Lecture 3: Operating Systems

Yonghao Lee

April 14, 2025

Introduction

We want to explore how an Operating System runs programs.

From Source Code to Process

The journey begins when we write source code and call the compiler to transform it into an executable file. This executable contains both instructions (the translated code) and data (such as global variables we declared).

The next stage is loading the program into memory. The OS provides a program called the loader to accomplish this task. The loader is responsible for reading the executable file, allocating appropriate memory regions, and copying the various section*s into these regions. When we talk about memory, we are referring to RAM. RAM consists of many addressable locations, and a portion of it will be allocated to create the living copy of our compiled program - this living instance is called a **process**.

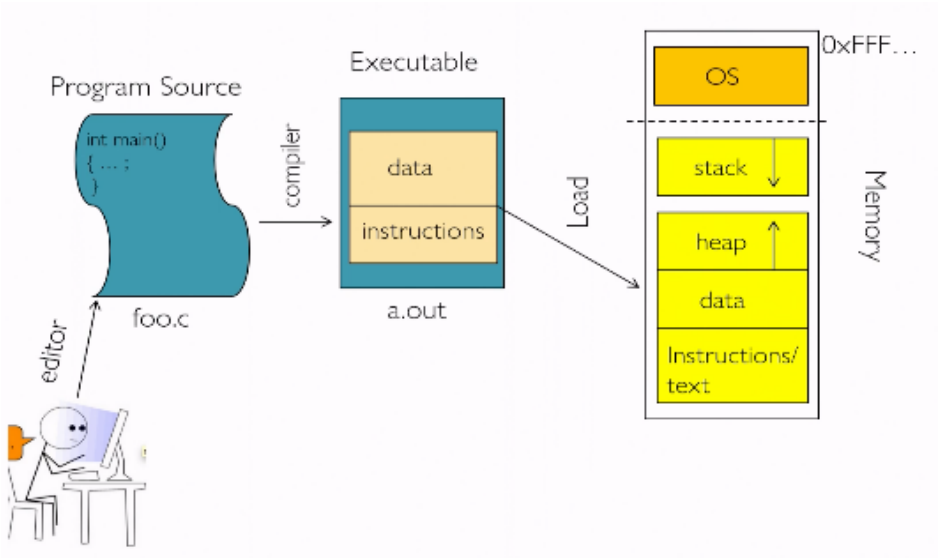

Figure 1: From source code to running program

Memory Layout of a Process

In this living copy (the process), memory is organized into several segments:

- **Text segment (Instructions)**: Contains the executable code we want to run. This section* is typically read-only to prevent a program from accidentally modifying its own instructions.
- **Data segment**: Stores initialized global and static variables. This region contains data that persists throughout the program's lifetime.
- **Stack**: This is where local variables are allocated. When a function is called, the stack is also where the OS saves the return address so execution can continue at the correct location after the function completes. The stack grows downward (toward lower memory addresses).
- **Heap**: This area is used for dynamically allocated memory (e.g., when using malloc() in C or new in C++). The heap grows upward (toward higher memory addresses).

Between the stack and heap exists a gap. This gap is crucial because we generally don't know in advance how large the stack or heap will become during execution. The gap allows both regions to grow as needed. If the stack and heap were to meet (a collision), it would typically result in a program crash such as a stack overflow or an out-of-memory error.

Memory Isolation

At the highest memory addresses, there is a portion allocated to the Operating System kernel. These addresses are very high and our process is forbidden from accessing this memory region. This protection is enforced by the CPU's memory protection mechanisms. This separation is essential for system stability and security.

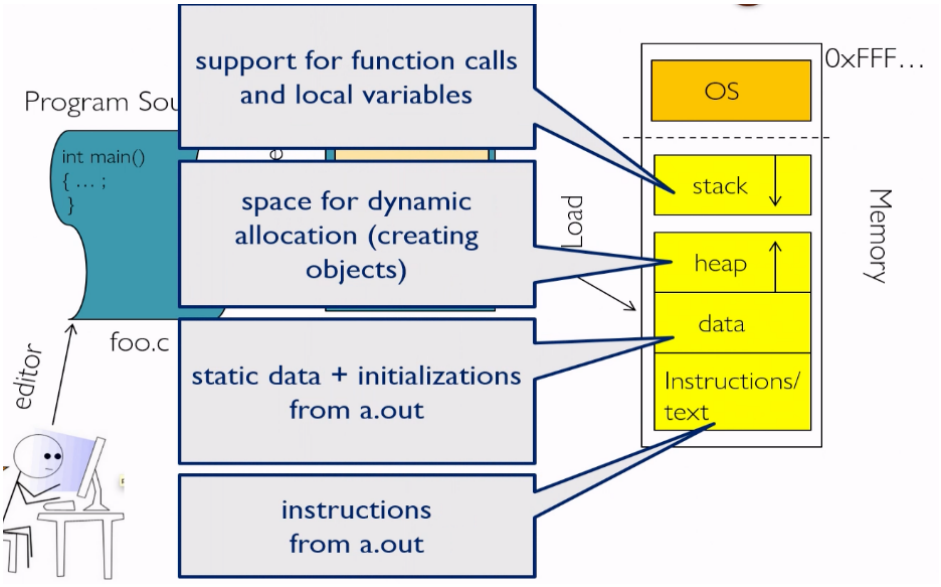

Figure 2: Process memory layout showing protection boundaries

Program Execution

When we have this loaded program on the RAM, the CPU can run it.

The Fetch-Execute Cycle

The CPU executes programs through a continuous cycle called the *fetch-execute cycle* (also known as the instruction cycle). As shown in the diagram, this cycle consists of several steps:

1. **Fetch**: The CPU reads the instruction at the memory address specified by the Program Counter (PC) and stores it in the Instruction Register (IR).

$$\text{IR} \leftarrow \text{Memory}[\text{PC}]$$

2. **Decode**: The CPU interprets the instruction in the IR, determining what operation should be performed and what data or memory locations are involved.

3. **Execute**: The CPU performs the operation specified by the instruction.

4. **Write Back**: The CPU stores the results of the execution in the appropriate location (register or memory).

