# K22 – Operating Systems: Design Principles and Internals

Fall 2026 @dit (α.μ. mod 2 = 1)

Vaggelis Atlidakis

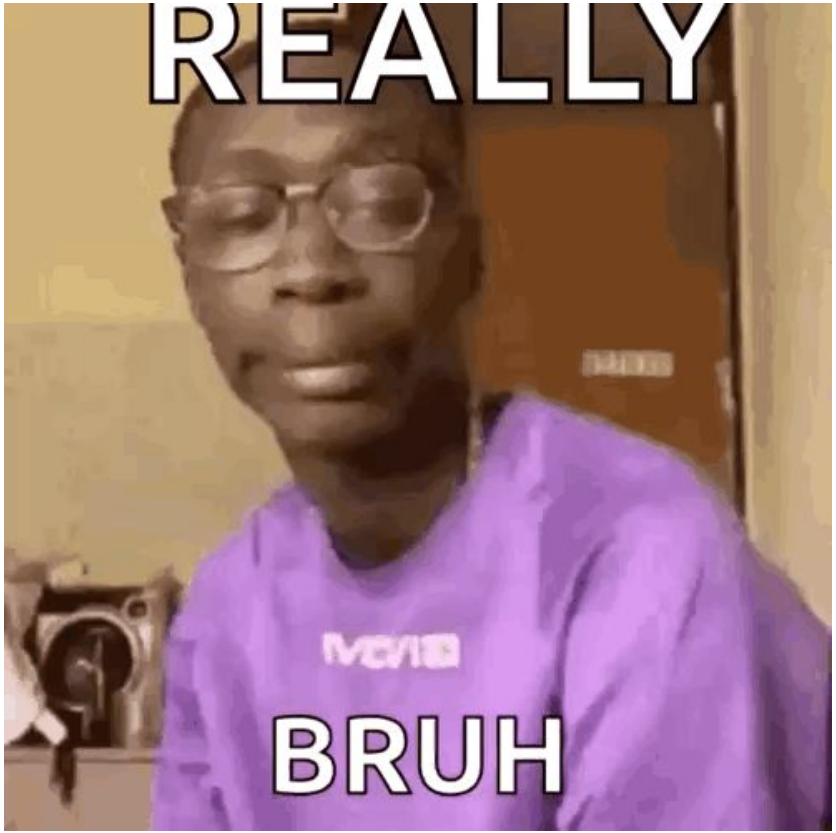
Lecture 05

---

References: Similar OS courses @Columbia, @Stanford, @UC San Diego, @Brown, @di (previous years); and textbooks: Operating Systems: Three Easy Pieces, Operating Systems: Principles and Practice, Operating System Concepts, Linux Kernel Development, Understanding the Linux Kernel

# Quiz 01

# Quiz 01



# Quiz 01



# Overview

- We'll start from hardware and follow a question-oriented approach
  - ~~Intro [Q: What is an OS?]~~
  - ~~Events [Q: When does the OS run?]~~
  - ~~Runtime [Q: How does a program look like in memory?]~~
  - **Processes [Q: What is a process?]**
  - IPC [Q: How do processes communicate?]
  - Threads [Q: What is a thread?]
  - Synchronization [Q: What goes wrong w/o synchronization?]
  - Time Management [Q: What is scheduling?]
  - Memory Management [Q: What is virtual memory?]
  - Files [Q: What is a file descriptor?]
  - Storage Management [Q: How do we allocate disk space to files?]

* Basic (H/W & S/W)
* Abstractions
* Primitives
* Mechanisms

# Overview

- We'll start from hardware and follow a question-oriented approach
  - ~~Intro [Q: What is an OS?]~~
  - ~~Events [Q: When does the OS run?]~~
  - ~~Runtime [Q: How does a program look like in memory?]~~
  - **Processes [Q: What is a process?]**
  - IPC [Q: How do processes communicate?]
  - Threads [Q: What is a thread?]
  - Synchronization [Q: What goes wrong w/o synchronization?]
  - Time Management [Q: What is scheduling?]
  - Memory Management [Q: What is virtual memory?]
  - Files [Q: What is a file descriptor?]
  - Storage Management [Q: How do we allocate disk space to files?]

