# Introduction to Operating Systems

## Operating System Design – MOSIG 1

Instructor: Arnaud Legrand
Class Assistants: Benjamin Negreverne, Sascha Hunold

September 16, 2010

# Outline

# Outline

# Administrivia

- **Class web page:** http://mescal.imag.fr/membres/arnaud.legrand/teaching/2010/M1_OS.php
  - All assignments, handouts, lecture notes on-line
- **References**
  - Textbook: *Operating System Concepts, 8th Edition*, by Silberschatz, Galvin, and Gagne
  - Slides heavily inspired by those from "CS140: Operating Systems by **David Mazieres** (Stanford)". Many thanks to him!!!
- **Staff email address:** Arnaud.Legrand@imag.fr, Benjamin.Negrevergne@imag.fr
  - Add [M1-OSD] to the subject of your emails (otherwise, we may not read them)
  - Write your name/email on the sheet currently going through the room. This way you will receive updates on the lecture
- **Key dates:**
  - Lectures: Tuesday & Wednesday 13:30–15:00, F111
  - Practical Sessions: Wednesday 15:15–18:00, F112
  - Midterm: Tuesday, November 9, 13:30–15:00, (F107;F204)
  - Final: to be determined (three hours).

# Course goals

- **Introduce you to operating system concepts**
  - Hard to use a computer without interacting with OS
  - Understanding the OS makes you a more effective programmer
  - The first minutes of the lecture can be devoted to re-explain some parts of the previous lecture.
  - I can also come earlier if you have questions but you should send me an email before.
- **Cover important systems concepts in general**
  - Caching, concurrency, memory management, I/O, protection
- **Teach you to deal with larger software systems**
- **Prepare you to take graduate OS classes (M1 Principles of Computer Networks, M2 Parallel Systems, Distributed Systems, . . . )**

# Programming Assignments

- **Among the different practical sessions, some of them will be graded:**
  - A simple memory allocator.
  - A Synchronization problem.
  - Implementing System Calls in Nachos.
- **The memory allocator session will be next week.**
- **Implement projects in groups of up to 3 people**
- **No incompletes**
  - Please, please, please turn in working code, or **no credit** here
- **Means design and style matter a lot**
  - Large software systems not just about producing working code
  - Need to produce code other people can understand
  - That's why we have group projects

# Grading

- **No incompletes**
  - Talk to me ASAP if you run into real problems
- **33% of grade from projects**
  - For each project, 50% of score based on passing test cases
  - Remaining 50% based on design and style
- **33% of grade from mid-term exam**
- **33% of grade from final exam**

# Style

- **Must turn in a design document along with code**
- **CAs will manually inspect code for correctness**
  - E.g., must actually implement the design
  - Must handle corner cases (e.g., handle `malloc` failure)
- **Will deduct points for error-prone code w/o errors**
  - Don't use global variables if automatic ones suffice
  - Don't use deceptive names for variables
- **Code must be easy to read**
  - Indent code, keep lines and (when possible) functions short
  - Use a uniform coding style (try to match existing code)
  - Put comments on structure members, globals, functions
  - Don't leave in reams of commented-out garbage code

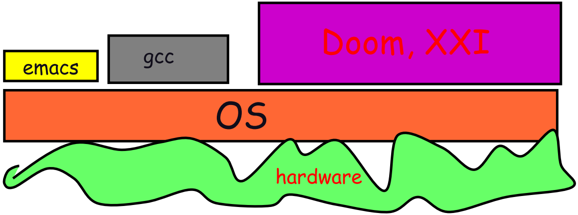
# Assignment requirements

- **Do not look at other people's solutions to projects**
- **Can read but don't copy other OSes**
  - E.g., Linux, Open/FreeBSD, etc.
- **Cite any code that inspired your code**
  - As long as you cite what you used, it's not cheating
  - Worst case we deduct points if it undermines the assignments
- **Projects due on the next Tuesday at start of lecture**
- **Ask me for extension if you run into trouble**

# Outline

# What is an operating system?

- **Layer between applications and hardware**



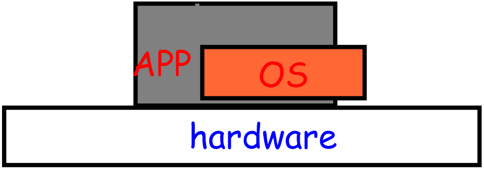

- **Makes hardware useful to the programmer**
- **[Usually] Provides abstractions for applications**
  - Manages and hides details of hardware
  - Accesses hardware through low/level interfaces unavailable to applications
- **[Often] Provides protection**
  - Prevents one process/user from clobbering another

