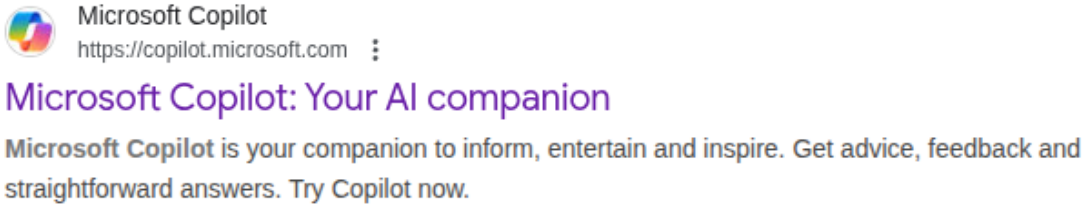
glibc Process Execution

20251104 Tue

I used Microsoft Copilot to prepare this document



linux exec family of functions

The Linux `exec` family of functions replaces the current process image with a new one. Unlike `fork()`, which creates a new process, `exec()` transforms the calling process into a different program.

🧠 Overview of `exec` Family

All `exec` functions are declared in `<unistd.h>` and share the same core behavior: they **load and run a new executable**, replacing the current process. If successful, they never return.

🔹 Common Members

Function	Description
`execl()`	Takes a variable list of arguments
`execv()`	Takes an array of arguments
`execle()`	Like `execl()` but allows environment specification
`execve()`	Like `execv()` but allows environment specification
`execlp()`	Like `execl()` but searches `PATH`
`execvp()`	Like `execv()` but searches `PATH`

📦 Function Signatures

```c
int execl(const char *path, const char *arg0, ..., NULL);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg0, ..., NULL, char *const envp[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char *arg0, ..., NULL);
int execvp(const char *file, char *const argv[]);
```

- `arg0` is conventionally the name of the program.
- `envp[]` is an array of environment variables.

🧪 Example: Using `execl()`

```c
#include <unistd.h>

int main() {
    execl("/bin/ls", "ls", "-l", NULL);
    // If exec fails, this line will execute
    perror("execl failed");
    return 1;
}
```

- This replaces the current process with `ls -l`.
- If successful, `main()` never returns.

🔍 Key Differences

Function	Uses `PATH`?	Accepts `argv[]`	Accepts `envp[]`
`execl`	No	No	No
`execv`	No	Yes	No
`execle`	No	No	Yes
`execve`	No	Yes	Yes
`execlp`	Yes	No	No
`execvp`	Yes	Yes	No

📚 Best Practices

- Use `fork()` before `exec()` if you want to keep the original process.
- Always check for errors — `exec()` only returns on failure.
- Prefer `execvp()` or `execlp()` when you want to search the executable in `PATH`.

For a deeper dive, check out Baeldung's guide on exec functions Baeldung or LinuxVox's comprehensive tutorial linuxvox.com.

why envp is used

The `envp` parameter in the `execve()` and related `exec` functions is used to **pass a custom environment** to the new process being executed. It allows you to define what environment variables the new process will see — independent of the current process's environment.

🧠 What Is `envp`?

- `envp` stands for **environment pointer**.
- It's an array of strings, each in the format:

```c
"KEY=value"
```

- It terminates with a `NULL` pointer.

🔧 Why Use `envp`?

1. Custom Environment

You can launch a process with a specific set of environment variables:

C

Copy

```c
char *envp[] = {
    "PATH=/custom/bin",
    "USER=guest",
    NULL
};
execve("/custom/bin/myapp", argv, envp);
```

2. Isolation

Useful for sandboxing or running processes with minimal or controlled environments.

3. Portability

You can ensure the new process has the exact environment it needs, regardless of the parent's state.

🧪 When Is `envp` Used?

Function	Uses `envp`?	Description
`execve()`	✅ Yes	Full control over environment
`execle()`	✅ Yes	Like `execl()` but with environment
`execv()`	❌ No	Inherits current environment
`execlp()`	❌ No	Searches `PATH`, inherits environment

📌 Summary

- `envp` lets you **explicitly define** the environment for a new process.
- It's essential for **controlled execution**, especially in system programming, scripting, and containerization.

on_exit

In Linux (and POSIX systems), `on_exit()` is a function used to register a callback that will be executed when a process terminates normally. It's part of the GNU C Library and is similar to `atexit()`, but provides more flexibility.

