSS), a hardware structure defined by the x86 architecture. In the xv6 operating system, its most critical function is to provide the CPU with the location of the kernel stack. When a trap, such as a system call or an interrupt, occurs while the CPU is in user mode, the hardware automatically references the TSS to find the kernel stack pointer (`ts.esp0`). It then switches to this secure kernel stack before handling the trap, ensuring the kernel's integrity.

## `struct segdesc gdt[NSEGS];`

This represents the Global Descriptor Table (GDT), another essential x86 hardware structure. The GDT's purpose is to define memory "segments" that the CPU can access, specifying attributes like the base address, size limit, and, most importantly, the privilege level (kernel or user). In xv6, each CPU has its own GDT primarily to accommodate a unique descriptor for its own Task State Segment (`ts`), which is a necessary part of managing per-CPU state.

## `volatile uint started;`

This is a synchronization flag used exclusively during the multi-processor boot sequence. The first CPU to boot, the Bootstrap Processor, is responsible for awakening the other CPUs, known as Application Processors. The Bootstrap Processor monitors this flag for each Application Processor. Once an Application Processor finishes its initial setup, it sets its `started` flag to 1. The `volatile` keyword is critical, as it prevents compiler optimizations and forces a fresh read from memory in the waiting loop, ensuring the main CPU sees the change immediately.

## `struct context *scheduler;`

This field is a pointer to the saved context (CPU registers) of the CPU's own dedicated scheduler loop. When a process yields control back to the kernel, the `swtch()` function saves the process's current kernel context and restores the scheduler's context from this pointer. This action effectively makes the CPU jump directly back into its `scheduler()` function, ready to select the next process to run. Each CPU core has its own independent scheduler loop and thus requires its own scheduler context.

## `struct proc *proc;`

This pointer serves as the definitive link between a physical CPU core and the software process it is currently executing. It points to the `struct proc` of the process that is in the `RUNNING`

state on that core. If no process is running—for instance, if the CPU is idle or executing its scheduler loop—this pointer will be `NULL`. Helper functions like `myproc()` use this pointer to easily identify the currently running process on the local CPU.

## `struct readyqueue rq;`

This is a per-CPU queue designed to hold all processes that are in the `RUNNABLE` state and are available to be scheduled on this specific core. By giving each CPU its own local queue of ready processes, this design avoids the high lock contention that would occur if all CPUs had to compete for a single global queue. This significantly improves performance and scalability on multi-core systems. A function like `load_balance_on_timer` is used to migrate processes between these queues to prevent starvation and balance the workload.

## `int core_type;`

This flag is used to differentiate between different types of cores in a heterogeneous architecture, such as Performance-cores (`CORE_P`) and Efficiency-cores (`CORE_E`). This identification enables the operating system's scheduler to implement distinct policies tailored to the capabilities of each core type. For example, the scheduler might use a standard time quantum for preemption on E-cores, while P-cores might use a more aggressive First-Come, First-Served (FCFS) strategy based on process creation time.

## `int ncli;`

This field is a depth counter for nested calls to `pushcli()`, a function used to disable interrupts and protect critical sections of kernel code. It functions as a lock counter: the first call to `pushcli()` (when `ncli` is 0) disables interrupts and increments `ncli` to 1. Subsequent nested calls simply increment the counter without touching the interrupt flag. This ensures that interrupts, disabled by an outer function, are not prematurely re-enabled by an inner function's corresponding `popcli()` call.

## `int intena;`

This flag preserves the state of the interrupt-enable flag before interrupts were first disabled by a `pushcli()` call. When `popcli()` is called and decrements `ncli` back to 0, it consults the `intena` flag. If interrupts were already disabled *before* the initial `pushcli()` was invoked, this

flag will be 0, and `popcli()` will not re-enable them. This ensures that the interrupt state is correctly restored to what it was before the critical section was entered.

