Q&A

1.

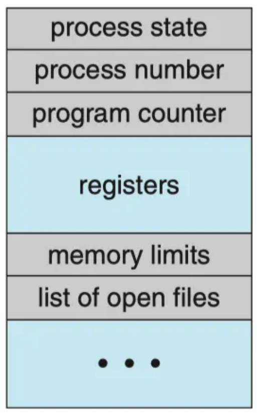


Figure 3.3 Process control block (PCB).

- Process state. The state may be new, ready, running, waiting, halted, and so on.
- Program counter. The counter indicates the address of the next instruction to be executed for this process.
- CPU registers. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.
- CPU-scheduling information. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (Chapter 5 describes process scheduling.)

- Memory-management information. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system (Chapter 9).
- Accounting information. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- I/O status information. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

And the PCB is:

```
// Per-process state
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
  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)
};
```

This structure is the **Process Control Block (PCB)** in xv6. It contains all the necessary information about a process, such as its state, memory layout, open files, and more. Here's a quick summary of its fields and their purposes:

1. **Memory Management**:
    - `uint sz`: Size of the process's memory in bytes.

- `pde_t* pgdir`: Page table pointer, used for virtual memory management.

2. **Kernel Stack**:
   - `char *kstack`: Pointer to the bottom of the kernel stack for the process.
3. **State and Identification**:
   - `enum procstate state`: Current state of the process (`UNUSED`, `RUNNABLE`, etc.).
   - `int pid`: Process ID, uniquely identifies the process.
4. **Parent and Child Process Relationships**:
   - `struct proc *parent`: Pointer to the parent process.
5. **System Call and Context**:
   - `struct trapframe *tf`: Trap frame for the current system call, used to save registers.
   - `struct context *context`: Used by `swtch()` to save and restore process context.
6. **Synchronization and Signals**:
   - `void *chan`: If non-zero, the process is sleeping on this channel.
   - `int killed`: Non-zero indicates the process has been killed.
7. **File and Directory Management**:
   - `struct file *ofile[NOFILE]`: Array of pointers to open files.
   - `struct inode *cwd`: Pointer to the current working directory.
8. **Debugging**:
   - `char name[16]`: Name of the process (used mainly for debugging).

So here's the mapping between the fields in the xv6 `struct proc` (Process Control Block) and the elements described in the source book:

1. **Process State**

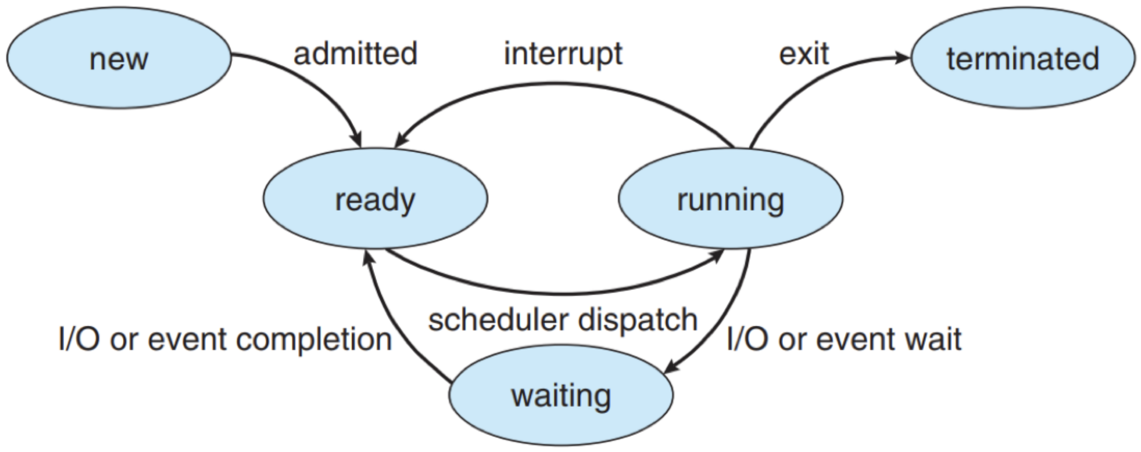
- **Figure Description**: Tracks whether the process is new, ready, running, waiting, terminated, etc.
- **xv6 Field**: `enum procstate state;`

2. **Process Number (PID)**
   - **Figure Description**: A unique identifier for the process.
   - **xv6 Field**: `int pid;`
3. **Program Counter**
   - **Figure Description**: Indicates the address of the next instruction to execute.
   - **xv6 Field**: Stored in the `struct trapframe *tf;`, which includes the program counter (e.g., `eip` on x86).
4. **CPU Registers**
   - **Figure Description**: Includes general-purpose registers, stack pointers, and condition codes.
   - **xv6 Field**: `struct trapframe *tf;` (includes all CPU registers).
5. **Memory-Management Information**
   - **Figure Description**: Tracks base/limit registers, page tables, or segment tables.
   - **xv6 Field**:
     - `pde_t* pgdir;` (page table).
     - `uint sz;` (size of process memory).
6. **CPU-Scheduling Information**
   - **Figure Description**: Includes priority, scheduling queues, etc.
   - **xv6 Field**:
     - `enum procstate state;` indirectly contributes to scheduling.
     - Priority is not explicitly implemented in xv6.
7. **I/O Status Information**

- **Figure Description**: Includes a list of allocated I/O devices and open files.
- **xv6 Field**:
    - `struct file *ofile[NOFILE];` (list of open files).
    - `struct inode *cwd;` (current directory).

8. **Accounting Information**
    - **Figure Description**: Tracks CPU time, process limits, etc.
    - **xv6 Field**: Not explicitly maintained in xv6.

2.



شکل ۱. چرخهٔ وضعیت یک پردازه

In xv6, we have these states:

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

- **UNUSED** → Not directly shown in the diagram. Represents processes that are not in use or allocated (e.g., not yet admitted into the system).
- **EMBRYO** → **New**: Represents a newly created process that is being initialized.
- **SLEEPING** → **Waiting**: Corresponds to processes waiting for an event or I/O to complete.
- **RUNNABLE** → **Ready**: Represents processes ready to run but waiting for CPU scheduling.
- **RUNNING** → **Running**: Represents processes actively running on the CPU.
- **ZOMBIE** → **Terminated**: Represents processes that have finished execution but are waiting to be reaped (cleaned up by the parent process).

3.

As you can see in

Transition for `NEW → READY`

1. A new process is allocated via `allocproc()` and is in the `EMBRYO` state.

And the `allocproc()` implementation is: (not that important)

```
//PAGEBREAK: 32
// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and initialize
// state required to run in the kernel.
// Otherwise return 0.
static struct proc*
allocproc(void)
{
struct proc *p;
char *sp;

acquire(&ptable.lock);
```

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

release(&ptable.lock);
return 0;

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

release(&ptable.lock);

// Allocate kernel stack.
if((p->kstack = kalloc()) == 0){
p->state = UNUSED;
return 0;
}
sp = p->kstack + KSTACKSIZE;

// Leave room for trap frame.
sp -= sizeof p->tf;
p->tf = (struct trapframe)sp;

// Set up new context to start executing at forkret,
// which returns to trapret.
sp -= 4;
(uint)sp = (uint)trapret;

sp -= sizeof p->context;
p->context = (struct context)sp;
memset(p->context, 0, sizeof *p->context);
p->context->eip = (uint)forkret;
```

```
return p;
}
```

2. After setup, it transitions to `RUNNABLE` (matching the `Ready` state in the diagram) using functions like `userinit()` or `fork()`.

We have this line of code:

```
np->state = RUNNABLE;
```

4.

- The maximum number of processes in xv6 is defined by the constant `NPROC`. This constant is typically set to **64** in `param.h`:

  ```
  #define NPROC 64 // maximum number of processes
  ```

- **What happens if a process creates too many child processes and exceeds the limit?**
  - If a process attempts to create more processes than the limit (`NPROC`), the `allocproc` function will fail to find an `UNUSED` process slot in the process table (`ptable`) and will return `0`. This means no new process will be created.
  - The user-level program attempting to fork will receive an error. Specifically, the `fork` system call will return `1` to indicate failure.
- **How does the kernel and user program react?**
  - **Kernel Reaction**:
    - The kernel doesn't crash or behave unpredictably. Instead, it gracefully handles the situation by not allocating any more processes when the limit is reached.
    - This behavior is implemented in the `allocproc` function, specifically in this part:If no `UNUSED` process is found, `allocproc` simply returns `0`.

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
  if(p->state == UNUSED) {
    goto found;
  }
}
release(&ptable.lock);
return 0;
```

- **User-Level Reaction**:
  - The user-level program will typically check the return value of `fork`. If it receives `1`, it can react by printing an error message or handling the failure gracefully.