* Basic (H/W & S/W)
* **Abstractions**
* Primitives
* Mechanisms

# Overview

- **Processes**
  - Q1: What is a process?
  - Q2: Why use a process?
  - Q3: What are the components of a process?
  - Q4: How do we create a process?
  - Q5: How do we run a program?
  - Q6: What are the possible states a process could be in?
  - Q7: How does the OS run multiple processes simultaneously?
  - Q8: What are the steps involved in process creation and process switching in Linux?

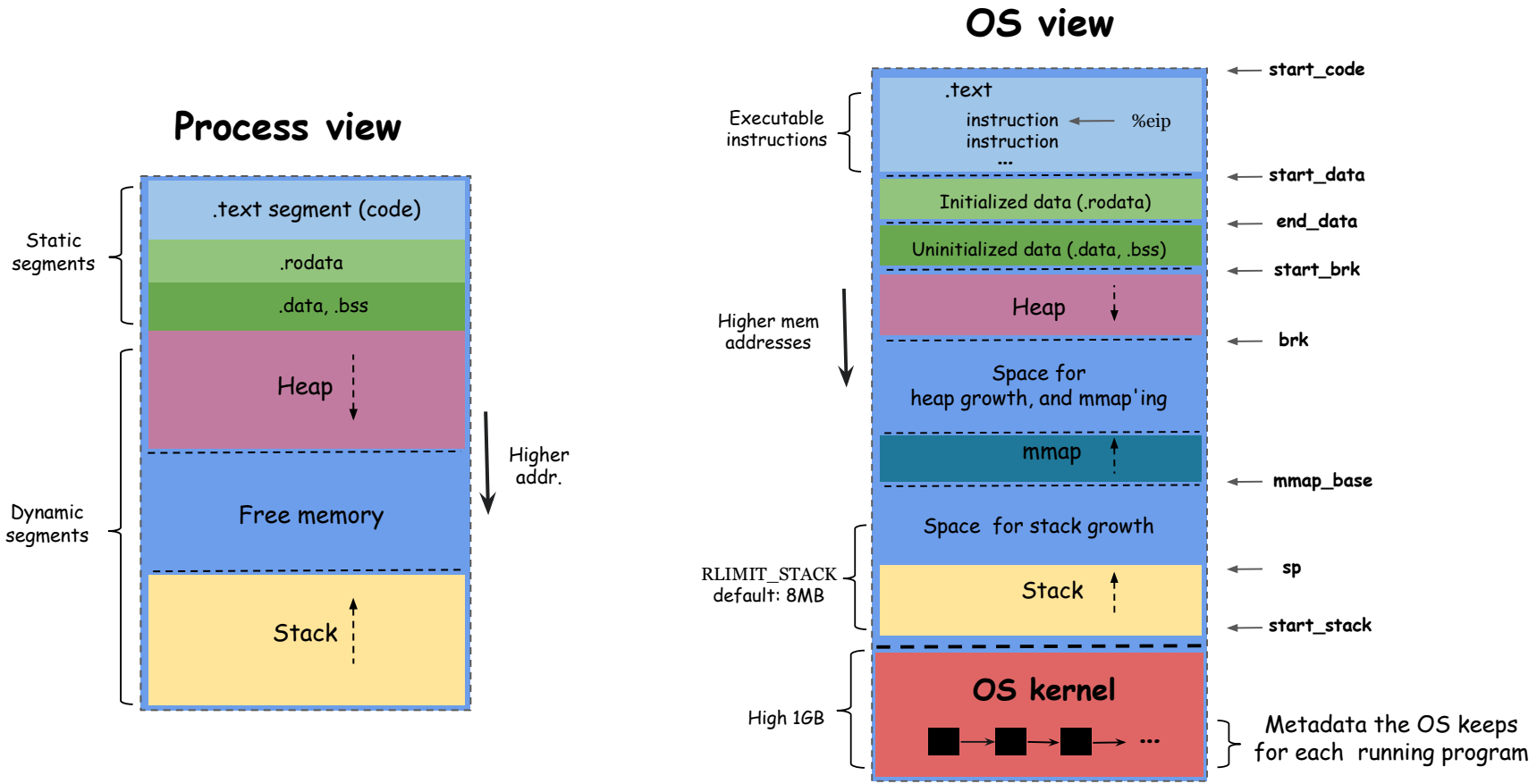
...