# 4.1 E-cores with RR (30ms time quantum)

## 1:

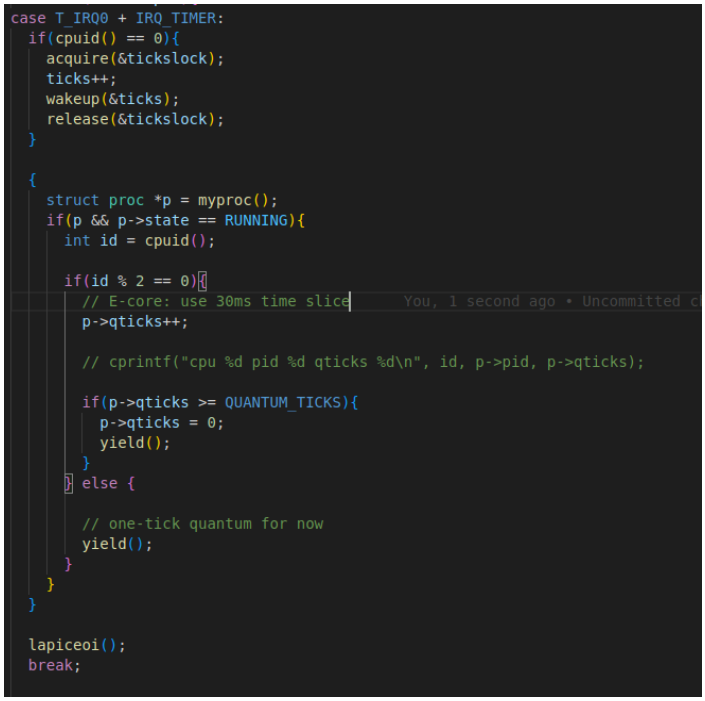
First we add a flag for the cpu type in cpu struct then we add

```
  // Determine core type for this CPU based on its index
int id = c - cpus;
c->core_type = (id % 2 == 0) ? CORE_E : CORE_P;
```

At start of scheduler function to know the cpu type.

## 2:

Now we add a qtick to count the number of time quantum a process can use, we have to be careful and reset the qtick everyi=time the process gets some time to execute again, and finally we add the section that checks for e cores and checks for how many ticks it has used in trap.c:



```
case T_IRQ0 + IRQ_TIMER:
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }

  {
    struct proc *p = myproc();
    if(p && p->state == RUNNING){
      int id = cpuid();

      if(id % 2 == 0){
        // E-core: use 30ms time slice      You, 1 second ago • Uncommitted c
        p->qticks++;

        // cprintf("cpu %d pid %d qticks %d\n", id, p->pid, p->qticks);

        if(p->qticks >= QUANTUM_TICKS){
          p->qticks = 0;
          yield();
        }
      } else {

        // one-tick quantum for now
        yield();
      }
    }
  }

  lapiceoi();
  break;
```

Edit: this step was done wrong at first , the picture above is wrong because we are calling yield before lapiceoi sometimes causing issues (lapiceoi's job is to like clear interrupts and let more interrupts come.)

## 3:

We reset the qticks every time the process gets chosen to run

```
    done1 = 1;
    c->proc = p;
    switchuvm(p);
    p->qticks = 0; // reset ticks used in timeslice
    p->state = RUNNING;
```

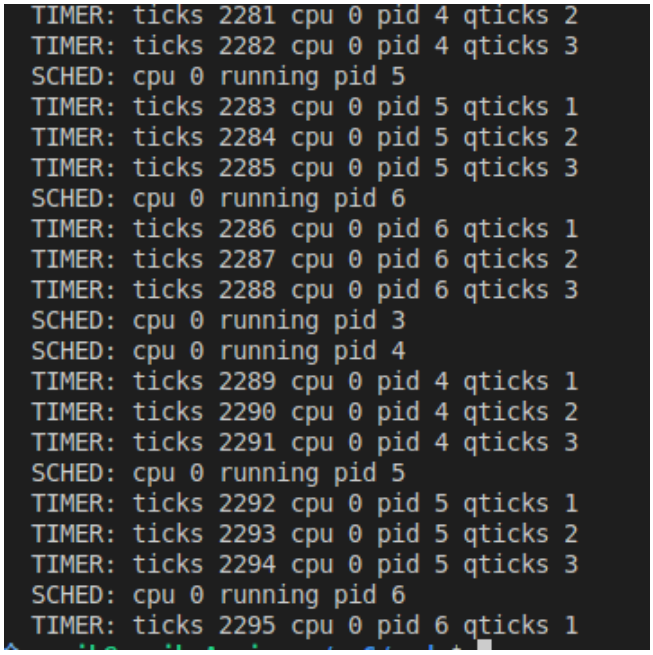
## 4:

And lastly we add a last index to know from which process we need to start looking for new runnables to run. We also add a p flag and we run only when p is not null or zero to avoid running dead processes.

```
scheduler(void){
  static int last = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);

    p = 0;

    int i;
    int idx;
    for(i = 0; i < NPROC; i++){
      idx = (last + i) % NPROC;
      if(ptable.proc[idx].state == RUNNABLE){
        p = &ptable.proc[idx];
        // next time, start searching from after this one
        last = (idx + 1) % NPROC;
        break;
      }
    }

    if(p){
      c->proc = p;
      switchuvm(p);

      // Start a fresh time slice for this process
      p->qticks = 0;
      p->state = RUNNING;

      // Switch to chosen process. It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      swtch(&(c->scheduler), p->context);
```

### 5:

And finally we write a program that makes a few children and runs them forever so we can see that our round robin is working




## 4.2: Adding per core Queue

### 1:

First we make a ready queue struct that holds the runnable processes , and we add it to cpu struct to show each cpu has a ready queue.

### 2:

So here we assume a home core for each process and assign a core for each process in a roud robin like fashion

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->priority = PRI_NORMAL;

  p->creation_time = ticks;

  p->qticks = 0; // no ticks used yet in its timeslice

  // Assign this process a "home" core similiar to round-robin.
  p->home_core = next_core;
  next_core = (next_core + 1) % ncpu;


      You, 1 hour ago • Uncommitted changes
```

```
You, 1 minute ago | 1 author (You)
struct readyqueue {
  struct proc *procs[NPROC];  // ready/runnable processes
  int count;                  // number of processes in the queue
};


// Per-CPU state
You, 1 minute ago | 2 authors (hmdmhmd2004 and one other)
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?

  int core_type;               // CORE_E or CORE_P (even/odd cpuid)

  struct proc *proc;           // The process running on this cpu or null
  struct readyqueue rq;          // per-CPU ready queue
};
```

## 3:

In fork and userinit we add the process to the desired cpu. And in scheduler we grab the process from its cpu’s ready queue.

```
scheduler(void)
  for (;;)


    if (rq->count > 0)
    {
      if (c->core_type == CORE_E)
      {
        p = rq->procs[0];
        rq_remove(c, p);
      }
      else
      {

        struct proc *best = 0;

        for (int i = 0; i < rq->count; i++)
        {
          struct proc *cand = rq->procs[i];
          if (cand->state != RUNNABLE)
            continue;

          if (best == 0 || cand->creation_time < best->creation_time)
            best = cand;
        }

        if (best)
        {
          p = best;
          rq_remove(c, p);
        }
      }
```

## 4:

Here we change yield,kill and wake up accordingly to add or remove the process from the desired ready queue

```
int
kill(int pid)
{
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING){
        p->state = RUNNABLE;
        rq_push(&cpus[p->home_core], p);
      }
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

```
static void
wakeup1(void *chan)
{
  struct proc *p;

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state == SLEEPING && p->chan == chan)
    {
      p->state = RUNNABLE;
      // enqueue on its home core's ready queue
      rq_push(&cpus[p->home_core], p);
    }
  }
}
```

```
void yield(void)
{
  struct proc *p = myproc();

  acquire(&ptable.lock); // DOC: yieldlock

  p->state = RUNNABLE;
  // put it back into its home core's ready queue
  rq_push(&cpus[p->home_core], p);      You, 3 min

  sched();
  release(&ptable.lock);
}
```