5.

For Multicore Systems:

In multicore systems, the `ptable.lock` plays a critical role in protecting the invariants of the process's state and context fields, which are not maintained during the execution of `swtch`. An example of an issue that could arise without `ptable.lock` occurs when a different CPU attempts to run a process after `yield` has set its state to `RUNNABLE` but before `swtch` stops using the process's kernel stack. In such a scenario, two CPUs might end up using the same stack simultaneously, resulting in undefined behavior and corruption. The lock prevents this by ensuring that no other CPU can schedule the process until `swtch` completes.

Additionally, `ptable.lock` safeguards several other critical operations:

1. **Allocation of Process IDs and Table Slots:** It ensures that process IDs and table slots are managed consistently across CPUs, preventing duplication or invalid assignments.
2. **Synchronization Between `exit` and `wait`:** It coordinates the termination of processes and parent-child relationships to avoid race conditions when a process exits while another waits for it.

3. **Lost Wakeups Avoidance:** The lock prevents issues where a wakeup signal is sent but not received due to timing inconsistencies.
4. **General Protection of Process States:** It ensures that the process state transitions (e.g., from `RUNNING` to `SLEEPING`) occur atomically.

It is worth considering whether the various functions of `ptable.lock` could be divided into separate, specialized locks for clarity and potential performance improvements.

For Single-Core Systems:

Even on single-core systems, locks are essential because concurrency arises from interrupts and kernel preemption. Xv6 uses spin-locks to protect shared data accessed by both interrupt handlers and threads. For instance, a timer interrupt might increment the `ticks` counter while a kernel thread reads `ticks` in `sys_sleep` . The lock `tickslock` serializes these accesses, preventing race conditions.

Interrupts can preempt kernel code at any moment, even in single-core systems. For example:

- A timer interrupt could trigger process scheduling or modify the process table while another thread is already performing operations on the same table.
- Hardware events could wake up sleeping processes, altering the state of the process table concurrently with other kernel operations.

Locks like `ptable.lock` ensure that such interrupts do not cause inconsistencies. While only one thread runs on the CPU at a time in single-core systems, the preemption caused by interrupts creates a need for synchronization mechanisms to serialize access to shared resources.

6.

- **Single-Core Round-Robin Scheduling**:
  - xv6 uses a round-robin scheduling policy, where the `scheduler` iterates over the process table, checking each process's state.
  - If a process becomes `RUNNABLE`, it will only be scheduled in the **next iteration** of the scheduler's loop after the current iteration completes.

    - This is because the `scheduler` continuously loops through the process table, and newly `RUNNABLE` processes will be considered in subsequent passes.
- **Scheduler Behavior**:
    - The `scheduler` acquires `ptable.lock` and iterates over all processes in the table. If it finds a process with `p->state == RUNNABLE`, it prepares that process to run.
    - If a process becomes `RUNNABLE` while the scheduler is in the middle of iterating, it won't be considered until the scheduler starts a new iteration.
    - This behavior ensures fairness and avoids starvation, as every `RUNNABLE` process gets a chance to be considered during the scheduler's iterations.
- **Timing and Context**:
    - When a process yields or is preempted (e.g., by a timer interrupt), the scheduler selects the next `RUNNABLE` process in its loop. If no processes are `RUNNABLE`, it waits for an interrupt or external event to mark a process as `RUNNABLE`.
    - Processes that just became `RUNNABLE` are not immediately scheduled in the middle of the current iteration; they wait until the next iteration.

7.

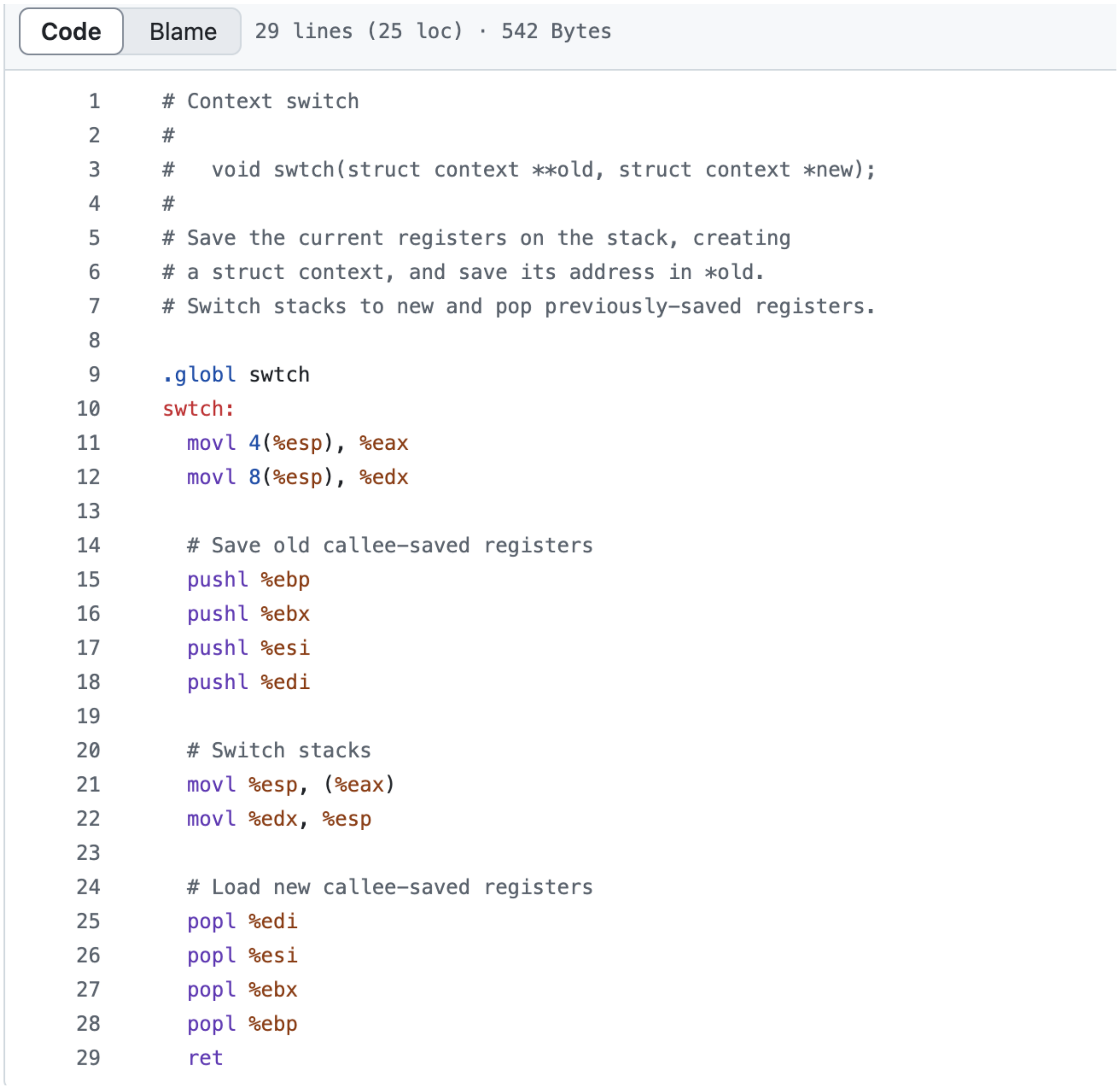
Registers in `context`:

