1 – Introduction

Operating Systems
ACSI 500
Class Times M-W 5:45-7:05

Instructor
W. A. Maniatty: maniatty@cs.albany.edu

Teaching Assistant
Du Ling: ld7441@cs.albany.edu
2 – Some General Design Goals

All good engineering pays close attention to:

- Correctness/Completeness — Does the solution handle all cases correctly?

- Cost — What are the resource requirements of the solution (e.g. run time, hardware, user time)?

- Performance — How efficiently does the solution solve my problem?
3 – Goals of the Course

This course is to provide a combined applied/theoretical background in Operating Systems and Systems Programming to improve students’:

1. Systems software design,
2. Systems programming,
3. Performance analysis,
4. Performance tuning and
5. Documentation design.

These things should improve academic/professional preparation and competitiveness when done with this course.
4 – Prerequisites

The Study of Operating systems relies on many areas:

1. Mathematics
   (a) Basic Calculus/Numerical Analysis
   (b) Probability (Exponential Distribution, Poisson Processes)
   (c) Statistics (Experiment design and regression)
   (d) Scheduling

2. Computer Science
   (a) Computer Architecture
   (b) Basic Performance Analysis
   (c) Programming Language Design
   (d) Software Design
   (e) Data Structures/Algorithms

3. Technical Writing Skills
5 – Who Should NOT Take this Class?

If you:

- Are not ready do your own hard work
- Do not have 20 hours a week
- Lack prerequisites (e.g. graduate standing and/or ACSI 511).
- Will not take the initiative
  - Only key concepts will be covered in class.
- Are a weak programmer
- Are not good at math
- Just want to sit and listen
- Are not interested in this topic.

This class will not be good for you.

Students lacking adequate background may be deregistered or fail the course.
6 – O/S Design Goals

An Operating System (O/S for short) provides the following:

1. A run time programming interface for applications.
2. Resource management.

O/S design goals include:

1. Convenience — Reduce complexity for programmers/users.
2. Efficiency — Allow “efficient” access to system resources.
3. Flexibility — Adding new features should be tractable.
Comer [2] lists the following things as not being an O/S:

1. A Programming Language/Compiler — Translators
2. An Interpreter — Shell/User Interface
3. A Library — A set of commands/utilities

These features are typically provided as part of the O/S but are strictly speaking not the O/S.
## Computer Architecture

The following layers of abstraction are used (from highest to lowest):

1. **Applications Software** — Custom programs to support users.
2. **Systems Software** — Controls the hardware.
3. **Hardware** — The physical control of the system (signals).

<table>
<thead>
<tr>
<th>Web Browser</th>
<th>Games</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Base</td>
<td>Word Processor</td>
</tr>
<tr>
<td>Compiler</td>
<td>Shell</td>
</tr>
</tbody>
</table>

**Operating Systems**

- **Applications Software**
- **Systems Software**
- **Hardware**

Figures 1: Computer Systems Architecture
Almasi and Gottlieb’s [1] computer design overview:

Figure 2: Computer Design, the Big Picture
Some O/S services for users are:

1. Program Execution — Read program into memory/run it return control when done.
5. Error Detection/Handling — Hardware failure, programming errors.
6. Program Creation Tools — Debugging/text editing, etc.
7. Accounting — Billing for computer access.
11 – O/S Resource Management

O/S controls are *intrusive* because they both consume the resources which they seek to control and mediate conflicting requests.

12 – Resource Attributes

Resources have properties governing their use including:

1. Preemption — How expensive is it for the device/resource?
2. Renewability — Does the resource get replenished?
3. Scheduling — Does order of allocation matter?
4. Persistence — Does it’s lifetime extend beyond that of the process which creates it?
5. Sharing — Is it suited to to be accessed by many users.

