TTIT61: Process Programming and Operating Systems

Alexandru Andrei, PhD
Embedded Systems Laboratory
Department of Computer and Information Science
Email: alean@ida.liu.se
Phone: 013-322828

Why Am I Here?

Process Programming and Operating Systems: TTIT61
http://www.ida.liu.se/~TTIT61
TTIT61=Course+Labs
Course=6 lectures; Exam=2 points
Labs=15 labs; Assignments=2 points
"Resurs"= 4 times (3 sessions dedicated to the labs)
Invited lecture: Thorbjörn Jemander, PhD, Manager
Embedded Platforms, Enea

Course Material

Seventh Edition
Avi Silberschatz
Peter Baer Galvin
Greg Gagne
John Wiley & Sons, Inc.

Lecture Plan

1. What is an operating system? What are its functions?
   Basics of computer architectures. (Part I of the textbook)
2. Processes, threads, schedulers (Part II, chap. III-V)
3. Synchronization & Deadlock (Part II, chap. VI, VII)
4. Primary memory management. (Part III, chap. VIII, IX)
5. File systems and secondary memory management (Part IV, chap. X, XI, XII)

What Is an Operating System?

- Examples of operating systems that you have seen/used:
  - Windows (95, 98, 2000, XP, Vista), DOS
  - UNIX (Solaris, AIX, HP-UX, Linux, BSD, Minix, SunOS)
  - MacOS X
  - OSE, Symbian, RTEMS, RT-Linux, QNX, VxWorks
  - Novell Netware
- When you buy/get one, what do you get actually?
  
  You get a piece of software!

- … but when you get Quake, you get a piece of software too! Is Quake an OS?
Four Components of a Computer System

Computer System Structure

- Computer system can be divided into four components
  - Users
    - People, machines, other computers
  - Hardware – provides basic computing resources
    - CPU, memory, I/O devices
  - Software
    - Application programs – define the ways in which the system resources are used to solve the computing problems of the users
      - Word processors, compilers, web browsers, database systems, video games
    - Operating system
      - Controls and coordinates use of hardware among various applications and users

What is an Operating System?

- A program that acts as an intermediary between a user of a computer or an application and the computer hardware.
- Operating system goals:
  - Execute user programs and make solving user problems easier.
  - Make the computer system convenient to use.
  - Use the computer hardware in an efficient manner.

Operating System Definition

- OS is a resource allocator
- Manages all resources
- Decides between conflicting requests for efficient and fair resource use
- OS is a control program
  - Controls execution of programs to prevent errors and improper use of the computer
Operating System Definition (Cont.)

- No universally accepted definition
- “Everything a vendor ships when you order an operating system” is good approximation
- But varies wildly
- “The one program running at all times on the computer” is the kernel. Everything else is either a system program (ships with the operating system) or an application program

Computer Startup

- The hardware doesn’t know where the operating system resides and how to load it
- Need a special program to do this job – bootstrap loader
- Typically stored in ROM or EPROM, generally known as firmware
- Bootstrap loader locates the OS kernel, loads it into main memory and starts its execution
- In some systems, a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel
  - GRUB – GRand Unified Bootloader
  - LILO – Linux LOader
  - NTLDR – NT Loader

What Are the Functions of an OS?

- You use your computer to run applications on it. Think of how you run an application.
  - The first thing we see at an OS is an interface to run applications.
- Could be command line (Unix, DOS) or graphical (MacOS, Windows). Because it is at the outer layers of an OS, it is typically called “shell.”
User-Operating System Interface:

**CLI**

- CLI allows the direct input of commands
- Sometimes implemented in the kernel, sometimes by system programs
- Sometimes multiple flavors implemented – shells
  - sh, bash, csh, tcsh (Unix)
- Primarily fetches a command from user and executes it
- Sometimes commands built-in the shell, sometimes just names of other programs
  - If the latter, adding new features doesn’t require shell modification
- Shell scripts to automate various jobs (ex. backup, find specific files and apply a certain command)