5. **Update PC**: The Program Counter is updated to point to the next instruction.

$$\text{PC} \leftarrow \text{Next\_Instruction}(\text{PC})$$

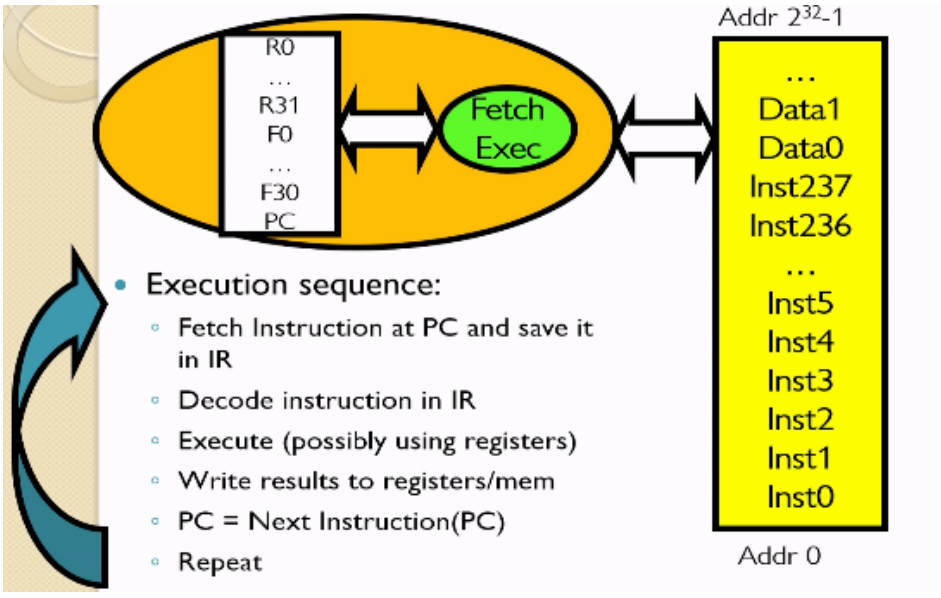

Figure 3: Fetch-execute cycle

Example: x = x + y

We demonstrate how a simple high-level statement `x = x + y` is translated into assembly instructions:

- Initial conditions:
  - Variable `x` is stored at memory location 100
  - Variable `y` is stored at memory location 104
- Assembly implementation:

```
lw r1, 100    #Copy value from memory location 100 (x) into register r1
lw r2, 104    #Copy value from memory location 104 (y) into register r2
add r3, r1, r2 #Compute r1 + r2 and store result in register r3
sw r3, 100    #Copy value from register r3 into memory location 100 (x)
```

The following example illustrates how the CPU executes the instruction sequence for computing $x = x + y$ when $x = 5$ and $y = 8$.

Step-by-Step Execution

Initial State (Figure 4)

- **Memory**:
  - Address 100: Value 5 (variable x)
  - Address 104: Value 8 (variable y)
  - Addresses 0-12: Program instructions
    - Address 0: `lw r1, 100` (Load value from memory location 100 into register r1)
    - Address 4: `lw r2, 104` (Load value from memory location 104 into register r2)
    - Address 8: `add r3, r1, r2` (Add values in r1 and r2, store in r3)
    - Address 12: `sw r3, 100` (Store value from r3 into memory location 100)
- **Registers**:
  - PC (Program Counter): 0 (Points to the first instruction)
  - All other registers have undefined or zero values

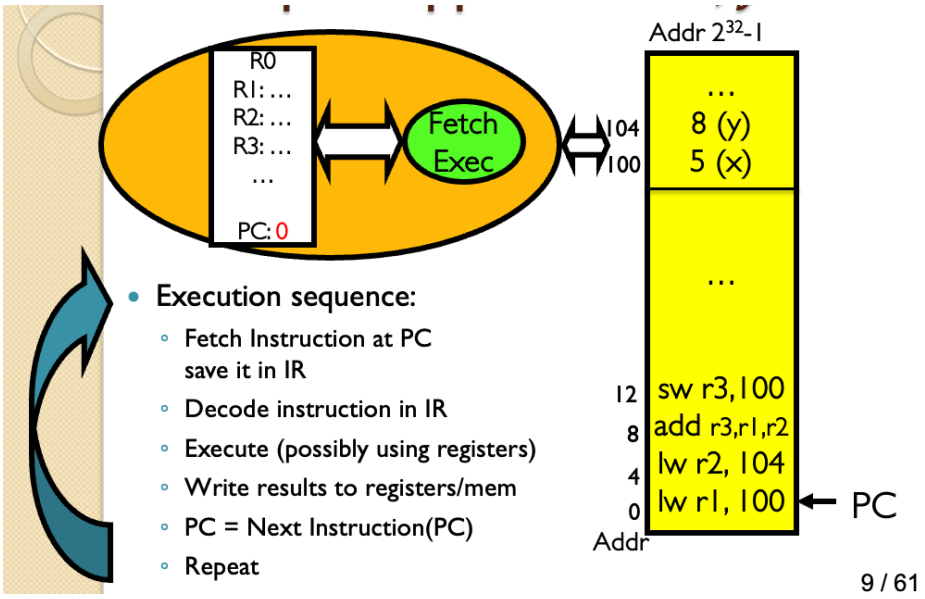

Figure 4: Initial state

After First Instruction (Figure 5)

- **What happened**:
  1. The CPU fetched the instruction at address 0: `lw r1, 100`
  2. It decoded the instruction: "Load the value at memory address 100 into register r1"
  3. It executed the instruction: Retrieved value 5 from memory location 100
  4. It stored the result in register r1
  5. It updated the PC to 4 (the address of the next instruction)
- **Updated state**:
  - Register r1: 5 (now contains the value of x)
  - PC: 4 (pointing to the next instruction)

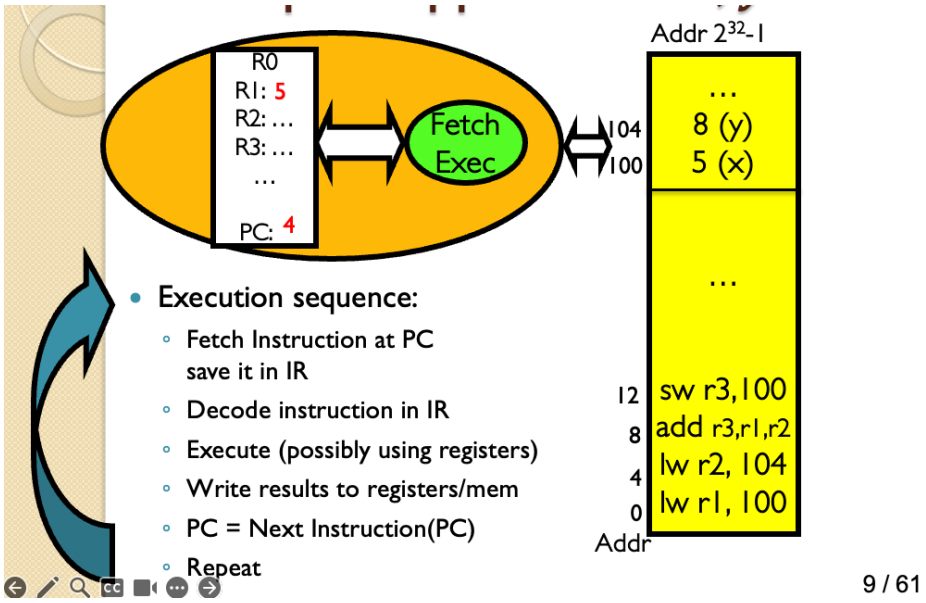

Figure 5: After first instruction

After Second Instruction (Figure 6)

