1. QUESTIONS FROM LAST LECTURE

Memory Management Q&A

Question: Do we need to free all memory allocated in test programs?

Answer: No, we don't need to.

Why?

- The operating system cleans up our allocated memory for us when our program exits
- In fact, it frees our entire heap, stack, code, everything

Important Context:

- While the OS automatically reclaims memory when a program terminates, it's still best practice to free memory properly in production code
- For long-running programs (servers, daemons), not freeing memory causes real memory leaks during execution
- Good habit: Always pair `malloc`/`calloc` with corresponding `free` calls

---

2. BRAIN TEASER: FORK() SYSTEM CALL

The Challenge:

How many processes will this code create?

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

int main() { int i; for (i = 0; i < 4; i++) fork(); return 0; }`

Analysis:

- This is a classic fork bomb demonstration
- Each `fork()` creates a new child process
- Both parent and child continue executing the loop
- **Total processes created:** 2^4 = 16 processes (including the original)

How it works:

- Iteration 0: 1 process calls fork() → 2 processes total
- Iteration 1: 2 processes each call fork() → 4 processes total
- Iteration 2: 4 processes each call fork() → 8 processes total
- Iteration 3: 8 processes each call fork() → 16 processes total

---

3. MEMORY LEAK EXAMPLE

The Problem:

How do we free the memory allocated by `func`?

`c#include <stdlib.h>

void func() { void *x = malloc(20); }

int main() { func(); // TODO – free x here return 0; }`

Answer:

- You **cannot** free `x` from `main()` because `x` is a local variable in `func()` and goes out of scope when `func()` returns
- **Solution 1:** Free `x` inside `func()` before returning
- **Solution 2:** Return `x` from `func()` so `main()` can free it later
- **Solution 3:** Pass a pointer to `x` from `main()` so both functions have access

Proper approach:

```
cvoid func() { void *x = malloc(20); // ... use x ... free(x); // Free before returning }
```

---

4. PROCESS CONCEPT

What is a Process?

Definition: Something that is **running** or **executing** on a CPU

Terminology:

- If initiated by a **user**, might be called a **program**
- If initiated by the **system**, may be called a **job**
- Might be called a **task** (Linux uses task more than process)
- Let's just called them all **processes**

Key Distinctions:

Passive vs Active:

- A program on disk is **passive**
- A process is an **active** entity – it is currently running

Important Notes:

- Though when we say "running," we might mean sleeping!
- Not every process is executing all the time (e.g., 5,000 threads but only 16 cores)
- An active process is listed in the process table

---

5. PROCESS STATES

State Model

A process is not always executing – it can have a number of **states**:

Five Basic States:

1. **New** - Process is being created
2. **Ready** - Process is waiting to be assigned to a processor
3. **Running** - Instructions are being executed
4. **Waiting/Blocked** - Process is waiting for some event (I/O, signal)
5. **Terminated** - Process has finished execution

State Transitions:

From Ready → Running:

- Only one process can run on one core at a time
- A process that is ready to run is added to a **ready queue**
- The **scheduler** will try to assign that process a CPU core as soon as one is available

From Running → Waiting:

- A process that makes a **blocking I/O call** or **sleeps** (blocks on a timer) is moved to the waiting state and removed from the run list

Monitoring:

- The program `top` shows the total number of **active** tasks (processes) as well as the **load average** which is the average length of the ready queue

---

6. PROCESS STATE TRANSITIONS

Detailed Transition Flow:

1. Process Creation:

- A process initially moves to **ready** after it has been created by a call to `fork()`

2. Scheduling:

- When scheduled, it moves to the **running** state

3. From Running State, Several Events May Happen:

a) I/O Request:

- The process might make an I/O request that causes it to become **blocked**

b) Time Slice Exceeded:

- The process may exceed its time slice, in which case the scheduler moves it back to the **ready** queue

c) Normal Termination:

- The process may finish (move to **terminated** state) by calling `exit()`

d) Forced Termination:

- The kernel may terminate the process

---

7. PROCESS CONTROL BLOCKS (PCBs)

What is a PCB?