**GUI**

- User-friendly desktop interface
- Usually mouse, keyboard, and monitor
- Icons represent files, programs, actions, etc
- Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)
- Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI “command” shell
  - Apple Mac OS X as “Aqua” GUI interface with UNIX kernel underneath and shells available
  - Solaris is CLI with optional GUI interfaces (Java Desktop, Gnome, CDE)

Starting Applications w/o an OS

- How would we run an application without having an interface for that?

```
<table>
<thead>
<tr>
<th>CPU</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr</td>
<td>Data</td>
</tr>
<tr>
<td>Ctrl</td>
<td></td>
</tr>
</tbody>
</table>
```

- Just put the code of the application in memory at address X, where X is the address from where the processor starts to fetch instructions after a power-on.
- How would we start a second application?
- Chaining? The first application has to know which is the second application and has to start it?

Summary

- The OS consists of
  - A user interface for controlling programs (starting, interrupting)

Hardware Abstraction

- What means “running an application”? Why do we run applications?
- To get some results! On screen, printer, network, files, soundcard, etc.
- How does an application put data on the screen, printer, files, etc., read data from keyboard, network, mouse, etc?
- It uses device drivers! (When you buy a network card, you get a CD with drivers too)
Device Drivers

- How would applications interact with the hardware if there was no OS?
- The device driver code has to be included in the application code.
  - Portable applications? What if I change my soundcard?
  - Security? If everyone has access to the whole hardware, any user could read any hard-disk sector.
  - Safety? What if I make errors in my program?

Hardware Abstraction

- The device drivers are part of the OS.
- The application requests a service from the drivers (OS).
- Known as "system calls", in this case device-oriented system calls.
- The set of system calls can be seen as the API of the OS.

System Calls

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>

/* System calls are written in bold italic. Type "man 2 sys_call_name" for info in Linux or "man –s 2 sys_call_name" in Solaris */