- **What happened**:
  1. The CPU fetched the instruction at address 4: `lw r2, 104`
  2. It decoded the instruction: "Load the value at memory address 104 into register r2"
  3. It executed the instruction: Retrieved value 8 from memory location 104
  4. It stored the result in register r2
  5. It updated the PC to 8 (the address of the next instruction)
- **Updated state**:
  - Register r1: 5 (unchanged)
  - Register r2: 8 (now contains the value of y)
  - PC: 8 (pointing to the next instruction)

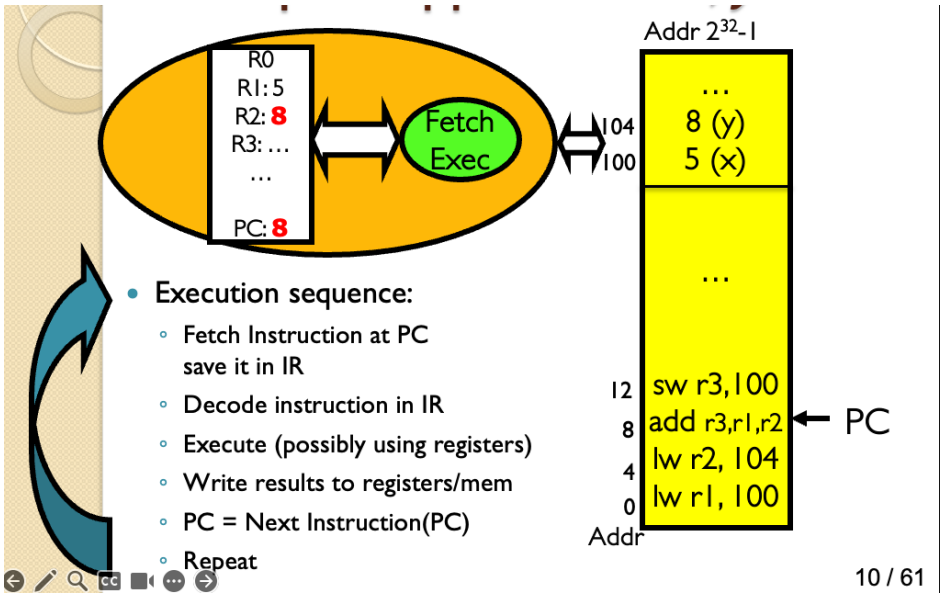

Figure 6: After second instruction

After Third Instruction (Figure 7)

- **What happened**:
  1. The CPU fetched the instruction at address 8: `add r3, r1, r2`
  2. It decoded the instruction: "Add the values in registers r1 and r2, and store the result in r3"
  3. It executed the instruction: Computed $5 + 8 = 13$
  4. It stored the result in register r3
  5. It updated the PC to 12 (the address of the next instruction)
- **Updated state**:
  - Register r1: 5 (unchanged)
  - Register r2: 8 (unchanged)
  - Register r3: 13 (now contains the sum of x and y)
  - PC: 12 (pointing to the next instruction)

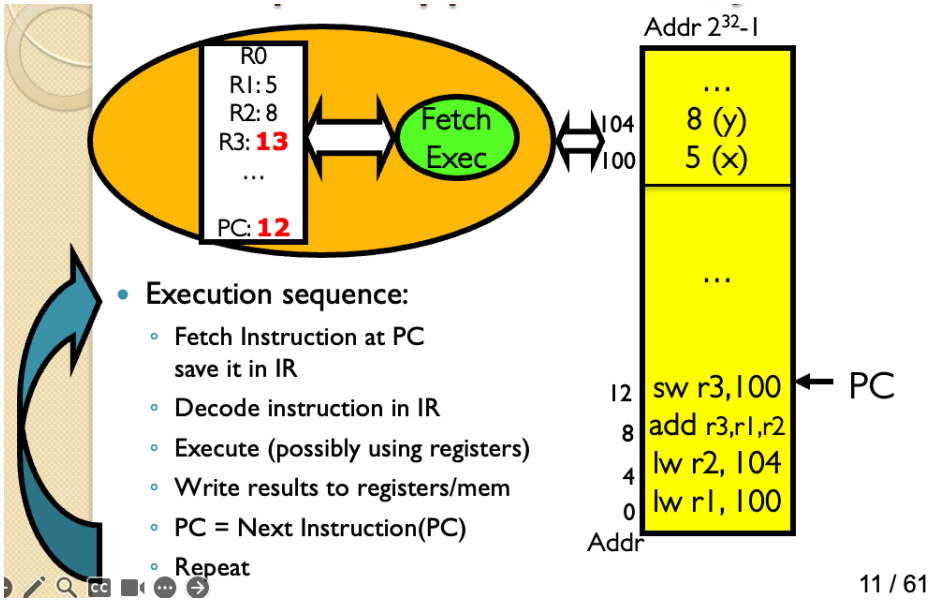

Figure 7: After third instruction

Final Step (Figure 8)

After the CPU finishes all the steps above, PC gets updated to 16, which is right after the last instruction and now at address 100, we have value 13 for the variable x.

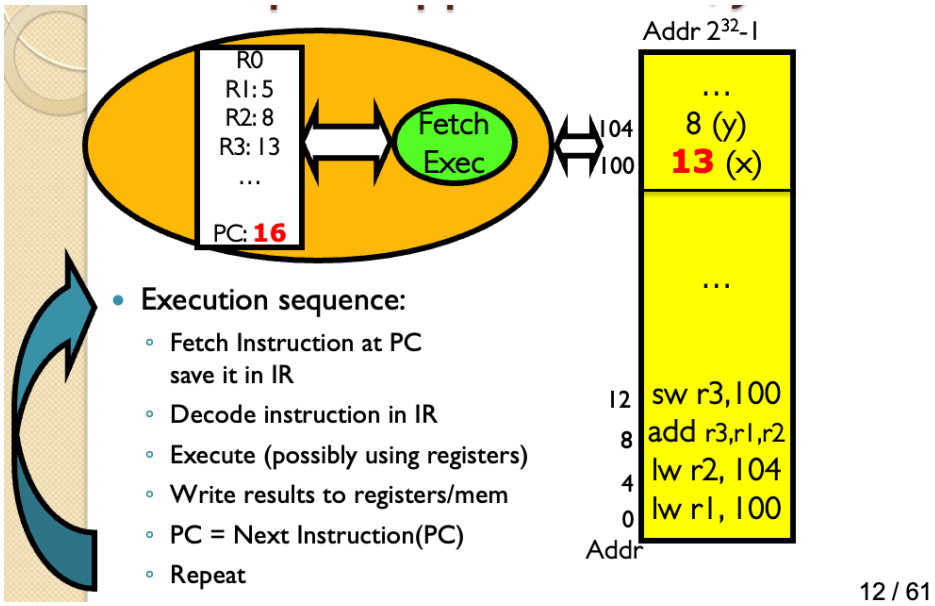

Figure 8: Final step

Context of Execution

The execution context refers to the complete set of information needed to represent the state of a running program at any given moment. It encompasses all the data that would be necessary to pause a program and later resume it exactly where it left off.