In the code (defined in `proc.h`), you see this:

```
//PAGEBREAK: 17
// Saved registers for kernel context switches.
// Don't need to save all the segment registers (%cs, etc),
// because they are constant across kernel contexts.
// Don't need to save %eax, %ecx, %edx, because the
// x86 convention is that the caller has saved them.
// Contexts are stored at the bottom of the stack they
// describe; the stack pointer is the address of the context.
// The layout of the context matches the layout of the stack in swtch.S
// at the "Switch stacks" comment. Switch doesn't save eip explicitly,
// but it is on the stack and allocproc() manipulates it.
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

- **edi**: Extended Destination Index register, used in data transfer operations.
- **esi**: Extended Source Index register, typically used in pointer arithmetic or string operations.
- **ebx**: A general-purpose register.
- **ebp**: Base Pointer register, used for stack frame references.
- **eip**: Instruction Pointer register, which holds the address of the next instruction to execute.

And for Context switch, we found this code snipped in the `swtch.S`



```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

- `eax`, `ebx`, `ecx`, `edx`: General-purpose registers.
- `edi`, `esi`: Used for data indexing and pointer operations.
- `ebp`: Base pointer (used for stack frames).
- `esp`: Stack pointer.
- `eip`: Instruction pointer (holds the address of the next instruction to execute).

These registers represent the minimal state needed to safely pause and resume a process.

8.

The register that indicates how far the program has executed is the **Program Counter (PC)**, which is referred to as the **Instruction Pointer (eip)** in x86 architecture.

The **Program Counter (eip)** in x86 holds the address of the next instruction to execute. During a context switch in xv6, `eip` is saved in the `context` structure (defined in `proc.h`) to ensure the program can resume from where it left off. The `swtch.S` assembly code handles saving and restoring `eip` along with other registers, enabling seamless multitasking.

- **Before the context switch**, the `eip` (along with other registers) is saved in the process's `context` structure.
- **During the context switch**, the kernel switches to a new process and loads the saved `eip` value, allowing the new process to resume execution from the exact point it was interrupted.

9.

I think the answer can be driven from the book xv6 chapter 5 page 64:

In the **scheduler** function of xv6, the `sti()` function is used to enable interrupts at the start of each iteration of the scheduling loop. This is crucial for ensuring the system remains responsive.

When the scheduler is running, it holds the `ptable.lock` for most of its operations but releases it when calling `switchuvm()` and explicitly enables interrupts. This is particularly important when the CPU is idle and no `RUNNABLE` process is found. Without enabling interrupts, no process could be scheduled, and system calls or context switches would be blocked, preventing processes from transitioning to the `RUNNABLE` state.

If interrupts were disabled and the `ptable.lock` held continuously, no other CPU could perform context switches or handle I/O operations. This would prevent I/O completions from being processed, leaving processes blocked and unable to become `RUNNABLE`. Therefore, enabling interrupts ensures that I/O can be completed, allowing processes to resume execution and the scheduler to function correctly.

Timer interrupts ensure that no process can monopolize the CPU for too long and allow the operating system to manage multiple processes effectively.

Interrupts allow the scheduler to preempt running processes and respond to I/O events. This enables processes to move from a `BLOCKED (sleeping)` state (e.g., waiting for I/O or a special event) to a `RUNNABLE` state when the required resources become available.

10.

The **timer interrupt** in xv6 occurs every **10 milliseconds**

If we look for the answer in the xv6 book, in page 45, we have this quote:

> We would like the timer hardware to generate an interrupt, say, 100 times per second so that the kernel can track the passage of time and so the kernel can time-slice among multiple running processes. The choice of 100 times per second allows for decent interactive performance while not swamping the processor with handling interrupts.

11.

In file `trap.c` in function trap(), we have this code snipped

```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

It checks whether our state is running and the timer is activated. If so, it calls the yield() function.

What does yield() do? As you know, when the timer interrupt occurs (it signals that your quantum time has finished), it abandons the process and changes its state from running to runnable.

```
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

Then, we call the `sched` function. As it is discussed, It performs the context switch (switching from the process's context to the scheduler's context at the end).

12.

How It Works:

1. **Timer Interrupt Trigger:**

   When the timer interrupt fires (based on the APIC timer in `lapic.c`), xv6 calls `trap()` in `trap.c`.

   - The **APIC timer** in `lapic.c` is initialized in `lapicinit()` with:

     ```
     lapicw(TDCR, X1);                          // Set timer di
     visor
     lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));  // Ena
     ble periodic timer interrupts
     lapicw(TICR, 10000000);                   // Set initial t
     imer count
     ```

- This makes the timer **fire periodically**, generating an **interrupt**.
- If a larger divisor like `X2`, `X4`, or `X16` is used, the countdown would be slower, reducing the timer's interrupt frequency. Now timer interrupt occurs every **10 milliseconds.**

Example:

If the CPU's bus clock runs at **1GHz** and the divisor is set to `X1`, the timer counts down from `TICR = 10000000` in **10ms**:

$$\text{Time} = \frac{\text{TICR}}{\frac{\text{CPU Frequency}}{\text{Divisor}}} = \frac{10{,}000{,}000}{\frac{1{,}000{,}000{,}000}{1}} = 0.01s = 10ms$$

With X2:

$$T = \frac{10{,}000{,}000}{\frac{1{,}000{,}000{,}000}{2}} = 0.02s = 20ms$$

With X4:

$$T = \frac{10{,}000{,}000}{\frac{1{,}000{,}000{,}000}{4}} = 0.04s = 40ms$$

2. **Tick Increment:**

```
if (cpuid() == 0) {
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);  // Wake processes waiting on ticks
    release(&tickslock);
}
```

3. **Yield Trigger:**

After incrementing `ticks`, the kernel checks if a running process needs to be preempted:

```
if (myproc() && myproc()->state == RUNNING &&
tf->trapno == T_IRQ0 + IRQ_TIMER)
    yield();  // Force a context switch
```

4. **Context Switch:**

```
void yield(void) {
    acquire(&ptable.lock);  // Lock the process table
    myproc()->state = RUNNABLE;  // Mark current process a
s ready
    sched();  // Call scheduler to pick the next process
    release(&ptable.lock);
}
```

`yield()` calls `sched()`, which triggers the **scheduler** to pick the next process from the ready queue.

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

    // Save process state
    swtch(&p->context, &cpu->scheduler);
}
```

So, every time the tick is incremented, the time quantum expires, and xv6 triggers a context switch to give the CPU to the next process. As we know (question 10 & question 11), a tick corresponds to approximately 10ms. Therefore, the default round-robin scheduling algorithm in xv6 assigns a **10ms quantum slice** to each process. This is because the timer interrupt, which increments the tick, is activated every 10ms.

13.

For example, in init.c, we have this:

```
while((wpid=wait()) >= 0 && wpid != pid)
  printf(1, "zombie!\n");
```

In wait() function, we have:

```
// Wait for children to exit.  (See wakeup1 call in proc_exit.)
sleep(curproc, &ptable.lock);  //DOC: wait-sleep
```

And in exit() function, we have this code:

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

So the parent sleeps in the wait function until its child wake it up when the child exits. So it uses the sleep() function to wait for the child to exit.

14.

What are the other usage examples of sleep()?

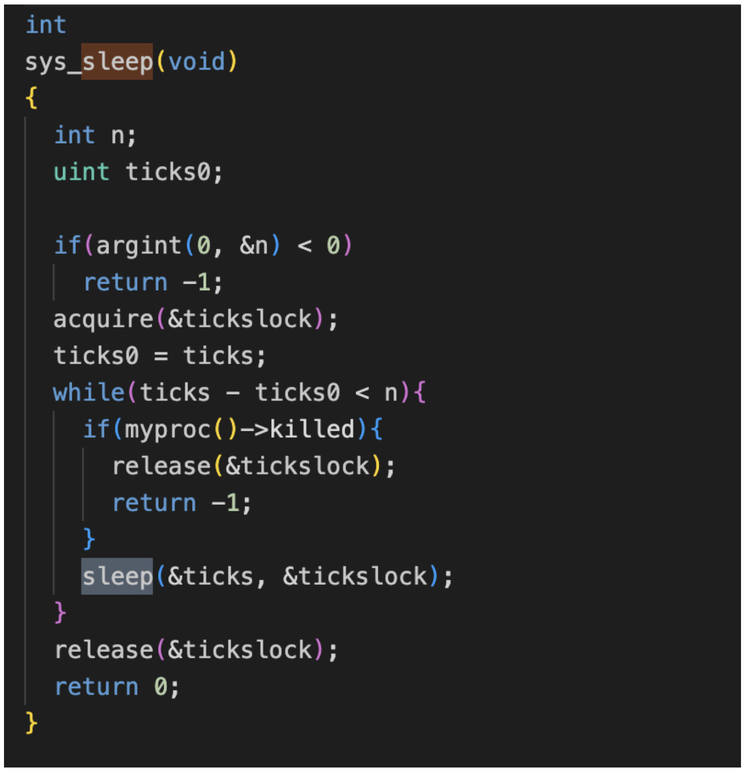
The `sleep` function in xv6 is a versatile system call used to block a process until a specific event or condition occurs. Apart from its use in the `wait` function, where a parent process waits for its child to exit, here are other usages of `sleep` in xv6, with examples:

---

1. Waiting for a Timer/Delay

- The `sys_sleep` function allows a process to sleep for a specified number of clock ticks.
- **Example**: A user program might use `sleep(n)` to pause execution for `n` clock ticks, effectively introducing a delay.

```
// User program example
sleep(100); // Sleep for 100 clock ticks
```

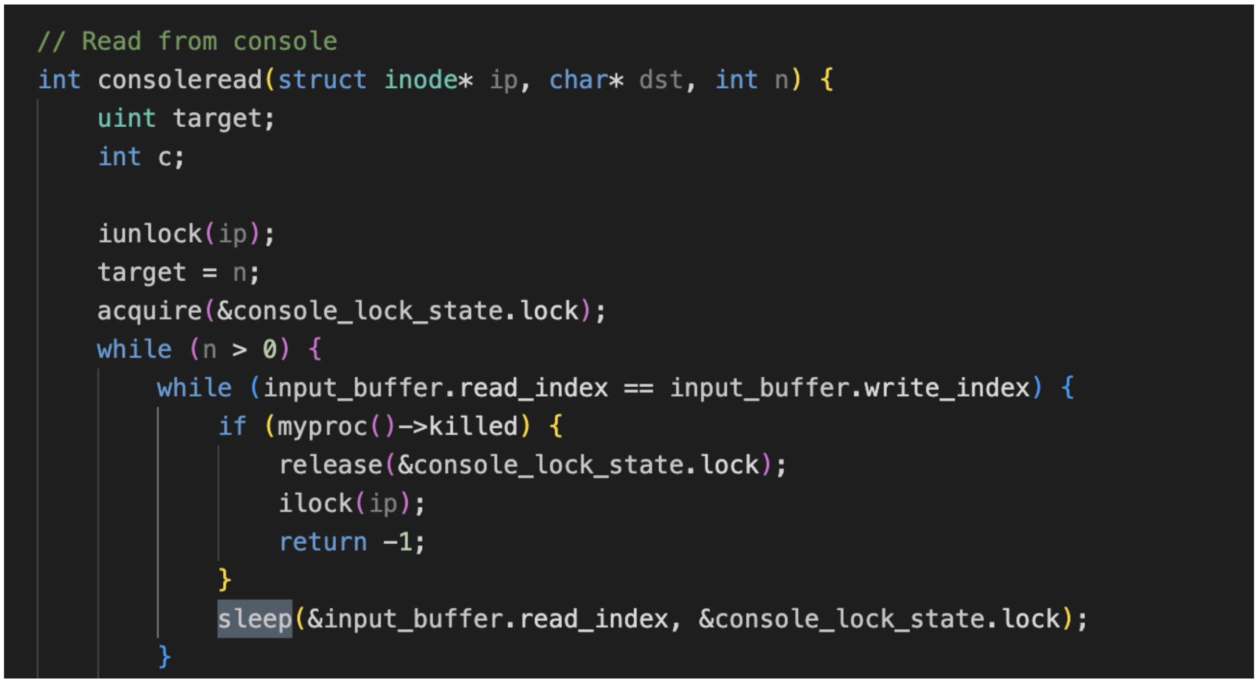


```
int
sys_sleep(void)
{
  int n;
  uint ticks0;

  if(argint(0, &n) < 0)
    return -1;
  acquire(&tickslock);
  ticks0 = ticks;
  while(ticks - ticks0 < n){
    if(myproc()->killed){
      release(&tickslock);
      return -1;
    }
    sleep(&ticks, &tickslock);
  }
  release(&tickslock);
  return 0;
}
```

2. Waiting for a Free Buffer in the File System

- In the xv6 file system, `sleep` is used to wait for a buffer to become available when all buffers are busy.
- **Example**: In `bget()` (buffer cache allocation), if no free buffer is available, the process sleeps until another process releases a buffer.

```
if (!buffer_free) {
    sleep(&buffer_table, &buffer_table_lock);
}
```



```c
// Read from console
int consoleread(struct inode* ip, char* dst, int n) {
    uint target;
    int c;

    iunlock(ip);
    target = n;
    acquire(&console_lock_state.lock);
    while (n > 0) {
        while (input_buffer.read_index == input_buffer.write_index) {
            if (myproc()->killed) {
                release(&console_lock_state.lock);
                ilock(ip);
                return -1;
            }
            sleep(&input_buffer.read_index, &console_lock_state.lock);
        }
```

3. Synchronization in Device Drivers

- `sleep` is used in device drivers to block processes waiting for I/O operations to complete.
- **Example**: In the console driver, a process may sleep while waiting for input from the user.