# The process abstraction

# The process abstraction

- What is a process? "An address space with one or more threads executing within it." (Strict POSIX definition, 3/189.)

# Virtual Address Space (VAS)



# The process abstraction

- What is a process? "An address space with one or more threads executing within it." (Strict POSIX definition, 3/189.)

- Popularized in 1974, in the context of the Bell Labs paper called "The UNIX Time-Sharing System."

# The process abstraction

- What is a process? "An address space with one or more threads executing within it." (Strict POSIX definition, 3/189.)

- Popularized in 1974, in the context of the Bell Labs paper called "The UNIX Time-Sharing System."

- In simple words: A process is an instance of a program in execution
  - A program is a recipe ⇒ A process is the mess in your kitchen

# The process abstraction

- What is a process? "An address space with one or more threads executing within it." (Strict POSIX definition, 3/189.)

- Popularized in 1974, in the context of the Bell Labs paper called "The UNIX Time-Sharing System."

- In simple words: A process is an instance of a program in execution
  - A program is a recipe ⇒ A process is the mess in your kitchen

- How we use it?
  - Want to run a program? Use a process
  - Want to run multiple programs? Use multiple processes

# The process abstraction

- Why we love it?

# The process abstraction

- Why we love it? The definite illusion: User programs can be written as if they will get isolated access to all system resources

# The process abstraction

- Why we love it? The definite illusion: User programs can be written as if they will get isolated access to all system resources
- Noone else will exist in memory
  - Remember printf("%x | %d\n", &x, x) from quiz01?
  - Two processes may see a different value at the same virt. address

# The process abstraction

- Why we love it? The definite illusion: User programs can be written as if they will get isolated access to all system resources
- Noone else will exist in memory
  - Remember printf("%x | %d\n", &x, x) from quiz01?
  - Two processes may see a different value at the same virt. address
- Noone else will be using the processor
  - Remember preemption and this timer ticking?
  - Every process has its own "virtual" processor time

# The process abstraction

- Why we love it? The definite illusion: User programs can be written as if they will get isolated access to all system resources
- Noone else will exist in memory
  - Remember printf("%x | %d\n", &x, x) from quiz01?
  - Two processes may see a different value at the same virt. address
- Noone else will be using the processor
  - Remember preemption and this timer ticking?
  - Every process has its own "virtual" processor time
- Noone else will be using using the storage
  - This is for later, but the ideas are similar…
  - Every process may read from a file as if no other is writing to it

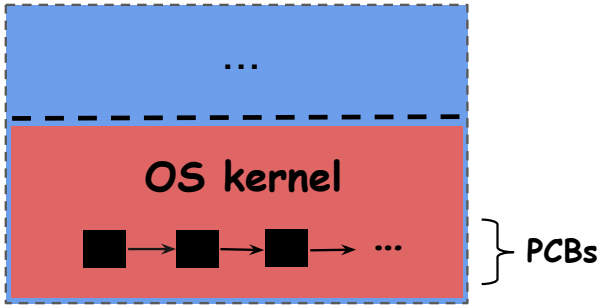
# Components of a process

# Components of a process

- Process Control Block (PCB): A struct used by the OS to keep track of each running process (see task_struct)

# Components of a process

- Process Control Block (PCB): A struct used by the OS to keep track of each running process (see task_struct)



# Components of a process

- Process Control Block (PCB): A struct used by the OS to keep track of each running process (see task_struct)
  - Virtual Address Space (see struct mm): A linear array of bytes with all static and dynamic segments in virtual memory of a running process

