# Linux Case Study

OPS Lecture 16, G53OPS/G52OSC

Geert De Maere
(Jason Atkin – OSC)
Geert.DeMaere@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2015

# Overview

## Goals for Today/Tomorrow

- Next **three lectures**:
  - **Structure** of a Linux system
  - **Process management** in Linux
  - **Memory management** in Linux
  - **File systems** in Linux
- After Easter: **revision lecture**

# Linux

## Operating System Structure

- Linux is a **monolithic multi-user** operating system that uses **preemptive multi-tasking** and has a high similarity to **Unix**
  - Monolithic systems are **high performant** and share a **single address space** for process schedulers, file systems, device drivers, etc.
  - I.e., **no context switches occur** and no costly interprocess communication is required
- Linux aims to be **POSIX compliant**, i.e. it implements a standard interface

# Linux

## Operating System Structure

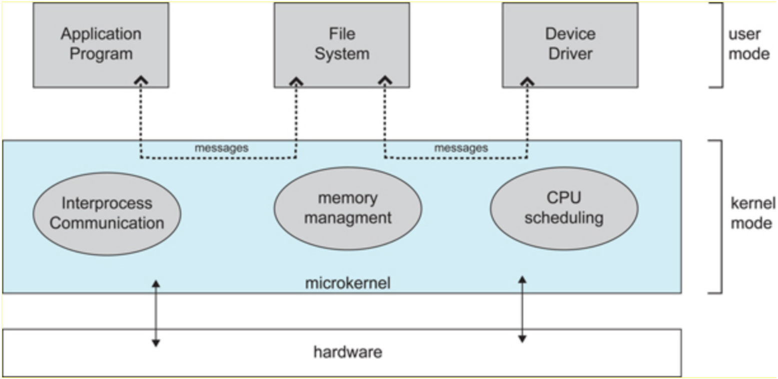



Figure: Structure of a Micro Kernel (Silberschatz)

# Micro Kernels

## Minimal Kernel Functionality

- All **non-essential functionality** are **extracted** from the kernel
  - **Communication**, **memory management** and **CPU scheduling** are likely to be included
  - The **file system**, **GUI**, **device drivers** are likely to be user processes
  - Some Unix version, Mac OS X, Minix, and early versions of Windows (NT4.0) were (partially) micro kernels
- Micro kernels are more **easy to extend**, more **portable**, and usually more **reliable**
- Frequent **system calls** and **kernel traps** cause significant **overhead**

# Linux

## Functions of an Operating System

**Remember:**

- **Manage/allocate resources** fairly and evenly amongst different competing processes
- **Hide low level hardware details** from the user/programmer and provide a layer of abstractions

# Linux

## Operating System Structure

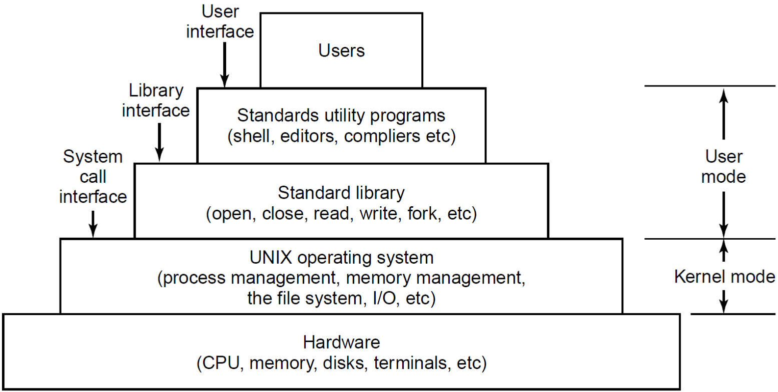



Figure: Layers and Interfaces in a Linux System (Tanenbaum)

# Linux

## The Four Layers in a Linux (Unix) System

- The **hardware layer** contains the CPU, memory, disks, terminals, . . .
- The **operating system** runs directly on top of the hardware:
  - It has full **full access** to the hardware and hides its details
  - It **controls access** to the hardware (e.g. process scheduler for the CPU, memory manager for RAM, etc.)
  - It provides a much simpler **system call interface** to the layers above

# Linux

## The Four Layers in a Linux (Unix) System

- The (`libc`) **library interface** (`Win32` for Windows) is provides a standard set of functions (wrapped around the true system calls) for applications to interact with the kernel
  - The library interface is defined in **POSIX** and provides a layer of indirection
  - Library functions themselves do not run in kernel mode
  - Library functions contain **machine code** to carry out a trap instruction
- A common set of **standard system/user utilities** is provided for Unix/Linux Systems, including user management, network configuration, shells, etc.

