CSE 4733/6733 - Operating System 1

S. Torri, T. Ritter

Mississippi State University

CSE 4733/6733 - Operating System 1

S. Torri, T. Ritter

Mississippi State University

Introduction

Introduction

Definition of an Operating System

- An Operating System (OS) is software that intermediates computer hardware and the user.
- It manages hardware resources and provides an environment for applications to run.
- Examples: Windows, Linux, macOS.

Definition of an Operating System
- An Operating System (OS) is software that intermediates computer hardware and the user.
- It manages hardware resources and provides an environment for applications to run.
- Examples: Windows, Linux, macOS.

In this slide, I introduce the concept of an operating system. The critical point is that the OS is a layer of software that sits between the hardware and the user or applications. This intermediary role is crucial as it allows users to interact with the computer without needing to understand the intricacies of the hardware. Common examples like Windows, Linux, and macOS help to ground the concept in familiar territory.

Primary Functions of an Operating System

- **Process Management**
- **Memory Management**
- **File System Management**
- **I/O System Management**
- **Security and Access Control**

Primary Functions of an Operating System

Process Management
Memory Management
File System Management
I/O System Management
Security and Access Control

- **Process Management:** Controls the execution of processes, including scheduling, creation, and termination.
- **Memory Management:** Manages the system's memory, including allocation and deallocation.
- **File System Management:** Organizes, stores, retrieves, and manages data on storage devices.
- **I/O System Management:** Manages input/output devices, facilitating communication between hardware and applications.
- **Security and Access Control:** Ensures data and resource security, enforcing access control policies.

Why Study Operating Systems?

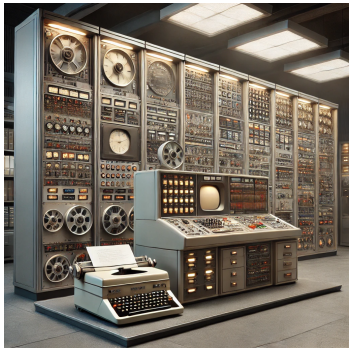
- Understanding OS principles is crucial for developing efficient, secure, and scalable software.
- Knowledge of OS internals aids in troubleshooting and optimizing system performance.
- OS concepts are foundational to other areas of computer science, such as distributed systems, networking, and cybersecurity.

Why Study Operating Systems?
- Understanding OS principles is crucial for developing efficient, secure, and scalable software.
- Knowledge of OS internals aids in troubleshooting and optimizing system performance.
- OS concepts are foundational to other areas of computer science, such as distributed systems, networking, and cybersecurity.

This slide addresses the importance of studying operating systems. The first point emphasizes that an understanding of OS principles is key to developing robust software. The second point highlights the practical benefits of this knowledge, particularly in system optimization and troubleshooting. Finally, the last point connects OS concepts to broader areas in computer science, reinforcing the foundational role of operating systems.

History

History of OS development

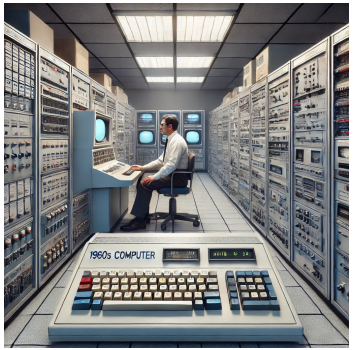


- Early 1950s: Bare-Metal era
- Late 1950s: Batch Processing Systems
- 1960s: Multiprogramming and Time-Sharing Systems
- 1970s: The UNIX Era
- 1980s: The Rise of Personal Computer Operating Systems
- 1990s: The Windows Era and the Birth of Linux
- 2000s: The Rise of Mobile and Embedded Operating Systems
- 2010s: Cloud Computing and Virtualization

History of OS development
- Early 1950s: Bare-Metal era
- Late 1950s: Batch Processing Systems
- 1960s: Multiprogramming and Time-Sharing Systems
- 1970s: The UNIX Era
- 1980s: The Rise of Personal Computer Operating Systems
- 1990s: The Windows Era and the Birth of Linux
- 2000s: The Rise of Mobile and Embedded Operating Systems
- 2010s: Cloud Computing and Virtualization

1. **Early 1950s: Bare-Metal era** - In the early days of computing, there were no operating systems. Programs ran directly on the hardware, with programmers manually setting switches to control the hardware. This era is often referred to as the "bare-metal" era.
2. **Late 1950s: Batch Processing Systems** - As computers became more common, there was a need for more efficient use of expensive resources. Batch processing systems were introduced, where jobs (programs to be run) were grouped and run as a batch, reducing the setup time required between jobs.