## 5:

Finally we write some debugging lines to confirm every core is running the process it is supposed to.

```
SCHED: cpu 0 core_type E running pid 5 (ct=1026, home=0)
SCHED: cpu 2 core_type E running pid 3 (ct=998, home=2)
SCHED: cpu 1 core_type P running pid 6 (ct=1030, home=1)
SCHED: cpu 3 core_type P running pid 4 (ct=1025, home=3)
SCHED: cpu 1 core_type P running pid 6 (ct=1030, home=1)
SCHED: cpu 2 core_type E running pid 3 (ct=998, home=2)
SCHED: cpu 1 core_type P running pid 6 (ct=1030, home=1)
SCHED: cpu 3 core_type P running pid 4 (ct=1025, home=3)
SCHED: cpu 1 core_type P running pid 6 (ct=1030, home=1)
SCHED: cpu 2 core_type E running pid 3 (ct=998, home=2)
```

# 4.3: P-cores with FCFS scheduling

## 1:

First we try to save the creation time of every process , for doing so we first add a variable to our proc struct to hold the creation time of the process , then in all proc where each new process is created we set the creation time of the process

```
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++;
  p->priority = PRI_NORMAL;

      You, 32 seconds ago • Uncommitted changes
  p->creation_time = ticks;



  p->qticks = 0;            //no ticks used yet in its timeslice
  release(&ptable.lock);
```

## 2:

We find the lowest creation times among all runnable processes in scheduler function

```
}
else
// P cores use FCFS
{
  for (int i = 0; i < NPROC; i++)
  {
    // find the first RUNNABLE process
    if (ptable.proc[i].state == RUNNABLE)
    {
      p = &ptable.proc[i];
      break;
    }

  }
  for (int i = 0; i < NPROC; i++)
  {
    // compare creation_time to implement FCFS among RUNNABLE processes
    if (ptable.proc[i].state == RUNNABLE)
    {
      if (p->creation_time>ptable.proc[i].creation_time)
      {
        p = &ptable.proc[i];
      }

    }

  }
```

## 3:

Here we add the preemption logic to trap function to yield the cpu every time a process with a lower creation time comes. We also added a lock when we are reading through ptable so some other process doesnt change anything in it, and added a should yield flag to avoid messing wth locks and potentially causing a deadlock when we call yield.

```
} else {
  // P-core: check for preemption every tick      You, 20 seconds ago • Uncommitted change
  int should_yield = 0;

  acquire(&ptable.lock);
  for (int i = 0; i < NPROC; i++) {
    if (ptable.proc[i].state == RUNNABLE) {
      if (p->creation_time > ptable.proc[i].creation_time) {
        should_yield = 1;
        if (SCHED_DEBUG) {
          cprintf("P-CORE PREEMPT: cpu %d curpid %d(ct=%d), better pid %d(ct=%d)\n",
                  id, p->pid, p->creation_time,
                  ptable.proc[i].pid, ptable.proc[i].creation_time);
        }
        break;
      }
    }
  }
  release(&ptable.lock);

  if (should_yield) {
    yield();
  }
}
```