🧠 What Is `on_exit()`?

C

Copy

```c
int on_exit(void (*function)(int, void *), void *arg);
```

- `function`: A pointer to a function that takes an `int` (exit status) and a `void *` (user-defined argument).
- `arg`: A user-defined pointer passed to the callback.

The registered function is called when `exit()` is invoked, allowing cleanup or logging before the process ends.

🔧 How It Differs from `atexit()`

Feature	`atexit()`	`on_exit()`
Parameters	No arguments	Receives exit status and user data
Portability	Standard C (ANSI, POSIX)	GNU extension (not portable)
Use case	Simple cleanup	Context-aware cleanup or logging

🧪 Example Usage

C Copy

```c
#include <stdlib.h>
#include <stdio.h>

void cleanup(int status, void *arg) {
    printf("Exiting with status %d, message: %s\n", status, (char *)arg);
}

int main() {
    on_exit(cleanup, "Goodbye!");
    exit(42);
}
```

Output:

Code Copy

```
Exiting with status 42, message: Goodbye!
```

📌 Notes

- Only works with `exit()`, not `_exit()` or abnormal termination.
- Multiple `on_exit()` calls are allowed; they execute in reverse order of registration.
- Not portable across all systems — use `atexit()` for cross-platform code.

status arg of the call back function of on_exit argument function

The `status` argument in the callback function registered with `on_exit()` represents the **exit status** of the process — the same value passed to `exit(status)` when the process terminates.

🧠 What Does `status` Mean?

- It's an `int` value that indicates how the process ended.
- Conventionally:
  - `0` → success
  - Non-zero → error or specific exit code

This value is passed to the callback so you can log, analyze, or respond to how the process exited.

🧪 Example

C

Copy

```c
#include <stdlib.h>
#include <stdio.h>

void goodbye(int status, void *arg) {
    printf("Process exited with status %d\n", status);
    printf("Message: %s\n", (char *)arg);
}

int main() {
    on_exit(goodbye, "Cleanup complete");
    exit(42);
}
```

Output:

Code

Copy

```
Process exited with status 42
Message: Cleanup complete
```

📌 Summary

- `status` is the exit code passed to `exit()`.
- It helps you understand why the process ended.
- Useful for logging, diagnostics, or conditional cleanup.

shell

In Linux, a **shell** is a command-line interface that allows users to interact with the operating system by typing commands. It acts as a bridge between the user and the kernel, interpreting input and launching programs.

🧠 What Is a Shell?

- A **shell** is a user-level program that provides a text-based interface.
- It reads user input, parses commands, and executes them.
- It can run interactively (typing commands) or non-interactively (via scripts).

🔧 Common Shells in Linux

Shell	Description
Bash	Bourne Again SHell — most common, powerful scripting
Zsh	Z Shell — advanced features, popular with developers
Fish	Friendly Interactive SHell — user-friendly, auto-suggestions
Dash	Lightweight shell used in system scripts
Tcsh	C Shell with enhancements

🧪 Shell Capabilities

- **Command execution**: Run programs, scripts, and system utilities
- **Job control**: Manage foreground/background processes
- **Redirection**: Control input/output (`>`, `<`, `|`)
- **Variables**: Store and manipulate data (`$HOME`, `$PATH`)
- **Scripting**: Automate tasks with shell scripts
- **Environment management**: Set environment variables and session behavior

---

📚 Shell vs Terminal vs Console

Term	Role
Shell	Software that interprets commands
Terminal	Interface to interact with the shell (e.g., GNOME Terminal)
Console	Physical or virtual device for system access

📚 Shell vs Terminal vs Console

Term	Role
Shell	Software that interprets commands
Terminal	Interface to interact with the shell (e.g., GNOME Terminal)
Console	Physical or virtual device for system access

linux terminal

A Linux terminal is a text-based interface that allows users to interact directly with the operating system by entering commands. It's a powerful tool for controlling, automating, and managing system tasks efficiently.

🧠 What Is the Linux Terminal?