History of OS development



- Early 1950s: Bare-Metal era
- Late 1950s: Batch Processing Systems
- 1960s: Multiprogramming and Time-Sharing Systems
- 1970s: The UNIX Era
- 1980s: The Rise of Personal Computer Operating Systems
- 1990s: The Windows Era and the Birth of Linux
- 2000s: The Rise of Mobile and Embedded Operating Systems
- 2010s: Cloud Computing and Virtualization

History of OS development
- Early 1950s: Bare-Metal era
- Late 1950s: Batch Processing Systems
- 1960s: Multiprogramming and Time-Sharing Systems
- 1970s: The UNIX Era
- 1980s: The Rise of Personal Computer Operating Systems
- 1990s: The Windows Era and the Birth of Linux
- 2000s: The Rise of Mobile and Embedded Operating Systems
- 2010s: Cloud Computing and Virtualization

1. **1960s: Multiprogramming and Time-Sharing Systems**
   - The 1960s saw the development of multiprogramming systems, which allowed multiple programs to be in memory simultaneously and was a significant step towards more efficient CPU use.
   - Time-sharing systems were also developed, allowing multiple users to interact with a computer simultaneously. A notable example is the Compatible Time-Sharing System (CTSS) developed at MIT.

History of OS development

- Early 1950s: Bare-Metal era
- Late 1950s: Batch Processing Systems
- 1960s: Multiprogramming and Time-Sharing Systems
- 1970s: The UNIX Era
- 1980s: The Rise of Personal Computer Operating Systems
- 1990s: The Windows Era and the Birth of Linux
- 2000s: The Rise of Mobile and Embedded Operating Systems
- 2010s: Cloud Computing and Virtualization

History of OS development
Early 1950s: Bare-Metal era
Late 1950s: Batch Processing Systems
1960s: Multiprogramming and Time-Sharing Systems
1970s: The UNIX Era
1980s: The Rise of Personal Computer Operating Systems
1990s: The Windows Era and the Birth of Linux
2000s: The Rise of Mobile and Embedded Operating Systems
2010s: Cloud Computing and Virtualization

1. **1970s: The UNIX Era** - In 1969, Ken Thompson, Dennis Ritchie, and others at AT&T Bell Labs developed UNIX, an operating system that was written in the C programming language, making it portable across different hardware platforms. UNIX introduced many concepts that are still in use today, such as a hierarchical file system and a command-line interface.

History of OS development

- Early 1950s: Bare-Metal era
- Late 1950s: Batch Processing Systems
- 1960s: Multiprogramming and Time-Sharing Systems
- 1970s: The UNIX Era
- 1980s: The Rise of Personal Computer Operating Systems
- 1990s: The Windows Era and the Birth of Linux
- 2000s: The Rise of Mobile and Embedded Operating Systems
- 2010s: Cloud Computing and Virtualization

History of OS development
- Early 1950s: Bare-Metal era
- Late 1950s: Batch Processing Systems
- 1960s: Multiprogramming and Time-Sharing Systems
- 1970s: The UNIX Era
- 1980s: The Rise of Personal Computer Operating Systems
- 1990s: The Windows Era and the Birth of Linux
- 2000s: The Rise of Mobile and Embedded Operating Systems
- 2010s: Cloud Computing and Virtualization

1. **1980s: The Rise of Personal Computer Operating Systems** - The 1980s saw the rise of operating systems for personal computers. Notable examples include MS-DOS (Microsoft Disk Operating System), developed by Microsoft for IBM PCs, and the original Mac OS for Apple Macintosh computers.

History of OS development

- Early 1950s: Bare-Metal era
- Late 1950s: Batch Processing Systems
- 1960s: Multiprogramming and Time-Sharing Systems
- 1970s: The UNIX Era
- 1980s: The Rise of Personal Computer Operating Systems
- 1990s: The Windows Era and the Birth of Linux
- 2000s: The Rise of Mobile and Embedded Operating Systems
- 2010s: Cloud Computing and Virtualization

History of OS development
Early 1950s: Bare-Metal era
Late 1950s: Batch Processing Systems
1960s: Multiprogramming and Time-Sharing Systems
1970s: The UNIX Era
1980s: The Rise of Personal Computer Operating Systems
1990s: The Windows Era and the Birth of Linux
2000s: The Rise of Mobile and Embedded Operating Systems
2010s: Cloud Computing and Virtualization

1. **1990s: The Windows Era and the Birth of Linux**
   - In the 1990s, Microsoft Windows became the dominant operating system for personal computers. Windows 95, released in 1995, was a significant release that included a graphical user interface integrated with the operating system.
   - In 1991, Linus Torvalds released the first version of the Linux kernel. Linux is a UNIX-like operating system that is free and open source. It has since become a popular choice for servers and other large systems.

