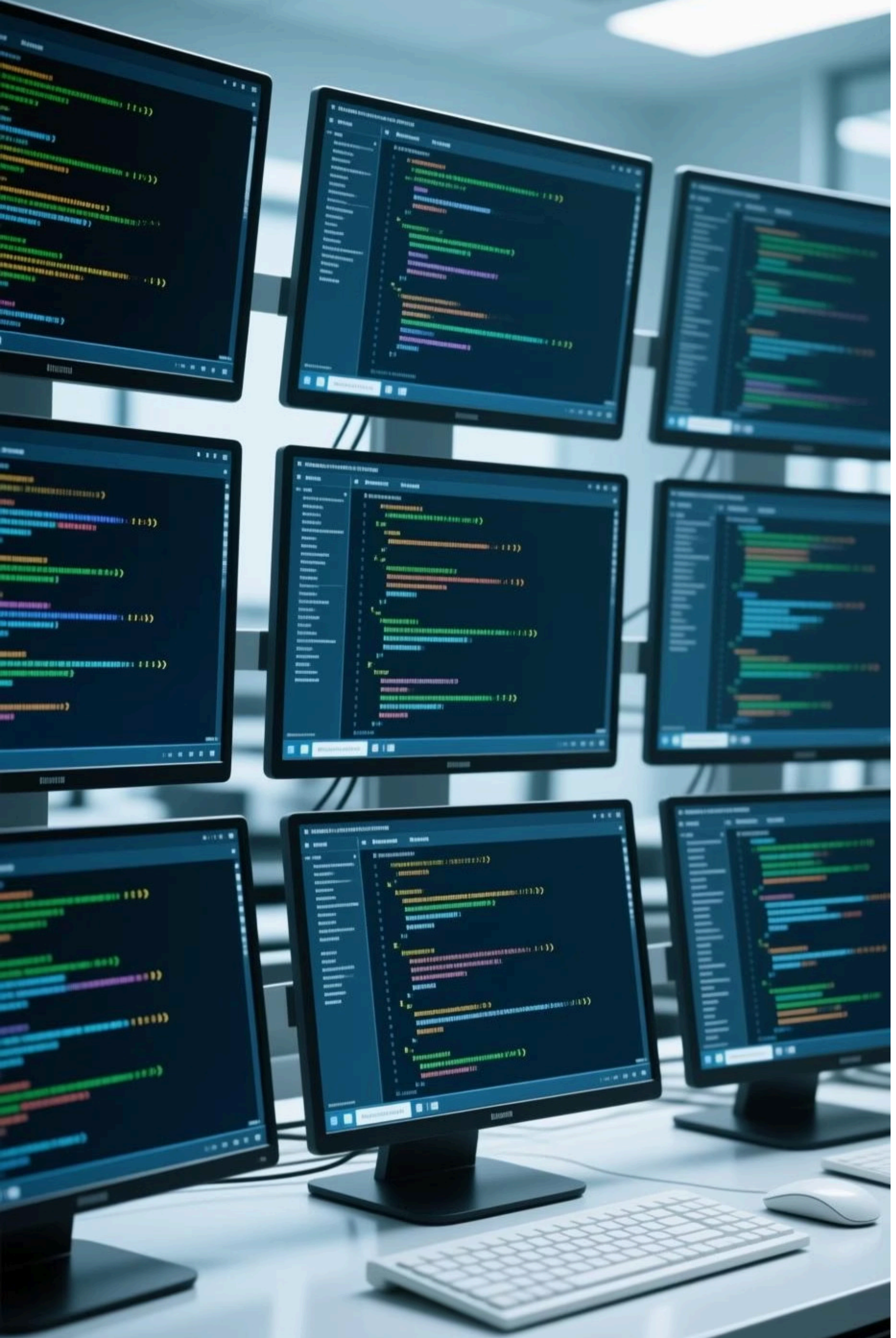