The **terminal** (also called the command-line interface or CLI) is a program that opens a **shell**, which interprets user commands and communicates with the Linux kernel. It's essential for developers, system administrators, and power users.

- **Terminal emulator**: A graphical application (like GNOME Terminal, Konsole, or xterm) that simulates a physical terminal.
- **Shell**: The actual command interpreter (e.g., Bash, Zsh) that runs inside the terminal.
- **Console**: The physical or virtual device that provides access to the terminal (e.g., `/dev/tty`).

🔧 Why Use the Terminal?

- **Speed**: Faster than graphical tools for many tasks.
- **Automation**: Enables scripting and batch operations.
- **Control**: Offers fine-grained access to system internals.
- **Remote access**: Essential for managing servers via SSH.

🧪 Common Terminal Tasks

Task	Example Command	
Navigate directories	`cd /home/user`	
List files	`ls -l`	
View file contents	`cat file.txt`	
Search text	`grep "error" log.txt`	
Install software	`sudo apt install nginx`	
Manage processes	`ps aux`, `kill`, `top`	
Redirect output	`ls > files.txt`	
Chain commands	`cat file.txt	grep "data"`	

🧭 Terminal vs Shell vs Console

Component	Role
Terminal	Interface to enter commands and view output
Shell	Program that interprets commands (e.g., Bash)
Console	Physical or virtual device for terminal access

📚 Learn More

- How-To Geek – Linux Terminal for Beginners How-To Geek
- LinuxVox – Mastering the Linux Terminal linuxvox.com
- GeeksforGeeks – Terminal vs Shell vs Console GeeksForGeeks

terminal control

Terminal control in Linux refers to how the operating system and shell manage access to terminal devices — especially in multi-process environments. It governs which process group can interact with the terminal, how input/output is handled, and how signals are delivered.

🧠 What Is Terminal Control?

A **terminal** (like `/dev/tty`) is a character device that provides a text interface between the user and the system. Terminal control determines which process group has the right to:

- **Read input** from the terminal
- **Write output** to the terminal
- **Receive signals** like `SIGINT` (`Ctrl+C`), `SIGTSTP` (`Ctrl+Z`), `SIGQUIT` (`Ctrl+\`)

Only the **foreground process group** of a terminal session can do these things.

🔧 Key Concepts

🔹 Controlling Terminal

- The terminal associated with a **session leader** (usually a shell)
- Only one session can control a terminal at a time

🔹 Foreground vs Background Process Groups

- **Foreground group**: Can read from the terminal and receives signals from keyboard
- **Background group**: Cannot read input; receives `SIGTTIN` or `SIGTTOU` if it tries

🔹 Terminal Modes

- **Canonical mode**: Input is line-buffered (processed after Enter)
- **Non-canonical (raw) mode**: Input is processed byte-by-byte
- Controlled via `termios` structure using `tcgetattr()` and `tcsetattr()`

🧪 System Calls for Terminal Control

Function	Purpose
`tcgetpgrp(fd)`	Get the foreground process group ID for a terminal
`tcsetpgrp(fd, pgid)`	Set the foreground process group ID
`setsid()`	Create a new session and detach from controlling terminal
`ioctl()`	Low-level terminal control (e.g., enabling raw mode)

📚 Example: Setting Foreground Group

C

Copy

```c
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>

int main() {
    int fd = open("/dev/tty", O_RDWR);
    if (fd == -1) {
        perror("open");
        exit(1);
    }

    pid_t pgid = getpgrp(); // Get current process group ID
    if (tcsetpgrp(fd, pgid) == -1) {
        perror("tcsetpgrp");
        exit(1);
    }

    printf("Terminal control assigned to process group %d\n", pgid);
    close(fd);
    return 0;
}
```

🔍 Related Topics

- **Job Control**: Uses terminal control to manage foreground/background jobs
- **Sessions**: Own the controlling terminal
- **Signals**: Routed based on terminal control

terminal and process

In Linux, **terminals** and **processes** are deeply intertwined through the concepts of sessions, process groups, and terminal control. Here's a clear breakdown of how they relate:

🧠 What Is a Terminal?