13 – Scheduling/Allocation Policy
The scheduled allocation of resources reflects policies including:

1. Fairness
2. Starvation Freedom
3. Maximum Throughput
4. Maximum Utilization
5. Minimize Response Time
14 – O/S Achievements

Denning’s list of conceptual breakthroughs (1980):

1. Processes — A program in execution
2. Memory Management
3. Information protection and security
4. Scheduling and resource management
5. System Structure
15 – A Traditional Hardware Architecture

The traditional von Neumann structure consists of:

1. Processor — Also called the CPU (Central Processing Unit) our example has the registers:
   (a) PC — Program Counter
   (b) IR — Instruction Register
   (c) MAR — Memory Address Register
   (d) MBR — Memory Buffer Register
   (e) I/O AR — I/O Address Register
   (f) I/O BR — I/O Buffer Register

2. Memory — (Volatile)

3. Peripherals — I/O Devices

4. Systems Interconnection — Links the Other Components (Bus)
Figure 3: von Neumann Architecture
The register structure comes from Stallings, the names mean:

1. PC — Program Counter — Address of next instruction

2. IR — Instruction Register — Current Opcode to execute

3. MAR — Memory Address Register — Memory Pointer

4. MBR — Memory Buffer Register — Buffer for memory access

5. I/O AR — I/O Address Register — I/O Pointer

6. I/O BR — I/O Buffer Register — Buffer for I/O access

User registers are not shown here.
16 – Process Context

The context of a process consists of:

1. The Current Instruction Pointer (the IP)
2. Register Contents
3. Memory used by that process

This is the machine’s current state used to determine the machines next state when running a particular program.
Traditionally called the *fetch/execute* cycle, in its simplest form:

![Diagram](image)

**Figure 4: Basic Fetch/Execute Cycle**
18 – Motivation for Interrupts

An interrupt is an external asynchronous event changing the flow of control of a process (my definition). It is also refers to triggering such an event.

19 – The Problem

Peripherals tend to have slow response time (especially if waiting on human supplied input).

20 – The Solution

To improve CPU utilization we want to allow the CPU to continue to execute while awaiting I/O as seen in Figure 5. When an interrupt is detected an interrupt handling routine is typically invoked. There are times (such as interrupt processing) where it may be desirable to disallow further interrupts.
21 – Interrupting the Fetch/Execute Cycle

The fetch/execute cycle is adjusted to support interrupts.

Figure 5: Fetch/Execute Cycle Supporting Interrupts
22 – Interrupt Processing

Processing an interrupt on most architectures involves the following steps:

1. Finish the current instruction
2. Push the IP on the stack
3. Set the IP to the interrupt handler’s address.
4. Preserve the process context (i.e. push registers).
5. Executing the interrupt handling routine.
6. Restore the process context (i.e. pop registers).
7. Pop the IP from the stack
23 – Where Software Designers Fit in?

End User

Application Programs

Utilities

Operating System

Computer Hardware

Programmer

Operating-System Designer

Figure 6: Software Design Types
24 – Technological History of O/S Design

This section considers the availability of tools. Tanenbaum [3, 4] describes 5 decades of electronic computing.

25 – 1st Generation: 1945 - 1955

1st Generation computers were made of Vacuum tubes and plugboards. Computers were expensive and had limited peripherals, programs were entered by toggling switches on the control panel. Minimal O/S were used.

26 – 2nd Generation: 1955-1965

2nd Generation computers are characterized by Transistors/Batch Programming. Non-interactive peripheral access improved (i.e. card readers) but computers were still relatively expensive relative to user/programmer time. Access to the machine was restricted to operators, users submitted jobs to the operators.
27 – 3rd Generation: 1965 - 1980

IC’s reduced computing costs and made interactive peripherals available. Computers are still expensive, and had to be shared. Time-Sharing permits many users to share the machine, and encouraged interactive applications.


Tanenbaum calls Personal Computing fourth generation.

Users had dedicated machines and initially multitasking was not used. Simple networking was prevalent.

29 – Parallel Computing: 1990-present

Transparent tools for interconnecting machines are prevalent.
Coupling in parallel systems refers to how processors are connected. Processors sharing common memory are said to be *tightly coupled*.

![Diagram of tightly coupled processors](image)

**Figure 7: Tightly Coupled Processors**

Tightly coupled systems have limited scalability.
Loosely coupled processors have distributed memory.

