CS 170 Week 5

Winter 2026

https://sites.cs.ucsb.edu/~rich/class/cs170/

Agenda

1. Mid-quarter TA Evaluation
2. Lab1 Rubric
3. Lab2 Overview
4. Questions

Mid-quarter TA Evaluation

Please fill out the survey before February 12th.

https://forms.gle/UN3qeXsc1ouxgwoa9

Lab 1 Rubric

- General Design: 60 points
  - 10 pts for successful build
  - 10 pts for creating/jumping to process
  - 10 pts for handling arguments correctly
  - 10 pts for scheduler
  - 5 pts for SYS_exit
  - 5 pts for SYS_halt
  - 5 pts for exception handling
  - 5 pts fort interrupt handling
- Read system call: 60 points
- Write system call: 40 points
- Durability: 40 points

Lab 1 Rubric

- General Design: 60 points
- Read system call: 60 points
  - 25 pts for incorrect reading implementation
  - 11 pts for EOF handling
  - 8 points for error checking, per argument (exact errno is not required for full points)
- Write system call: 40 points
  - 25 pts for incorrect writing implementation
  - 5 points for error checking, per argument (exact errno is not required for full points)
- Durability: 40 points
  - 19 pts - No crashes / faults / deadlocks (excluding EOF hang)
  - 3 pts per autograder read/write test (7)

Lab 2: Multiple Processes

Lab2 Overview

Goal: *Multiprogramming* - Support multiple (8) processes concurrently

Lab2 Overview

Goal: *Multiprogramming* - Support multiple (8) processes concurrently

Rough roadmap:

- Restructuring processes
- More system calls

Lab2 Overview

Goal: *Multiprogramming* - Support multiple (8) processes concurrently

Rough roadmap:

- Restructuring processes
  - The main memory will be split in 8 equal sections. Each process gets one.
- More system calls

User “main_memory”

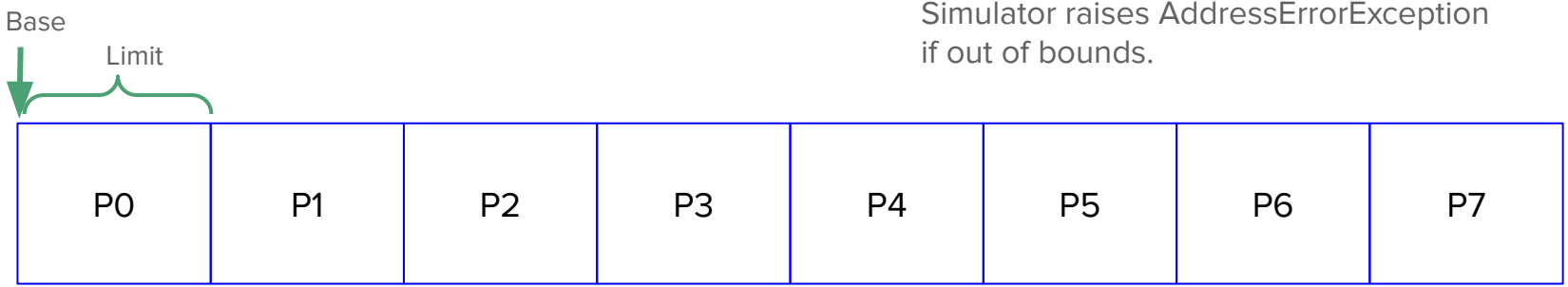
- -12 offset for stack pointer

User “main_memory”

- -12 offset for stack pointer
- How user memory address translates to “real” KOS addresses
  - User_Base: offset in main_memory for that particular process
  - User_Limit: Size in bytes allocated to that particular process
  - ...

User “main_memory”

- -12 offset for stack pointer
- How user memory address translates to “real” KOS addresses
  - User_Base: offset in main_memory for that particular process
  - User_Limit: Size in bytes allocated to that particular process
  - ...