History of OS development

- Early 1950s: Bare-Metal era
- Late 1950s: Batch Processing Systems
- 1960s: Multiprogramming and Time-Sharing Systems
- 1970s: The UNIX Era
- 1980s: The Rise of Personal Computer Operating Systems
- 1990s: The Windows Era and the Birth of Linux
- 2000s: The Rise of Mobile and Embedded Operating Systems
- 2010s: Cloud Computing and Virtualization

History of OS development
- Early 1950s: Bare-Metal era
- Late 1950s: Batch Processing Systems
- 1960s: Multiprogramming and Time-Sharing Systems
- 1970s: The UNIX Era
- 1980s: The Rise of Personal Computer Operating Systems
- 1990s: The Windows Era and the Birth of Linux
- 2000s: The Rise of Mobile and Embedded Operating Systems
- 2010s: Cloud Computing and Virtualization

1. **2000s: The Rise of Mobile and Embedded Operating Systems** - the 2000s saw the development of new operating systems for mobile devices. Notable examples include https://www.techtarget.com/searchmobilecomputing/definition/iOS (developed by Apple), Android (developed by Google), and https://microsoft.fandom.com/wiki/Windows_Mobile (developed by Microsoft).

History of OS development

- Early 1950s: Bare-Metal era
- Late 1950s: Batch Processing Systems
- 1960s: Multiprogramming and Time-Sharing Systems
- 1970s: The UNIX Era
- 1980s: The Rise of Personal Computer Operating Systems
- 1990s: The Windows Era and the Birth of Linux
- 2000s: The Rise of Mobile and Embedded Operating Systems
- 2010s: Cloud Computing and Virtualization

History of OS development
- Early 1950s: Bare-Metal era
- Late 1950s: Batch Processing Systems
- 1960s: Multiprogramming and Time-Sharing Systems
- 1970s: The UNIX Era
- 1980s: The Rise of Personal Computer Operating Systems
- 1990s: The Windows Era and the Birth of Linux
- 2000s: The Rise of Mobile and Embedded Operating Systems
- 2010s: Cloud Computing and Virtualization

1. **2010s: Cloud Computing and Virtualization** - The 2010s saw the rise of cloud computing, which relies heavily on virtualization—an operating system technology that allows multiple virtual machines to run on a single physical machine. Examples include Amazon's EC2 and Google Cloud Platform.

History of OS development

- Early 1950s: Bare-Metal era
- Late 1950s: Batch Processing Systems
- 1960s: Multiprogramming and Time-Sharing Systems
- 1970s: The UNIX Era
- 1980s: The Rise of Personal Computer Operating Systems
- 1990s: The Windows Era and the Birth of Linux
- 2000s: The Rise of Mobile and Embedded Operating Systems
- 2010s: Cloud Computing and Virtualization

History of OS development
- Early 1950s: Bare-Metal era
- Late 1950s: Batch Processing Systems
- 1960s: Multiprogramming and Time-Sharing Systems
- 1970s: The UNIX Era
- 1980s: The Rise of Personal Computer Operating Systems
- 1990s: The Windows Era and the Birth of Linux
- 2000s: The Rise of Mobile and Embedded Operating Systems
- 2010s: Cloud Computing and Virtualization

1. **2020s: Edge Computing and IoT**: As we move into the 2020s, operating systems are being developed for a wider variety of devices than ever before, including Internet of Things (IoT) devices. Edge computing, which involves processing data closer to where it is generated rather than in a centralized data center, is also becoming increasingly important.

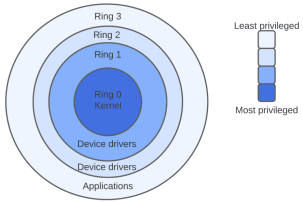
What is the Role of the OS?

- OS abstracts the user from the details of the machine
- Timeshared between processes (jobs)
- Memory - abstracts the details of placement, addressing
- I/O Devices - abstracts disk/network/keyboard/etc.

What is the Role of the OS?
OS abstracts the user from the details of the machine
Timeshared between processes (jobs)
Memory - abstracts the details of placement, addressing
I/O Devices - abstracts disk/network/keyboard/etc.