```
if (input_not_available) {
    sleep(&input_queue, &input_queue_lock);
}
```

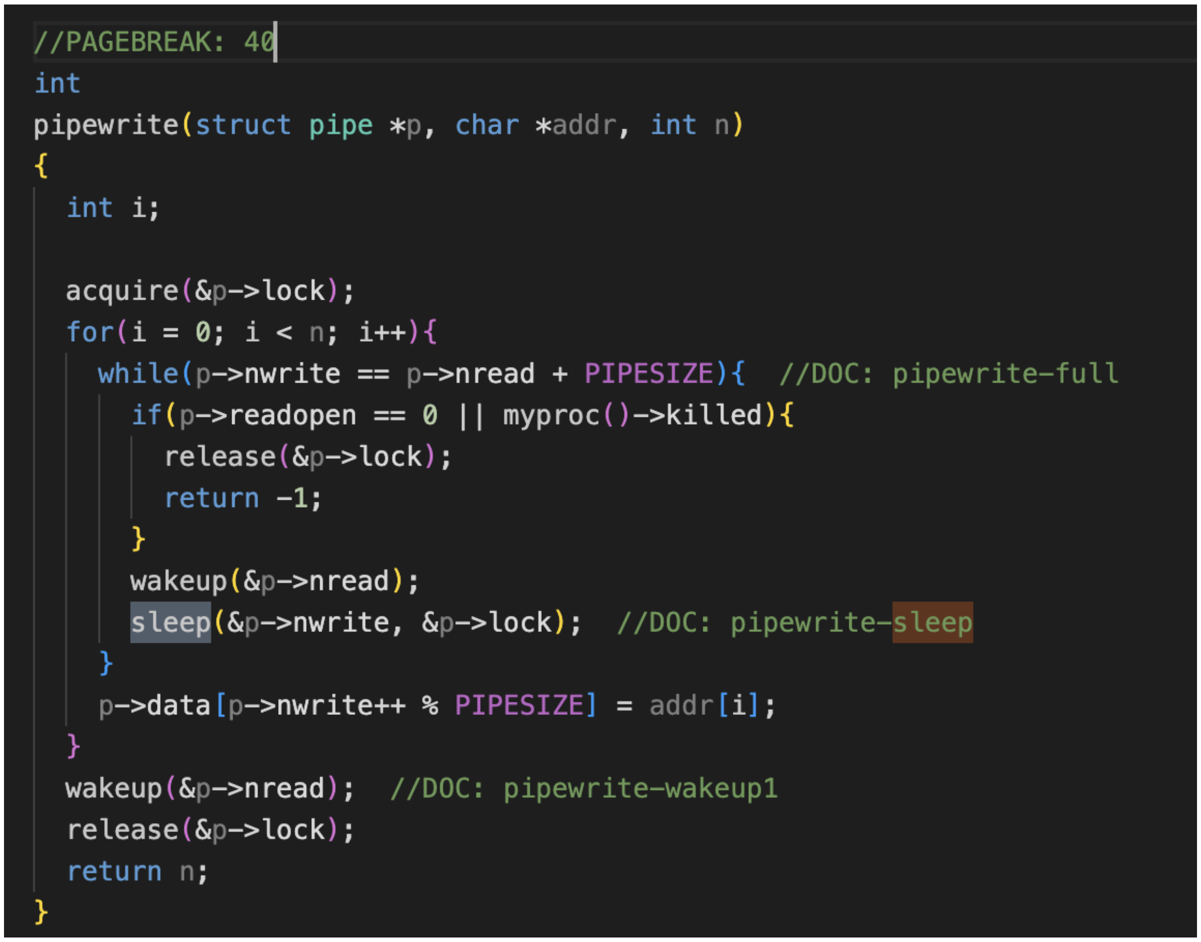
---

4. Process Synchronization with Events

- Processes often sleep while waiting for certain events or conditions to be satisfied.
- **Example**: In `pipe.c`, processes sleep while waiting for data to be written to or read from a pipe.

```
if (pipe_empty) {
    sleep(&pipe->read_queue, &pipe->lock);
}
```

Or it sleeps when it is full and it wants to write



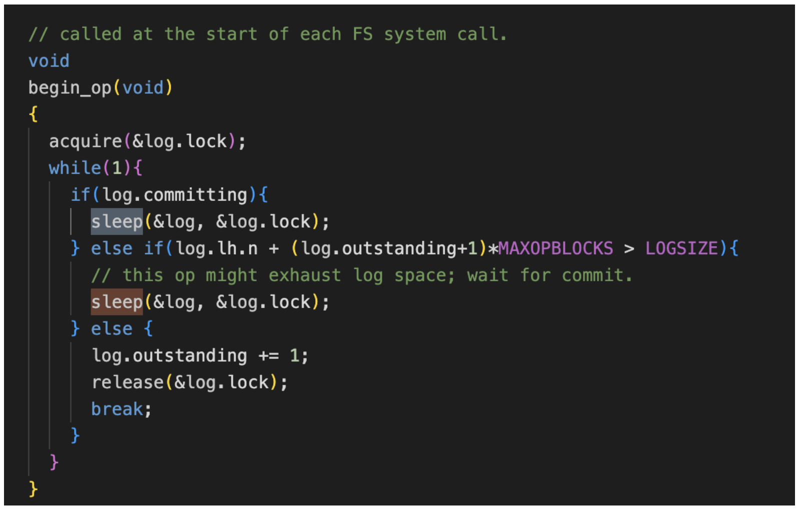
```
//PAGEBREAK: 40
int
pipewrite(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  for(i = 0; i < n; i++){
    while(p->nwrite == p->nread + PIPESIZE){  //DOC: pipewrite-full
      if(p->readopen == 0 || myproc()->killed){
        release(&p->lock);
        return -1;
      }
      wakeup(&p->nread);
      sleep(&p->nwrite, &p->lock);  //DOC: pipewrite-sleep
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
  }
  wakeup(&p->nread);  //DOC: pipewrite-wakeup1
  release(&p->lock);
  return n;
}
```

5. Memory Management

- `sleep` is used to handle memory allocation scenarios where processes need to wait for memory to be freed.
- **Example**: In `kalloc.c`, a process may sleep if it cannot allocate memory immediately.

```
if (memory_not_available) {
     sleep(&memory_pool, &memory_pool_lock);
}
```

6. File System's Logging Mechanism



```
// called at the start of each FS system call.
void
begin_op(void)
{
  acquire(&log.lock);
  while(1){
    if(log.committing){
      sleep(&log, &log.lock);
    } else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
      // this op might exhaust log space; wait for commit.
      sleep(&log, &log.lock);
    } else {
      log.outstanding += 1;
      release(&log.lock);
      break;
    }
  }
}
```

- The `begin_op()` function is called at the start of each file system operation (e.g., creating a file, writing to a file). The function is designed to manage concurrent operations that modify the file system while ensuring consistency and preventing log overflow. In this function:
- **Condition 1**: If a commit is in progress (`log.committing`), the process sleeps.
- **Condition 2**: If adding another operation would exceed the log size, the process sleeps.

15.

As I told you in question 13, the function in the kernel that is responsible for **notifying a process about an event it was waiting for** is the `wakeup()` function. like:

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

The definition of wakeup functions are here:

```
//PAGEBREAK!
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
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;
}

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

16.

The `wakeup()` function causes a transition in the state of a process from `SLEEPING` **(waiting** for an event**)** to `RUNNABLE` **(ready** to be scheduled**).**

17.

There is another function in **xv6** that can cause the same transition from `SLEEPING` to `RUNNABLE`. This function is `kill()`.

Explanation:

- The `kill()` function sends a signal to a process, typically to terminate it.
- If the process is in the `SLEEPING` state, the `kill()` function forces the process to wake up and transitions it to the `RUNNABLE` state so it can handle the signal.
- This ensures that even sleeping processes can respond to termination signals.

Code Example in xv6:

Here's the relevant code snippet from `proc.c` (in the `kill` function):

```
c
Copy code
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;  // Mark process as killed
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING)
        p->state = RUNNABLE; // Transition to RUNNABLE state
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1; // Process not found
}
```

Transition in State:

- **From:** `SLEEPING`
- **To:** `RUNNABLE`

When is this Transition Triggered?

- If a process is sleeping and receives a termination signal (via `kill()`), the `state` of the process is set to `RUNNABLE`, ensuring the scheduler can pick it up and terminate it appropriately.

18.

If a parent did not invoke wait() and instead terminated, It leaves its child processes as orphans.

In **xv6**, orphan processes are handled by reassigning their parent to a special process called `init`. This ensures that no process in the system remains without a parent, maintaining proper process hierarchy and resource management.

How xv6 Handles Orphans:

1. **When a Parent Process Exits**:
    - When a process terminates, its children (if any) are reassigned to the `init` process.
    - The `init` process is always running in the background and serves as the "adoptive parent" for orphaned processes.
    - The `init` process is responsible for calling `wait()` to clean up these orphaned processes when they exit.
2. **Code Explanation**:
    - The reassignment happens in the `exit()` function, located in `proc.c`.
    - Here's the relevant part of the code from `proc.c`:

```
c
Copy code
void
exit(void)
{
    struct proc *p;
    int fd;

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

    // Reparent any children to init
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->parent == myproc()){
            p->parent = initproc;  // Reassign orphaned ch
ildren to init
            if(p->state == ZOMBIE)
                wakeup1(initproc); // Wake up init to clea
n up zombies
        }
    }

    // Finalize current process exit
    myproc()->state = ZOMBIE;
    sched();
}
```

3. **Key Steps in the Code**:
   - The kernel iterates through all processes in the `ptable` (process table).
   - Any process whose parent is the exiting process (`myproc()`) is reassigned to the `initproc`.
   - If any of these orphaned processes are in the `ZOMBIE` state, the `init` process is woken up so it can reap (clean up) these processes.
4. **Role of `init`**:
   - The `init` process periodically calls `wait()` to reap zombie processes.
   - This ensures that resources held by terminated processes are freed and the system remains clean.