A **terminal** is a character device (like `/dev/tty`) that provides a text-based interface between the user and the system. It allows users to input commands and view output. In modern systems, this is usually a **pseudo-terminal (PTY)** managed by terminal emulators like GNOME Terminal or xterm.

👥 What Is a Process?

A **process** is an instance of a running program. Each process has:

- A **PID** (process ID)
- A **PPID** (parent process ID)
- A **PGID** (process group ID)
- A **SID** (session ID)

Processes interact with terminals through **sessions** and **process groups**.

🔗 How Terminals and Processes Interact

1. Controlling Terminal

- When a shell starts, it becomes the **session leader** and attaches to a terminal.
- That terminal becomes the **controlling terminal** for the session.

2. Foreground vs Background Process Groups

- Only the **foreground process group** can read from the terminal.
- Background processes are blocked from reading input and receive signals like `SIGTTIN` or `SIGTTOU` if they try.

3. Signal Routing

- The terminal sends signals to the foreground process group:
  - `Ctrl+C` → `SIGINT`
  - `Ctrl+Z` → `SIGTSTP`
  - `Ctrl+\` → `SIGQUIT`

4. Session and Job Control

- A session can contain multiple process groups.
- The shell uses job control to manage which g p is in the foreground.

🧪 Example: Shell and Terminal

```bash
$ ps -eo pid,ppid,sid,pgid,tty,cmd
```

This shows:

- `SID`: Session ID (shared by all processes in a session)
- `PGID`: Process Group ID (used for job control)
- `TTY`: Terminal device (e.g., `pts/0`)

📚 Summary

Component	Role in Terminal Interaction
Terminal	Interface for user input/output
Process	Executes commands and interacts with the terminal
Session	Groups related processes with a controlling terminal
Process Group	Enables job control and signal routing

process' loading of an executable

When a Linux process loads an executable, it replaces its current memory image with the new program using the `exec` family of functions. This involves parsing the ELF file, setting up memory segments, and preparing the runtime environment.

⚙️ Step-by-Step: How a Process Loads an Executable in Linux

Here's a detailed breakdown of what happens when a process executes a new program:

1. System Call Invocation

- The process calls one of the `exec` functions (e.g., `execve()`).
- This passes the path to the executable, arguments (`argv[]`), and optionally environment variables (`envp[]`).

2. File Validation

- The kernel verifies the file exists and is executable.
- It checks permissions and file format (usually ELF — Executable and Linkable Format).

3. ELF Parsing

- The kernel parses the ELF header to understand:
  - Entry point
  - Program headers
  - Required memory segments (text, data, bss)

4. Memory Setup

- The kernel clears the current process memory.
- It maps new segments into memory:
  - **Text segment**: executable code
  - **Data segment**: initialized variables
  - **BSS segment**: uninitialized variables
  - **Stack**: for function calls and local variables
  - **Heap**: for dynamic memory allocation

5. Dynamic Linking (if needed)

- If the executable is dynamically linked:
  - The **dynamic linker** (`ld-linux.so`) is loaded.
  - It resolves symbols and loads shared libraries (`libc.so`, etc.).

6. Environment Setup

- The kernel sets up:
  - `argv[]` and `envp[]` on the stack
  - Auxiliary vectors (`auxv`) for runtime info
  - File descriptors (stdin, stdout, stderr)

7. Transfer Control

- The kernel sets the instruction pointer to the program's entry point.
- The process begins execution — starting with `_start`, then `main()`.

🧪 Example: `execve()` Flow

```c
execve("/bin/ls", argv, envp);
```

- Kernel loads `/bin/ls`
- Parses ELF headers
- Maps memory segments
- Resolves dynamic libraries
- Starts execution at `_start`

loding an executable into a memory

Loading an executable into memory in Linux is a complex, well-orchestrated process handled by the kernel. It involves parsing the executable file (usually in ELF format), setting up memory regions, and preparing the runtime environment. Here's a detailed breakdown:

🧠 Overview: What Happens When an Executable Is Loaded

When a process calls an `exec` function (e.g., `execve()`), the kernel replaces the current process image with a new one. This includes:

1. **Validating the executable**
2. **Parsing the ELF headers**
3. **Mapping memory segments**
4. **Setting up the stack and environment**
5. **Transferring control to the entry point**

⚙️ Step-by-Step Breakdown

1. System Call Invocation

- The process calls `execve(path, argv, envp)`.
- The kernel begins replacing the current process image.