1. Early OSs were complicated hardware-specific assembler programs
2. Unix was developed in 1969 at Bell Labs, a research lab.
3. Kernel re-written in C, a high-level language with minimal assembler for devices
4. Portablility and readability more important than speed

User versus Kernel mode



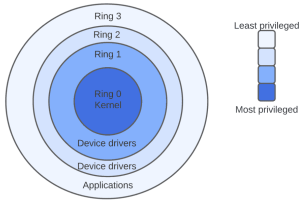
- OS wants user code to run as a user, not have complete access to memory
- Kernel wants to isolate users from other users
- Kernel wants some privileged code ONLY to be run by the kernel

User versus Kernel mode
OS wants user code to run as a user, not have complete access to memory
Kernel wants to isolate users from other users
Kernel wants some privileged code ONLY to be run by the kernel

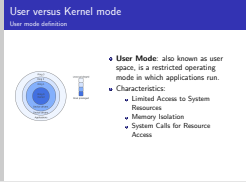
1. Kernel does not want malicious or incorrect programs to crash the system
2. User code can ask for resources/services by making a system call
3. Programs ask for services frequently

User versus Kernel mode

User mode definition



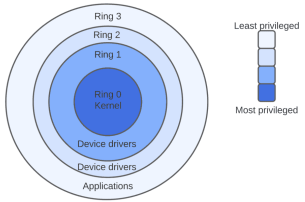
- **User Mode**: also known as user space, is a restricted operating mode in which applications run.
- Characteristics:
  - Limited Access to System Resources
  - Memory Isolation
  - System Calls for Resource Access



1. In user mode, the executing code has no direct access to hardware or memory resources. Instead, it must request access to the operating system via system calls.
2. Programs running in user mode have limited access to system resources. They cannot directly access hardware or reference memory addresses outside their allocated space.
3. Each user-mode process runs in private memory space. This isolation prevents one user-mode process from reading or altering the memory contents of another process.
4. To access hardware or any privileged resource, user-mode processes must request the operating system to perform these tasks on their behalf through system calls.

User versus Kernel mode

Kernel mode definition



- **Kernel Mode**: a.k.a system mode is a privileged operating mode in which the operating system runs.
- Characteristics:
  - Full Access to System Resources
  - Direct Hardware Interaction
  - Critical for System Stability and Security

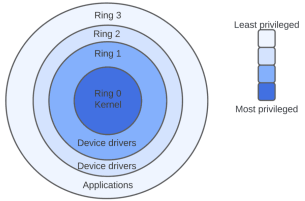
User versus Kernel mode
Kernel mode definition

Kernel Mode: a.k.a system mode is a privileged operating mode in which the operating system runs.
Characteristics:
Full Access to System Resources
Direct Hardware Interaction
Critical for System Stability and Security

1. In kernel mode, the executing code has complete and unrestricted access to the underlying hardware.
2. The operating system, running in kernel mode, has unrestricted access to the system's memory and hardware.
3. In kernel mode, the operating system can directly interact with hardware components, making performing tasks like memory management, process scheduling, and I/O operations possible.
4. Because the kernel has such a high level of control over the system, bugs or security vulnerabilities in the kernel can have severe consequences.

User versus Kernel mode

Important distinctions



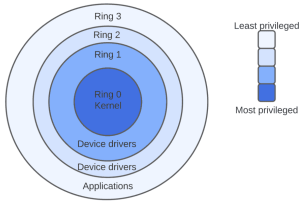
- Security to prevent direct access.
- Process stability
- Policy enforcement in kernel mode
- Error Containment
- Operating system performance

User versus Kernel mode
Important distinctions
Security to prevent direct access.
Process stability
Policy enforcement in kernel mode
Error Containment
Operating system performance

1. By separating user and kernel modes, the operating system can prevent user applications from directly accessing hardware and critical system data. This separation is crucial for maintaining system security.
2. The separation ensures that user applications cannot accidentally or intentionally interfere with each other or the operating system's core parts. This isolation is key to maintaining system stability.
3. The kernel, running in kernel mode, can enforce policies (like access control policies) and manage resources effectively, as it has complete control over all hardware and software on the system.

User versus Kernel mode

Important distinctions



- Security to prevent direct access.
- Process stability
- Policy enforcement in kernel mode
- Error Containment
- Operating system performance

User versus Kernel mode
Important distinctions
Security to prevent direct access.
Process stability
Policy enforcement in kernel mode
Error Containment
Operating system performance

1. If a user-mode application fails or attempts to perform an illegal operation, the impact is generally limited to that single process. The operating system can terminate that process without affecting the rest of the system.
2. Kernel mode allows the operating system to manage and interact with hardware efficiently, making implementing features like multitasking, advanced memory management, and fast I/O operations possible.