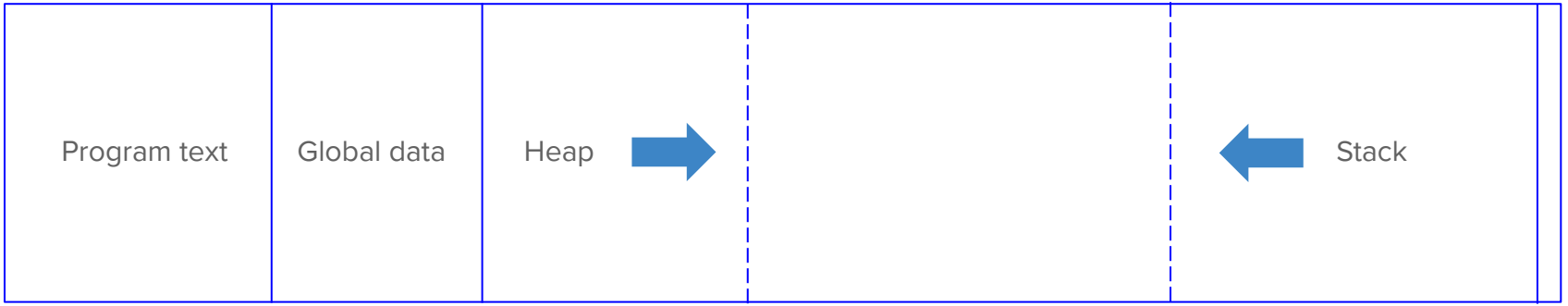


*NOTE: The process numbers (P#) are *not* the pid of the process. It is only used to highlight the memory partitions.

Memory from a user process perspective

Per user process memory (only relevant segments are shown):

P0:



Note: May be demonstrated on the board for better understanding.

Lab2 Overview

Goal: *Multiprogramming* - Support multiple (8) processes concurrently

Rough roadmap:

- Restructuring processes
  - The main memory will be split in 8 equal sections. A process gets one.
- More system calls

Lab2 Overview

Goal: *Multiprogramming* - Support multiple (8) processes concurrently

Rough roadmap:

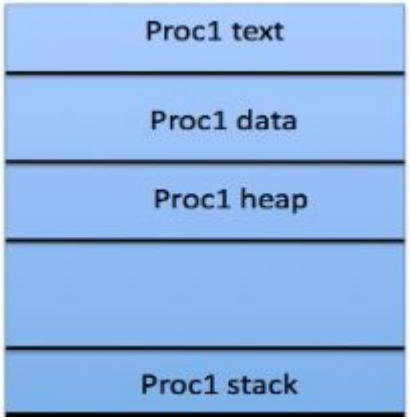
- Restructuring processes
  - The main memory will be split in 8 equal sections. A process gets one.
- More system calls
  - Major system calls
  - “Housekeeping” system calls

Lab2 Overview – System calls

- **Major system calls**
  - **fork()**
  - **execve()**
  - **wait()**
- Housekeeping system calls
  - sbrk()
  - ioctl()
  - fstat()
  - getpagesize()
  - getpid()
  - getppid()
  - getdtablesize()
  - close()

Creating new processes: fork (part 1)

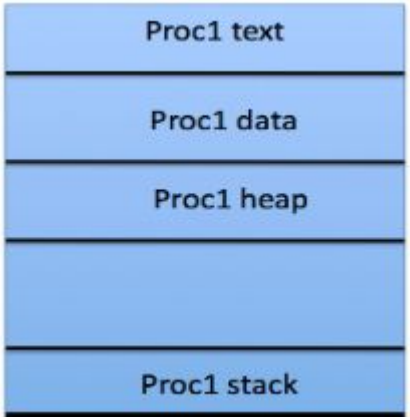
Parent



Register Values
(and more)

Creating new processes: fork (part 1)

Parent



Register Values
(and more)

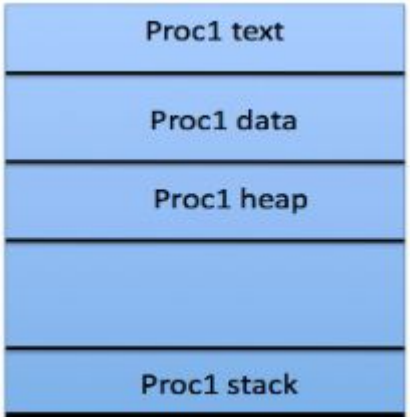
Child

Register Values
(and more)

NOTE: Unix/Unix-like systems use this mechanism (i.e. macOS/OS X, Linux, and KOS). Windows do something different.

Creating new processes: fork (part 1)

Parent



Register Values (and more)

"Clones"/copy almost everything (except PID).

Child

Register Values (and more)

NOTE: Unix/Unix-like systems use this mechanism (i.e. macOS/OS X, Linux, and KOS). Windows do something different.