# Linux

## The Four Layers in a Linux (Unix) System

- CPU Modes include:
  - **Kernel mode** has access to all instructions of the CPU
  - **User mode** has access to a controlled subset of instructions and system resources
- Transition between user and kernel mode happens via a **trap instruction**

# Linux

## Three Interfaces to Linux

- The **true system call interface**, i.e. what is called by the libraries
- The **library interface** (e.g. `libc`), providing a set of functions defined in **POSIX** to access functionality in the kernel through system calls and to extend it (e.g., I/O buffering)
- The **utility interface**, i.e. what is used by the shell

# Linux

## Structure of the Linux Kernel

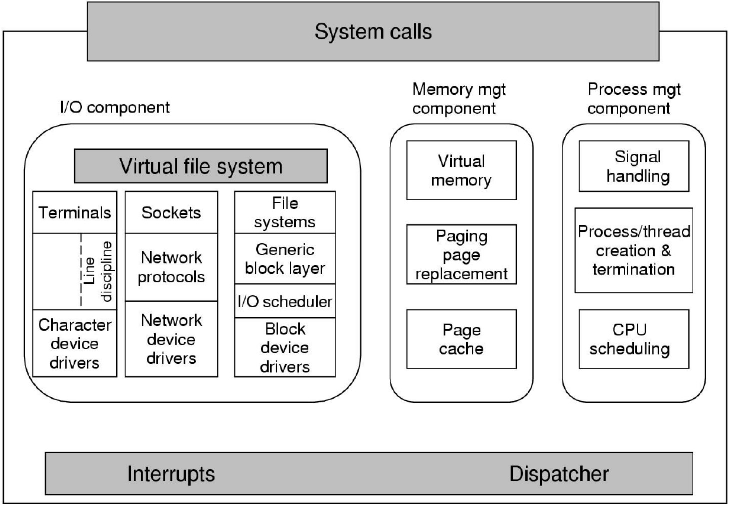



Figure: Structure of the Linux Kernel (Tanenbaum)

# Linux
## Structure of the Linux Kernel

- The kernel provides the **core functionality** such as process scheduling, memory management, file system management and runs in **kernel mode**
- The kernel interacts with the hardware through **dispatching routines** and **interrupts**
- “User layers” and libraries interact with the kernel through **system calls**

# Linux

## Structure of the Linux Kernel

- Key components in the kernel:
  - **Process management component** is responsible for the creation / termination of processes, scheduling processes / threads, . . .
  - **Memory management component** maintains the **virtual** to **physical memory mapping**, caching, page replacement, page faults, . . .
  - **I/O Component** manages devices, network (character device), and file systems/disks (block devices)
    - The I/O scheduler is used in the case of disks to minimise head movements
    - I/O is hidden behind the virtual file system
- These components are **interacting** with one another, e.g. caching a file from disk requires access to the memory management component

# Processes

## Recap

- “A process is a **running instance** of a program”, i.e., it is **active** and has a **process control block** and **resources** associated with it (e.g. I/O devices, memory, processor)
- A process has a **life cycle** with different states and multiple transitions between them (new, ready, running, exit, blocked, suspended)
- **Multi-programming** allows for concurrent execution of multiple processes (context switching is required)

# Processes
## Recap

- A process contains **two fundamental units**:
  - A set of **resources** (memory, files)
  - An **execution trace**
- A process can **share its resources** between multiple execution traces / **threads**, with each thread representing a **separate execution trace** with its own states, **thread control block**, and program counter
- The **process scheduler** (single and multi-core/processor) determines the order in which the processes run on the CPU (pre-emptive, non-preemptive, turn around time, response time)