Why This Approach?

- Reassigning orphaned processes to `init` ensures that no process is left in an undefined state, preventing resource leaks or inconsistencies in the process table.
- The `init` process acts as a central, stable parent for orphan processes, which is a standard design in UNIX-like operating systems.

19.

If the number of CPUs (`CPUS`) in xv6 is set to **1** versus **2**, and we are using the **Round Robin** scheduling policy, the results differ because of the way the CPUs share the workload and handle processes. Here's why:

Key Differences Between 1 CPU and 2 CPUs in Round Robin Scheduling:

1. **Single CPU (`CPUS = 1`):**
    - All processes are executed on a single CPU. This means:
        - Only **one process** can run at any given time.
        - The kernel switches between `RUNNABLE` processes in the process table in a round-robin fashion.
        - Context switches happen when a process voluntarily yields (e.g., through I/O or `yield()`), or when its time quantum expires (due to a timer interrupt).
    - Since there is no parallelism, the overall throughput is limited. Processes waiting for I/O or sleeping will block the CPU, although the kernel can still switch to other `RUNNABLE` processes.
2. **Two CPUs (`CPUS = 2`):**
    - Processes can run **in parallel** on two CPUs. This introduces:
        - **Improved throughput**: More processes can execute simultaneously, reducing the overall time to complete tasks.

    - **Concurrency**: Processes are not limited to waiting for a single CPU; while one CPU is busy, the other can execute another process.
- **Load balancing**: The scheduler ensures that both CPUs are kept busy if there are enough `RUNNABLE` processes.
- Potential **conflicts**: Since both CPUs share the process table (`ptable`), synchronization is required (e.g., using locks) to ensure correct scheduling behavior and avoid race conditions.

Why Do the Results Differ?

1. **Parallelism vs. Serial Execution**:
    - With 1 CPU, all processes share the same CPU and are executed one at a time in a round-robin order. With 2 CPUs, processes can run in parallel, which reduces the time required to complete a set of processes.
2. **Idle Time**:
    - With a single CPU, if the CPU is idle (e.g., no `RUNNABLE` processes), the scheduler waits. With two CPUs, while one CPU may be idle, the other can still execute another process, leading to better utilization.
3. **Contention for Locks**:
    - With multiple CPUs, synchronization (e.g., on the process table or other shared data structures) introduces slight overhead due to locking mechanisms. However, this is offset by the gains in parallelism.
4. **Fairness in Scheduling**:
    - In single-CPU round robin, the time quantum for each process is strictly enforced since there's only one CPU. With two CPUs, fairness is still enforced, but processes may be assigned to different CPUs, creating slight variations in scheduling behavior depending on the workload.

Example to Illustrate the Difference:

- Suppose we have **4 processes** (`P1`, `P2`, `P3`, `P4`) that are all CPU-bound and `RUNNABLE`.

- **With 1 CPU**: The processes execute as follows:

  ```
  Time:  | 0-1  | 1-2  | 2-3  | 3-4  | 4-5  | ...
  CPU1:  | P1   | P2   | P3   | P4   | P1   | ...
  ```

  Only one process runs at any time.

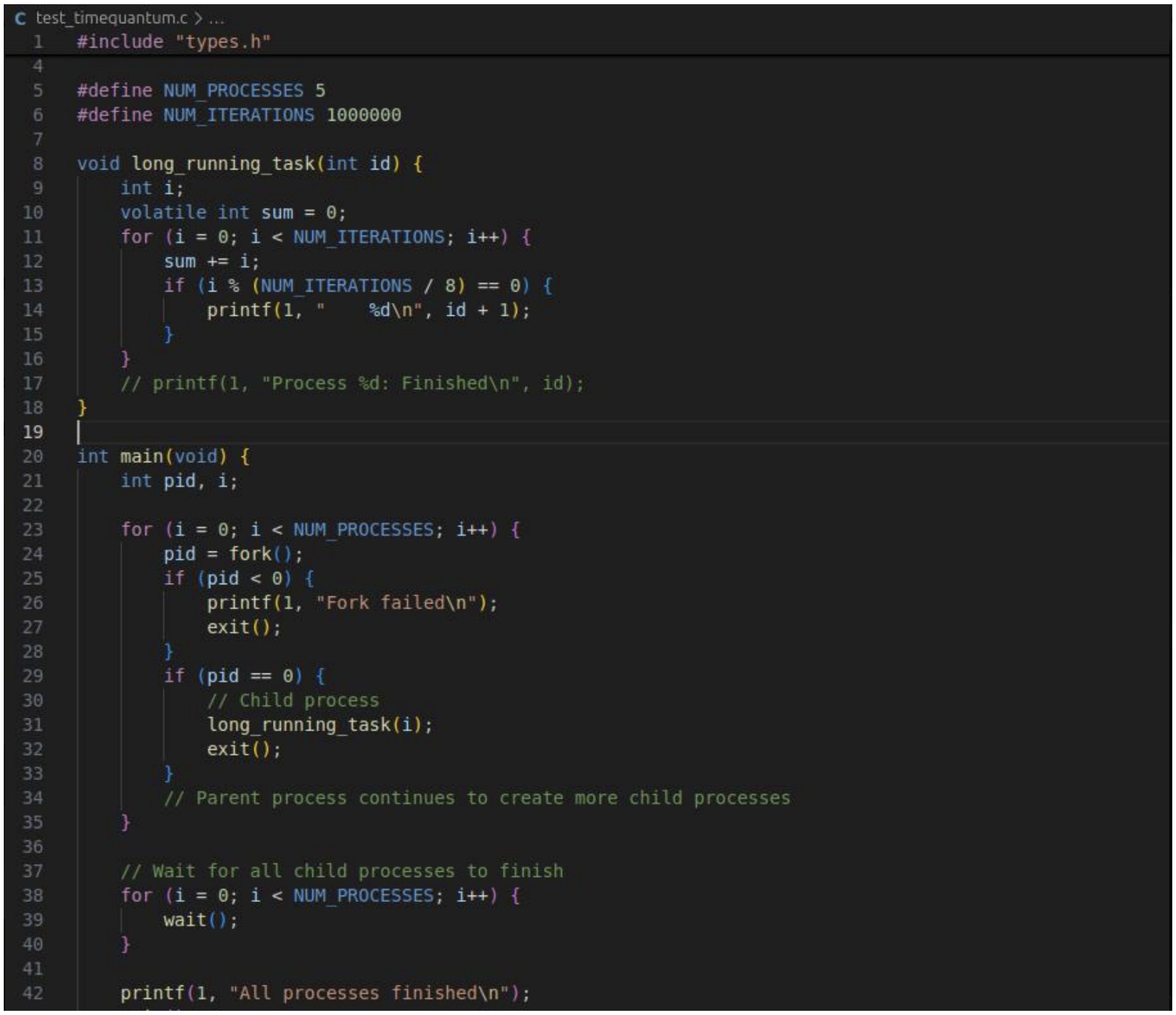
- **With 2 CPUs**: Two processes can run in parallel:

  ```
  Time:  | 0-1  | 1-2  | 2-3  | 3-4  | ...
  CPU1:  | P1   | P3   | P1   | P3   | ...
  CPU2:  | P2   | P4   | P2   | P4   | ...
  ```

  The total time to complete all processes is halved because of parallelism.