2. Executable Validation

- Checks file existence and permissions.
- Verifies the file format (typically ELF).
- Ensures it's marked executable.

3. ELF Header Parsing

- Reads the ELF header to determine:
  - Entry point address
  - Program headers (segment layout)
  - Interpreter path (for dynamically linked binaries)

4. Memory Mapping

- Clears the current memory image.
- Maps segments into memory:
  - **Text segment**: executable code (read + execute)
  - **Data segment**: initialized variables (read + write)
  - **BSS segment**: uninitialized variables (zeroed memory)
  - **Heap**: for dynamic allocations
  - **Stack**: for function calls and local variables

5. Dynamic Linking (if needed)

- Loads the dynamic linker (`ld-linux.so`).
- Resolves shared library dependencies.
- Maps required libraries into memory.

6. Stack and Environment Setup

- Pushes `argv[]`, `envp[]`, and auxiliary vectors (`auxv`) onto the stack.
- Sets up initial registers and stack pointer.

7. Transfer of Control

- Sets the instruction pointer to the entry point.
- Begins execution at `_start`, which eventually calls `main()`.

🧪 Visual Summary

Text

Copy

```
[ Executable File (ELF) ]
        ↓
[ Kernel parses ELF headers ]
        ↓
[ Memory segments mapped ]
        ↓
[ Stack and environment prepared ]
        ↓
[ Control transferred to entry point (_start) ]
```

📚 Related Concepts

- **ELF format**: Standard binary format for executables and shared libraries
- **Virtual memory**: Each process gets its own address space
- **Dynamic linker**: Loads and links shared libraries at runtime
- **Auxiliary vectors**: Metadata passed to the new process (e.g., page size, CPU features)

virtual memory and a process

Virtual memory is a foundational concept in Linux that allows each process to operate as if it has its own private, contiguous address space — even though physical memory is shared across the system. It's essential for process isolation, efficient memory use, and system stability.

🧠 What Is Virtual Memory?

Virtual memory is an abstraction provided by the operating system that maps **virtual addresses** used by processes to **physical addresses** in RAM. This mapping is handled by the **MMU (Memory Management Unit)** and the **kernel**.

👤 How It Works for a Process

Every process in Linux gets its own **virtual address space**, which is divided into segments:

Segment	Purpose	Access Rights
Text	Executable code	Read + Execute
Data	Initialized global/static variables	Read + Write
BSS	Uninitialized global/static variables	Read + Write
Heap	Dynamic memory (`malloc` , `new`)	Read + Write
Stack	Function calls, local variables	Read + Write
Memory-mapped	Shared libraries, files	Varies

Each segment is mapped to physical memory (or disk) as needed, using **paging**.

🔧 Key Benefits

- **Isolation**: Each process is protected from others — no accidental memory overlap.
- **Efficiency**: Only needed pages are loaded into RAM; unused parts stay on disk.
- **Flexibility**: Processes can use more memory than physically available (via swap).
- **Security**: Permissions (read/write/execute) are enforced per segment.

🧪 Example: Viewing a Process's Memory Map

Bash Copy

```
$ cat /proc/<pid>/maps
```

This shows the virtual memory layout of a running process, including addresses, permissions, and mapped files.

📚 Related Concepts

- **Paging**: Divides memory into fixed-size blocks (pages)
- **Page tables**: Track virtual-to-physical address mappings
- **Swap space**: Disk area used when RAM is full
- **Copy-on-write**: Efficient memory sharing during `fork()`

copy-on-write

Copy-on-Write (CoW) is a memory optimization technique used in Linux (and other operating systems) to efficiently handle process creation — especially during `fork()`. It allows the parent and child processes to share the same memory pages until one of them modifies the data.

🧠 What Is Copy-on-Write?

When a process is duplicated using `fork()`, the kernel doesn't immediately copy all memory pages. Instead, it:

- Marks shared pages as **read-only**
- Duplicates **page tables**, not the actual memory
- Defers copying until a **write** occurs

This saves time and memory when the child process doesn't modify the data — which is common in cases like `fork()` followed by `exec()`.