# Processes

## Creating Processes in Linux

- **Two approaches** exist for a parent process to **create** a child process
  - The traditional **Unix approach** using `fork()` and `exec()`
  - The **Linux specific approach** using `clone()` which blurs the boundaries between processes and threads by allowing fine grained control over memory, files, etc. and
- If all resources are shared, a traditional **thread** is created, if none of them are shared, a **new process** is created
- **Processes** and **threads** are both called **tasks** in Linux

# Processes

## Process Creation in Linux

- The `fork()` system call creates a **child process** that is an **exact copy** of the parent process
  - The child has its own **task structure** or **process control block**
- The child has its own **private memory image** and shares the files with its parent:
  - Changes to registers, file descriptors, main memory, variables are not visible to the parent
  - Changes to files are visible

# Processes
## Process Creation in Linux

```
while (TRUE) {                                  /* repeat forever /*/
    type_prompt( );                             /* display prompt on the screen */
    read_command(command, params);              /* read input line from keyboard */

    pid = fork( );                              /* fork off a child process */
    if (pid < 0) {
        printf("Unable to fork0);               /* error condition */
        continue;                               /* repeat the loop */
    }

    if (pid != 0) {
        waitpid (–1, &status, 0);               /* parent waits for child */
    } else {
        execve(command, params, 0);             /* child does the work */
    }
}
```

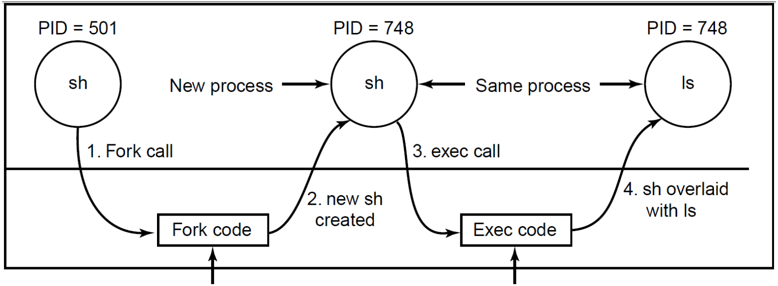
Figure: Use of the fork() and exec() system calls (Tanenbaum)

# Processes
## Process Creation in Linux

- Any process (e.g. the child) can **overwrite its memory image** using the `exec()` system call
- Process creation is optimised by applying **“copy on write”**
  - Every child gets its own **page table**
  - The page entries **point to the parent’s frames** (however they are marked as read only)
  - A **copy** of the parent’s page is made when ever **either of the processes writes** to it

# Processes

## Process Creation in Linux




Allocate child' s task structure
Fill child' s task structure from parent
Allocate child' s stack and user area
Fill child' s user area from parent
Allocate PID for child
Set up child to share parent' s text
Copy page tables for data and stack
Set up sharing of open files
Copy parent' s registers to child

Find the executable program
Verify the execute permission
Read and verify the header
Copy arguments, environ to kernel
Free the old address space
Allocate new address space
Copy arguments, environ to stack
Reset signals
Initialize registers

Figure: Process Creation in Linux (Tanenbaum)

# Processes

## Process Creation in Linux

- The **process control blocks** in Linux are represented by **task structures**:
  - They are used for **threads** as well as for **processes**, i.e. one single task structure per thread/process
  - They contain, e.g., process/thread scheduling information, open file table, memory information, registers, etc.
  - Some of this information is contained in **separate structures** (which can be **swapped out**) to which the task structure holds a pointer
- The process table (containing task structures) is implemented as a **doubly linked list** with a **mapping** based on the PID

# Processes

## Thread Creation in Linux

- Traditionally, **threads share address space**, global variables, files, etc. with the processes and have their **own program counter, registers, stack**, states, and local variables
- In contrast to `fork()`, the `clone()` allows **fine grained control** over which aspects the thread shares (e.g. address space, file descriptors)
    - This becomes possible because the process control block is based on **several sub-blocks**, e.g. for the file descriptors