# Components of a process

- Process Control Block (PCB): A struct used by the OS to keep track of each running process (see task_struct)
  - Virtual Address Space (see struct mm): A linear array of bytes with all static and dynamic segments in virtual memory of a running process
  - Execution state (see thread_struct): An instruction pointer, a stack pointer, and a set of general-purpose registers with their values

# Components of a process

- Process Control Block (PCB): A struct used by the OS to keep track of each running process (see task_struct)
  - Virtual Address Space (see struct mm): A linear array of bytes with all static and dynamic segments in virtual memory of a running process
  - Execution state (see thread_struct): An instruction pointer, a stack pointer, and a set of general-purpose registers with their values
  - Control metadata: Scheduling (see sched_entity and sched_statistics), identity (see struct cred), and more...

# Components of a process

- Process Control Block (PCB): A struct used by the OS to keep track of each running process (see task_struct)
  - Virtual Address Space (see struct mm): A linear array of bytes with all static and dynamic segments in virtual memory of a running process
  - Execution state (see thread_struct): An instruction pointer, a stack pointer, and a set of general-purpose registers with their values
  - Control metadata: Scheduling (see sched_entity and sched_statistics), identity (see struct cred), and more...
  - Shared system resources: Open files (see files_struct) and open network connections (see struct sock), and more...

# Process creation: To POSIX or not to POSIX?

# **Process creation:** To POSIX or not to POSIX?

- Create a new process by cloning an existing process

# **Process creation:** To POSIX or not to POSIX?

- Create a new process by cloning an existing process
  - Pause the current process and save its state

# **Process creation:** To POSIX or not to POSIX?

- Create a new process by cloning an existing process
  - Pause the current process and save its state
  - Duplicate its address space and its PCB

# **Process creation:** To POSIX or not to POSIX?

- Create a new process by cloning an existing process
  - Pause the current process and save its state
  - Duplicate its address space and its PCB
  - Add the new PCB to the in-kernel queue of PCBs

# **Process creation:** To POSIX or not to POSIX?

- Create a new process by cloning an existing process
  - Pause the current process and save its state
  - Duplicate its address space and its PCB
  - Add the new PCB to the in-kernel queue of PCBs
  - Use the return value to diverge

# **Process creation:** To POSIX or not to POSIX?

- Create a new process by cloning an existing process
  - Pause the current process and save its state
  - Duplicate its address space and its PCB
  - Add the new PCB to the in-kernel queue of PCBs
  - Use the return value to diverge
  - Requires distinguished init process…

# Process creation: To POSIX

**DESCRIPTION**

pid_t fork(void): Creates a new process by duplicating the calling process

# Process creation: To POSIX

**DESCRIPTION**

pid_t fork(void): Creates a new process by duplicating the calling process

- Returns process ID (PID) of the new process in the "parent" process
- Returns 0 in the "child" process

# Process creation: To POSIX

**DESCRIPTION**

pid_t fork(void): Creates a new process by duplicating the calling process

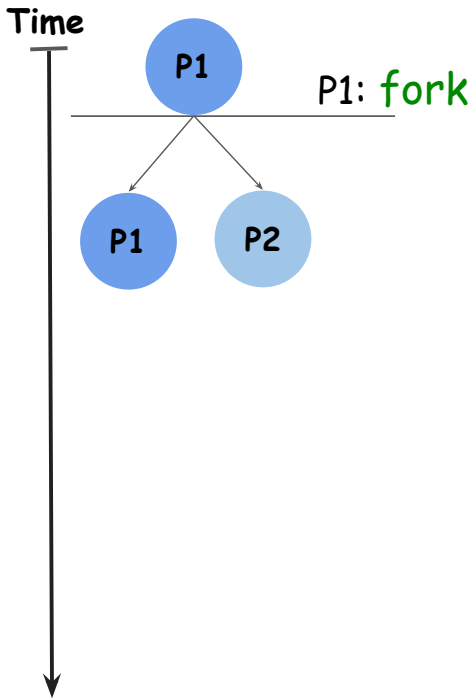
- Returns process ID (PID) of the new process in the "parent" process
- Returns 0 in the "child" process

```
SYSCALL_DEFINE0(fork)
{
    ...
    return kernel_clone(&args);
}
```

