Parent & Child Processes — Memory Segments (Study Notes)

1. Introduction to Processes

A **process** is a program in execution. In Linux/UNIX, new processes are created using the `fork()` system call.

When a process calls `fork()`:

- A **new Child Process** is created
- The original process becomes the **Parent Process**
- Both start execution from the **next instruction after** `fork()`

Both processes run the **same program (same code base)** but have:

- different Process IDs (PIDs)
- separate memory spaces

---

2. Memory Layout of a Process

A running program is divided into five major memory segments:

1. **Text / Code Segment** — program instructions
2. **Data Segment** — initialized global & static variables
3. **BSS Segment** — uninitialized global & static variables
4. **Heap** — dynamic memory (`malloc`, `calloc`, `realloc`)
5. **Stack** — local variables, parameters, and function frames

---

3. Behaviour of Memory Segments After `fork()`

When `fork()` is executed:

- The child process receives a **copy of the parent's memory**
- Linux applies **Copy-On-Write (COW)**

Meaning:

> Memory pages are shared initially and are duplicated **only when modified**.

This improves performance and reduces memory usage.

3.1 Text / Code Segment

Stores program instructions and constant string literals.

Characteristics:

- Read-only
- Same for Parent and Child
- Not duplicated
- Copy-On-Write not required

Both processes execute the **same code**, but take different paths depending on the `fork()` return value.

3.2 Data Segment (Initialized Data)

Contains:

- global variables with values
- initialized static variables

Behaviour after `fork()`:

- logically copied to the child
- initially shared using COW
- becomes **separate when modified**

Parent and Child do **not** affect each other's values after modification.

3.3 BSS Segment (Uninitialized Data)

Contains:

- global/static variables without initialization

Behaviour:

- initially appears identical in both processes
- separated when changed

Same Copy-On-Write rules as the Data segment.

3.4 Heap Segment

Used for dynamic memory allocation via:

- `malloc()`
- `calloc()`
- `realloc()`

Behaviour after `fork()`:

- content is logically copied
- initially shared through COW
- becomes **independent when either process modifies it**

So the Parent heap and Child heap eventually become different.

3.5 Stack Segment

Stores:

- local variables
- function arguments
- return addresses
- stack frames

Behaviour after `fork()`:

- child receives a copy of parent stack
- appears same initially
- becomes separate when modified

Each process maintains its **own stack frames** during execution.

---

4. Parent Process vs Child Process

Parent Process

- The process that calls `fork()`
- Continues execution after fork
- Receives **child PID (> 0)** as return value

Child Process

- Newly created process
- Starts execution from the same instruction

- Receives **0** as return value from `fork()`

---

5. Process ID (PID) Behaviour

Process	`fork()` Return Value
Parent	Child PID (> 0)
Child	0
Failure	-1

Each process has its **own PID**.

The child process also stores the **Parent PID (PPID)**.

---

6. File Descriptors After `fork()`

File descriptors are **shared** between parent and child:

Examples:

- open files
- pipes
- sockets

They share the **same file offset**, meaning:

- both can read/write to the same file
- the OS synchronizes access

---

7. Copy-On-Write (COW) — Key Concept

Instead of copying full memory immediately:

- Parent and Child share the same physical memory pages
- The OS copies a page **only when one process modifies it**

Advantages:

- saves RAM
- faster process creation
- improves performance

8. Summary Table

Memory Segment	Parent & Child Behaviour
Text / Code	Shared (read-only)
Data	Copy-on-write — separates when modified
BSS	Copy-on-write — separates when modified
Heap	Copy-on-write — separates when modified
Stack	Copy-on-write — separates when modified
File Descriptors	Shared file handle & offset
PID	Different for each process

---

9. Final Key Points

- Parent and Child run the **same program code**
- Execution starts from the same point after `fork()`
- They have **separate logical address spaces**
- Linux optimizes memory using **Copy-On-Write**
- Data, Heap, and Stack diverge when modified
- Code segment remains shared
- File descriptors are shared
- PIDs are different

> These concepts are very important for Linux internals, OS fundamentals, and C programming interviews.

If you want, we can extend these notes with:

- Zombie & Orphan processes
- `exec()` family behaviour
- Inter-Process Communication (IPC)
- Real program examples & diagrams

Tell me what to add next 👍