A running program's execution context consists of:

CPU Context:
- Program Counter (tracking the next instruction)
- General-purpose registers (holding computation values)
- Special registers (stack pointer, etc.)

Memory Context:
- Code segment (program instructions)
- Data segment (global variables)
- Stack (local variables and function calls)
- Heap (dynamically allocated memory)

OS Context:
- Process metadata (user ID, priority level)
- Resource allocations
- Scheduling information

Program Execution Context	
In CPU	**In Memory**
• Program Counter	• Code (Text)
• General Registers	• Data
• Stack Pointer	• Stack
	• Heap
OS Management Data	
• Process ID, User, Priority	

Processes

A process is an instance of a program in execution. We can conceptualize it as an abstraction of an individual computer—to the code being executed, the process appears to be the entire computing environment.

Processes are managed by the operating system: when we request the OS to run our programs, it creates a new process and allocates the necessary resources (registers, memory, etc.) for execution. The process provides the complete context for program execution.

A process exists for the duration of the execution and is therefore temporary; it is sometimes referred to as a "job" in certain contexts.

A process is said to be *executing* on a CPU when it is *resident* in the CPU registers. When the system has multiple processes to run, only one can be resident at any given time. For non-resident processes, the execution context (register values, program counter, etc.) is stored in a data structure called the *Process Control Block (PCB)* maintained by the operating system in memory.

Process vs. Program

A program is a *passive entity* (a set of instructions stored in a file), whereas a process is an *active entity*.

A program becomes a process when an executable file is loaded into memory by the loader (a component of the operating system). At this point, memory for the stack and heap is allocated, which doesn't exist for a program in its passive state.

The relationship between programs and processes is nuanced:

1. A process encompasses more than just a program, as the program code is only part of the process state. The process also includes resources such as memory allocation, open files, and execution context.

2. Conversely, a single program can invoke multiple processes, as seen in multi-threaded applications or when a program spawns child processes.

Address Space of Processes (Simplified)

The *address space* of a process refers to the range of memory addresses that the process can access during its execution. In a simplified memory management model, the address space can be represented by two parameters: the *base* and the *bound*, which define that a process can access addresses only in the range $[\text{base}, \text{base} + \text{bound})$.

When a process attempts to access a memory address, the operating system verifies whether this address falls within the permitted interval. This mechanism creates protection and isolation: processes cannot access each other's memory, preventing potential interference or security breaches.

Address translation is a key concept here: when a process references address x, the system translates it to the physical address $\text{base} + x$. Since the base value varies between processes, identical pointer addresses in different processes actually refer to different physical memory locations. This is the foundation of virtual memory addressing.

The operating system's own address space is designated as *kernel memory* and is inaccessible to user processes. The remaining portion is *user memory*, where processes operate. This separation is crucial for system security and stability.

Let's look at an example of multiple processes running with process 2 resident on the CPU.

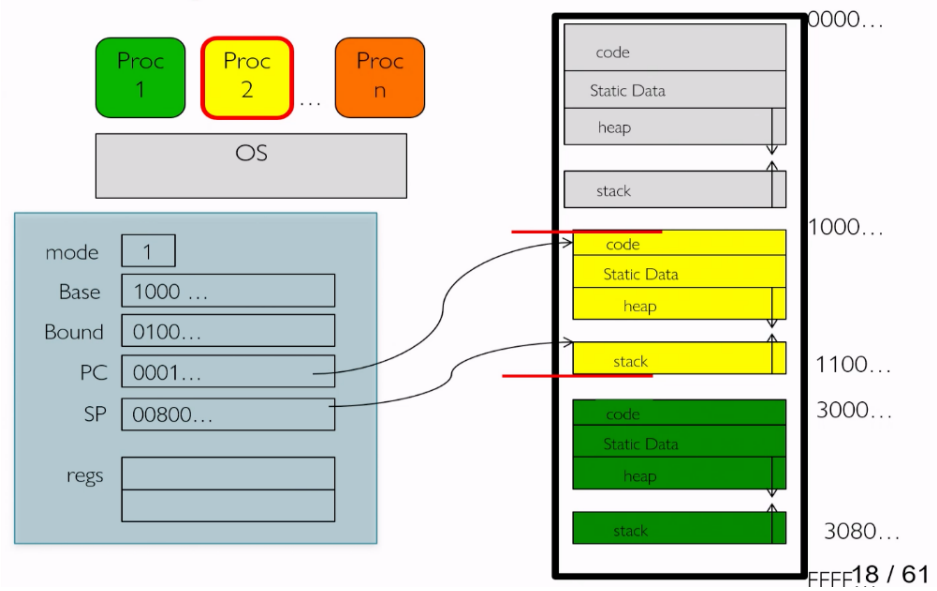

Figure 9: Multiple processes

- **Left side:** Process Control Blocks maintained by the OS, containing:
  - Mode flag (user/kernel mode)
  - Base address (starting point in physical memory)
  - Bound (size limit of allocated memory)
  - Register values (PC, SP, general registers)
- **Right side:** Physical memory layout showing how different processes are allocated separate memory regions:
  - Process 1 (green): Physical addresses 3000-3080
  - Process 2 (yellow): Physical addresses 1000-1100
  - Other processes (gray): Elsewhere in memory
- **Connecting arrows:** Demonstrate address translation—when a process accesses its memory, the system adds the base address to translate virtual addresses to physical locations

This diagram demonstrates memory isolation between processes and the base-bounds approach to memory protection and address translation.

Now we switch to run process 1:

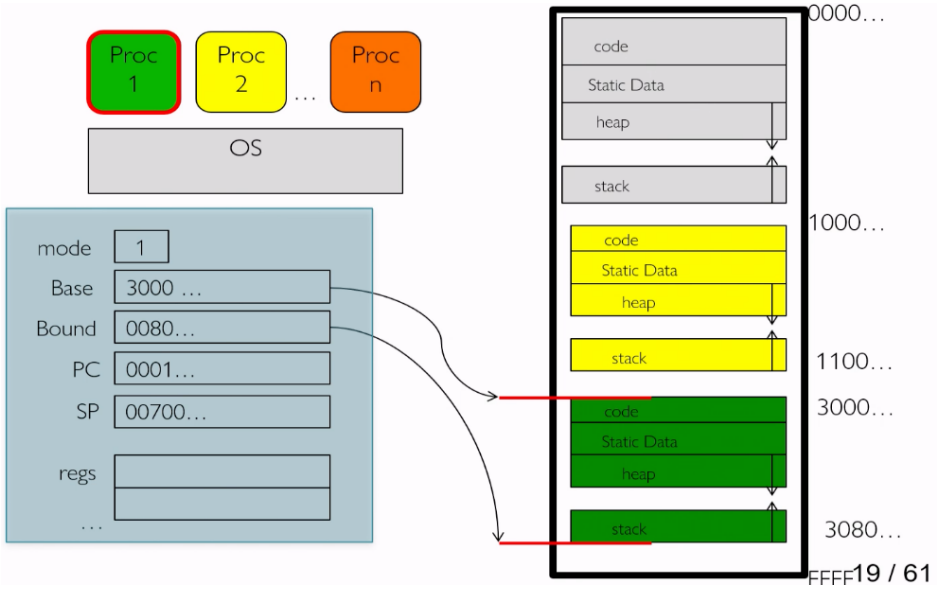

Figure 10: Multiple processes - process 1

We see that the base and the bound change accordingly.