Operating System Processes

Operating System Processes

Introduction to Operating System Processes

- How does the OS keep track of different instances of Google Chrome?
- How does the OS allow different processes to run on the same computer?

Introduction to Operating System Processes

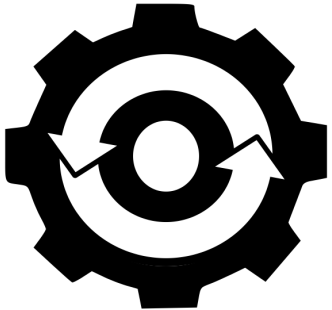
- How does the OS keep track of different instances of Google Chrome?
- How does the OS allow different processes to run on the same computer?

1. Process Control Block(PCB) is a kernel data structure to keep track of running processes, "top" can show you what is running
2. There are user limits on the number of processes a user can run and typically limits on how many a system can run
3. User limits are set with "ulimit"
4. Kernel limits are configuration settings (MAXUSERS)

Process Abstraction

OS provides process abstraction

Ref: M. Vutukuru (2016)



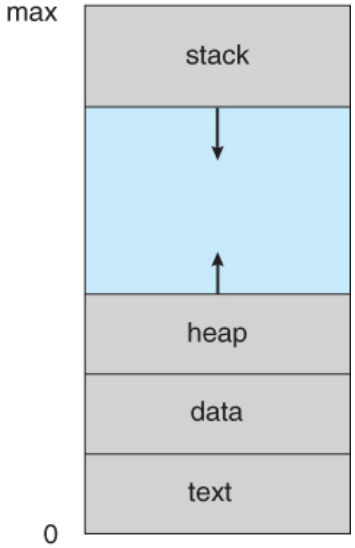
- When you run an application, the OS creates a process (an instance of an application).
- OS shares the CPU across multiple processes by keeping the machine state for each.
- OS has a CPU scheduler that uses a scheduling policy to switch between them. Also called a "context switch."

OS provides process abstraction
Ref: M. Vutukuru (2016)
When you run an application, the OS creates a process (an instance of an application).
OS shares the CPU across multiple processes by keeping the machine state for each.
OS has a CPU scheduler that uses a scheduling policy to switch between them. Also called a "context switch."

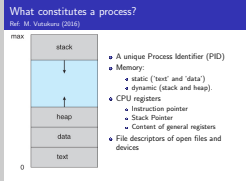
1. Process: an instance of a program that includes its instructions, data, and CPU register information.
2. Machine State: Information that the operating system uses to keep track of the current program state.
3. Scheduling policies tell the operating system how to switch between each process.

What constitutes a process?

Ref: M. Vutukuru (2016)



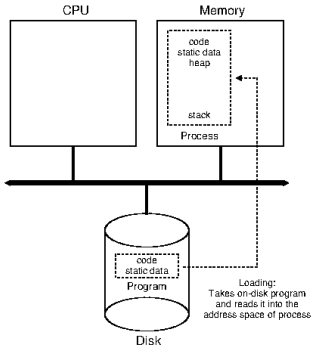
- A unique Process Identifier (PID)
- Memory:
  - static ('text' and 'data')
  - dynamic (stack and heap).
- CPU registers
  - Instruction pointer
  - Stack Pointer
  - Content of general registers
- File descriptors of open files and devices



1. A process identifier in Linux is an integer value the operating system assigns when it starts an application.
2. A memory image contains application instructions and data loaded from the file. It also has a stack or heap.
3. Code: Application instructions found in the file. For example, the '.text' section for a Linux executable
4. Stack: Operating system feature to keep track of functions called. The operating system uses this information to keep track of execution.
5. Heap: Dynamically allocated memory where data can be stored.

How does an OS create a process?

Ref: M. Vutukuru (2016)

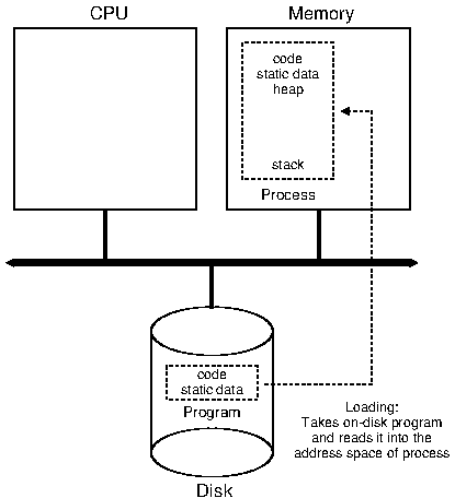


- OS allocates memory for a new process.
- OS reads application code from the file and loads it into the '.text' block.
- OS reads application data from the file and puts it into the '.data' bock.
- It will set up the stack frame with the location of the 'main' function. Also called an 'entry point.'

