# Child Process - fork()

The `fork()` system call in Unix-based systems is used to create a new process, known as the child process.   
The newly created child process is a copy of the parent process except for the returned value.  
The `fork()` call returns a value in both the parent and the child processes. In the parent, it returns the PID of the child; in the child, it returns 0.

**Key Features:**  
- Used for process creation.  
- Each process has its own separate memory space.  
- Useful in concurrent programming.  
- Combined with `exec()` family of functions for new program execution.

**Code Snippet:**  
  
pid\_t pid = fork();  
if (pid == 0) {  
 // Child process  
} else if (pid > 0) {  
 // Parent process  
} else {  
 // fork failed  
}

**Applications:**  
- Creating independent tasks.  
- Managing services like web servers.

# Handling Common Signals

Signals are asynchronous notifications sent to a process to notify it of various events like illegal memory access or termination requests.

**Common Signals:**  
- `SIGINT` – Interrupt from keyboard (Ctrl+C)  
- `SIGTERM` – Termination request  
- `SIGKILL` – Kill signal  
- `SIGSEGV` – Invalid memory reference  
- `SIGCHLD` – Child process stopped or terminated

**Handling Signals:**  
Signals can be handled using the `signal()` function in C.

#include <signal.h>  
#include <stdio.h>

void handler(int signum) {  
 printf("Caught signal %d\n", signum);  
}

int main() {  
 signal(SIGINT, handler);  
 while (1);  
}

**Importance:**  
- Ensures graceful shutdown.  
- Enables process cleanup.  
- Enhances robustness in concurrent environments.

# Exploring Different Kernel Crashes

Kernel crashes, often referred to as panics, occur when the operating system encounters a critical error it cannot recover from.

**Causes:**  
- Null pointer dereferencing in kernel space.  
- Invalid memory access.  
- Race conditions or deadlocks.  
- Improper hardware driver implementations.

**Tools:**  
- `dmesg`: Displays kernel ring buffer messages.  
- `kdump`: Captures kernel dumps on crash.  
- Kernel debugging tools like GDB with `crash` utility.

**Prevention and Handling:**  
- Write robust kernel modules.  
- Use mutex/spinlocks carefully.  
- Follow best practices in memory management.

**Real-World Scenarios:**  
- Device driver development.  
- Embedded systems running real-time OS.

# Locking Mechanism - Mutex / Spinlock

Locks are used to protect shared data from being concurrently accessed by multiple threads.

**Mutex (Mutual Exclusion):**  
- Blocks the thread if the lock is unavailable.  
- Suitable for general-purpose locking.  
- Used with `pthread\_mutex\_lock()` in C.

**Spinlock:**  
- Continuously checks lock availability.  
- Efficient for short locking durations in multicore systems.  
- Used in kernel-level synchronization.

**Code Example (Mutex):**  
  
pthread\_mutex\_t lock;  
pthread\_mutex\_lock(&lock);  
// critical section  
pthread\_mutex\_unlock(&lock);

**Key Differences:**

|  |  |  |
| --- | --- | --- |
| Feature | Mutex | Spinlock |
| Blocking | Yes | No |
| CPU Usage | Low | High (Busy Wait) |
| Use Case | User-space apps | Kernel/RTOS |

Proper usage of locks ensures thread safety and avoids data inconsistency in concurrent applications.