Note that we do not need to do anything to the memory when we switch between processes since everything in the memory stays in the memory and each process has its own slice, no conflict.

Everything in the CPU will need to be copied and saved them somewhere else.

Process Control Block (PCB)

First, how is a process represented in the OS? We can see in the following image:

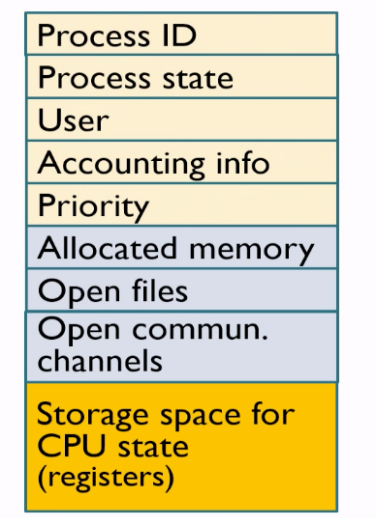

Figure 11: Process representation in the OS

The Life-cycle of a Process

A process transitions through several states during its lifetime:

New: The process is being created by the OS

Ready: The process is waiting to be assigned to a processor

Running: Process instructions are being executed on the CPU

Waiting: The process is blocked, waiting for some event (I/O, timer, etc.)

Terminated: The process has completed execution

- A newly created process enters the ready queue
- The scheduler selects a ready process to run on the CPU
- A running process may:
  - Be preempted (returned to ready state)
  - Block itself waiting for I/O, timer, or terminal
  - Terminate upon completion
- Processes in waiting state return to ready once their wait condition is satisfied

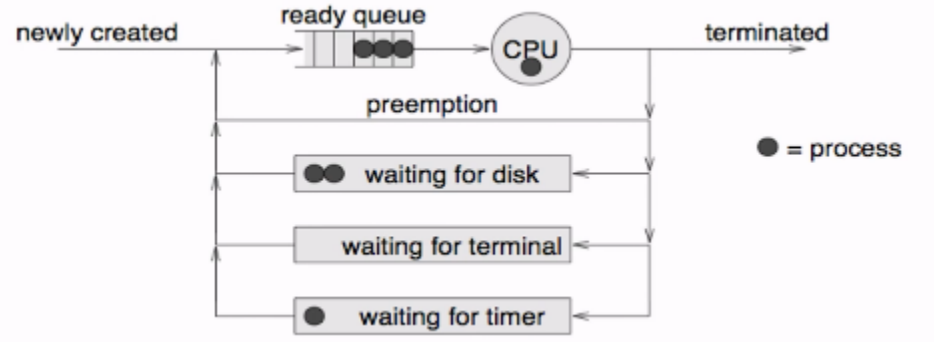

Figure 12: Life-cycle of a process

We look closely to the process states:

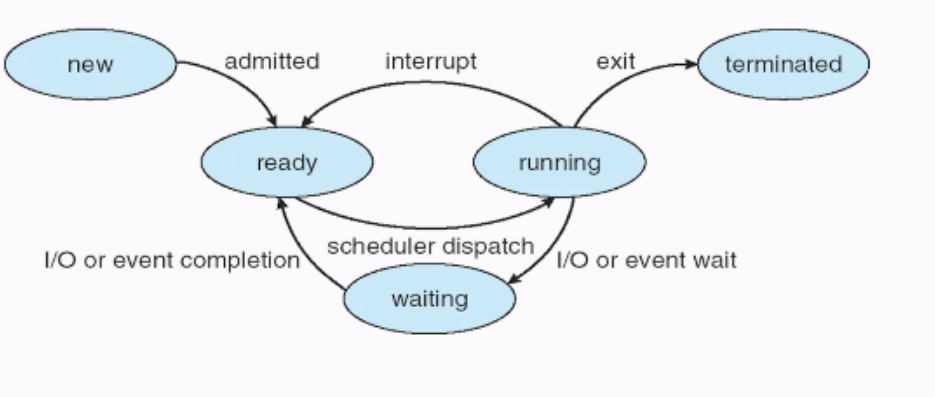

Figure 13: Process States

- **New → Ready (Admitted)**: The operating system admits the process into the ready queue when system resources are available.
- **Ready → Running (Scheduler Dispatch)**: The process scheduler selects this process for execution based on scheduling algorithms.
- **Running → Ready (Interrupt)**: The running process can be interrupted by the scheduler when its time quantum expires (in time-sharing systems) or when a higher priority process becomes ready.
- **Running → Waiting (I/O or Event Wait)**: The process issues an I/O request or waits for some event and cannot continue execution until the operation completes or event occurs.
- **Waiting → Ready (I/O or Event Completion)**: When the event the process was waiting for occurs (e.g., I/O completion), the process returns to the ready state.
- **Running → Terminated (Exit)**: The process completes its execution or is terminated by the operating system.

Scheduler & Dispatcher

The CPU Scheduler decides which process should run on the CPU and sometimes also for how long. The Dispatcher is the module that is responsible for implementing the CPU scheduler's decisions by performing the context switch.

Looking at the diagram, we can observe a snapshot of the system's state at a particular moment:

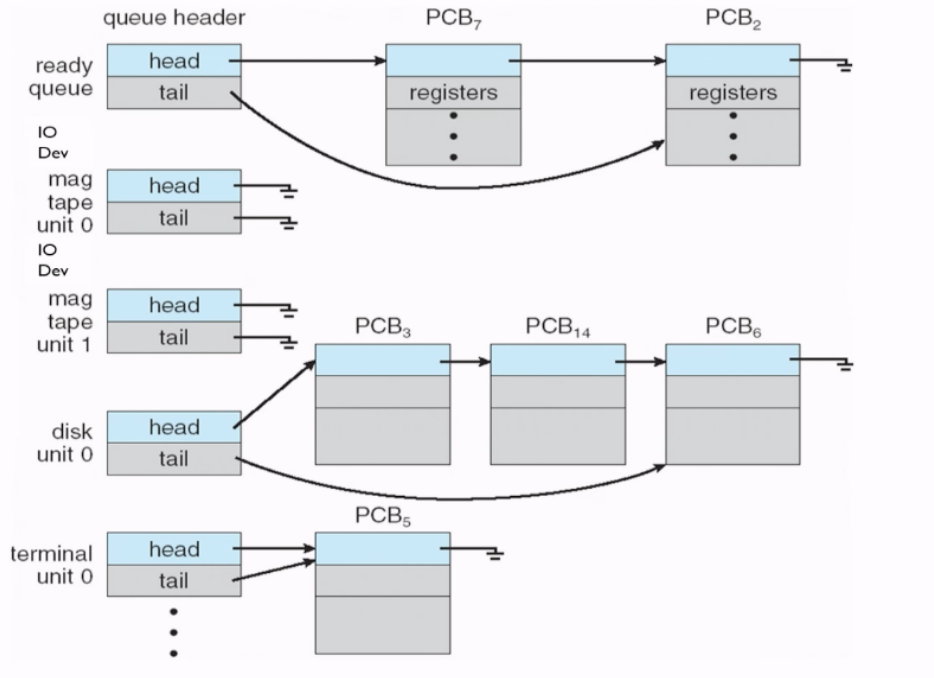

Figure 14: Ready queue and various I/O device queues

The diagram shows how processes are organized in different queues:

- **Ready Queue**: At the top, we see the ready queue containing PCBs (Process Control Blocks) of processes that are ready to execute. When the scheduler needs to select a process, it chooses from this queue.
- **I/O Device Queues**: Below the ready queue are several device-specific queues:
  - Magnetic tape unit queues (units 0 and 1)
  - Disk unit queue
  - Terminal unit queue

Each PCB represents a process in the system. When a process requests I/O from a specific device, its PCB is moved from the ready queue to the corresponding device queue, where it remains blocked until the I/O operation completes. When the device generates an interrupt signaling completion, the OS moves the process's PCB back to the ready queue.

The diagram illustrates how the OS manages multiple queues to efficiently handle processes in different states. By separating blocked processes into device-specific queues, the system can quickly identify which processes are waiting for particular resources and move them appropriately when those resources become available.

Context Switching Between Processes

A context switch is the process of saving the state of a currently executing process and restoring the state of a different process for execution. This operation is necessary for

multitasking operating systems.

When the OS decides to do a context switch, it needs to:

- Save the processor context of the current process, including the program counter and other registers
- Update the PCB with the new state and accounting information (CPU time used, time when process started, etc.)
- Move the PCB to the appropriate queue: ready, blocked/waiting, etc.
- Select another process for execution, update its PCB
- Update memory management data structures
- Restore the context of the selected process

When to Switch a Process

- Interrupts:
  1. From the clock (process has executed a full time slice)
  2. I/O completion
- Memory fault
- System call
- Exception

An Example of Process Creation Using fork()

Let us examine how a process can create another process using the `fork()` system call:

- The `fork()` system call creates a new process
- Child and parent processes are identical copies but execute different paths
- The processes are differentiated by the return value of `fork()`:
  - Parent receives child's PID (a positive integer)
  - Child receives 0
  - Negative value indicates an error

```
int pid;
int status = 0;
pid = fork();
if (pid != 0)
{
```

```
   /* parent */
   ....
}
else
{
   /* child */
   ....
}
```

For a concrete example, consider the following code:

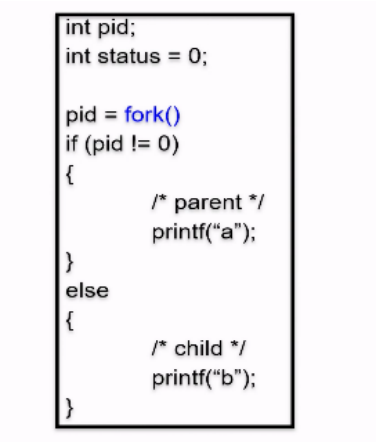

Figure 15: Using fork()

In this example, we want the parent to print a and the child to print b. Initially, we have a single process running on the CPU:

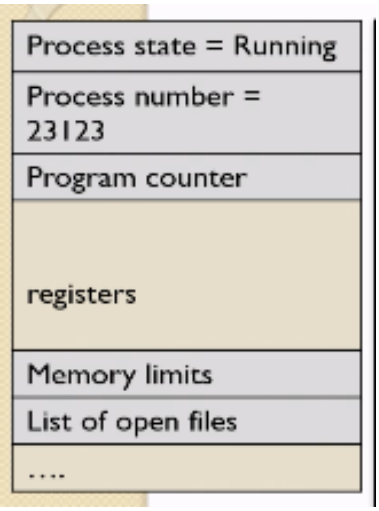

Figure 16: Original process before fork()

After calling `fork()`, which is a system call, an interrupt occurs that changes the process state from Running to Ready.

The OS then creates a copy that is identical to the original process except for the process number:

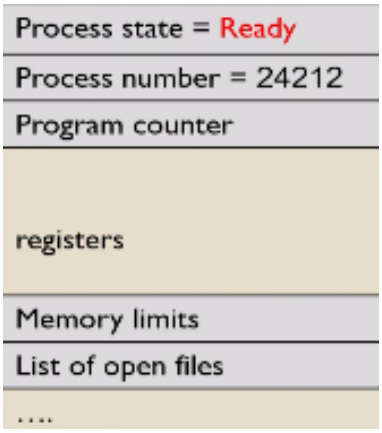

Figure 17: Duplicated process

Both the parent and child processes are now in the Ready state. The child process receives its own distinct memory space, separate from the parent's memory:

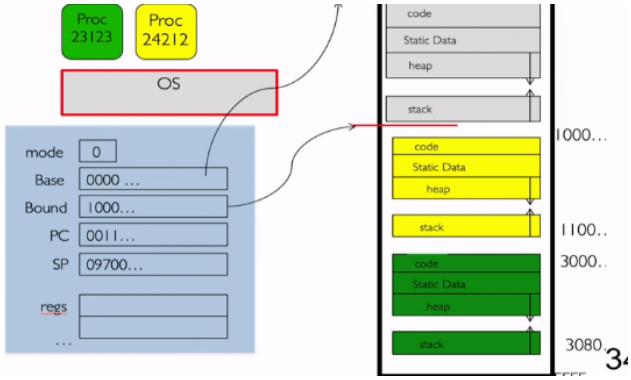

Figure 18: The processes have distinct memory

When `fork()` returns, different values are stored in the EAX register of each process: the child process receives 0, and the parent process receives the child's PID.

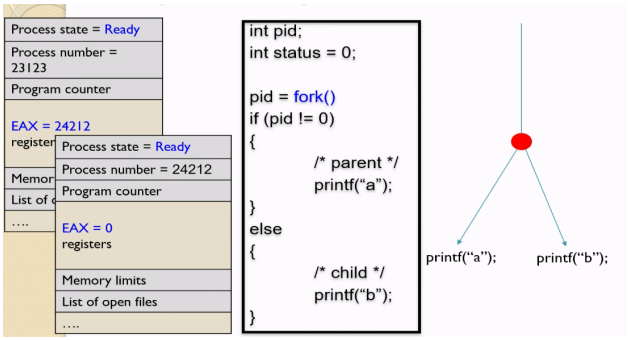

Figure 19: PID return values

Communication between Processes

Sometimes processes need to cooperate with each other: for example, when they share information or break large tasks into multiple subtasks. Yet we want to protect the processes from each other. For this purpose, inter-process communication (IPC) is provided by the operating system through specific system calls. The disadvantage is that IPC comes with very high overhead.