# Why study operating systems?

- **Operating systems are a maturing field**
  - Most people use a handful of mature OSes
  - Hard to get people to switch operating systems
  - Hard to have impact with a new OS
- **High-performance servers are an OS issue**
  - Face many of the same issues as OSes
- **Resource consumption is an OS issue**
  - Battery life, radio spectrum, etc.
- **Security is an OS issue**
  - Hard to achieve security without a solid foundation
- **New "smart" devices need new OSes**
- **Web browsers increasingly face OS issues**

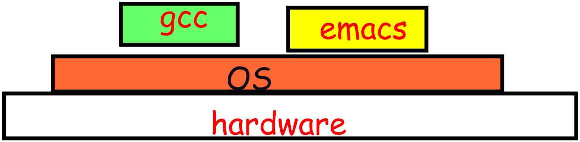
# Primitive Operating Systems

- **Just a library of standard services [no protection]**



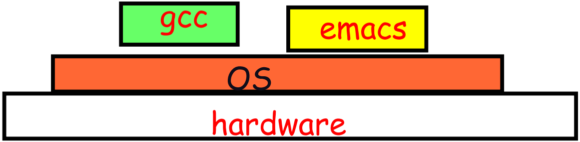

  - Standard interface above hardware-specific drivers, etc.
- **Simplifying assumptions**
  - System runs one program at a time
  - No bad users or programs (often bad assumption)
- **Problem: Poor utilization**
  - . . . of hardware (e.g., CPU idle while waiting for disk)
  - . . . of human user (must wait for each program to finish)

# Multitasking



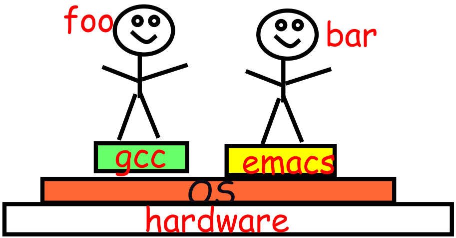

- **Idea: Run more than one process at once**
  - When one process blocks (waiting for disk, network, user input, etc.) run another process
- **Problem: What can ill-behaved process do?**

# Multitasking



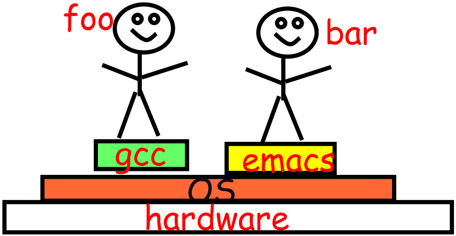

- **Idea: Run more than one process at once**
  - When one process blocks (waiting for disk, network, user input, etc.) run another process
- **Problem: What can ill-behaved process do?**
  - Go into infinite loop and never relinquish CPU
  - Scribble over other processes' memory to make them fail
- **OS provides mechanisms to address these problems**
  - *Preemption* – take CPU away from looping process
  - *Memory protection* – protect process's memory from one another

# Multi-user OSes




- **Many OSes use protection to serve distrustful users**
- **Idea: With $N$ users, system not $N$ times slower**
  - Users' demands for CPU, memory, etc. are bursty
  - Win by giving resources to users who actually need them
- **What can go wrong?**

# Multi-user OSes



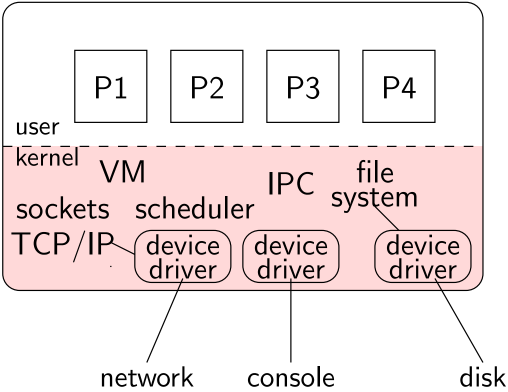

- **Many OSes use protection to serve distrustful users**
- **Idea: With $N$ users, system not $N$ times slower**
  - Users' demands for CPU, memory, etc. are bursty
  - Win by giving resources to users who actually need them
- **What can go wrong?**
  - Users are gluttons, use too much CPU, etc. (need policies)
  - Total memory usage greater than in machine (must virtualize)
  - Super-linear slowdown with increasing demand (thrashing)

