Assignment1

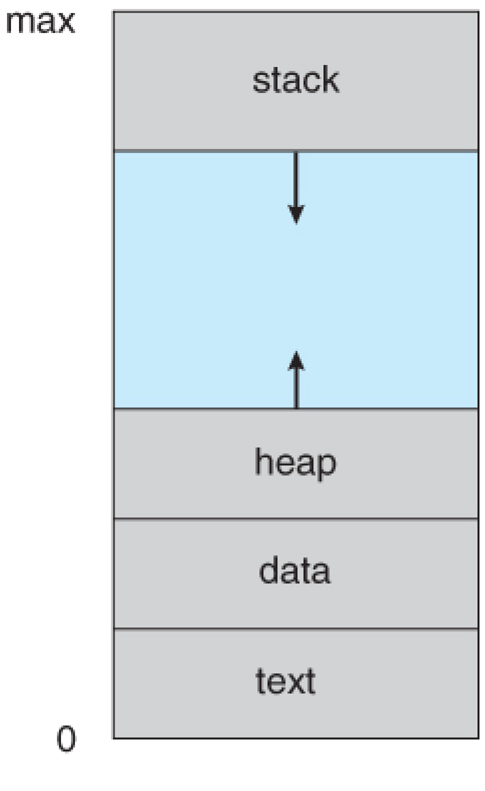
Process

A process is a running program on your computer.
When you run any program (like Chrome, Word, or a C program), your operating system creates a process for it.
Each process is an independent "unit of work" that the operating system manages.

What Happens When a Process is Created?

Every process gets its own memory space, which is protected from other processes. This space is divided into sections:

- Code (text): The program's compiled instructions.
- Data: Global/static variables.
- Heap: Dynamically allocated memory.
- Stack: Local variables and function calls.



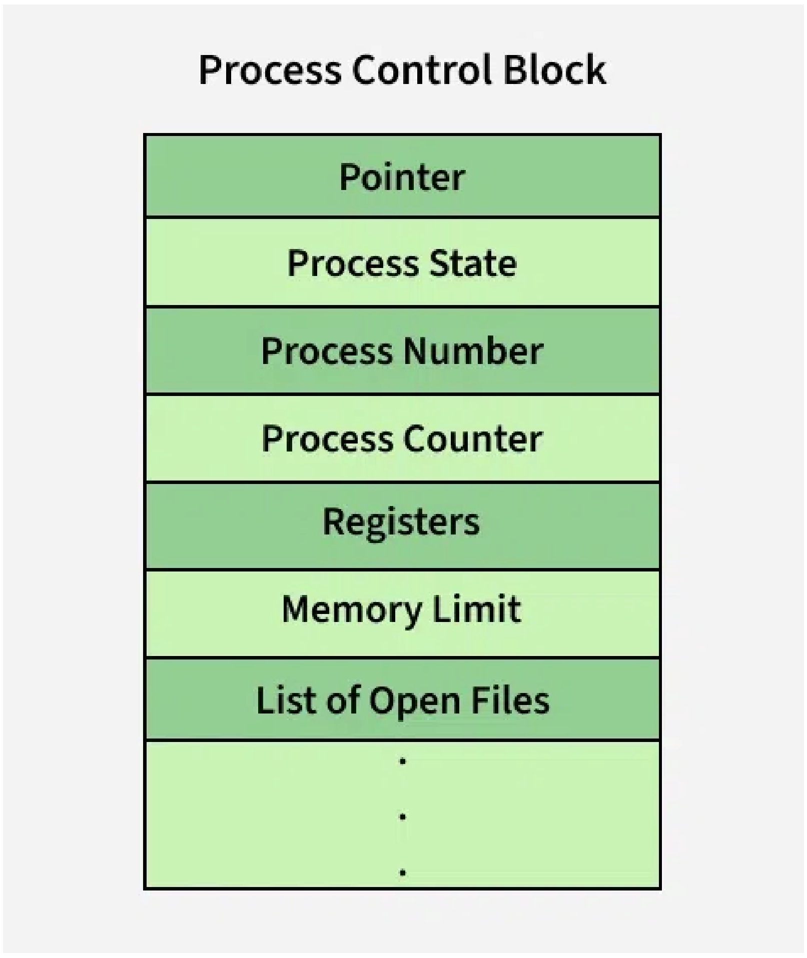
When a new process is created, the os not only gives it its own memory space, but also creates a Process Control Block (PCB) for that process.