Figure 8: Loosely Coupled Processors

Interprocessor communication latency is larger in a loosely coupled system than in a tightly coupled system, however such systems scale well.
31 – Types of Applications

How are computers used, and how do people want to use them?

1. By Resource Utilization:
   (a) Computation intensive jobs
   (b) I/O intensive jobs

2. By Application Driven Needs:
   (a) Real time jobs
   (b) Fault Tolerant/High Availability
   (c) Virtual Machine
   (d) Multi User
   (e) Multitasking
   (f) Interactive Jobs
   (g) Batch Jobs
   (h) On Line Transaction Processing
   (i) Transparent Interconnection
32 – Some well Known Operating Systems

We can characterize well known current O/Ss:

1. Apple’s Mac OS
2. IBM’s OS/2
3. UNIX/Linux/SunOs/AIX/IRIX
4. Microsoft Windows
5. QNX
6. IBM’s VM
7. IBM’s MVS
8. Inferno
IBM developed the System/360 and System/370 and their O/Ss:

1. PCP - Principle control program
2. MFT - Multiprogramming with a fixed number of tasks
3. MVT - Multiprogramming with a variable number of tasks
4. MVS - Multiple virtual storages.

MVS is designed to Support:

1. High performance (I/O throughput)
2. On Line Transaction Processing (OLTP)
3. Maintain backward compatibility
4. Have high availability/fault tolerance
5. Support tightly coupled multiprocessing.
34 – MVS Jargon and Components

MVS has internals and an outer layer of support tools:

35 – MVS Outer Layer

MVS has the following outer layers shown in 9:

1. Compilers, Link Edigors, Loader
2. Error Recovery Management
3. Job Management — The command environment (shell):
   (a) Interprets operator commands
   (b) Read and write Job input data to peripherals
   (c) Allocate I/O devices and notify operator of mount requests
   (d) Convert the Job into tasks for task management
36 – MVS Internals

Under the hood MVS has:

1. Dispatcher — Schedules tasks on processors
2. Task Management — process control
3. Interrupt Handling
4. Program Management — Runs Programs (run time loader)
5. Storage Management — Virtual and real memory
6. Systems resource Management — Partitions resource among tasks
7. Access Methods — User I/O interface
8. I/O supervisor — Low level device access
Figure 9: MVS Systems Structures
Figure 10: General Structure of the UNIX O/S
Figure 11: (Now Ancient) History of UNIX
40 – Some History

- 1969 Multics Abandoned by Bell Labs GE 645 Removed
- Thompson begins building UNIX file system on GECOS
- Ritchie and Thompson ported/extended UNIX for PDP 7 “Space Travel” — Assembler/Command interpreter added
- 1972 Version 1 on PDP 11/20 — C created (no structs or global variables yet).
- 1973 — UNIX Rewritten in C
- 1975 — UNIX Version 6 made publicly available (inexpensive)
- 1977 — UNIX ported to Interdata 8/32 (eliminating many machine dependencies)
- 1979 — UNIX System 7 released (first widely
UNIX programming environment features:

1. Text Editors
2. Text Processing
3. C/C++ compilers
4. make utility
5. Debuggers (dbx/sdb/adb/gdb)
6. Profilers
7. Compiler Construction Aids (Lex and YACC)
8. Source code version control (sccs, rcs, cvs)
42 – UNIX Programming Philosophy

Program design philosophy:

1. Arrange each program to perform a single function.

2. Avoid extraneous output, another program might use it as input.

3. If possible, use or modify existing tools rather write a new one.

4. Create the design first, then start with a small prototype and add features incrementally.
Some Central Concepts to UNIX include:

1. The File System
   (a) Every file is a sequence of bytes (characters) representing either a program or data. No record structure is imposed.
   (b) A directory holds the names of other files or directorys, hence the file system is hierarchichal.
   (c) Input or output devices are treated as special files using the standard file interface. Information is provided from/to the device directly.
2. Processes do all user work in UNIX.

   (a) Process creation is done by copying, the original is the parent, the copy is the child.

   (b) Parent and Child are identical except the parent may wait for the child to finish.

   (c) A process may replace itself by running another program.