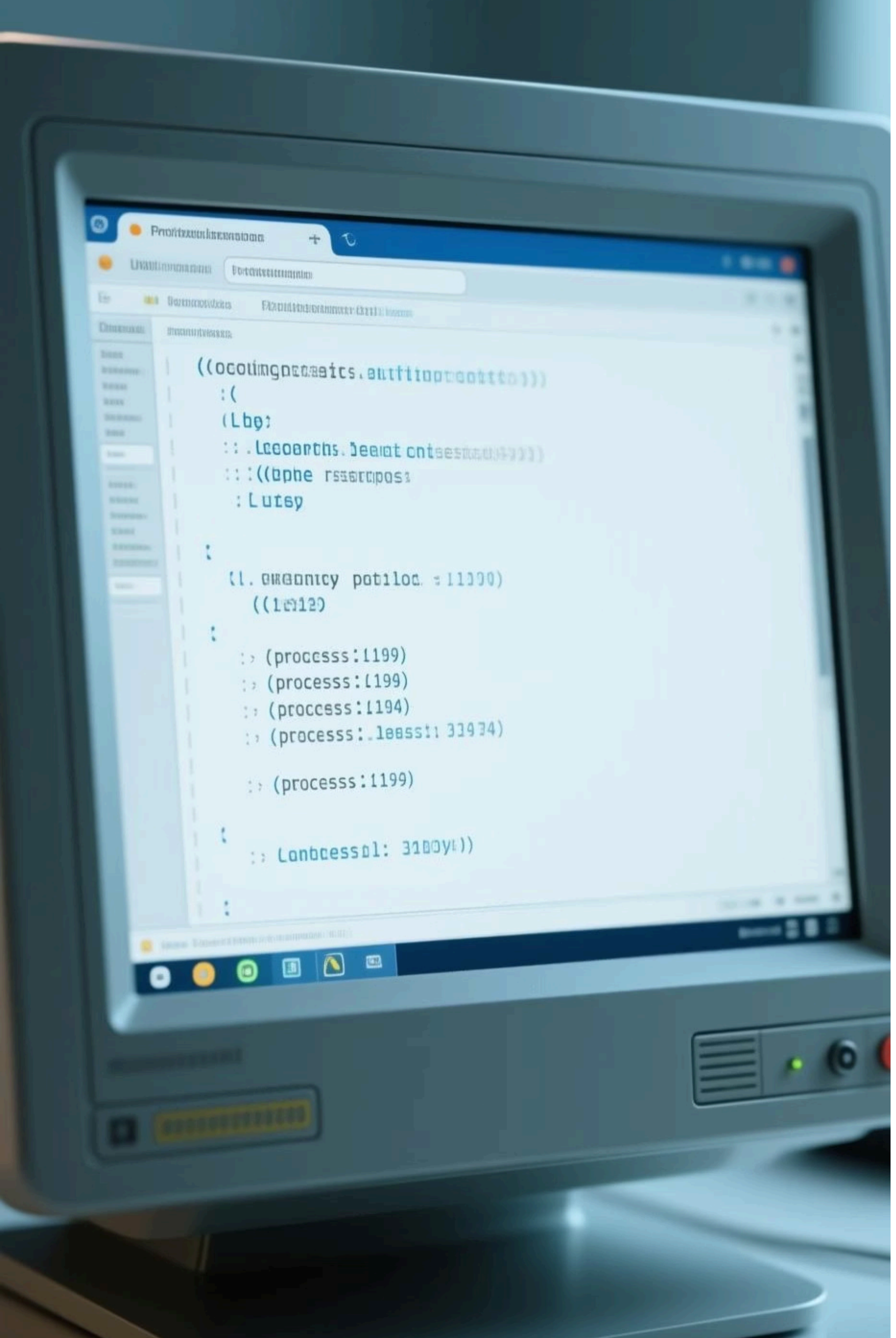