## 4:

Finally we made a program to test our FCFS, we can clearly see it is working(note that pid 3 is shell so it runs when we type something on the screen)

```
SCHED: cpu 0 core_type P running pid 3 (ct=823)
SCHED: cpu 0 core_type P running pid 3 (ct=823)
SCHED: cpu 0 core_type P running pid 3 (ct=823)
SCHED: cpu 0 core_type P running pid 3 (ct=823)
SCHED: cpu 0 core_type P running pid 3 (ct=823)
SCHED: cpu 0 core_type P running pid 4 (ct=829)
child1 (pid 4) starting long work
SCHED: cpu 0 core_type P running pid 4 (ct=829)
SCHED: cpu 0 core_type P running pid 4 (ct=829)
SCHED: cpu 0 core_type P running pid 4 (ct=829)
SCHED: cpu 0 core_type P running pid 4 (ct=829)
child1 (pid 4) finished
SCHED: cpu 0 core_type P running pid 3 (ct=823)
SCHED: cpu 0 core_type P running pid 5 (ct=829)
child2 (pid 5) starting long work
SCHED: cpu 0 core_type P running pid 5 (ct=829)
SCHED: cpu 0 core_type P running pid 5 (ct=829)
SCHED: cpu 0 core_type P running pid 5 (ct=829)
SCHED: cpu 0 core_type P running pid 5 (ct=829)
SCHED: cpu 0 core_type P running pid 5 (ct=829)
SCHED: cpu 0 core_type P running pid 5 (ct=829)
SCHED: cpu 0 core_type P running pid 5 (ct=829)
SCHED: cpu 0 core_type P running pid 5 (ct=829)
SCHED: cpu 0 core_type P running pid 5 (ct=829)
SCHED: cpu 0 core_type P running pid 5 (ct=829)
SCHED: cpu 0 core_type P running pid 5 (ct=829)
SCHED: cpu 0 core_type P running pid 5 (ct=829)
SCHED: cpu 0 core_type P running pid 5 (ct=829)
SCHED: cpu 0 core_type P running pid 5 (ct=829)
SCHED: cpu 0 core_type P running pid 5 (ct=829)
```

# 4.4: Load Balancing

## 1:

First we change all proc so everytime a new process is created we find the e core with lowest load and set the home core to that

```c
static int
find_least_loaded_ecore(void)
{
  int best_core = 0;
  int best_load = cpus[0].rq.count;

  for (int i = 1; i < ncpu; i++) {        You, 2 m
    if (i % 2 != 0)
      continue;

    if (cpus[i].rq.count< best_load) {
      best_core = i;
      best_load = cpus[i].rq.count;
    }
  }


  return best_core;
}
```

## 2:

We add a function that check if a given e core is overloaded compared to a given p core we also make a function that checks an e core's ready queue for a process to give to a p core.

```
static int
ecore_is_overloaded(int e_core, int p_core)
{
  if (p_core < 0) return 0;

  int E_load = cpus[e_core].rq.count;
  int P_load = cpus[p_core].rq.count;

  return (E_load >= P_load + 3);
}

static struct proc*
select_ecore_proc(int e_core)
{
  struct readyqueue *rq = &cpus[e_core].rq;

  for (int i = 0; i < rq->count; i++) {      You, 54 seco
    struct proc *p = rq->procs[i];

    if (p->pid == 1)
      continue;
    if (strcmp(p->name, "sh") == 0)
      continue;

    return p;
  }

  return 0;
}
```

## 3:

And lastly we make a function that every 5 ticks gets called that is supposed to do the job of balancing the load.

```
void
load_balance_on_timer(int cpu_id)
{
  if (cpu_id % 2 != 0)
    return;

  balance_ticks[cpu_id]++;
  if (balance_ticks[cpu_id] < BALANCE_TICKS)
    return;

  balance_ticks[cpu_id] = 0;

  acquire(&ptable.lock);

  int p_core = find_least_loaded_pcore();
  if (!ecore_is_overloaded(cpu_id, p_core)) {
    release(&ptable.lock);
    return;
  }

  struct proc *proc_to_move = select_ecore_proc(cpu_id);
  if (proc_to_move == 0) {
    // nothing we can push
    release(&ptable.lock);
    return;
  }

  rq_remove(&cpus[cpu_id], proc_to_move);

  proc_to_move->home_core = p_core;

  rq_push(&cpus[p_core], proc_to_move);

#ifdef BALANCE_DEBUG
  cprintf("BALANCE: moved pid %d from E%d -to P%d (new Eload=%d, new Pload=%d)\n",
          proc_to_move->pid,
          cpu_id, p_core,
          cpus[cpu_id].rq.count,
          cpus[p_core].rq.count);
#endif

  release(&ptable.lock);
}
```

