**Assignment -2 Advance C Training**

**1. Child Process – fork()**

The fork() system call in Unix/Linux is used to create a new process, called a child process, by duplicating the current process (the parent).

* Syntax: pid\_t pid = fork();
* Return Value:
  + 0 → inside the child process
  + Positive PID → inside the parent process (PID of the child)
  + -1 → fork failed

//sample code

pid\_t pid = fork();

if (pid == 0) {

printf("Child process\n");

} else if (pid > 0) {

printf("Parent process\n");

} else {

perror("fork failed");

}

**2. Handling Common Signals**

Signals are asynchronous notifications sent to a process to notify it of events like termination requests, segmentation faults, etc.

**Common Signals:**

* SIGINT – Interrupt (e.g., Ctrl+C)
* SIGTERM – Termination request
* SIGKILL – Kill signal (cannot be caught)
* SIGSEGV – Segmentation fault
* SIGCHLD – Child process stopped or terminated

**Handling:**

Use signal() or sigaction() to define custom handlers.

//sample code

#include <signal.h>

void handle\_sigint(int sig) {

printf("Caught signal %d\n", sig);

}

signal(SIGINT, handle\_sigint);

**3. Exploring Different Kernel Crashes**

Kernel crashes or **panics** are critical failures in the OS kernel. They often lead to system reboot or hang.

**Common Causes:**

* Null pointer dereference
* Buffer overflows
* Illegal memory access
* Deadlocks
* Misbehaving device drivers

**Debugging Tools:**

* **dmesg** – Print kernel messages
* **/var/log/kern.log** – Kernel log
* **kdump/crash** – Capture crash dumps for analysis
* **gdb + vmlinux** – Kernel debugging with symbols

**4. Time Complexity**

Time complexity describes the **computational complexity** of an algorithm in terms of input size n.

**Common Notations:**

* **O(1)** – Constant time
* **O(log n)** – Logarithmic (e.g., binary search)
* **O(n)** – Linear (e.g., array traversal)
* **O(n log n)** – Merge sort, quicksort (average case)
* **O(n²)** – Nested loops (e.g., bubble sort)

Used to evaluate performance and scalability of code, especially in competitive programming and systems optimization.

**5. Locking Mechanism – Mutex / Spinlock**

Locking mechanisms prevent **race conditions** in multithreaded environments by controlling access to shared resources.

**Mutex (Mutual Exclusion):**

* **Blocking lock**
* If locked, the thread sleeps until it gets the lock
* Suitable for user-space applications (e.g., pthread\_mutex\_t)

pthread\_mutex\_t lock;

pthread\_mutex\_lock(&lock);

// critical section

pthread\_mutex\_unlock(&lock);

**Spinlock:**

* **Busy-wait lock** (non-blocking)
* Continuously checks the lock variable
* Preferred in kernel or real-time applications where sleeping is costly

spin\_lock(&my\_spinlock);

// critical section

spin\_unlock(&my\_spinlock);

| **Feature** | **Mutex** | **Spinlock** |
| --- | --- | --- |
| Blocking | Yes | No |
| CPU Usage | Low (sleeps) | High (busy-waits) |
| Use case | User space | Kernel space, short critical sections |

Top of Form

Bottom of Form