Process Creation in UNIX: fork(), exec(), waitpid() and Process States

An exploration of how processes are created, transformed, and managed in UNIX-based operating systems, with practical examples and potential pitfalls.



Chapter 1: The Birth of a Process – fork()

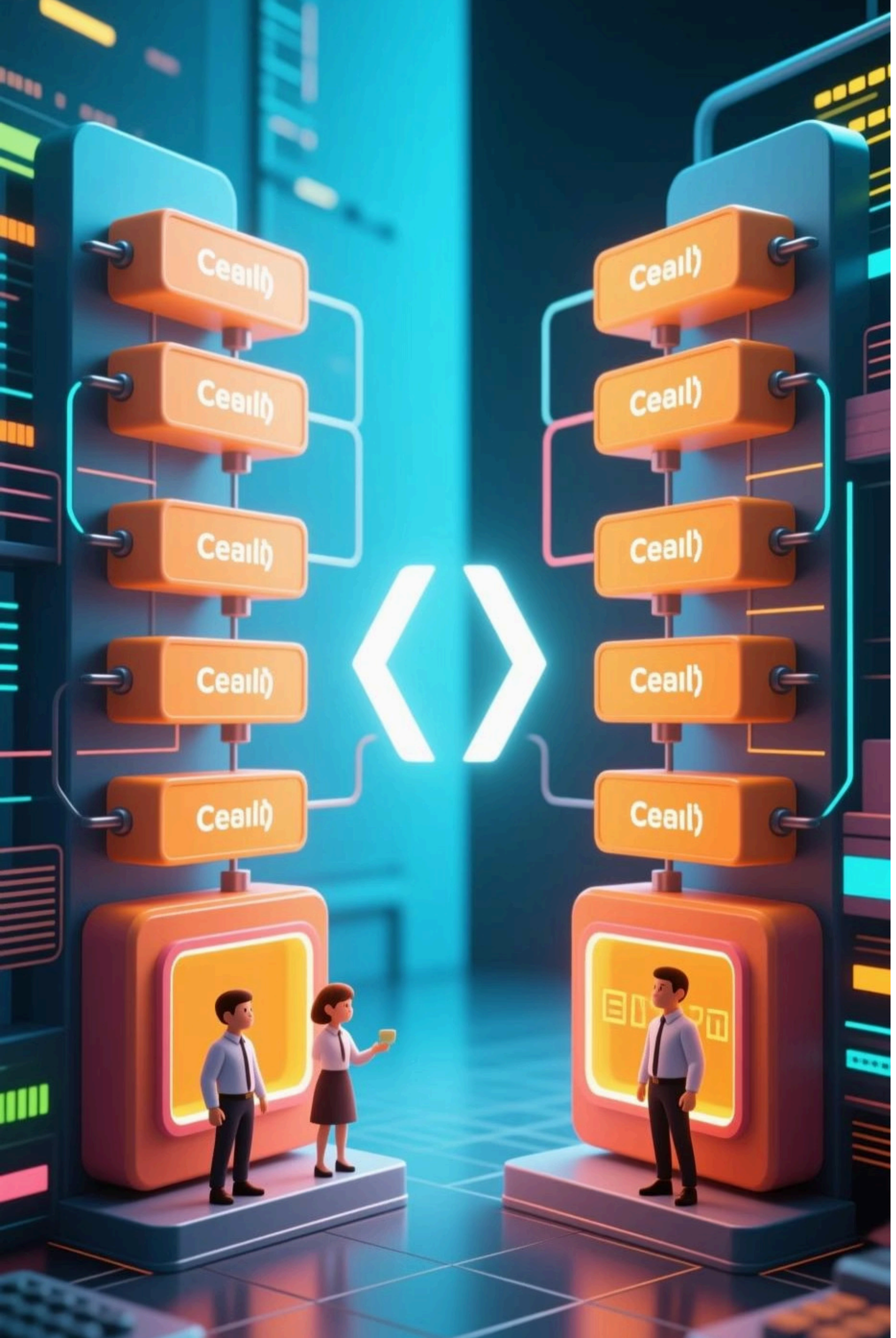
Understanding how new processes come into existence in UNIX systems

The fork() system call is the fundamental mechanism by which UNIX systems create new processes, establishing the parent-child relationship that underlies all process management.

fork(): Cloning the Parent Process

- fork() creates a child process by duplicating the parent's memory and state
- Returns 0 to the child, child's PID to the parent, and -1 on failure
- Both parent and child continue execution from the fork() call concurrently

```
pid_t pid = fork();
if (pid < 0) {
 // Error handling
} else if (pid == 0) {
 // Child process code
} else {
 // Parent process code
}
```



Process Bifurcation at fork()

After fork(), the parent and child processes execute the same code but can follow different paths based on the return value. They have separate memory spaces but initially identical content.