## 4:

Results show that our balancing is working.

```
SCHED: cpu 0 core_type E running pid 4 (ct=808, home=0)
SCHED: cpu 0 core_type E running pid 3 (ct=784, home=0)
SCHED: cpu 0 core_type E running pid 4 (ct=808, home=0)
BALANCE: moved pid 6 from E0 -to P1 (new Eload=2, new Pload=1)
SCHED: cpu 1 core_type P running pid 6 (ct=813, home=1)
SCHED: cpu 0 core_type E running pid 3 (ct=784, home=0)
SCHED: cpu 1 core_type P running pid 6 (ct=813, home=1)
SCHED: cpu 0 core_type E running pid 4 (ct=808, home=0)
SCHED: cpu 1 core_type P running pid 6 (ct=813, home=1)
SCHED: cpu 1 core_type P running pid 6 (ct=813, home=1)
SCHED: cpu 0 core_type E running pid 5 (ct=809, home=0)
SCHED: cpu 1 core_type P running pid 6 (ct=813, home=1)
SCHED: cpu 1 core_type P running pid 6 (ct=813, home=1)
SCHED: cpu 1 core_type P running pid 6 (ct=813, home=1)
SCHED: cpu 1 core_type P running pid 6 (ct=813, home=1)
BALANCE: moved pid 4 from E0 -to P1 (new Eload=2, new Pload=1)
SCHED: cpu 1 core_type P running pid 4 (ct=808, home=1)
SCHED: cpu 0 core_type E running pid 5 (ct=809, home=0)
SCHED: cpu 1 core_type P running pid 4 (ct=808, home=1)
SCHED: cpu 1 core_type P running pid 4 (ct=808, home=1)
SCHED: cpu 0 core_type E running pid 3 (ct=784, home=0)
SCHED: cpu 1 core_type P running pid 4 (ct=808, home=1)
```

# 5.4: Time scheduling  algorithms test

So we first need to  add the variables needed:

```
struct spinlock eval_lock;
int measurement_active = 0;
uint start_tick = 0;
int finished_processes = 0;
```

Then the two functions start and stop need to be implemented;In start we just need to set the variables to 0

```
int
start_measuring_impl(void)
// In this function we just need to set the variables to a start mode so
setting them to
//their start values, finished process should be zero and start tick is
required for measuring the time
// We also set measuring to active to we know we actually have started a
test and can print it
{
 acquire(&ptable.lock);
 measurement_active = 1;
 finished_processes = 0;
 start_tick = ticks;
 release(&ptable.lock);
 return 0;
}
```

And in stop we need to first check if we have already started measuring or not, if so then set the finish time using ticks, and also the count of processes finished. Then we need throughput to be a rate of number of finished processes and duration and an appropriate formula is count*1000/duration which is end_time - start_time. And at last we print them at the end.

```
int
stop_measuring_impl(void)
{
 acquire(&ptable.lock);
 if (!measurement_active) {
     release(&ptable.lock);
     return -1;
 }
  uint end_tick = ticks;
```

```
 int count = finished_processes;
 measurement_active = 0; // Stop by setting this variable to zero meaning
false and inactive
 release(&ptable.lock);

 uint duration = end_tick - start_tick;
 if (duration == 0) duration = 1;

 int throughput = (count * 1000) / duration;

 cprintf("\n[EVALUATION RESULT]\n");
 cprintf("Finished Processes: %d\n", count);
 cprintf("Total Time: %d ticks\n", duration);
 cprintf("Throughput: %d.%d processes/tick\n", throughput / 1000,
throughput % 1000);
 cprintf("---------------------\n");

 return 0;
}
```