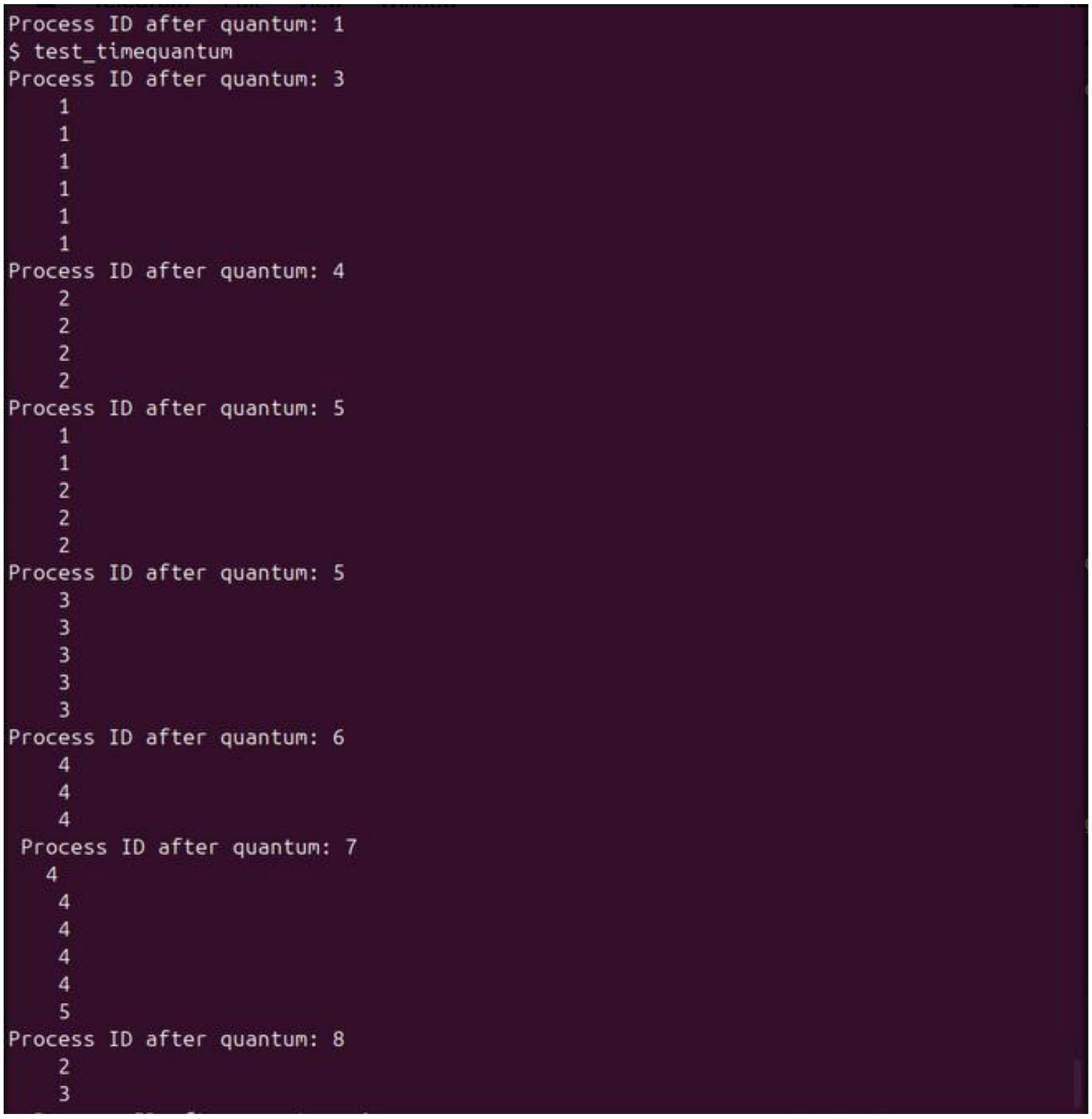
Example In Code

A user-level application forks five processes, each performing 1 million additions. Every 125,000 additions, each process prints its ID.



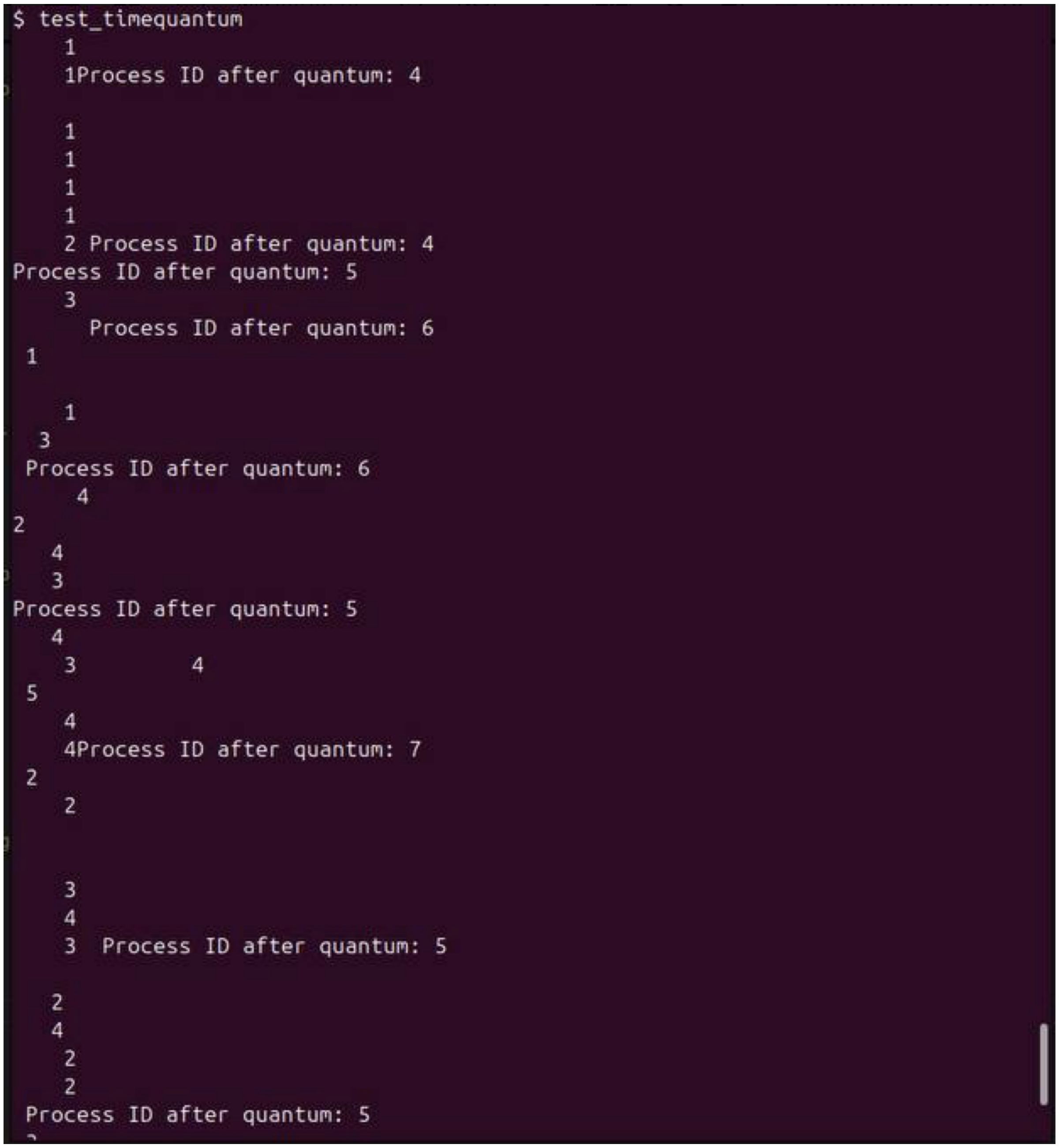
```c
#include "types.h"

#define NUM_PROCESSES 5
#define NUM_ITERATIONS 1000000

void long_running_task(int id) {
    int i;
    volatile int sum = 0;
    for (i = 0; i < NUM_ITERATIONS; i++) {
        sum += i;
        if (i % (NUM_ITERATIONS / 8) == 0) {
            printf(1, "    %d\n", id + 1);
        }
    }
    // printf(1, "Process %d: Finished\n", id);
}

int main(void) {
    int pid, i;

    for (i = 0; i < NUM_PROCESSES; i++) {
        pid = fork();
        if (pid < 0) {
            printf(1, "Fork failed\n");
            exit();
        }
        if (pid == 0) {
            // Child process
            long_running_task(i);
            exit();
        }
        // Parent process continues to create more child processes
    }

    // Wait for all child processes to finish
    for (i = 0; i < NUM_PROCESSES; i++) {
        wait();
    }

    printf(1, "All processes finished\n");
```