# Outline

# Protection

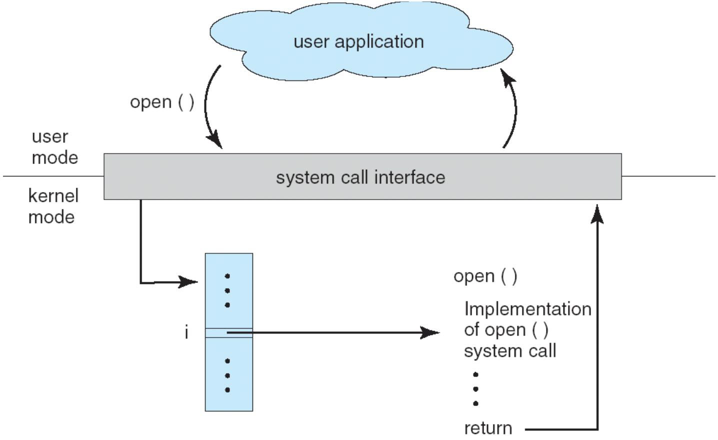
- **Mechanisms that isolate bad programs and people**
- **Pre-emption:**
  - Give application a resource, take it away if needed elsewhere
- **Interposition:**
  - Place OS between application and "stuff"
  - Track all pieces that application allowed to use (e.g., in table)
  - On every access, look in table to check that access legal
- **Privileged & unprivileged modes in CPUs :**
  - Applications unprivileged (user/unprivileged mode)
  - OS privileged (privileged/supervisor mode)
  - Protection operations can only be done in privileged mode

# Typical OS structure




- **Most software runs as user-level processes (P[1-4])**
- **OS kernel runs in privileged mode [shaded]**
  - Creates/deletes processes
  - Provides access to hardware

# System calls

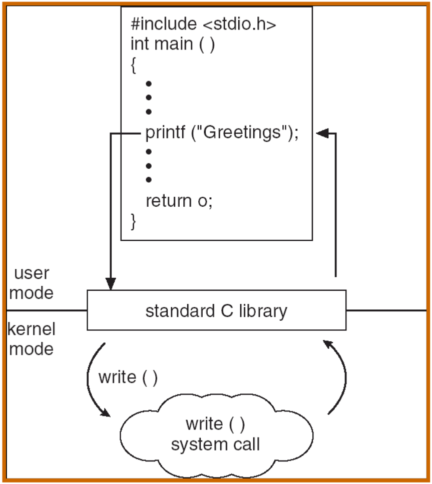



- **Applications can invoke kernel through system calls**
  - Special instruction transfers control to kernel
  - ...which dispatches to one of few hundred syscall handlers

# System calls (continued)

- **Goal: Do things app. can't do in unprivileged mode**
  - Like a library call, but into more privileged kernel code
- **Kernel supplies well-defined system call interface**
  - Applications set up syscall arguments and *trap* to kernel
  - Kernel performs operation and returns result
- **Higher-level functions built on syscall interface**
  - `printf`, `scanf`, `gets`, etc. all user-level code
- **Example: POSIX/UNIX interface**
  - `open`, `close`, `read`, `write`, ...

# System call example




- **Standard library implemented in terms of syscalls**
  - *printf* – in libc, has same privileges as application
  - calls *write* – in kernel, which can send bits out serial port

# Different system contexts

- **A system can typically be in one of several contexts**
- **User-level – running an application**
- **Kernel process context ("top half" in Unix)**
  - Running kernel code on behalf of a particular process
  - E.g., performing system call
  - Also exception (mem. fault, numeric exception, etc.)
  - Or executing a kernel-only process (e.g., network file server)
- **Kernel code not associated w. a process ("bottom half" in Unix)**
  - Timer interrupt (hardclock)
  - Device interrupt
  - "Softirqs", "Tasklets", . . . in Linux
- **Context switch code – changing address spaces**