# 5.5: Required system calls

In this part we need to add another function for printing info of the processes. Then we need to handle the system calls for all three functions we have written.

```
print_info_impl(void)
{
 struct proc *p;
 int core;
 int home;
 int creation;
 int pid;
 char *name;

 // 1. DISABLE INTERRUPTS to safely read CPU and Process data
 pushcli();
  if(myproc() == 0) {
     // Safety check: If called from scheduler when no process is running
     popcli();
     return -1;
 }

 p = myproc();
 core = cpuid(); // Now safe because interrupts are off
  // Copy data to local variables so we can print safely later
 pid = p->pid;
 name = p->name;
 home = p->home_core;
 creation = p->creation_time;
  popcli(); // 2. RE-ENABLE INTERRUPTS
  // 3. Logic
 char *algo = (core % 2 == 0) ? "Round Robin" : "FCFS (Modified)";

 // 4. Print (It is safer to print with interrupts enabled,
 //    so we do this after popcli)
 cprintf("[INFO] PID: %d | Name: %s | Core: %d (%s) | Home: %d | Created:
%d\n",
         pid, name, core, algo, home, creation);

 return 0;
}
```

In printing info we can see the algorithm used by knowing whether the cpu has been changed or not, also we can get the pid and its name and its origin, but since we change the cpu for a process completely, home and code are gonna be the same and also created is simply the time which in the process was made.
Then just handling the system calls remain and here are the steps for it:
Changes in defs.h:

```
int             start_measuring_impl(void);
int             stop_measuring_impl(void);
int             print_info_impl(void);
```

Change in sysproc.c:

```
int sys_start_measuring(void) {
 return start_measuring_impl();
}

int sys_stop_measuring(void) {
 return stop_measuring_impl();
}

int sys_print_info(void) {
 return print_info_impl();
}
```

Changes in syscall.h

```
#define SYS_start_measuring  27
#define SYS_stop_measuring   28
#define SYS_print_info       29
```

Change in user.h:

```
int start_measuring(void);
int stop_measuring(void);
int print_info(void);
```

Change in usys.S:

```
36  SYSCALL(start_measuring)
37  SYSCALL(stop_measuring)
38  SYSCALL(print_info)
```

Change in syscall.c:

```
extern int sys_start_measuring(void);
extern int sys_stop_measuring(void);
extern int sys_print_info(void);
```

And in static-int *syscalls we add:

```
   [SYS_start_measuring] sys_start_measuring,
   [SYS_stop_measuring]  sys_stop_measuring,
   [SYS_print_info]      sys_print_info,
```

# 6.5: User level program for testing

Then we are asked to run a use level program and use the three system call we have added to test the functionality of our scheduling algorithms:
For this part we make this user level  program named schedtest.c:

```c
#include "types.h"
#include "stat.h"
#include "user.h"

void stress_cpu(long iterations) {
   volatile long x = 0;
   volatile long i;
   for(i = 0; i < iterations; i++) {
       x = x + 1;
       x = x * 2;
       x = x / 2;
   }
}

int main(int argc, char *argv[])
{
   int num_children = 8;
   int i;
   int pid;

   printf(1, "SCHEDTEST: Starting with 3 System Calls...\n");

   start_measuring();

   for(i = 0; i < num_children; i++) {
       pid = fork();

       if(pid == 0){
           // --- CHILD ---
           print_info();
           stress_cpu(50000000);
           print_info();
           exit();
       }
   }
```

```
    for(i = 0; i < num_children; i++) {
        wait();
    }

    stop_measuring();

    exit();
}
```

We then need to make the required modification in Makefile:
We this line to UPROGS:

```
    _schedtest\
```