# Process creation: To POSIX

int fork ()

- **Creates a new process by duplicating the calling process**
- Returns: Pid of the new process in "parent" process
- Returns: 0 in the "child" process



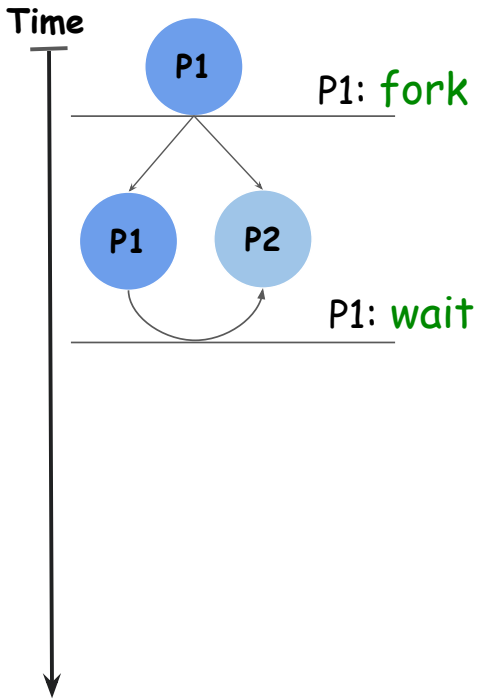
# Process creation: To POSIX

int fork ()

- **Creates a new process by duplicating the calling process**
- Returns: Pid of the new process in "parent" process
- Returns: 0 in the "child" process

int waitpid (int pid, int *stat, ...) ⇐ Parent calls this

- **pid**: process to wait for, or -1 for all
- **stat**: will contain exit value, or signal
- Returns process ID, or -1 on error



# Process creation: To POSIX

int fork ()

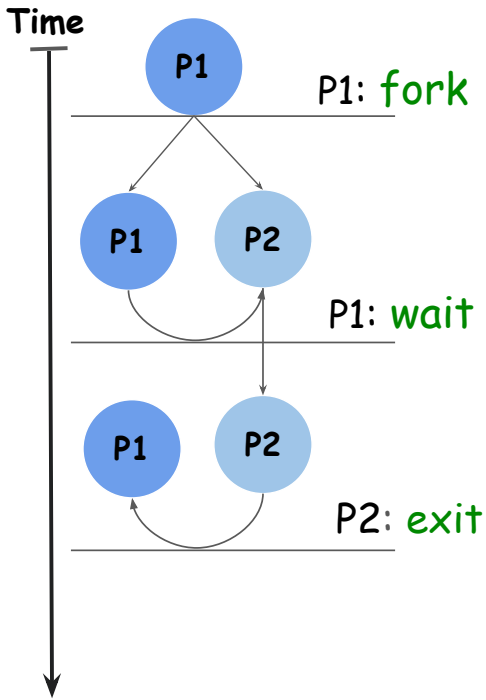
- **Creates a new process by duplicating the calling process**
- Returns: Pid of the new process in "parent" process
- Returns: 0 in the "child" process

int waitpid (int pid, int *stat, ...) ⇐ Parent calls this

- **pid**: process to wait for, or -1 for all
- **stat**: will contain exit value, or signal
- Returns process ID, or -1 on error

void exit (int status)

- **status**: shows up in waitpid (shifted)
- Current process ceases to exist
- Convention: pass 0 on success, non-zero on error



# Process creation: To POSIX

int fork ()