3. The Shell is the Unix command language.

   (a) The shell executes command from a terminal or a file.

   (b) Users can create commands using script files.

   (c) Many commands use standard input or standard output.

   (d) Pipes send the standard output from one process to the standard input of another.
Recall that a process is a program in execution. In UNIX a process has access to the following files:

1. Standard Input (stdin) — The keyboard by default, but could be a file or the output of another process (pipe)
2. Standard Output (stdout) — Where normally generated output goes.
3. Standard Error (stderr) — Reserved for error messages (so they don’t get piped as input to another process).

Additional files can be read or written.

Redirection: `sort < junk > /tmp/junk.srt`
Pipe: `du -a -s | sort -r -n | more`
Steve Johnson constructed the following prototype of spell using filters. (Bentley [?]).

The sequence of actions is:

```
Input File -> Break into Words -> Input Words -> Sort and Discard Duplicates -> Sorted Words -> Compare to Dictionary -> Errors
```

Figure 12: Simple Spell Checker Structure

The program looks like:

```
prepare filename
   tr '[A-Z]' '[a-z]' |
   tr -c '[a-z]' '\n' |
   sort -u |
   comm -2 dict
```
The program looks like:

```
prepare filename       # Remove Formatting
 tr '[A-Z]' '[a-z]'    # Convert to lower case
 tr -c '[a-z]' '\n'   # Separate into words
 sort -u              # Sort step
 comm -2 dict          # In Dictionary?
```
Some UNIX Contributions to O/S design and practice include:

1. Simplicity
2. Portable implementation using a high level language
3. Uniform treatment of peripherals/files
4. Ease of I/O redirection
5. Flexibility/ease of extension
6. (Initially) System size
7. Good software development tools
8. (Initially) easily obtained source code
9. Interactive multitasking
47 – Problems With UNIX

1. (Historically) Easy to crash the system due to limited error detection and recovery

2. Command names are not obvious

3. Documentation assumes you know what you are doing

4. Administration tools assume UNIX expertise

5. Limited system security

6. Code Bloat (particularly in the Kernel)

7. Filter model initially unintuitive to users (sometimes no output is generated)
David Cutler was the chief architect of NT [5]. NT is a portable single user multitasking system with highly modular components, designed to support multi processing.

Figure 13: Structure of Windows NT
NT has the following layers:


2. Kernel: Provides basic O/S services not belonging in user space: scheduling, context switching, interrupt handling, etc...

3. Subsystems: Modules designed to provide specific functions within the O/S outside the Kernel.

4. System Services: Modules designed to provide functions for application level support.
Models of parallelism employed:

1. Client Server Model
2. Threads
3. Symmetric Multiprocessing (SMP)

NT has some object oriented features, with:

1. Encapsulation
2. Objects and Instantiations (named instances have security)
Figure 14: Speed of various Memory Devices
Programmers and Users demand:

1. More Memory
2. Fast Memory Access

The *von Neumann bottleneck*: Memory speed limits systems performance.

Ways of improving memory performance:

1. Parallelize Memory Access — Possible Asynchrony
2. Hierarchical Memory — Has two common approaches:
   (a) Large amounts of cheap slow memory extend available memory.
   (b) Small amounts of expensive fast memory improve systems performance.
53 – Parallelizing Memory Access

As per Problem 1.3, Memory tends to be written sequentially. The cost of sequential memory access can be described:

\[ T_A = T_L + T_C \]  \hspace{0.5cm} (1)

Where:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_A)</td>
<td>Total Memory access time</td>
</tr>
<tr>
<td>(T_C)</td>
<td>Completion time (service cost)</td>
</tr>
<tr>
<td>(T_L)</td>
<td>Memory latency (startup cost)</td>
</tr>
</tbody>
</table>

A technique to improve memory access is to initiate several memory accesses in parallel when \(T_C \gg T_L\) so that the processor can continue operating while waiting for a memory operation to finish. Typically this is done by distributing sequential memory addresses across different memory units.
54 – Hierarchical Memory

Hierarchical memory was first used in the Atlas operating system in 1962.

Figure 15: A typical memory hierarchy on a networked machine.
References