🔧 How It Works

🔹 Step-by-Step

1. **Parent calls** `fork()`
   - Kernel creates a child process with identical page tables.
   - All memory pages are marked read-only and shared.
2. **Child or parent writes to a page**
   - A **page fault** occurs due to read-only protection.
   - Kernel allocates a new page and copies the original content.
   - The writing process gets its own copy — hence “copy-on-write.”

🧪 Example Scenario

C

Copy

```c
pid_t pid = fork();
if (pid == 0) {
    // Child process
    buffer[0] = 'X'; // triggers CoW if buffer was shared
}
```

- Before the write: both processes share `buffer`
- After the write: child gets a private copy of the page

📚 Benefits

Advantage	Description
Efficiency	Avoids unnecessary memory copying
Speed	Faster process creation
Isolation	Ensures memory integrity when changes occur

🔍 Where It's Used

- `fork()` system call
- Virtual memory management
- Filesystems like **Btrfs** and **ZFS**
- Container runtimes and snapshotting

how to create many children using fork

To create multiple child processes using `fork()` in C, you can use a loop where each iteration spawns a new child. Here's a simple and effective pattern:

🧪 Example: Creating Multiple Children with `fork()`

C

Copy

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

#define NUM_CHILDREN 5

int main() {
    for (int i = 0; i < NUM_CHILDREN; i++) {
        pid_t pid = fork();

        if (pid < 0) {
            perror("fork failed");
            exit(1);
        } else if (pid == 0) {
            // Child process
            printf("Child %d: PID = %d\n", i, getpid());
            exit(0); // Important: prevent child from continuing the loop
        }
        // Parent continues loop to create next child
    }

    // Parent waits for all children
    for (int i = 0; i < NUM_CHILDREN; i++) {
        wait(NULL);
    }

    printf("Parent: All children have exited.\n");
    return 0;
}
```

🔍 Key Points

- Each `fork()` creates a new child.
- The child must `exit()` immediately to avoid creating more children.
- The parent uses `wait()` to reap all children and avoid zombies.

🧠 What Happens Internally

- After each `fork()`, two processes exist: parent and child.
- Only the parent continues the loop to create more children.
- Each child exits after printing its message.

wait

The `wait()` system call in Linux is used by a **parent process** to pause execution until one of its **child processes** terminates. It's essential for proper process cleanup and preventing zombie processes.

🧠 What `wait()` Does

C

Copy

```c
#include <sys/wait.h>
pid_t wait(int *status);
```

- **Blocks** the parent until any child process exits.
- **Returns** the PID of the terminated child.
- **Fills** `status` with exit information (if not `NULL`).

🧪 Example: Using `wait()` After `fork()`

C

Copy

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

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        // Child process
        printf("Child: PID = %d\n", getpid());
        sleep(2);
        exit(5); // Exit with status 5
    } else {
        // Parent process
        int status;
        pid_t child_pid = wait(&status);

        if (WIFEXITED(status)) {
            printf("Parent: Child %d exited with status %d\n",
                   child_pid, WEXITSTATUS(status));
        } else {
            printf("Parent: Child %d did not exit normally\n", child_pid);
        }
    }

    return 0;
}
```

🔍 Key Macros

Macro	Purpose
`WIFEXITED(status)`	Child exited normally
`WEXITSTATUS(status)`	Exit code from child
`WIFSIGNALED(status)`	Child terminated by signal
`WTERMSIG(status)`	Signal number that caused termination

🆚 `wait()` vs `waitpid()`

Function	Behavior
`wait()`	Waits for **any** child
`waitpid()`	Waits for a **specific** child

waitpid

`waitpid()` is a powerful system call in Linux used to **wait for a specific child process** to terminate. It gives you more control than `wait()`, especially when managing multiple children or using non-blocking behavior.