- **With 1 CPU:** Since only one process can run at a time, the operating system schedules processes sequentially. In each time quantum, only one process prints its message, resulting in one output line per time quantum.



```
Process ID after quantum: 1
$ test_timequantum
Process ID after quantum: 3
    1
    1
    1
    1
    1
    1
Process ID after quantum: 4
    2
    2
    2
    2
Process ID after quantum: 5
    1
    1
    2
    2
    2
Process ID after quantum: 5
    3
    3
    3
    3
    3
Process ID after quantum: 6
    4
    4
    4
 Process ID after quantum: 7
   4
    4
    4
    4
    4
    5
Process ID after quantum: 8
    2
    3
```

- **With 2 CPUs:** Two processes can run simultaneously, meaning two processes may print messages concurrently. As a result, output lines may contain messages from two processes, and the execution order may appear non-deterministic due to parallel execution.



```
$ test_timequantum
    1
    1Process ID after quantum: 4

    1
    1
    1
    1
    2 Process ID after quantum: 4
Process ID after quantum: 5
    3
      Process ID after quantum: 6
 1

    1
  3
 Process ID after quantum: 6
     4
2
   4
   3
Process ID after quantum: 5
   4
    3         4
 5
    4
    4Process ID after quantum: 7
 2
    2


    3
    4
    3  Process ID after quantum: 5

   2
   4
    2
    2
 Process ID after quantum: 5
```

20.

If you need to initialize additional fields in the `cpu` structure, the appropriate place to do this would be during the **CPU initialization process**, which is performed in the `startothers()` function or possibly earlier in the boot process.

Where to Initialize Fields in `cpu` Structure:

1. **In `startothers()` Function**:
   The
   `startothers()` function is responsible for starting additional CPUs (Application Processors, or APs). It's here where the system sets up and prepares other processors to begin execution.

   If you add fields to the `cpu` structure, this would be a logical place to initialize them, as this function is responsible for setting up the state for all CPUs in the system (except the boot processor).

   In particular, after the entry code (`entryother.S`) is copied into memory and the appropriate parameters are set (e.g., stack pointer, page directory, etc.), you could add any necessary initialization of new `cpu` structure fields within this function, especially since it's specifically designed to set up each CPU for execution.

   Example:

   ```
   c
   Copy code
   // Initialize new fields in cpu structure
   c->new_field = initial_value;
   ```

2. **Earlier in the Boot Process**:
   If you're initializing a field that is part of the
   `cpu` structure but needs to be set before `startothers()` is called, such as during the early CPU setup or bootstrapping phase, you would do this in the **initialization code before the APs are started**. This might involve adding initialization code in functions like `mpinit()` or earlier setup steps.

The `mpinit()` function, which is typically responsible for initializing the multiprocessor (MP) system, can also be a place where you initialize new fields in the `cpu` structure.

Where Exactly to Add the Initialization:

- **In `startothers()`**: After the loop where the entry code for each CPU is set up (and before the CPU is started), you can add your new field initializations for each CPU.

  Example:

  ```
  for(c = cpus; c < cpus + ncpu; c++){
    if(c == mycpu())  // Skip the boot CPU
      continue;

    // Initialization for new cpu fields
    c->new_field = initial_value;  // Initialize the added f
  ield

    // Existing setup code for starting other CPUs
    stack = kalloc();
    *(void**)(code-4) = stack + KSTACKSIZE;
    *(void(**)(void))(code-8) = mpenter;
    *(int**)(code-12) = (void *) V2P(entrypgdir);
    lapicstartap(c->apicid, V2P(code));

    while(c->started == 0)
      ;
  }
  ```

- **Earlier in `main.c`**: If the initialization of the `cpu` structure fields is needed before `startothers()` is invoked, consider doing so right after initializing the `cpu` structure during the system boot phase (in functions like `mpinit()` or in the CPU setup phase).

This is the whole code of this function:

```
// Start the non-boot (AP) processors.
static void
startothers(void)
{
  extern uchar _binary_entryother_start[], _binary_entryother_si
  uchar *code;
  struct cpu *c;
  char *stack;

  // Write entry code to unused memory at 0x7000.
  // The linker has placed the image of entryother.S in
  // _binary_entryother_start.
  code = P2V(0x7000);
  memmove(code, _binary_entryother_start, (uint)_binary_entryoth

  for(c = cpus; c < cpus+ncpu; c++){
    if(c == mycpu())  // We've started already.
      continue;

    // Tell entryother.S what stack to use, where to enter, and
    // pgdir to use. We cannot use kpgdir yet, because the AP pr
    // is running in low  memory, so we use entrypgdir for the A
    stack = kalloc();
    *(void**)(code-4) = stack + KSTACKSIZE;
    *(void(**)(void))(code-8) = mpenter;
    *(int**)(code-12) = (void *) V2P(entrypgdir);

    lapicstartap(c->apicid, V2P(code));

    // wait for cpu to finish mpmain()
    while(c->started == 0)
      ;
    }
}
```

21.

Let’s consider a scenario for **Level 2** that could lead to starvation.

As you know, **Shortest Job First (SJF)** is used in Level 2, meaning that a process with a shorter burst time has higher priority. Suppose there is a process with a long burst time in the second queue, and at the same time, several processes with shorter burst times keep arriving in the queue (with confidence of 100%). As a result, there is always a process with a shorter burst time ready to run. The long burst time process will never get a chance to execute, causing **starvation**.

While **WRR** can adjust the time slice per process based on weights, **SJF** inherently favors short processes, so long processes may still be delayed indefinitely if shorter ones keep arriving.

Now, let’s discuss a scenario where **starvation** can occur in **Level 3**, which uses the **First Come First Served (FCFS)** algorithm. Suppose there is a process that enters an infinite loop, and no interrupts occur during its execution. This process will continuously run and prevent any other processes in the same level from executing. In this case, **starvation** happens because, in **FCFS**, if a process doesn't finish its work, it doesn't give the CPU to the next process in line.

This scenario would not occur in **Level 1**, where **Round Robin (RR)** is used. In Round Robin scheduling, each process has a fixed time quantum. Even if a process enters an infinite loop, its time slice will eventually end, causing it to be placed at the end of the ready queue. As a result, the next process in line will get a chance to execute, preventing starvation.

WRR doesn’t affect FCFS, because FCFS is based on a first-come-first-served principle, and once a process starts executing, it runs to completion without interruption, regardless of weights or time slices.

22.

- The key reason for this distinction is that **waiting time** is meant to measure how long a process waits to be executed by the CPU. While in the **SLEEPING** state, the process is not competing for CPU time, and thus its time spent sleeping does not impact the scheduling of other processes.
- Including time spent in **SLEEPING** state would distort the waiting time measurement because the process is not actively waiting for CPU resources but is instead idle until an event occurs that will wake it up.
- Some scheduling algorithms, like **Shortest Job First (SJF)** or **Priority Scheduling**, may rely on accurate waiting time calculations to make decisions about process execution.
- If **SLEEPING** time is added to the waiting time, a process might appear to be in the queue longer than it actually was. This could result in:
    - **Incorrect priority levels**: Processes that are not actively competing for CPU time (because they are sleeping) may be unfairly penalized or prioritized based on inaccurate metrics.
- If we add the time a process spends in the **SLEEPING** state to the **waiting time** and apply **aging** to this new waiting time, it would mean that **sleeping processes** would have their priority increased, even though they aren't actively waiting for the CPU.