Why fork() is Powerful but Dangerous

Fork Bombs

Recursive fork() calls can exponentially create processes, overwhelming system resources in seconds

```
:(){ :|:& };:
```

Protection Measures

Use `ulimit -u 40` to limit max processes per user to prevent system crashes

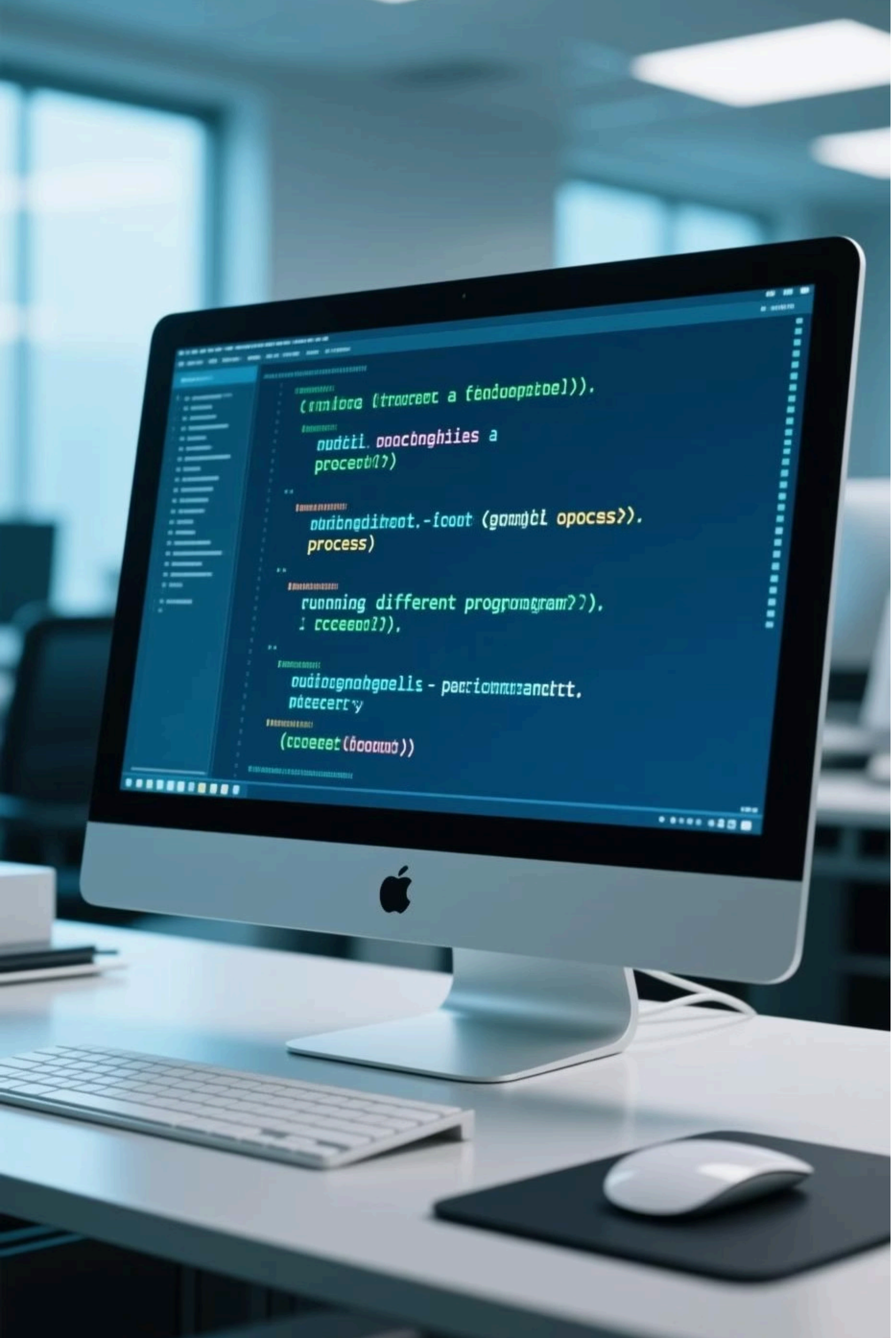
Testing Environment

Always test fork() code with caution, preferably with root access or on systems with physical access

fork() Return Values: How to Tell Parent from Child

The fork() system call has three possible return values, each indicating a different outcome:

0	>0	-1
Child Process	**Parent Process**	**Error**
A return value of zero indicates this is the newly created child process	A positive integer (the child's PID) indicates this is the parent process	Check errno for causes like EAGAIN (resource limits) or ENOMEM (memory)