# Processes

## Process Scheduling in Linux

- Process scheduling has evolved over different versions of Linux to account for **multiple processors / cores**, processor **affinity**, and **load balancing** between cores
- Linux distinguishes between two types of tasks for scheduling:
  - **Real time tasks** (to be POSIX compliant), divided into:
    - Real time FIFO tasks
    - Real time Round Robin tasks
  - **Time sharing tasks** using a preemptive multitasking approach
- The most recent scheduling algorithm in Linux is the “**completely fair scheduler**” (CFS)

# Processes

## Process Scheduling in Linux

- **Real time FIFO** tasks have the **highest priority** and are scheduled using a FCFS approach, using preemption only if a higher priority job shows up
- **Real time round robin tasks** are preemptable by **clock interrupts** and have a **time slice** associated with them
- Both approaches **cannot guarantee hard deadlines**

# Processes

## Process Scheduling in Linux: FCFS

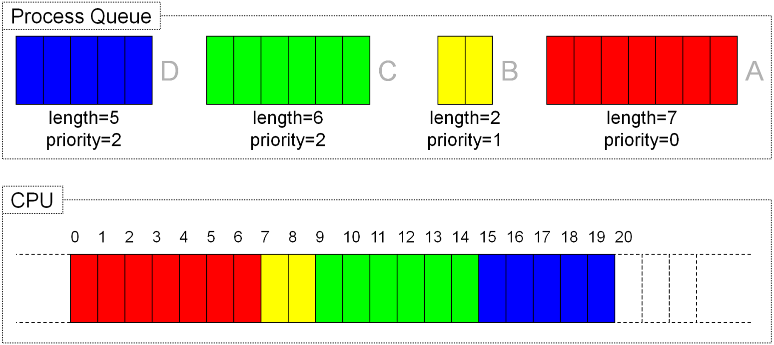



Figure: FCFS Scheduling

# Processes

## Process Scheduling in Linux: Round Robin

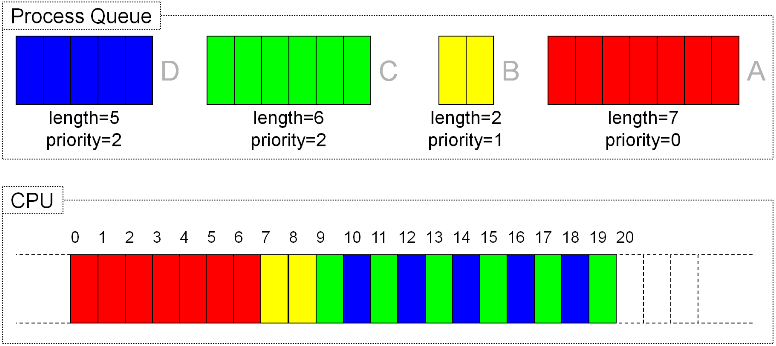



Figure: Round Robin Scheduling with Priorities

# Processes

## Process Scheduling in Linux: Time Sharing Tasks

- **Time sharing tasks** are scheduled using the “**completely fair scheduler**” (CFS, before the 2.6 kernel, this was an $O(1)$ scheduler)
- The CFS **divides the CPU time** between all processes, using a **weighting scheme** to take into account their priorities
- The length of the **time slice** and the “available CPU time” are based on the **targeted latency**
  - Every process should run at least once during the “target latency interval”

# Processes

## Process Scheduling in Linux: Time Sharing Tasks

- If all $N$ processes have the **same priority**:
  - They will be allocated a "time slice" equal to $\frac{1}{N}$ times the available CPU time
  - I.e., if $N$ equals 5, every process will receive 20% of the processor's time
- If process have **different priorities**:
  - Every process $i$ is allocated a **weight** $w_i$ that reflects its priority
  - The "time slice" allocated to process $i$ is then **proportional to** $\frac{w_i}{\sum_{j \in N} w_j}$

# Processes

## Process Scheduling in Linux: Time Sharing Tasks

- The tasks with the **lowest amount** of “used **CPU time**” are **selected first**
- If *N* is very large, the context switch time will be dominant, hence a lower bound on the “time slice” is imposed by the **minimum granularity**
  - A process’s time slice can be no less than the minimum granularity (response time will deteriorate)

# Summary
## Take-Home Message

- **Structure of a Linux system**, monolithic, interfaces, core components of the kernel
- **Processes in Linux**, process creation, process scheduling