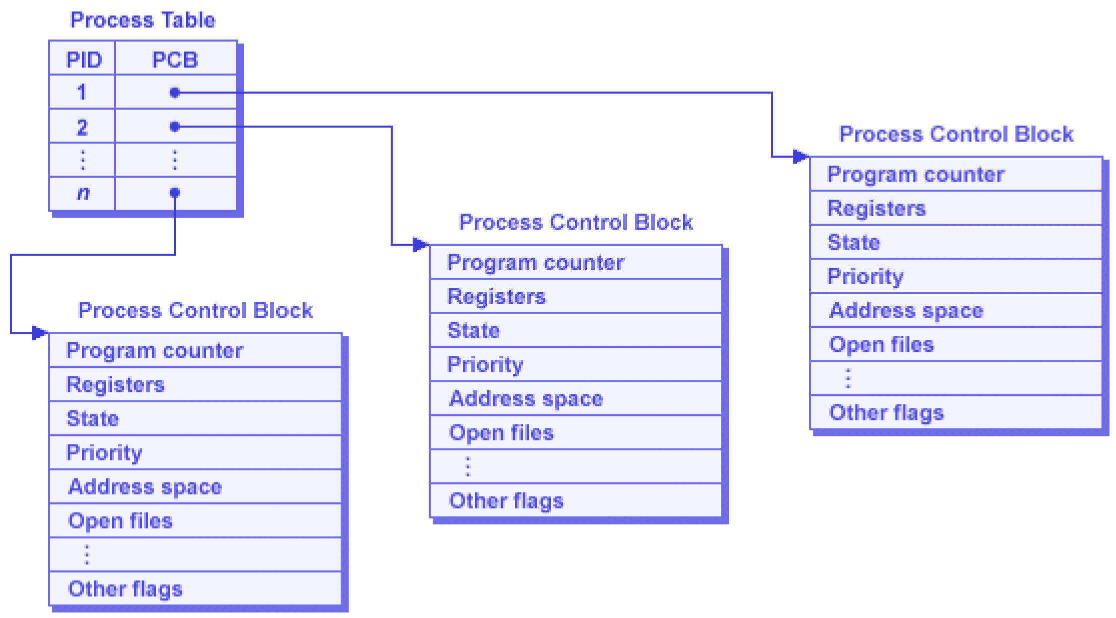
- The PCB is a special data structure the operating system uses to keep track of each process.
- The PCB is stored and managed by the OS in the kernel space.
- The PCB stores everything the OS needs to manage the process, such as:

Field	What It Means
PID	Process ID: unique number for this process

Field	What It Means
Process State	Running, waiting, stopped, etc.
Program Counter (PC)	The address of the next instruction to execute
Memory Information	Pointers to the process's code, data, heap, stack
Open Files	List of files the process has open
Parent PID (PPID)	PID of the parent process

...





Fork

Fork() is a function in C (on Unix/Linux systems) that creates a new process.
After calling fork(), two processes are running concurrently your program:

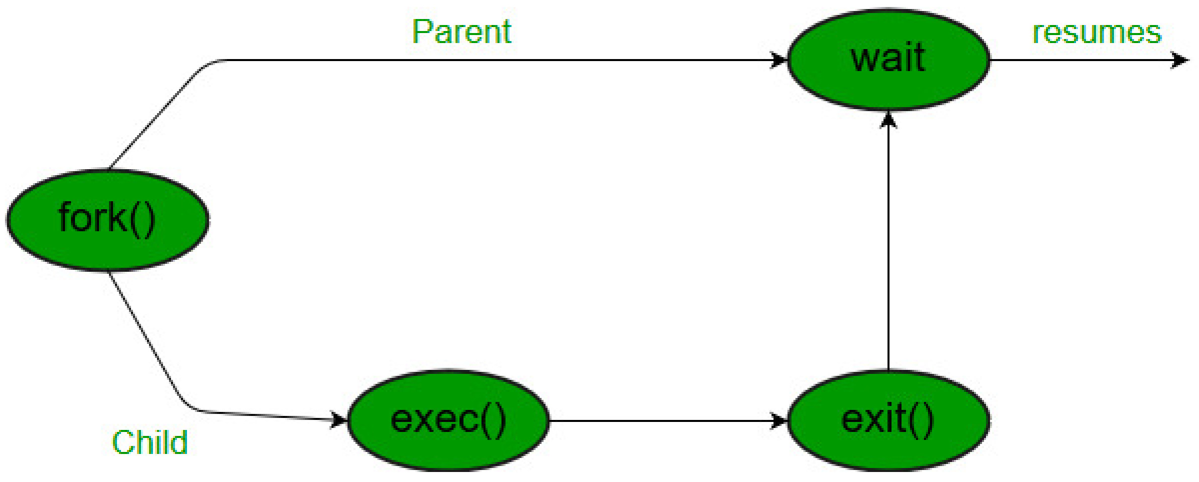
- The parent (the original process)
- The child (the new process created by fork()).

The child process uses the same pc(program counter), same CPU registers, and same open files which use in the parent process. It takes no parameters and returns an integer value.

How does it work?

When you call fork(), here’s what happens: Both parent and child run the code after the fork() line. Fork() returns a value:

- In the child process: it returns 0
- In the parent process: it returns the process ID (PID) of the child
- If something goes wrong: it returns -1



```c
pid_t pid = fork();
if (pid < 0) {
    perror("fork failed");
    exit(1);
}
if (pid == 0) {
    // This is the child
    printf("I am the child, my pid is %d\n", getpid());
    _exit(0);
} else {
    // This is the parent
    printf("I created a child with pid %d\n", pid);
    wait(NULL);
}
```

Status in fork()

Returned by wait(&status) or waitpid(&status) in the parent.
An integer that encodes how the child terminated.

```
   15                       8  7  6                             0
   +-----------------------+---+-----------------------------+
   | Exit code (if normal) | C | Signal number (if signaled) |
   +-----------------------+---+-----------------------------+
```

How to read it?

- WIFEXITED(status) → true if exited normally
- WEXITSTATUS(status) → extracts high byte (exit code)
- WIFSIGNALED(status) → true if killed by signal
- WTERMSIG(status) → extracts signal number

```c
int status;
pid_t done = wait(&status);

if (WIFEXITED(status)) {
    printf("Child exit code = %d\n", WEXITSTATUS(status));
} else if (WIFSIGNALED(status)) {
    printf("Child killed by signal %d\n", WTERMSIG(status));
}
```

IPC

Inter-Process Communication (IPC) is how two or more processes (programs running at the same time) exchange data or messages with each other.
Since each process has its own private memory (for safety and security), they cannot directly access each

other's variables.
IPC provides methods for processes to communicate, share data, or synchronize their actions.
Many real-world applications need processes to work together.

IPC Method	Description	Example Use
Pipe	Data flows in one direction (parent to child or vice versa)	Send output of one program to another
Shared Memory	Multiple processes read/write to a common memory region	Fast communication, like databases
Semaphore	Used to synchronize processes (not for sending data)	Preventing two processes from using a resource at the same time
Signals	Used to notify a process that some event has happened	Tell a process to stop or reload
Message Queue	Processes put messages in a queue to be read by others	Printing tasks in order
Socket	Processes (even on different machines) can send data back and forth	Chat applications, web servers

Signals

Signals are software interrupts sent to a process to notify it that an event has occurred.
The OS or users can send signals. Common signals include:

- SIGKILL:
  - forcefully stop a process
  - You want to be sure a process stops right now, even if it's stuck.
- SIGTERM
  - politely ask a process to terminate
  - You want to nicely stop a process, giving it a chance to clean up.
- SIGINT
  - interrupt (like Ctrl+C in terminal)
  - User wants to interrupt a program.