And add this line to EXTRAS:

```
    schedtest.c\
```

And so our user level program is ready, but we need to run it in two different ways. First we need to keep our algorithms running for changing cpu so we run it and the output looks as followed where multiple tasks can start and be swapped to other cpus when finished.

```
$ schedtest
SCHEDTEST: Starting with 3 System Calls...
SCHEDTEST: start of process
[INFO] PID: 23 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 16304
BALANCE: moved pid 30 from E0 -to P1 (new Eload=6, new Pload=1)
SCHEDTEST: start of process
[INFO] PID: 30 | Name: schedtest | Core: 1 (FCFS (Modified)) | Home: 1 | Created: 16309
SCHEDTEST: start of process
[INFO] PID: 29 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 16308
BALANCE: moved pid 23 from E0 -to P1 (new Eload=5, new Pload=1)
SCHEDTEST: start of process
[INFO] PID: 28 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 16308
BALANCE: moved pid 28 from E0 -to P1 (new Eload=4, new Pload=2)
SCHEDTEST: start of process
[INFO] PID: 27 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 16308
SCHEDTEST: end of process
[INFO] PID: 23 | Name: schedtest | Core: 1 (FCFS (Modified)) | Home: 1 | Created: 16304
BALANCE: moved pid 27 from E0 -to P1 (new Eload=4, new Pload=2)
SCHEDTEST: start of process
[INFO] PID: 26 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 16308
SCHEDTEST: end of process
[INFO] PID: 29 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 16308
SCHEDTEST: start of process
[INFO] PID: 25 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 16308
SCHEDTEST: end of process
[INFO] PID: 27 | Name: schedtest | Core: 1 (FCFS (Modified)) | Home: 1 | Created: 16308
SCHEDTEST: end of process
[INFO] PID: 28 | Name: schedtest | Core: 1 (FCFS (Modified)) | Home: 1 | Created: 16308
BALANCE: moved pid 25 from E0 -to P1 (new Eload=2, new Pload=1)
SCHEDTEST: start of process
[INFO] PID: 24 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 16304
SCHEDTEST: end of process
[INFO] PID: 26 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 16308
SCHEDTEST: end of process
[INFO] PID: 25 | Name: schedtest | Core: 1 (FCFS (Modified)) | Home: 1 | Created: 16308
SCHEDTEST: end of process
[INFO] PID: 24 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 16304
SCHEDTEST: end of process
[INFO] PID: 30 | Name: schedtest | Core: 1 (FCFS (Modified)) | Home: 1 | Created: 16309

[EVALUATION RESULT]
Finished Processes: 8
Total Time: 284 ticks
Throughput: 0.28 processes/tick
--------------------
```

Also we saw both cpu's were used.

Then we comment the line for swapping cpu code in trap.c:

```
116   // periodic load balancing on timer interrupts
117   // if (tf->trapno == T_IRQ0 + IRQ_TIMER)
118   // {
119   //   load_balance_on_timer(cpuid());
120   // }
```

And we run the test again:

```
$ schedtest
SCHEDTEST: Starting with 3 System Calls...
SCHEDTEST: start of process
[INFO] PID: 4 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 186
SCHEDTEST: start of process
[INFO] PID: 11 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 191
SCHEDTEST: end of process
[INFO] PID: 4 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 186
SCHEDTEST: end of process
[INFO] PID: 11 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 191
SCHEDTEST: start of process
[INFO] PID: 10 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 191
SCHEDTEST: start of process
[INFO] PID: 9 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 191
SCHEDTEST: end of process
[INFO] PID: 10 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 191
SCHEDTEST: end of process
[INFO] PID: 9 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 191
SCHEDTEST: start of process
[INFO] PID: 8 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 191
SCHEDTEST: start of process
[INFO] PID: 7 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 191
SCHEDTEST: end of process
[INFO] PID: 8 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 191
SCHEDTEST: end of process
[INFO] PID: 7 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 191
SCHEDTEST: start of process
[INFO] PID: 6 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 187
SCHEDTEST: start of process
[INFO] PID: 5 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 187
SCHEDTEST: end of process
[INFO] PID: 6 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 187
SCHEDTEST: end of process
[INFO] PID: 5 | Name: schedtest | Core: 0 (Round Robin) | Home: 0 | Created: 187

[EVALUATION RESULT]
Finished Processes: 8
Total Time: 566 ticks
Throughput: 0.14 processes/tick
--------------------
```

So as we can see processors start one after another and only run on a single CPU.
And throughput is actually double the speed (0.28 processes/tick compared to 0.14 processes/tick) when using the algorithms.

For these three parts we use gemini with this history: chat-history

Although it seems a part of the chat history which is the main part of it is missing at the beginning and we couldn't figure out why. Maybe because the chat was getting too long. So the chat history is basically useless for the parts it contains.