Purpose:

- The kernel must maintain state **information** for every **active process**
- It uses a struct called the **process control block (PCB)**

In Linux:

- In Linux, this is the `task_struct` defined in `<linux/sched.h>`
- Kernel code is heavy reading, but the samples show states and linkages for parents and children

PCB Contents Include:

- Process ID (PID)
- Process state (ready, running, waiting, etc.)
- Program counter (next instruction address)
- CPU registers
- CPU scheduling information (priority, scheduling queue pointers)
- Memory management information (page tables, memory limits)
- Accounting information (CPU time used, time limits)
- I/O status information (open files, I/O devices allocated)

Accessing Process Information:

- The directory `/proc/<pid>` contains lots of information on a process

---

8. PROCESS QUEUES

Queue Organization:

Chaining Mechanism:

- PCBs are **chained** together in lists
- The **ready list** is the list of processes that are ready to run

Process Distribution:

- The number of currently running processes can be up to the number of CPU cores

- All other processes are in another state: **ready**, **blocked**, **terminated**, etc

Device Queues:

- Processes are placed in a **device_queue** when **waiting** on a device
- Each device has its own **device_queue**

Example Queue Structure:

- **Ready Queue:** All processes ready for CPU
- **Disk I/O Queue:** Processes waiting for disk operations
- **Network I/O Queue:** Processes waiting for network operations
- **Printer Queue:** Processes waiting for printer
- **Keyboard Queue:** Processes waiting for keyboard input

---

9. PROCESS OPERATION FUNCTION CALLS

Core System Calls:

We have looked at the **functions** that start and stop processes:

1. fork()

Purpose: Create a new process by making a copy of the current process

How it works:

- Creates an identical copy of the calling process
- Both processes continue execution from the point after `fork()`
- Returns different values to parent and child

Return values:

- Returns `0` in the child process
- Returns child's PID in the parent process
- Returns `1` on error

2. exec...()

Purpose: Execute a new program over the current process (overlay)

Variants:

- There are variants of exec depending on how you want to pass parameters and/or env vars to a child process
- See `man exec` for more info

Common variants:

- `execl()`, `execle()`, `execlp()`
- `execv()`, `execve()`, `execvp()`

Important: exec only returns on error (returns `-1` and sets `errno`)

3. exit()

Purpose: Terminate the current process

Usage:

- You can explicitly call `exit()` but return from main does this for you implicitly

4. wait()

Purpose: Wait for a child process to complete

Who uses it:

- In addition, parents have an extra call: `wait()` is called to wait for a child process to complete

Return value:

- Returns the PID of the terminated child
- Returns `1` on error

---

10. SIGNALS

What are Signals?

- Linux/Unix processes can receive **signals** from other processes

- These are **numeric** but have macros defined for them as well
- Many signals range metaphorically from a light tap on the shoulder to being shot by a sniper on the roof

Common Signal Types:

#	Signal	Meaning
1	SIGHUP	(Hangup) Please reconfigure yourself
2	SIGINT	I'd like you to stop please if you can (Ctrl-C)
15	SIGTERM	Please terminate now
3	SIGQUIT	Terminate NOW
9	SIGKILL	I wasn't asking
4	SIGILL	Illegal instruction
6	SIGABRT	Abort - abnormal termination
8	SIGFPE	Floating point exception
11	SIGSEGV	Segmentation fault

Signal Handling:

Sending Signals:

- The `kill` command sends signals from shell
- `kill()` function sends signals from a program
- Called "kill" presumably because most signals related to process termination

Signal Trapping:

- Many signals can be trapped, but some cannot (e.g. SIGKILL – you don't see that one coming)
- You can register signal handlers to catch and respond to signals

---

11. ZOMBIE PROCESSES

What is a Zombie?