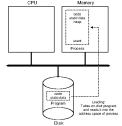
How does an OS create a process?
Ref: M. Vutukuru (2016)
- OS allocates memory for a new process.
- OS reads application code from the file and loads it into the '.text' block.
- OS reads application data from the file and puts it into the '.data' bock.
- It will set up the stack frame with the location of the 'main' function. Also called an 'entry point.'

How does an OS create a process?

Ref: M. Vutukuru (2016)

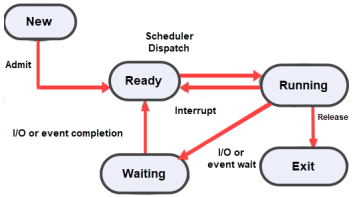


How does an OS create a process?
Ref: M. Vutukuru (2016)



States of a process

Ref: M. Vutukuru (2016)



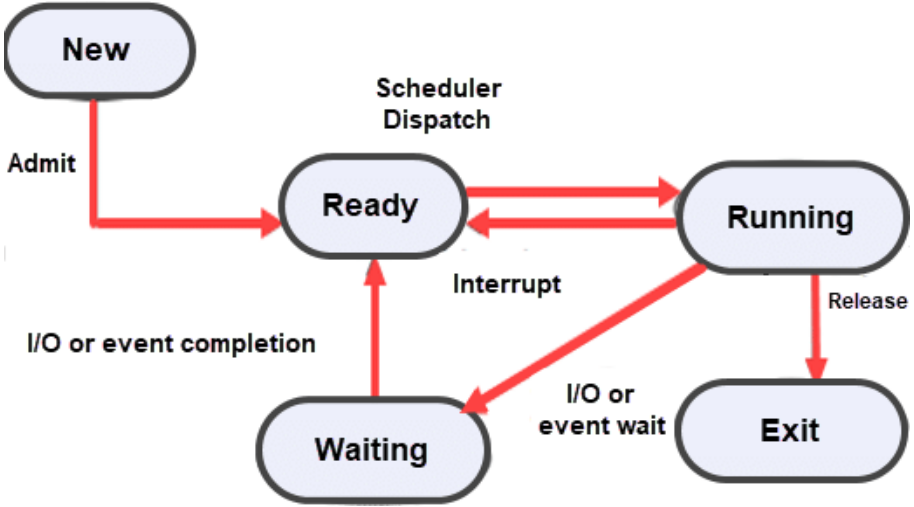
- **Running:** currently executing on CPU.
- **Ready:** waiting to be scheduled.
- **Waiting / Blocked:** Suspended execution
- **New:** Process being set up to execute.
- **Exit:** Process stopped.

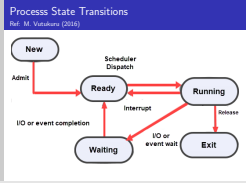
States of a process
Ref: M. Vutukuru (2016)
Running: currently executing on CPU.
Ready: waiting to be scheduled.
Waiting / Blocked: Suspended execution
New: Process being set up to execute.
Exit: Process stopped.

1. When the OS switches to a process, it will set up the CPU registers with the current state in the stack. The special register, instruction pointer, tells the OS where to resume.
2. Blocked processes can be waiting on an event like reading data from a disk. Once that data is retrieved, the process is ready to resume running.
3. When a process is terminated, all of its shared resources are returned to the OS.

Processs State Transitions

Ref: M. Vutukuru (2016)





States of a process

Ref: M. Vutukuru (2016)

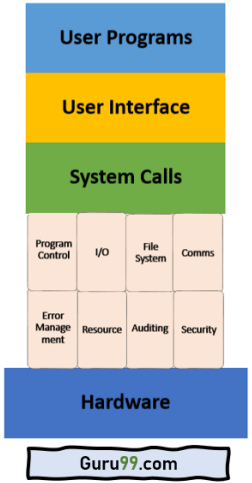
Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Waiting	Running	$Process_0$ is blocked $Process_1$ can start
5	Waiting	Running	
6	Waiting	Running	
7	Ready	Running	I/O for $Process_0$ is done.
8	Ready	Running	$Process_1$ is done.
9	Running	Exit	
10	Exit	-	

States of a process
Ref: M. Vutukuru (2016)
Time | $Process_0$ | $Process_1$ | Notes
1 | Running | Ready |
2 | Running | Ready |
3 | Running | Ready | $Process_0$ initiates I/O
4 | Waiting | Running | $Process_0$ is blocked; $Process_1$ can start
5 | Waiting | Running |
6 | Waiting | Running |
7 | Ready | Running | I/O for $Process_0$ is done.
8 | Ready | Running | $Process_1$ is done.
9 | Running | Exit |
10 | Exit | - |

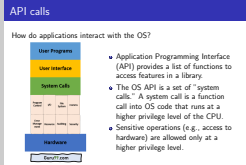
Process API

API calls

How do applications interact with the OS?

