Chapter 2: Book Overview

- Lectures skip chapter 1
  - Mostly 312 background with some examples

- Our goal in chapter 2
  - Understand the motivation for building an OS
  - Introduce basic terminology and history
  - Glance over the main concepts studied later
Chapter 2: Motivation

• Early computers (1940-1950s) did not have an OS

• Programs (called jobs) were loaded manually from punch cards
  – Errors were indicated by lights
  – Printer output signaled successful completion

• Three main problems:
  – Scheduling inefficiency
  – Setup delays
  – Hardware awareness

IBM punch card (invented in 1928)
Chapter 2: Motivation

• Scheduling inefficiencies
  - Sign-up sheet to reserve computer time
  - Wasted resources if job finishes quicker than reserved time
  - Forced termination and repeated visits if taking too long

• Setup delays
  - Loading compiler, source code, libraries, input data, and linking involved mounting tapes and/or card decks
  - If an error occurred, the user had to restart the process
  - Considerable time dedicated to setting up the program to run

• Hardware awareness
  - Programmer had to write directly into device registers in every program, keep track of hardware changes
  - Time wasted on largely irrelevant code development
Chapter 2: Roadmap

2.1 OS objectives and functions
2.2 Evolution of the OS
2.3 Major achievements
2.4 Other developments
2.5 Virtual Machines
2.6 Multi-core considerations
2.7 MS Windows
2.6 Traditional UNIX
2.7 Modern UNIX
2.8 Linux
Evolution of the OS

- Manual job control in the 1940s was known as **serial processing**
- Extreme inefficiency and inconvenience prompted automation of the process and development of an OS
- Main functions
  - Controls the execution of application programs
  - Provides an interface to hardware
Simple Batch System (1955)

• Early computers were extremely expensive
  - Was important to maximize processor utilization

• With an OS present, user no longer had direct access to CPU or devices
  - Instead, submitted jobs into a FIFO queue that was read and executed by a monitor

• When programs were done, they returned control to the monitor

Job Control Language (JCL)
- Directives how to run the job (e.g., compiler, input data, job owner)

OS = monitor

Device drivers
Job sequencing
JCL interpreter
User program area
Simple Batch System

Hardware features

• Memory protection
  – Jobs with access violations (e.g., trying to wipe out the monitor) were aborted

• Timer
  – Prevented jobs from monopolizing system or infinitely looping
  – Each job had a fixed deadline by which it had to finish

• Privileged instructions
  – Execution allowed only by the monitor
  – Prevented jobs from crashing the system or reading unauthorized data (e.g., the next job)
  – Monitor controlled all I/O

• Interrupts
  – Were not needed as all I/O was synchronous
Multi-Programmed Batch System (1959)

• Even in batched systems, the CPU was often idle
  – Automatic job sequencing helped reduce the delay *between* the jobs, but not *within* them
  – Reason: I/O devices are slow compared to processor

• **Example**: a job spends 15 ms reading a record from the file, then processes it for 1 ms, and finally writes one record to another file (also 15 ms)
  – What is the CPU utilization?

• This is often called *uni-programming*
Multi-Programmed Batch System

- **Idea**: when one job needs to wait for I/O, the monitor can switch the CPU to another job
  - Various scheduling algorithms are possible
  - Example below uses strict priority scheduling from A to C

- Interrupts are now needed for monitor to regain control
- This is called **multi-programming** (or **multi-tasking**) and is now the central theme of modern OSes
Example

- At time 0, three jobs are submitted to a monitor in a system with 250 MB of RAM
  - CPU in table means % of time task is not blocked on I/O
  - Assume jobs never conflict on the same I/O device

**Uni-programming**

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>70%</td>
<td>10%</td>
<td>10%</td>
</tr>
<tr>
<td>Duration</td>
<td>5 min</td>
<td>15 min</td>
<td>10 min</td>
</tr>
<tr>
<td>RAM</td>
<td>50 MB</td>
<td>100 MB</td>
<td>75 MB</td>
</tr>
</tbody>
</table>