🧠 What `waitpid()` Does

C

Copy

```c
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

- `pid`: Which child to wait for
  - `> 0`: Wait for child with that PID
  - `-1`: Wait for any child (like `wait()`)
  - `0`: Wait for any child in same process group
- `status`: Pointer to int where exit info is stored
- `options`:
  - `0`: Block until child exits
  - `WNOHANG`: Return immediately if no child has exited
  - `WUNTRACED`: Also report stopped children

🧪 Example: Wait for Specific Child

C

Copy

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

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        // Child process
        printf("Child: PID = %d\n", getpid());
        sleep(2);
        exit(42); // Exit with status 42
    } else {
        // Parent process
        int status;
        pid_t result = waitpid(pid, &status, 0); // Wait for specific child

        if (result == -1) {
            perror("waitpid failed");
        } else if (WIFEXITED(status)) {
            printf("Parent: Child %d exited with status %d\n", result, WEXITSTATUS(s
        }
    }

    return 0;
}
```

🔍 Key Macros

Macro	Purpose
`WIFEXITED(status)`	Child exited normally
`WEXITSTATUS(status)`	Exit code from child
`WIFSIGNALED(status)`	Child terminated by signal
`WTERMSIG(status)`	Signal number that caused termination

🧠 When to Use `waitpid()`

- You want to wait for a **specific child**.
- You need **non-blocking** behavior (`WNOHANG`).
- You're managing **multiple children** concurrently.

how wait can be used for multiple children process

To use `wait()` for **multiple child processes**, you simply call it **repeatedly** in a loop. Each call to `wait()` will block until one child terminates, allowing the parent to reap all children one by one.

🧪 Example: Reaping Multiple Children with `wait()`

C

Copy

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

#define NUM_CHILDREN 5

int main() {
    for (int i = 0; i < NUM_CHILDREN; i++) {
        pid_t pid = fork();

        if (pid < 0) {
            perror("fork failed");
            exit(1);
        } else if (pid == 0) {
            // Child process
            printf("Child %d: PID = %d\n", i, getpid());
            sleep(i + 1); // Simulate work
            exit(i);      // Exit with unique status
        }
        // Parent continues loop to create next child
    }
```

```c
    // Parent waits for all children
    for (int i = 0; i < NUM_CHILDREN; i++) {
        int status;
        pid_t child_pid = wait(&status);

        if (WIFEXITED(status)) {
            printf("Parent: Child %d exited with status %d\n",
                   child_pid, WEXITSTATUS(status));
        } else {
            printf("Parent: Child %d did not exit normally\n", child_pid);
        }
    }

    printf("Parent: All children have been reaped.\n");
    return 0;
}
```

🔍 How It Works

- Each `fork()` creates a child.
- Each child sleeps for a different time and exits with a unique code.
- The parent calls `wait()` in a loop to reap all children.
- `wait()` returns the PID of the child that exited, and fills `status`.

🧠 Tips

- You can use `waitpid(-1, &status, 0)` instead of `wait()` — it behaves the same but gives more control.
- If you want **non-blocking** behavior, use `waitpid()` with `WNOHANG`.

fork, return, exit

Here's a concise breakdown of how `fork()`, `return`, and `exit()` behave in a C program — especially in the context of parent-child process control.

🧠 `fork()`: Creating a New Process

C ^ Copy

```c
pid_t pid = fork();
```

- Creates a **child process**.
- Returns:
  - `0` to the **child**
  - Child's PID to the **parent**
  - `-1` on error

🔁 `return`: Exiting a Function

- Used to exit from a function (like `main()`).
- In `main()`, `return` is equivalent to calling `exit()` with the same status.

```c
int main() {
    return 0; // Same as exit(0)
}
```

🚪 `exit()`: Terminating a Process

```c
exit(status);
```

- Terminates the current process.
- Cleans up resources, flushes I/O buffers.
- Sends exit status to the parent (retrievable via `wait()` or `waitpid()`).

🧪 Example: `fork`, `return`, and `exit`

C

Copy

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

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        // Child process
        printf("Child: PID = %d\n", getpid());
        exit(42); // Exit with status 42
    } else {
        // Parent process
        int status;
        wait(&status);
        printf("Parent: Child exited with status %d\n", WEXITSTATUS(status));
        return 0; // Exit parent
    }
}
```

🔍 Summary

Function	Purpose	Scope
`fork()`	Creates a new process	System call
`return`	Exits a function (e.g., `main`)	Language keyword
`exit()`	Terminates the process	Library function

if parent exit the main, then what happened to child process

If a **parent process exits its** `main()` **function**, the behavior of the **child process** depends on whether the parent terminates **before or after** the child finishes. Here's what happens in each case:

🧠 Case 1: Parent Exits Before Child

- The child becomes an **orphan**.
- The **init process** (`PID 1`, or `systemd` on modern Linux) adopts the orphaned child.
- The child continues running normally.
- When the child exits, `init` will reap it to prevent a zombie.