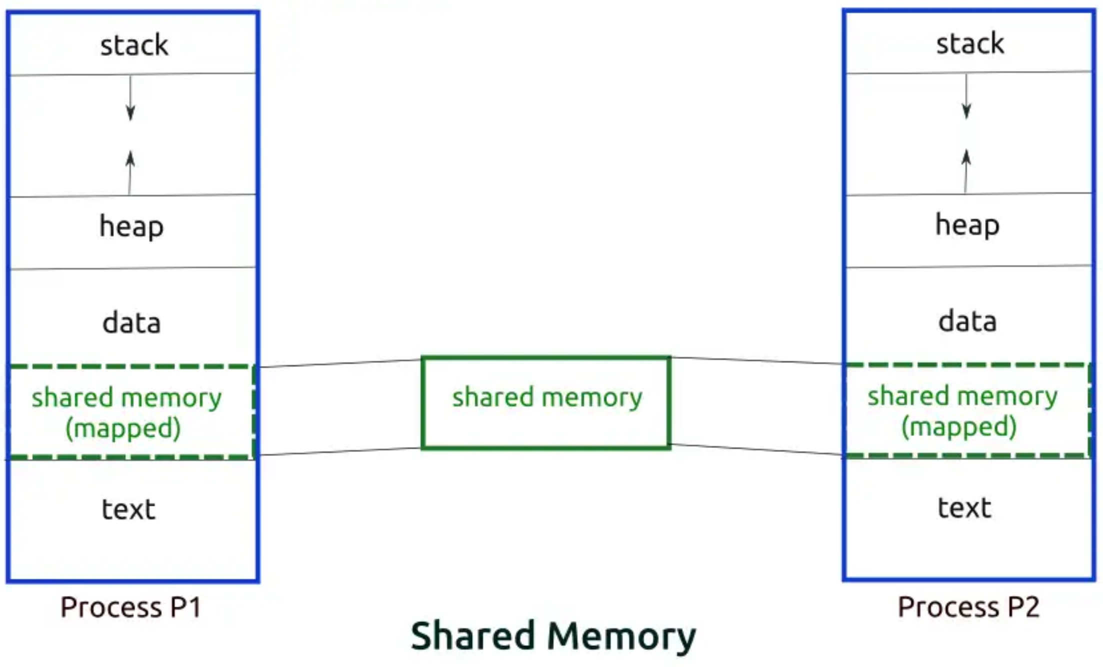
```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();
    if (pid < 0) {
        perror("fork failed");
        return 1;
    }
```

```c
    if (pid == 0) {
        // Child: Run forever
        while (1) {
            printf("Child alive! PID=%d\n", getpid());
            sleep(1);
        }
    } else {
        // Parent: Wait and then kill child
        sleep(3);  // Let child run for a bit
        printf("Parent: Sending SIGKILL to child %d\n", pid);
        kill(pid, SIGKILL); // Sends signal to kill the child
        wait(NULL); // Wait for child to terminate
        printf("Parent: Child killed.\n");
    }
    return 0;
}
```

Shared Memory

Shared memory is a method of Inter-Process Communication (IPC).
It allows multiple processes to access the same region of memory.
This is the fastest IPC because processes can read and write directly to shared data (instead of sending messages or using files).
Used when you want to share data (like variables or arrays) between related processes (for example, a parent and its children created by fork()).



`shmget()`

Creates a new shared memory segment or accesses an existing one.

```c
int shmget(key_t key, size_t size, int shmflg);
```

Parameters:

- key:
  - A unique key (identifier) is required for creating or accessing a shared memory segment.
  - This key is often generated using ftok(), or you can use IPC_PRIVATE for creating a private segment.
    - IPC_PRIVATE (which has a value of 0) is a special key for IPC system calls. When used with shmget(), it always creates a new, unique shared memory segment that is only accessible by related processes (such as parent and child).
    - ftok() (“file to key”) is a function that generates a unique numeric key for shared memory based on a filename and a project identifier.
- size:
  - The size (in bytes) of the shared memory segment.
- shmflg:
  - Flags and permissions.
  - Examples:
    - IPC_CREAT: Tells the operating system to create the shared memory segment if it does not already exist. If it’s already there, the OS just returns its ID.
    - IPC_EXCL: Fail (return -1) if segment already exists (used with IPC_CREAT).
    - Permissions: Sets who can read or write the shared memory segment (just like file permissions). The three digits mean permissions for owner, group, and others.
      - 6 = read + write (4 + 2)
      - 4 = read only
      - 0 = no access
      - 0666: Anyone can read and write.
      - 0600: Only the owner can read and write.