Chapter 2: Transforming the Child - exec()

While fork() creates an identical copy of a process, the exec() family of functions allows a process to completely transform itself by loading and running a different program.

exec(): Replacing Process Image

- exec() family replaces current process memory with a new program
- Child process after fork() often calls exec() to run a different program
- exec() never returns on success; returns only on failure

exec() Variants and Usage

execv(path, argv[])

Requires full path to executable

```
execv("/bin/ls", args);
```

execvp(file, argv[])

Searches PATH environment for executable

```
execvp("ls", args);
```

execve(path, argv[], envp[])

Allows specification of environment variables

```
execve("/bin/ls", args, env);
```

Example: `execv("/bin/ls", ["ls", "-l", NULL])` runs 'ls -l' in child process

Why fork() is Often Followed by exec()

The fork-exec pattern is the cornerstone of UNIX process creation:

- fork() creates a duplicate process; exec() transforms it into a new program
- Keeps parent process intact while child runs new code
- Fits UNIX philosophy: small tools combined via process creation

This pattern enables shells, service managers, and other tools to spawn diverse programs while maintaining their own execution context.

Chapter 3: Synchronising Processes - waitpid()

How parent processes can monitor and synchronize with their children

The waitpid() system call allows a parent process to wait for state changes in a child process, particularly waiting for the child to terminate before continuing execution.

waitpid(): Parent Waits for Child Termination

- Parent calls `waitpid(child_pid, &status, 0)` to block until child finishes
- Prevents zombie processes by reaping child's exit status
- Allows parent to know child's termination and resource cleanup

```
pid_t child_pid = fork();
if (child_pid == 0) {
 // Child process
 exit(42);
} else {
 // Parent process
 int status;
 waitpid(child_pid, &status, 0);
 if (WIFEXITED(status)) {
 printf("Child exited with %d\n",
 WEXITSTATUS(status));
 }
}
```

Example: fork(), exec(), and waitpid() Cycle

Parent forks child

`pid = fork();`

Child calls exec()

`execl("/bin/ls", "ls", NULL);`

Parent waits

`waitpid(pid, &status, 0);`

Parent continues

After child completes

This pattern ensures orderly process lifecycle and resource management, preventing zombie processes and allowing coordinated multi-process applications.

Chapter 4: Parent and Child Processes in Action

Understanding the dynamic relationships between processes and how they behave during normal operation and exceptional circumstances is crucial for robust systems.

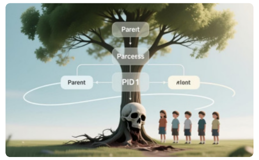
Parent and Child Process IDs

Every process in UNIX has unique identifiers:

- `getpid()` returns current process ID
- `getppid()` returns parent process ID
- Parent can spawn multiple children; children can spawn their own children

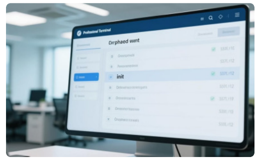
```
#include
#include

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

 if (pid == 0) {
 printf("Child: PID=%d, Parent=%d\n",
 getpid(), getppid());
 } else {
 printf("Parent: PID=%d, Child=%d\n",
 getpid(), pid);
 }
 return 0;
}
```

Orphan Processes: When Parent Dies Early



Orphan Definition

A child process whose parent terminates before the child does



Adoption by init

The init process (PID 1) automatically adopts orphaned children

Continued Execution

Orphans continue running normally, preventing premature termination

Zombie Processes: Dead but Not Reaped

Zombie processes occur when:

- Child terminates but parent hasn't called wait() or waitpid()
- Process entry remains in process table as "Z" or "defunct"
- Accumulation can exhaust system resources if not handled

Zombies hold minimal resources but each consumes a process table entry, which is a limited system resource.

Process Lifecycle States

A process moves through various states from creation to termination. Normal processes are created, run, and terminate with proper cleanup. Orphans continue running under init's supervision, while zombies remain in the process table until reaped.