🔍 Example

C ^ Copy

```c
pid_t pid = fork();
if (pid == 0) {
    sleep(5); // Child runs longer
    printf("Child: still alive after parent exited\n");
} else {
    return 0; // Parent exits immediately
}
```

- The child runs independently.
- The system ensures cleanup via `init`.

🧠 Case 2: Parent Exits After Child

- The parent can use `wait()` or `waitpid()` to reap the child.
- If the parent doesn't wait, the child becomes a **zombie** until the parent exits.

🔍 Zombie Process

- A child that has exited but hasn't been reaped by its parent.
- Occupies a slot in the process table.
- Cleared when the parent calls `wait()` or exits.

🧠 Summary

Parent Action	Child Outcome
Exits before child	Child becomes orphan, adopted by `init`
Exits after child	Child may become zombie if not reaped
Calls `wait()`	Child is reaped properly

kill(getppid(), 0)

The expression `kill(getppid(), 0)` in C is a **clever way to check if the parent process is still alive** — without actually sending a signal.

🧠 What It Does

- `getppid()` returns the **parent process ID**.
- `kill(pid, 0)` doesn't send a signal — it just checks:
  - If the process with that PID exists
  - If the caller has permission to signal it

✅ If the parent is alive and accessible:

```c
kill(getppid(), 0) == 0
```

❌ If the parent is dead or inaccessible:

```c
kill(getppid(), 0) == -1
```

And `errno` will be:

- `ESRCH`: No such process (parent has exited)
- `EPERM`: Permission denied

🧪 Example Usage

C

Copy

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>

int main() {
    if (kill(getppid(), 0) == -1) {
        if (errno == ESRCH) {
            printf("Parent process has exited.\n");
        } else if (errno == EPERM) {
            printf("No permission to signal parent.\n");
        }
    } else {
        printf("Parent process is alive.\n");
    }
    return 0;
}
```

📚 Use Cases

- **Child process monitoring parent** (e.g., for cleanup or shutdown)
- **Daemon supervision**
- **Detecting orphaned processes**

how to wait on parent process

In Linux, **you cannot directly wait on a parent process** using `wait()` or `waitpid()` — those system calls are strictly for waiting on **child processes**. However, if you need a child process to monitor or synchronize with its parent, there are alternative techniques.

🧠 Why `wait()` and `waitpid()` Don't Work for Parents

- These functions are designed for **parent processes to wait on their children**.
- If a child tries to wait on its parent, the kernel will return an error (`ECHILD`) because the parent is not a child of the calling process.

✅ Alternative Ways to Synchronize with a Parent

1. Polling with `kill(pid, 0)`

Check if the parent is still alive:

```c
if (kill(getppid(), 0) == -1) {
    // Parent has exited
}
```

2. Signals

Set up signal handlers in the child and have the parent send a signal when it's ready or exiting.

3. Pipes or Sockets

Use inter-process communication (IPC) to send messages between parent and child.

4. Shared Memory + Semaphores

Create shared memory regions and use semaphores or flags to coordinate.

5. `prctl(PR_SET_PDEATHSIG)`

In the child, you can request a signal when the parent dies:

```c
#include <sys/prctl.h>
prctl(PR_SET_PDEATHSIG, SIGTERM);
```

This causes the child to receive `SIGTERM` if the parent exits.

🧪 Example: Detect Parent Exit with `prctl`

C

Copy

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

void handler(int sig) {
    printf("Parent exited, received signal %d\n", sig);
    _exit(0);
}

int main() {
    signal(SIGTERM, handler);
    prctl(PR_SET_PDEATHSIG, SIGTERM);

    while (1) {
        sleep(1);
        printf("Child still running\n");
    }
}
```