There are multiple ways to achieve this communication. For instance, we can designate a segment in memory that is shared and accessible to all processes. This can be implemented using files through the file system, where one process writes and another reads. We can also exchange messages through communication channels, such as the socket interface. These approaches are generally classified into either shared-memory or message-passing systems.

Threads

Threads are essentially paths of execution. Until now, each process we've discussed had one thread of control, defined by its program counter (PC) and stack.

We would like to implement multithreading: multiple independent execution paths that allow different parts of the same code to execute concurrently while sharing context (memory contents, open files, etc.). The motivation here is that, in contrast to IPC, communication between threads is much more efficient and less costly.

Examples of Concurrent Tasks

Many applications need to perform more than one task simultaneously. For example:

- A web browser needs to retrieve data from the network while displaying images of data already retrieved.
- A word processor needs to process every keystroke while displaying graphics and performing spell checking in the background.
- A web server processes many requests while listening for new connections.

With and Without Concurrency

Imagine a scenario without concurrency: a word processor would wait for a keystroke, and the OS would remove it from the CPU. No other tasks like spell checking could be performed until the keystroke is processed after a keyboard interrupt.

One might think we could run each task as a separate process, but this approach has significant costs: extensive IPC would be required, which is costly and cumbersome. Additionally, the program code would need to be duplicated for each process.

Multi-Threaded Processes

To address these issues, we introduce multi-threaded processes. In this model, each thread maintains its own registers (including PC) and stack, but threads share the same code, data, and files. The heap is shared among all threads in a process.

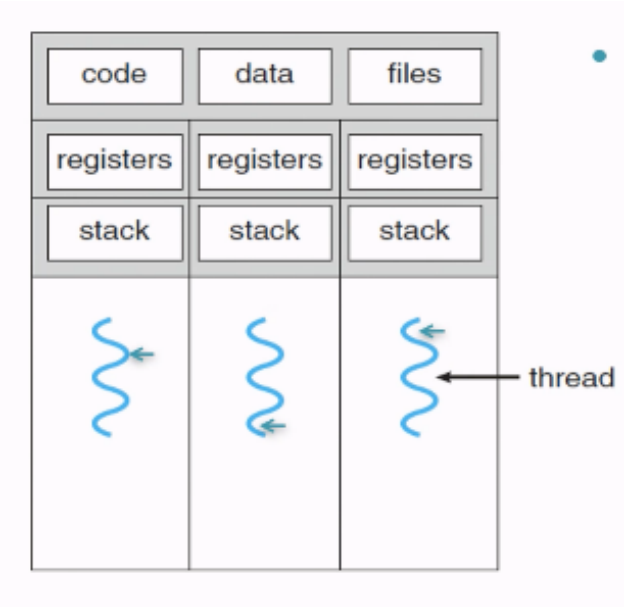

Figure 20: Multi-threaded processes

From our perspective, a process virtualizes the computer, allowing each running application to see an abstract computer dedicated only to itself. Similarly, a thread virtualizes the processor core, enabling an application to be structured as if it will run on multiple cores simultaneously. This approach combines the protection benefits of separate processes with the efficiency of shared memory access, making it ideal for concurrent programming in modern applications.

Example: Multi-Threaded Web Server

When a client connects to the server, the listener thread in the front-end accepts a connection request from a remote client on port 80 and immediately creates a worker thread to handle this request. This design is critical: if the listener were responsible for processing the entire request, it would be occupied and unable to accept new connections during that time, creating a bottleneck in server performance.

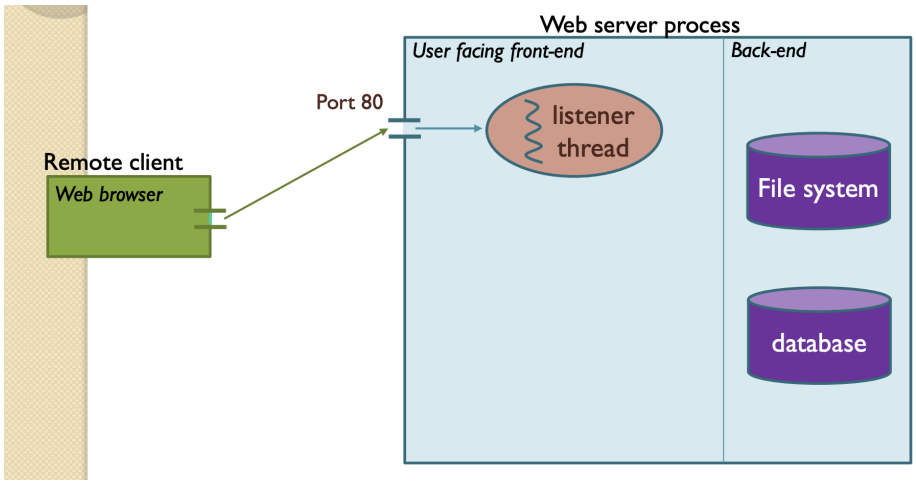

Figure 21: When a client connects

Once created, the worker thread independently handles the client request, communicating with back-end resources (file system and database) as needed. Meanwhile, the listener thread remains available to accept new connections from other clients. This separation ensures that request processing (often time-consuming for database operations) does not block the server's ability to accept new connections.

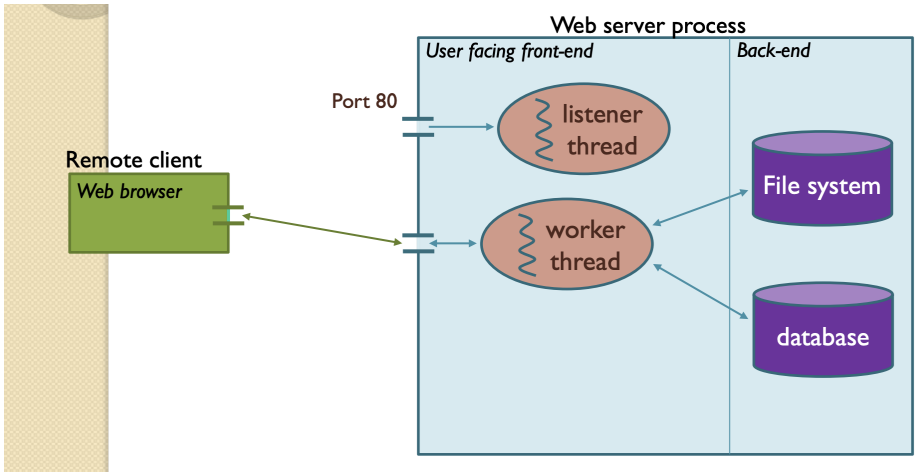

Figure 22: The worker thread handles the request

Here we see the full advantage of multi-threading: three different clients (A, B, and C) connect simultaneously, each served by its own worker thread. All three worker threads operate in parallel, each accessing necessary back-end resources independently. The listener thread continues to accept new connections concurrently. Without this architecture, clients would be served sequentially, significantly reducing server throughput and responsiveness under load.