Real-World Example: Handling Zombies and Orphans

Zombie Prevention

```
signal(SIGCHLD, SIG_IGN);
// OR
signal(SIGCHLD, child_handler);
```

Custom Reaping

```
void child_handler(int sig) {
  while(waitpid(-1, NULL,
        WNOHANG) > 0);
}
```

Orphan Handling

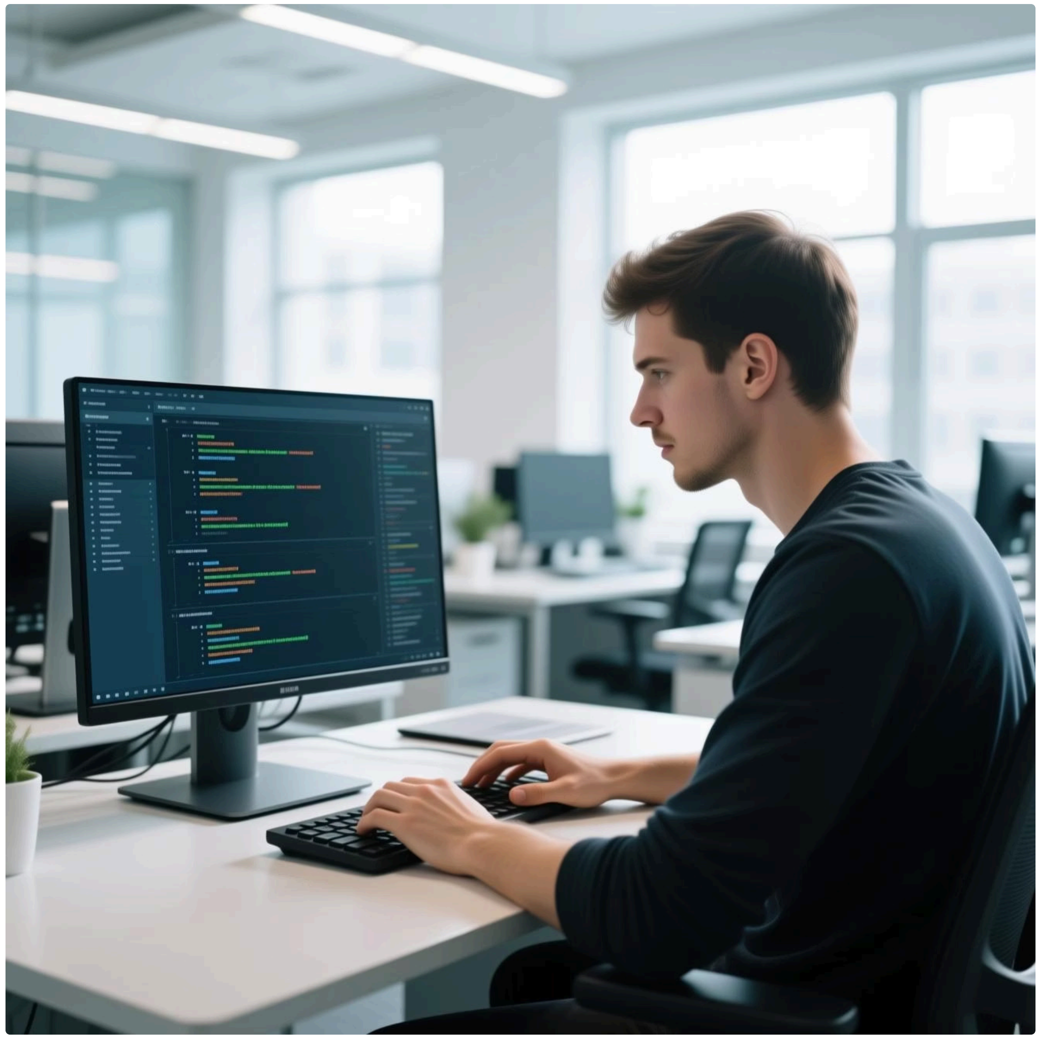
```
// For daemons
if (fork() > 0) exit(0);
setsid();
```

Proper signal handling and wait calls ensure system stability and prevent resource exhaustion from process accumulation.

Conclusion: Mastering Process Creation and Management

Key takeaways from our exploration:

- fork() births a new process; exec() transforms it; waitpid() synchronises
- Understanding parent-child dynamics prevents resource leaks and system issues
- Essential knowledge for robust UNIX/Linux programming and system design



Mastering these system calls gives you powerful tools to build efficient, reliable multi-process applications on UNIX systems.