int fd, n;
char buf[1024];
fd = open("datafile.txt", O_RDWR);
read(fd, buf, 1024);
printf("Have read %d bytes: %s\n", n, buf);
lseek(fd, 0, SEEK_SET);
write(fd, "Some other text", strlen("Some other text") + 1);
write(fd, buf, n);
close(fd);
```

Summary

- The OS consists of
  - A user interface for controlling programs (starting, interrupting)
  - A set of device drivers for accessing the hardware

System Call Execution
Summary

The OS consists of
- An user interface for controlling programs (starting, interrupting)
- A set of device drivers for accessing the hardware
- An set of system calls as a program interface to hardware (a lot more during the labs!)

Process Management

Can you run two or more programs at the same time on your computer?
Do they really run at the same time (i.e. concurrently)?

Many desktop PCs and laptops have 1 CPU, and 1 CPU may run only one program at a time. The programs run pseudo-concurrently. A pseudo-parallelism obtained by execution interleaving (multiprogramming).

Process Execution

<table>
<thead>
<tr>
<th>Process 1</th>
<th>I/O</th>
<th>waiting</th>
<th>executing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 2</td>
<td></td>
<td>I/O</td>
<td>waiting</td>
</tr>
<tr>
<td>Process 3</td>
<td>executing</td>
<td>I/O</td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>executing</td>
<td>executing</td>
<td>...</td>
</tr>
</tbody>
</table>

Process Management

- The OS schedules several processes
- It manages process data such that
  - Processes may resume their execution from where they were interrupted
  - Their requests for resources (I/O, memory) do not interfere
  - E.g.: Process A accesses the hard disk through some system call at time 0. Process B is scheduled at time 1ms. The hard disk sends an interrupt at time 2ms signalling that it has completed the request made at time 0. The interrupt should not reach to process B, just that the hard disk controller knows nothing about processes.

Could we implement pseudo-concurrent execution without an OS?
Each program would have to know about at least one other program, to relinquish the processor and to activate the next program.
We would need a way to chain programs. Almost impossible.

Summary

- The OS consists of
  - A user interface for controlling programs (starting, interrupting)
  - A set of device drivers for accessing the hardware
  - A set of system calls as a program interface to hardware (and not only, we’ll see later)
  - Process manager (scheduler) that schedules process execution and manages the process state
### Memory in MS-DOS

![Diagram of memory layout](image)

(a) at system startup  
(b) running a program

### Memory Management

**Naive solutions to the memory management problem:**
- Only one program in memory at the same time. Whenever the scheduler selects a new program to execute, the old one is swapped out on a secondary device (HDD). The memory image of the new program is swapped in from the secondary device.
- Pre-allocation by a global authority of the memory space to all software developers.
- All addresses that appear in the program are relative to a base address given by the OS upon loading the program in memory.

**Modern solutions:**
- Programs know about virtual memory and do not know the physical address that will be put on the address bus.
- Operating system translates virtual addresses to physical addresses (more during the course and labs).

### FreeBSD Running Multiple Programs

- process D
- free memory
- process C
- interpreter
- process B
- kernel

### Memory layout

**What to remember from the previous slide:**
- Programs are laid out in memory as code, data (read-only, initialized, not-initialized), and stack segments.
- Variables in high-level programs correspond to memory locations.
- Accesses to a variable (for reading and writing them) corresponds in the code to accesses in the memory at the address of the memory location that corresponds to the variable.
- Items from the memory are accessed using their address.

**How should a compiler decide at which address to put each variable? How are clashes avoided when there are several programs that run on the same system?**

### Summary

**The OS consists of:**
- A user interface for controlling programs (starting, interrupting).
- A set of device drivers for accessing the hardware.
- A set of system calls as a program interface to hardware (and not only, we'll see later).
- Process scheduler that schedules process execution and manages the process state.
- **Memory management**
File System

- The organization of a HDD as it comes from the manufacturer is given in C/H/S (cylinders, heads, sectors)
- Unsuitable for a hierarchical file system
- The OS abstracts away the C/H/S and presents to the user a file-system interface

Summary

- The OS consists of
  - A user interface for controlling programs (starting, interrupting)
  - A set of device drivers for accessing the hardware
  - A set of system calls as a program interface to hardware (and not only, we’ll see later)
  - Process scheduler that schedules process execution and manages the process state
  - Memory management
  - File system

Other Functions

- Support for inter-process communication (message queues, synchronization mechanisms, shared memory segments)
- Support for security (system of access control lists, capability lists, etc.)
- Interfaces for performance estimation, profiling, debug, statistics, board configurations, on-board sensor monitoring, etc. (see the /proc file system on Linux for example)
- Support for fault-tolerance (RAID arrays)
- Support for various optimizations (battery life, for example)

Summary

- The OS consists of
  - A user interface for controlling programs (starting, interrupting)
  - A set of device drivers for accessing the hardware
  - A set of system calls as a program interface to hardware (and not only, we’ll see later)
  - Process scheduler that schedules process execution and manages the process state
  - Memory management
  - File system
  - Others

Operating System Design and Implementation

- Design and Implementation of OS not easy, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start by defining goals and specifications
- Affected by choice of hardware, type of system
- User goals and System goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Simple Structure

- MS-DOS – written to provide the most functionality in the least space
- Not divided into modules
- Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated
Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers.

UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts:
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

Microkernel

- Minimum essential functionality in the kernel, the rest implemented as system services
- Client-server system on same system
  - Clients request services from microkernel which passes message onto appropriate server
Microkernel System Structure

- Moves as much from the kernel into "user" space
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

Modules

- Most modern operating systems implement kernel modules
- Uses object-oriented approach
- Each core component is separate
- Each talks to the others over known interfaces
- Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible

Solaris Modular Approach

- Most modern operating systems implement kernel modules
- Uses object-oriented approach
- Each core component is separate
- Each talks to the others over known interfaces
- Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible

Windows XP Architecture

Solaris Kernel Overview

- Reading
  - Silberschatz, Galvin, Gagne, 7th edition, Part I
    - Chapter 1: 1.1-1.9
    - Chapter 2: 2.1-2.7, 2.10