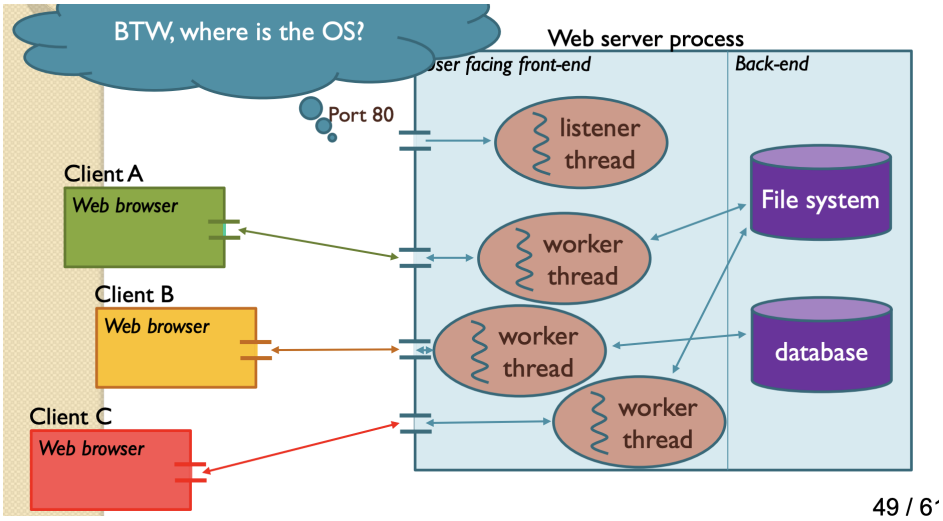

Figure 23: More worker threads

Kernel-Level Threads

- Kernel manages context information for the process and its threads.
- Blocking one thread does not necessarily block the entire process (like waiting for disk I/O or network data), the kernel knows that only that specific thread is blocked.
- **Disadvantage:** Switching between threads requires kernel involvement.
- Terminology:
  - *Kernel-level threads (Application):* Managed by kernel, run in **user space**. Used for application structuring.
  - *Kernel threads (OS):* Managed by kernel, run in **kernel space**. Used for OS internal structuring.

Implementing Kernel-Level Threads

The figure illustrates the Process Control Block (PCB), a fundamental data structure in operating systems that maintains process information:

- The PCB contains essential components:
  - **State**: Current execution state of the process
  - **Memory management**: Pointers to text, data, and heap segments
  - **Files**: File descriptors for open files
  - **User**: Process owner information
  - **Threads**: Thread control information

- Each thread descriptor maintains:
  - Thread execution state
  - Stack pointer
  - Accounting information
  - Priority level
  - CPU register storage
- The diagram shows the mapping between kernel space (PCB structure) and user space (memory segments and thread stacks).
- Multiple threads within a process share common resources (text, data, heap) while maintaining independent execution stacks.

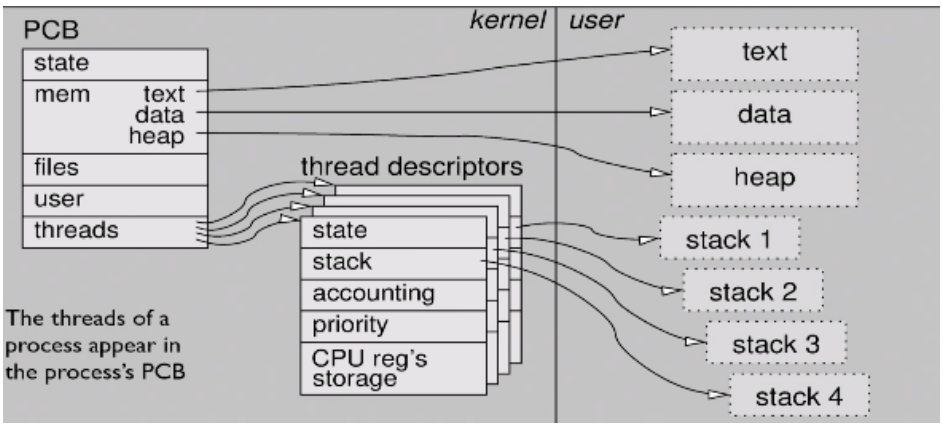

Figure 24: PCB Structure

Thread Control Block (TCB)

Threads are managed within the process context. The Thread Control Block (TCB) is the data structure that maintains thread-specific information:

- **Thread ID**: A unique identifier for each thread within the process
- **Thread state**: The current execution state (running, ready, blocked, etc.)
- **Pointer to PCB**: Since threads exist within the context of a process, each TCB maintains a reference to its parent PCB
- **Accounting information**: Resource usage statistics for this specific thread
- **Priority**: Scheduling priority value for the thread
- **Storage space for CPU state**: Registers and execution context specific to this thread

While the PCB maintains process-wide resources like memory segments (text, data, heap) and file descriptors, the TCB handles thread-specific execution contexts. This separation is critical because:

- Each process has at least one thread (the main thread)
- CPU state (registers, program counter, etc.) is stored per thread, not per process
- The scheduler operates at the thread level, determining which thread gets CPU time
- The TCB conceptually points back to its parent PCB, allowing the OS to track which process a thread belongs to

User-Level Threads

All thread management is done by the user app/library (in user mode). The kernel is not aware of the existence of the threads and thread switching does not require kernel mode privileges. The scheduling will be application specific.

This presents one significant disadvantage:

System calls (including I/O) by threads block the process: imagine that we have 3 threads of a process managed by the kernel, the kernel can run other threads when one is blocked. But with user-level threads, the OS is not aware that you have more than 1 thread. When we have I/O, the OS will block the process, therefore a single thread can monopolize the time slice thus starving the other threads within the task.

Summary: Benefits of Threads over Processes

- 30-100 times faster to create a new thread than a process
- Less time to terminate a thread than a process
- 5 times faster to switch between 2 threads within the same process
- Threads within the same process share memory and files, therefore they can communicate without invoking the kernel (IPC)

It's important to note that the CPU itself is not aware of the existence of either user or kernel level threads. The CPU was designed long before complex operating systems, providing the fundamental execution capabilities upon which threading models are built.

	Processes	**Kernel-level threads**	**User-level threads**
Protection	Protected from each other, require operating system to communicate	Share address space, simple communication, useful for application structuring	Share address space, simple communication, useful for application structuring
Overhead	High overhead: all operations require a kernel trap, significant work	Medium overhead: operations require a kernel trap, but little work	Low overhead: everything is done at user level
Blocking	Independent: if one blocks, this does not affect the others	Independent: if one blocks, this does not affect the others	If a thread blocks, the whole process is blocked
Parallelism	Can run in parallel on different processors in a multiprocessor system	Can run in parallel on different processors in a multiprocessor system	All share the same processor so only one runs at a time
API/ Portability	System-specific API, programs are not portable	System specific API, programs are not portable	The same thread library may be available on several systems
Flexibility	One size fits all	One size fits all	Application-specific thread management is possible

Table 1: Comparison of Processes, Kernel-level Threads, and User-level Threads