- A parent process must call `wait()` to clean up children when they terminate
- If the parent doesn't call `wait()`, the child process becomes a **zombie**

Why Zombies Matter:

- Zombie processes stay in the process table even though they're terminated
- They consume system resources (PCB entries)
- This is important for daemons and other child processes that outlive their parents

How to Prevent Zombies:

- Parent must call `wait()` or `waitpid()` to reap terminated children
- Use signal handlers to catch `SIGCHLD` and call `wait()` asynchronously
- Or use double-fork technique to orphan children to init process

---

12. PROCESS SCHEDULING

The Scheduler's Role:

- The **scheduler** in a modern OS is the CPU scheduler (or short-term scheduler)
- It essentially decides which process(es) get to run on the CPU core(s) next
- In older systems there was also a long-term or job scheduler but this isn't needed any more

Scheduling Goals:

- Depending on policy, schedulers will try and achieve a degree of fairness and/or a degree of predictability to task scheduling

Linux Scheduling Algorithms:

- **Completely Fair Scheduler (CFS)** (e.g. Linux)
- Since 6.6, Linux uses **earliest eligible virtual deadline first (EEVDF)** scheduling
- iOS traditionally did not allow true multi-tasking and only allowed one foreground app

Scheduling Complexity:

- Schedulers vary a lot in complexity
- The simplest scheduling algorithm is **round-robin** where every runnable task gets an equal turn
- Task priority, user interactiveness, battery life and others are factors
- You could do an entire PhD on scheduling – many have

---

13. RETURNING VALUES FROM PROCESSES

How Processes Return Values:

- Processes can **return** a numeric status to their parent
- Calling `exit(42);` will pass back the value 42 to the parent
- `return 42;` from within main has exactly the same effect

Extracting Return Values:

- The parent can extract the return value using the `WEXITSTATUS` macro on the value set by `wait()`
- If the parent is the bash shell, then `echo $?` will display the value

Code Example:

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

int main() { pid_t p = fork(); if (p == 0) { printf("Child normal return\n"); return 42; } else { int status; wait(&status); if (WIFEXITED(status)) printf("Parent: child normal return value is %d\n", WEXITSTATUS(status)); else printf("Parent: Child terminated abnormally\n"); } return 0; }`

Key Macros:

- `WIFEXITED(status)` - checks if child terminated normally
- `WEXITSTATUS(status)` - extracts the return value (0-255)

KEY TAKEAWAYS

1. **Processes are fundamental:** They represent running programs with their own memory space and execution state
2. **Process lifecycle:** Processes move through states (new → ready → running → waiting/terminated)
3. **OS manages processes:** Through PCBs, queues, and the scheduler
4. **System calls are essential:** `fork()`, `exec()`, `wait()`, and `exit()` are the core process management functions
5. **Memory is automatically cleaned:** OS reclaims all process memory on termination, but good practice is still to free explicitly
6. **Understanding fork is critical:** Each fork doubles the number of processes, creating exponential growth in loops
7. **Signals enable IPC:** Processes can communicate and control each other via signals
8. **Zombies must be reaped:** Always call `wait()` to clean up terminated children
9. **Scheduling is complex:** Modern schedulers balance fairness, performance, and responsiveness
10. **Return values matter:** Processes can communicate success/failure status to their parents

---

PRACTICAL TIPS FOR STUDYING

1. **Practice fork():** Write small programs to understand how parent and child processes behave
2. **Use process monitoring tools:** Experiment with `top`, `ps`, `htop` to see processes in action
3. **Read /proc:** Explore `/proc/<pid>/` directories to understand what information the kernel tracks
4. **Trace system calls:** Use `strace` to see what system calls programs make
5. **Experiment with states:** Write programs that block on I/O to see state transitions
6. **Handle signals:** Practice writing signal handlers for different signals
7. **Check for zombies:** Use `ps aux | grep Z` to find zombie processes
8. **Understand return codes:** Practice checking exit status with `$?` in bash scripts