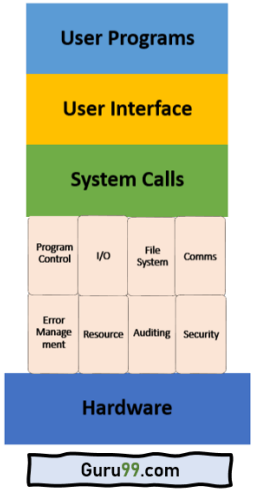


- Application Programming Interface (API) provides a list of functions to access features in a library.
- The OS API is a set of "system calls." A system call is a function call into OS code that runs at a higher privilege level of the CPU.
- Sensitive operations (e.g., access to hardware) are allowed only at a higher privilege level.



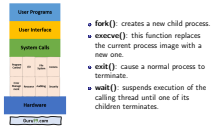
Process system calls in Linux

Ref: M. Vutukuru (2016)



- **fork()**: creates a new child process.
- **execve()**: this function replaces the current process image with a new one.
- **exit()**: cause a normal process to terminate.
- **wait()**: suspends execution of the calling thread until one of its children terminates.

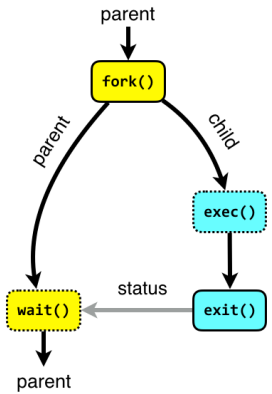
Process system calls in Linux
Ref: M. Vutukuru (2016)



1. Fork() is found in unistd.h. Fork() creates a new process by duplicating the calling process. This new process is called the *child* process. The process that called 'fork()' function is called the *parent* process.
2. The 'execve()' function executes the program referred to by the path name and causes the current program to be replaced with a new program with its stack, heap, and data segments. Any string arguments are passed to the new program. If there is a problem, the value of -1 is returned, and the 'errno' is set to indicate the error.

What happens during a fork?

Ref: M. Vutukuru (2016)

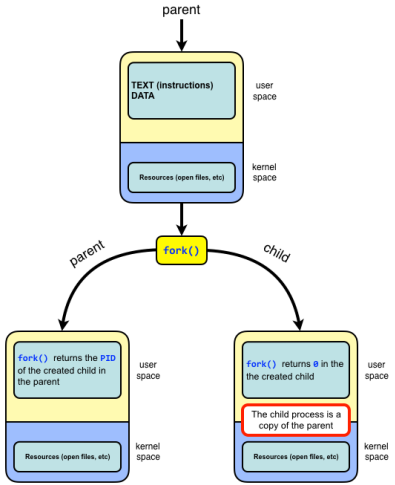


- A new process is created by copying a parent's memory image.
- The new process is added to the OS process list and scheduled.
- Parent and child process start execution after the call to 'fork().' The return values for each process will be different.
- Parent and child process will execute and modify memory independently.

What happens during a fork?
Ref: M. Vutukuru (2016)
A new process is created by copying a parent's memory image.
The new process is added to the OS process list and scheduled.
Parent and child process start execution after the call to 'fork().' The return values for each process will be different.
Parent and child process will execute and modify memory independently.

What happens during 'fork()'?

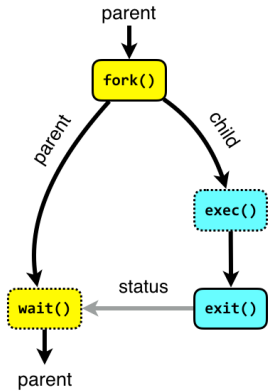
Ref: M. Vutukuru (2016)



What happens during 'fork()'?
Ref: M. Vutukuru (2016)

Waiting for children to finish.