Returns:

- On success: Returns a shared memory ID (int), used in later calls.
- On failure: Returns -1, sets errno.

```
// key_t key = ftok(const char *pathname, int proj_id);
// pathname: Path to an existing file
// proj_id: A single character (used to make the key more unique).

key_t key = ftok("/tmp/myfile", 'A');
int shmid = shmget(key, 100, IPC_CREAT | 0666);
// Both programs using ftok("/tmp/myfile", 'A') will get the same key!

// IPC_PRIVATE = 0 just means "create a new, unique segment" every time, and only
processes that inherit it (like children) can access it.
int shmid = shmget(IPC_PRIVATE, size, IPC_CREAT | 0666);
```

`shmat()`

Attaches the shared memory segment to your process's address space, giving you a pointer you can use. shmget() only reserves the memory in the system and gives you a shared memory ID number (called the "shmid"). It does not give you a pointer or direct access to the memory yet.

```
// void *shmat(int shmid, const void *shmaddr, int shmflg);

void *ptr = shmat(shmid, NULL, 0);
// Now ptr points to the shared memory area.
```

Parameters:

- shmid: The ID returned by shmget().
- shmaddr: Address where you want to attach the memory. Usually set to NULL (let the OS choose).
- shmflg: Attachment options. Usually 0.

Returns:

- On success: Returns a pointer to the shared memory.
- On failure: Returns (void *) -1, sets errno.

`shmdt()`

Detaches the shared memory segment from your process (stops using it).

```
// int shmdt(const void *shmaddr);

shmdt(ptr); // Detach the shared memory from your process
```

Parameters:

- shmaddr: Pointer returned by shmat() (the address you want to detach).

Returns:

- On success: Returns 0.
- On failure: Returns -1, sets errno.

`shmctl()`

Performs control operations on the shared memory segment (like removing it, getting status, etc.).

```
// int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

```
shmctl(shmid, IPC_RMID, NULL);
// Removes the shared memory segment from the system.
```

Parameters:

- shmid: The shared memory ID from shmget().
- cmd: What you want to do. Common commands:
    - IPC_RMID: Remove (delete) the shared memory segment.
    - IPC_STAT: Get info about the segment.
    - IPC_SET: Change the permissions, etc.
- buf: Used with IPC_STAT or IPC_SET for info/changes (can be NULL for IPC_RMID).

Returns:

- On success: Returns 0 (or sometimes data, depending on the command).
- On failure: Returns -1, sets errno.

Chunking Strategy

Example:

```
./prime 10 39 4
```

Range Formula

$$\text{Range} = \frac{\text{Upper} - \text{Lower} + 1}{N}$$

$$\frac{39 - 10 + 1}{4} = \frac{30}{4} = 7.5 \quad \Rightarrow \quad \text{chunk size} = 7$$

Child Ranges (slices)

- Child 0 $\rightarrow$ [10 ... 16]
- Child 1 $\rightarrow$ [17 ... 23]
- Child 2 $\rightarrow$ [24 ... 30]
- Child 3 $\rightarrow$ [31 ... 39] --> Remainder(Upper)

Shared Memory Layout
Total size = N * MAX_PRIMES_PER_CHILD
Each child writes to its own block:

```
Child 0 → indices [0 … MAX_PRIMES_PER_CHILD - 1]
Child 1 → indices [MAX_PRIMES_PER_CHILD … 2*MAX_PRIMES_PER_CHILD - 1]
Child 2 → indices [2*MAX_PRIMES_PER_CHILD … 3*MAX_PRIMES_PER_CHILD - 1]
Child 3 → indices [3*MAX_PRIMES_PER_CHILD … 4*MAX_PRIMES_PER_CHILD - 1]
```