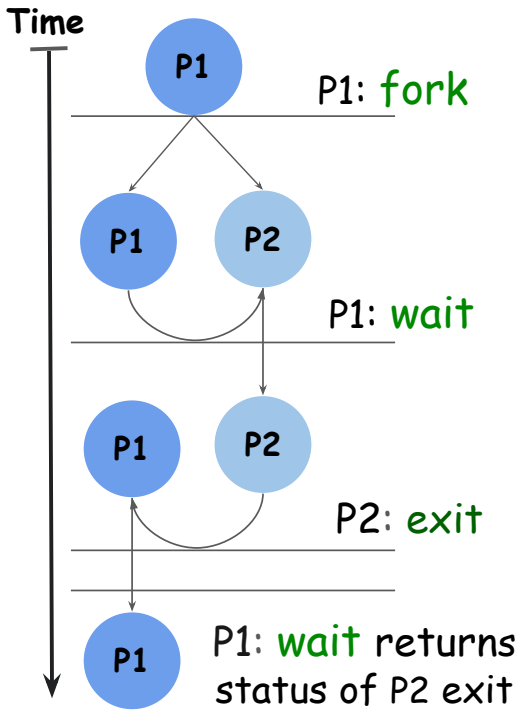
- **Creates a new process by duplicating the calling process**
- Returns: Pid of the new process in "parent" process
- Returns: 0 in the "child" process

int waitpid (int pid, int *stat, ...) ⇐ Parent calls this

- **pid**: process to wait for, or -1 for all
- **stat**: will contain exit value, or signal
- Returns process ID, or -1 on error

void exit (int status)

- **status**: shows up in waitpid (shifted)
- Current process ceases to exist
- Convention: pass 0 on success, non-zero on error



# Process creation: To POSIX or not to POSIX?

- Create a new process by cloning an existing process
  - Pause the current process and save its state
  - Duplicate its address space and its PCB
  - Add the new PCB to the in-kernel queue of PCBs
  - Use the return value to diverge
  - Requires distinguished init process...
- Creating a new process from scratch

# Process creation: To POSIX or not to POSIX?

- Create a new process by cloning an existing process
  - Pause the current process and save its state
  - Duplicate its address space and its PCB
  - Add the new PCB to the in-kernel queue of PCBs
  - Use the return value to diverge
  - Requires distinguished init process...
- Creating a new process from scratch
  - Create and initialize a new PCB

# Process creation: To POSIX or not to POSIX?

- Create a new process by cloning an existing process
  - Pause the current process and save its state
  - Duplicate its address space and its PCB
  - Add the new PCB to the in-kernel queue of PCBs
  - Use the return value to diverge
  - Requires distinguished init process...
- Creating a new process from scratch
  - Create and initialize a new PCB
  - Add a new PCB to the in-kernel queue of PCBs

# Process creation: To POSIX or not to POSIX?

- Create a new process by cloning an existing process
  - Pause the current process and save its state
  - Duplicate its address space and its PCB
  - Add the new PCB to the in-kernel queue of PCBs
  - Use the return value to diverge
  - Requires distinguished init process…
- Creating a new process from scratch
  - Create and initialize a new PCB
  - Add a new PCB to the in-kernel queue of PCBs
  - Does not require a distinguished process…but….

# Process creation: Not to POSIX

**DESCRIPTION**

CreateProcessA: Creates a new process and its primary thread

```
BOOL CreateProcessA(
        [in, optional]          LPCSTR                  lpApplicationName,
        [in, out, optional]     LPSTR                   lpCommandLine,
        [in, optional]          LPSECURITY_ATTRIBUTES   lpProcessAttributes,
        [in, optional]          LPSECURITY_ATTRIBUTES   lpThreadAttributes,
        [in]                    BOOL                    bInheritHandles,
        [in]                    DWORD                   dwCreationFlags,
        [in, optional]          LPVOID                  lpEnvironment,
        [in, optional]          LPCSTR                  lpCurrentDirectory,
        [in]                    LPSTARTUPINFOA          lpStartupInfo,
        [out]                   LPPROCESS_INFORMATION   lpProcessInformation
);
```

# The elegant simplicity of POSIX!

**DESCRIPTION**

**pid_t fork(void)**: Creates a new process by duplicating the calling process

- Returns process ID of new process in the "parent" process
- Returns 0 in the "child" process

```
SYSCALL_DEFINE0(fork)
{
        ...
        return kernel_clone(&args);
}
```