Ref: M. Vutukuru (2016)

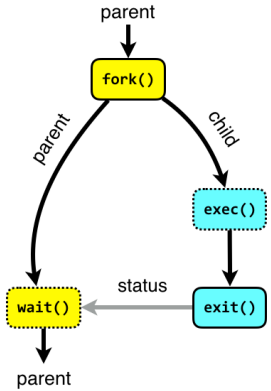


- A process can terminate by calling 'exit()' or the OS stopping a misbehaving process.
- Any terminated process exists as a zombie (stop state).
- When a parent process calls 'wait()' any zombie children are cleaned up. The parent process blocks until the children terminate.
- If a parent process terminates before a child, then the parent of all processes, 'init', becomes the parent to a child process.

Waiting for children to finish.
Ref: M. Vutukuru (2016)
- A process can terminate by calling 'exit()' or the OS stopping a misbehaving process.
- Any terminated process exists as a zombie (stop state).
- When a parent process calls 'wait()' any zombie children are cleaned up. The parent process blocks until the children terminate.
- If a parent process terminates before a child, then the parent of all processes, 'init', becomes the parent to a child process.

What happens during 'exec()'?

Ref: M. Vutukuru (2016)



- A process can run 'exec()' to load another executable to its memory image.
- Child process can be a completely different program.
- The command line arguments from the parent process can be passed to the child process.

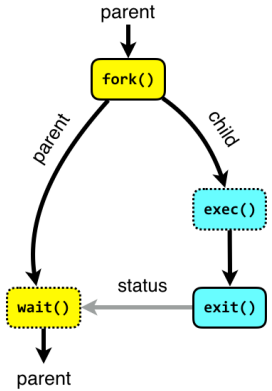
What happens during 'exec()'?
Ref: M. Vutukuru (2016)
A process can run 'exec()' to load another executable to its memory image.
Child process can be a completely different program.
The command line arguments from the parent process can be passed to the child process.

Case Study: Shell environment

Case Study: Shell environment

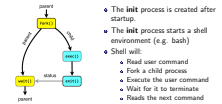
How does a shell work?

Ref: M. Vutukuru (2016)



- The **init** process is created after startup.
- The **init** process starts a shell environment (e.g. bash)
- Shell will:
  - Read user command
  - Fork a child process
  - Execute the user command
  - Wait for it to terminate
  - Reads the next command

How does a shell work?
Ref: M. Vutukuru (2016)

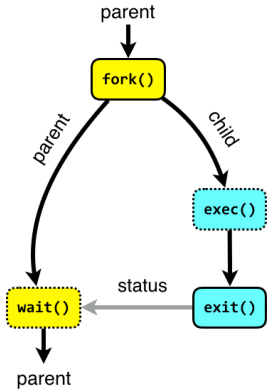


- The **init** process is created after startup.
- The **init** process starts a shell environment (e.g. bash)
- Shell will:
  - Read user command
  - Fork a child process
  - Execute the user command
  - Wait for it to terminate
  - Reads the next command

1. If the user command is 'ls,' the bash shell will:
   - Fork a child shell process of the parent shell process.
   - Execute 'ls' in the child shell process
   - Child shell process waits for 'ls' to finish
   - Child shell process terminates after the command is done.

How does a shell work?

Ref: M. Vutukuru (2016)



- Suppose you want to redirect output from a command to a file.
  prompt$ ls > foot.txt
- Shell will create a child process via fork().
- Set the standard output of the child to point to a file.
- Calls exec() on child process to execute the command.

How does a shell work?
Ref: M. Vutukuru (2016)
Suppose you want to redirect output from a command to a file.
prompt$ ls > foot.txt
Shell will create a child process via fork().
Set the standard output of the child to point to a file.
Calls exec() on child process to execute the command.

Conclusion

Conclusion

Conclusion

- How does the OS keep track of different instances of Google Chrome?
  - OS uses a process image to store state, code and handles to memory.
- How does the OS allow different processes to run on the same computer?
  - OS uses the stack to store information about a process.
  - OS uses scheduling policies to coordinate the use of the CPU.

Conclusion
How does the OS keep track of different instances of Google Chrome?
OS uses a process image to store state, code and handles to memory.
How does the OS allow different processes to run on the same computer?
OS uses the stack to store information about a process.
OS uses scheduling policies to coordinate the use of the CPU.

Conclusion (cont.)

- **fork()** creates copy of current process. (e.g., bash parent process produces bash child process)
- **execve()** creates a new command process. (e.g., bash process creates a new command process (e.g., ls))
- **exit()** is how the current process stops itself.
- **wait()** is how a parent process blocks until a child process terminates.

Conclusion (cont.)
fork() creates copy of current process. (e.g., bash parent process produces bash child process)
execve() creates a new command process. (e.g., bash process creates a new command process (e.g., ls))
exit() is how the current process stops itself.
wait() is how a parent process blocks until a child process terminates.