Creating new processes: exec (optional part 2)

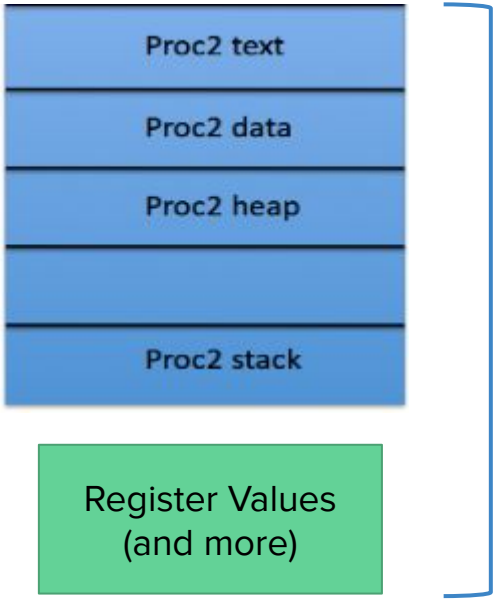
If the intent is to run a *different* program, **exec(ve)** is called after forking.

Register Values
(and more)

NOTE: Unix/Unix-like systems use this mechanism (i.e. macOS/OS X, Linux, and KOS). Windows do something different.

Creating new processes: exec (optional part 2)

If the intent is to run a *different* program, **exec(ve)** is called after forking.



“Everything” is cleared and the program text is replaced by the new program. PID is preserved.

NOTE: Unix/Unix-like systems use this mechanism (i.e. macOS/OS X, Linux, and KOS). Windows do something different.

Creating new processes: exec (optional part 2)

If the intent is to run a *different* program, **exec(ve)** is called ~~after forking~~.

Register Values
(and more)

exec does not require forking to work. You can call exec by itself!

NOTE: Unix/Unix-like systems use this mechanism (i.e. macOS/OS X, Linux, and KOS). Windows do something different.

Parent-child relationship

Because of the fork/exec mechanism, parent-child processes form a “hierarchical tree.”

Parent-child relationship

Because of the fork/exec mechanism, parent-child processes form a “hierarchical tree.”

This is required for the wait syscall. More on this later (or next week)

Parent-child relationship

Because of the fork/exec mechanism, parent-child processes form a “hierarchical tree.”

This is required for the wait syscall. More on this later (or next week)

Question: If the only way a process is created is by a fork (and execve), where does the first process come from? 🤔

Parent-child relationship

Because of the fork/exec mechanism, parent-child processes form a “hierarchical tree.”

This is required for the wait syscall. More on this later (or next week)

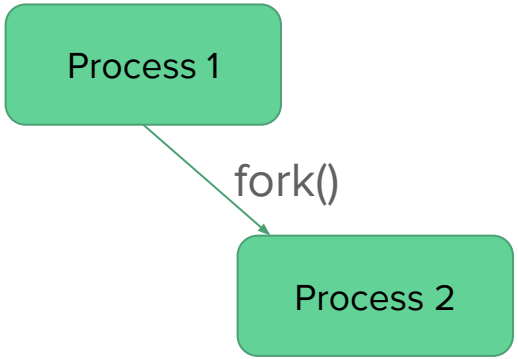
Question: If the only way a process is created is by a fork (and execve), where does the first process come from? 🤔

Answer: The first process (typically PID 1) is spawned by the OS kernel. In KOS, the first process is determined by the “-a” flag.

Wait syscall

The wait() syscall

Basic terms: wait for a child process to finish

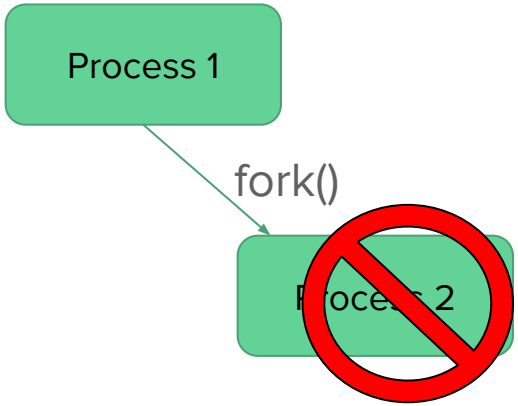


If process 1 calls wait(), it will block until process 2 completes.

Wait, a zombie apocalypse?