# Transitions between contexts

- **User $\rightarrow$ top half: syscall, page fault**
- **User/top half $\rightarrow$ device/timer interrupt: hardware**
- **Top half $\rightarrow$ user/context switch: return**
- **Top half $\rightarrow$ context switch: sleep**
- **Context switch $\rightarrow$ user/top half**

# CPU preemption

- **Protection mechanism to prevent monopolizing CPU**
- **E.g., kernel programs timer to interrupt every 10 ms**
  - Must be in supervisor mode to write appropriate I/O registers
  - User code cannot re-program interval timer
- **Kernel sets interrupt to vector back to kernel**
  - Regains control whenever interval timer fires
  - Gives CPU to another process if someone else needs it
  - Note: must be in supervisor mode to set interrupt entry points
  - No way for user code to hijack interrupt handler
- **Result: Cannot monopolize CPU with infinite loop**
  - At worst get $1/N$ of CPU with $N$ CPU-hungry processes

# Protection is not security

- **How can you monopolize CPU?**

# Protection is not security

- **How can you monopolize CPU?**
- **Use multiple processes**
- **Until recently, could wedge many OSes with**

  ```
  int main() { while(1) fork(); }
  ```

  - Keeps creating more processes until system out of proc. slots
- **Other techniques: use all memory (`chill` program)**
- **Typically solved with technical/social combination**
  - Technical solution: Limit processes per user
  - Social: Reboot and yell at annoying users
  - Social: Pass laws (often debatable whether a good idea)

# Address translation

- **Protect mem. of one program from actions of another**
- **Definitions**
  - *Address space*: all memory locations a program can name
  - *Virtual address*: addresses in process' address space
  - *Physical address*: address of real memory
  - *Translation*: map virtual to physical addresses
- **Translation done on every load and store**
  - Modern CPUs do this in hardware for speed
- **Idea: If you can't name it, you can't touch it**
  - Ensure one process's translations don't include any other process's memory

# More memory protection

- **CPU allows kernel-only virtual addresses**
  - Kernel typically part of all address spaces,
    e.g., to handle system call in same address space
  - But must ensure apps can't touch kernel memory
- **CPU allows disabled virtual addresses**
  - Catch and halt buggy program that makes wild accesses
  - Make virtual memory seem bigger than physical
    (e.g., bring a page in from disk only when accessed)
- **CPU enforced read-only virtual addresses useful**
  - E.g., allows sharing of code pages between processes
  - Plus many other optimizations
- **CPU enforced execute disable of VAs**
  - Makes certain code injection attacks harder

# Resource allocation & performance

- **Multitasking permits higher resource utilization**
- **Simple example:**
  - Process downloading large file mostly waits for network
  - You play a game while downloading the file
  - Higher CPU utilization than if just downloading
- **Complexity arises with cost of switching**
- **Example: Say disk 1,000 times slower than memory**
  - 100 MB memory in machine
  - 2 Processes want to run, each use 100 MB
  - Can switch processes by swapping them out to disk
  - Faster to run one at a time than keep context switching

# Useful properties to exploit

- **Skew**
  - 80% of time taken by 20% of code
  - 10% of memory absorbs 90% of references
  - Basis behind cache: place 10% in fast memory, 90% in slow, usually looks like one big fast memory
- **Past predicts future (a.k.a. temporal locality)**
  - What's the best cache entry to replace?
  - If past = future, then least-recently-used entry
- **Note conflict between fairness & throughput**
  - Higher throughput (fewer cache misses, etc.) to keep running same process
  - But fairness says should periodically preempt CPU and give it to next process

# Goals

**Main Goals of an OS:**

- Provide **abstraction** of hardware through sound APIs
- Make **efficient** use of hardware
- Ensure **protection**
- Ensure **fairness**

You should always study the lectures (including this one) with these goals in mind.

We will see how these different issues are adressed when dealing with the different parts of a computer system (memory, CPU, storage, network, . . . ).

The course will be organized accordingly.

# Course Organization

Memory (Virtual memory)

- Fragmentation and segmentation
- Pagination, caching

CPU

- Processes & Threads
- Scheduling
- Concurrency, Synchronization & Communication

Storage

- File systems
- Network file systems

Network

- Distributed Systems

**Note:** Lectures will often take Unix as an example

- Most current and future OSes heavily influenced by Unix
- Windows is exception; we will mostly ignore it