The wait() syscall

Basic terms: wait for a child process to finish

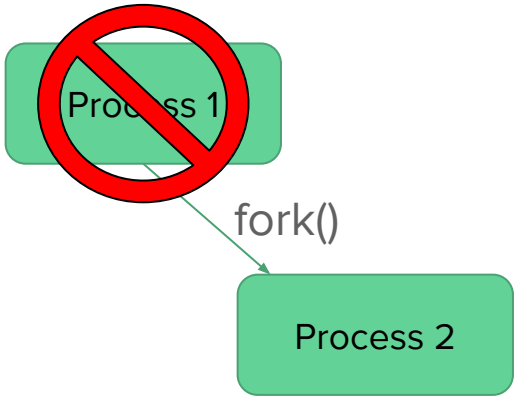


What if process 1 never calls wait?
And, process 2 exits?

Process 2 is now a “zombie”, as the OS must still remember the PCB. *Why???*

Wait, orphans?

The wait() syscall

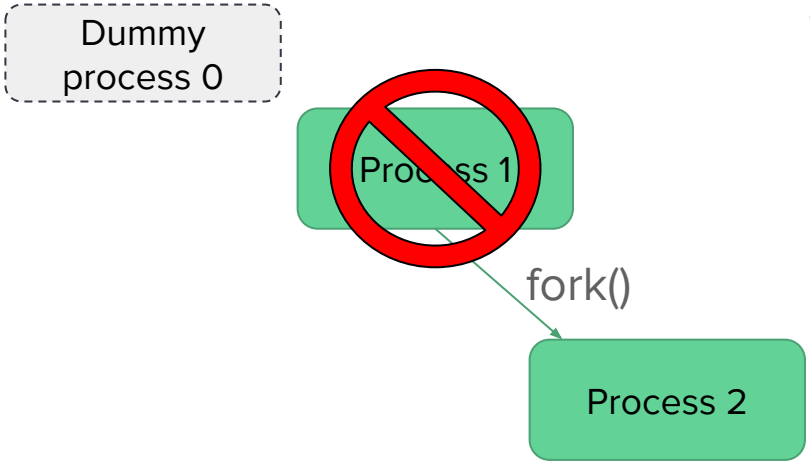


What if process 1 exits before calling wait()?

Process 2 is now an orphan. What to do for parent?

Wait, orphans?

The wait() syscall



What if process 1 exits before calling wait()?

Process 2 is now an orphan. What to do for parent?

Ah, we could just say P2 has a parent of a dummy PCB with a dummy process ID (like 0)

Lab2 Overview – System calls

- Major system calls
  - fork()
  - execve()
  - wait()
- Housekeeping system calls
  - sbrk()
  - ioctl()
  - fstat()
  - getpagesize()
  - getpid()
  - getppid()
  - getdtablesize()
  - close()

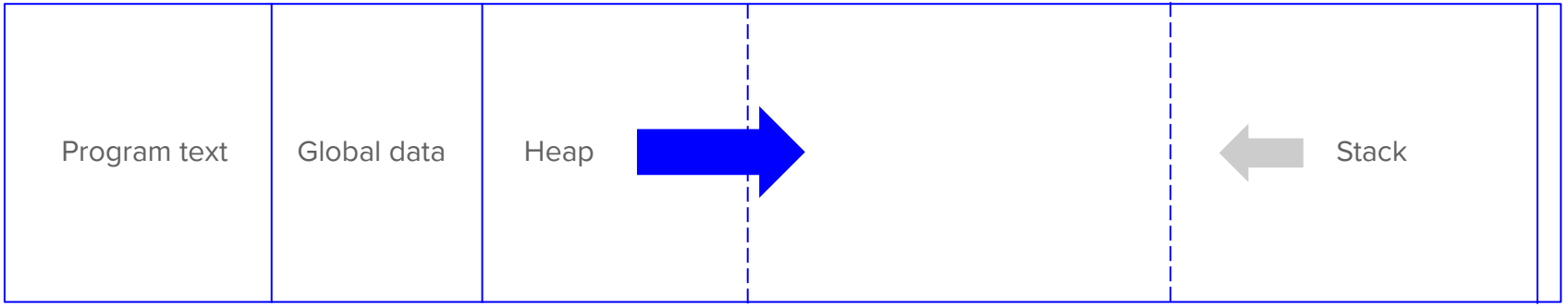
Lab2 Overview – System calls

- Major system calls
  - fork()
  - execve()
  - wait()
- **Housekeeping system calls**
  - **sbrk()**
  - **ioctl()**
  - **fstat()**
  - **getpagesize()**
  - **getpid()**
  - **getppid()**
  - **getdtablesize()**
  - **close()**

sbrk()

Increment (or decrement) heap size.

P0:



Note: May be demonstrated on the board for better understanding.

Lab2 Overview – System calls

- The remainder act as “pass through” to simulator / global values.
  - ioctl()
  - fstat()
  - getpagesize()
  - getpid()
  - getppid()
  - getdtablesize()
  - close()

See website: https://sites.cs.ucsb.edu/~rich/class/cs170/labs/kos_mp_v2/index.html

Time sharing

One function of the OS is the ensure every process is able get fair* cpu time.

*The concept of “fair” in OS is subjective. That is why different scheduling strategies exist and even well adopted policies change sometimes (i.e. EEVDF scheduler was only recently implemented in the Linux kernel)

Time sharing

One function of the OS is the ensure every process is able get fair* cpu time.

What really happens:

*The concept of “fair” in OS is subjective. That is why different scheduling strategies exist and even well adopted policies change sometimes (i.e. EEVDF scheduler was only recently implemented in the Linux kernel)

Time sharing

One function of the OS is the ensure every process is able get fair* cpu time.

What really happens:

1. The OS configures a (hardware) timer during initialization

*The concept of “fair” in OS is subjective. That is why different scheduling strategies exist and even well adopted policies change sometimes (i.e. EEVDF scheduler was only recently implemented in the Linux kernel)

Time sharing

One function of the OS is the ensure every process is able get fair* cpu time.

What really happens:

1. The OS configures a (hardware) timer during initialization
2. The timer trigger an interrupt at set intervals

*The concept of “fair” in OS is subjective. That is why different scheduling strategies exist and even well adopted policies change sometimes (i.e. EEVDF scheduler was only recently implemented in the Linux kernel)

Time sharing

One function of the OS is the ensure every process is able get fair* cpu time.

What really happens:

1. The OS configures a (hardware) timer during initialization
2. The timer trigger an interrupt at set intervals
3. The OS handles the interrupt

*The concept of “fair” in OS is subjective. That is why different scheduling strategies exist and even well adopted policies change sometimes (i.e. EEVDF scheduler was only recently implemented in the Linux kernel)

Time sharing

One function of the OS is the ensure every process is able get fair* cpu time.

What really happens:

1. The OS configures a (hardware) timer during initialization
2. The timer trigger an interrupt at set intervals
3. The OS handles the interrupt
4. The OS schedule processes according to some concept of “fairness”
    a. The reference KOS uses a simple round robin scheduling algorithm

*The concept of “fair” in OS is subjective. That is why different scheduling strategies exist and even well adopted policies change sometimes (i.e. EEVDF scheduler was only recently implemented in the Linux kernel)

Time sharing

One function of the OS is the ensure every process is able get fair* cpu time.

What really happens:

1. The OS configures a (hardware) timer during initialization
2. The timer trigger an interrupt at set intervals
3. The OS handles the interrupt
4. The OS schedule processes according to some concept of “fairness”
    a. The reference KOS uses a simple round robin scheduling algorithm
5. Repeat step 2-4

*The concept of “fair” in OS is subjective. That is why different scheduling strategies exist and even well adopted policies change sometimes (i.e. EEVDF scheduler was only recently implemented in the Linux kernel)

ksh and background tasks

One of the test executables is a shell (ksh).

ksh and background tasks

One of the test executables is a shell (ksh).

The “&” tells the shell to launch the process in the background (i.e. “hw &”)

ksh and background tasks

One of the test executables is a shell (ksh).

The “&” tells the shell to launch the process in the background (i.e. “hw &”)

Launch programs in the background to observe time sharing behavior

ksh and background tasks

One of the test executables is a shell (ksh).

The “&” tells the shell to launch the process in the background (i.e. “hw &”)

Launch programs in the background to observe time sharing behavior

Warning: This is extremely tricky to test! Also, your correct implementation might behave slightly differently than our reference implementation.

ksh and background tasks

One of the test executables is a shell (ksh).

The “&” tells the shell to launch the process in the background (i.e. “hw &”)

Launch programs in the background to observe time sharing behavior

Warning: This is extremely tricky to test! Also, your correct implementation might behave slightly differently than our reference implementation.

Demo ksh background task:

Questions