**Multi-programming**

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>90%</td>
<td>20%</td>
<td>10%</td>
</tr>
<tr>
<td>RAM</td>
<td>90%</td>
<td>70%</td>
<td>40%</td>
</tr>
</tbody>
</table>
Example

- **Task 1**: completion time of last job in uni-programming?
- **Task 2**: what is the average CPU and RAM utilization?
  - Metric computed over the entire interval
- **Uni-programming**
  - CPU: \((70\% \times 5 + 10\% \times 15 + 10\% \times 10)/30 = 20\%\)
  - RAM: \((20\% \times 5 + 40\% \times 15 + 30\% \times 10)/30 = 33.3\%\)
- **Task 3**: what is the **throughput** of the system?
  - Number of jobs finished per time unit (e.g., 1 hour)
- **Task 4**: what is the **mean response time**?
  - Average delay from job submission to its completion
  - Uniprocessing: \((5 + 20 + 30)/3 = 18.333\) min

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>70%</td>
<td>10%</td>
<td>10%</td>
</tr>
<tr>
<td>Duration</td>
<td>5 min</td>
<td>15 min</td>
<td>10 min</td>
</tr>
<tr>
<td>RAM</td>
<td>50 MB</td>
<td>100 MB</td>
<td>75 MB</td>
</tr>
</tbody>
</table>
**Time Sharing System (1961)**

- Batch mode favors long CPU-bound jobs
  - Response time for other tasks may be minutes or hours
- Maximizing CPU utilization does not suit interactive jobs
  - E.g., a text editor cannot wait 3 hours for its turn
- Under time-sharing, CPU is periodically provided to all jobs not waiting for I/O
  - **Goal**: minimize response delay
- Time divided into *slices*
  - E.g., 200 ms in early systems, 1-10 ms in modern OSes
- The kernel rotates through all jobs scheduling them to run on the CPU
- Max delay before getting on the CPU
  - Slice * (number of jobs in system – 1)
Time Sharing System

• Comparison

- multi-programmed batch mode
- time sharing

A   CPU
B   wait   CPU
C   wait

1 hour

20 ms to process keyboard

60 ms

• Response time of C with 10-ms slices?

• First time-sharing OS
  - Compatible Time-Sharing System (CTSS), MIT 1961

• Modern OSes derived from these early concepts
Real-Time System

• In regular OSes, job switching delays are random and depend on the immediate backlog of CPU-bound tasks and their priority
  - Under worst-case scenarios, a job may not receive its turn for many slices

• This presents certain problems in mission-critical applications
  - E.g., car traction control, helicopter missile-guidance system

• **Real-time OS (RTOS)** provides guarantees on scheduling and interrupt delays
  - Examples include Windows CE, RTLinux, VxWorks
OS Growth

- OSes are complex pieces of software
  - MIT’s CTSS (1961-3): 32,000 machine words
  - IBM’s OS/360 (1964): 1M CPU instructions
  - Multics (1978): 20M CPU instructions
- Later, software was measured in source lines of code (SLOC)
  - Estimates from Wikipedia:

<table>
<thead>
<tr>
<th>Year</th>
<th>OS</th>
<th>SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>93</td>
<td>NT 3.1</td>
<td>4M</td>
</tr>
<tr>
<td>94</td>
<td>NT 3.5</td>
<td>7M</td>
</tr>
<tr>
<td>96</td>
<td>NT 4.0</td>
<td>11M</td>
</tr>
<tr>
<td>00</td>
<td>2000</td>
<td>29M</td>
</tr>
<tr>
<td>01</td>
<td>XP</td>
<td>45M</td>
</tr>
<tr>
<td>03</td>
<td>Server 2003</td>
<td>50M</td>
</tr>
<tr>
<td>91</td>
<td>Linux kernel</td>
<td>10K</td>
</tr>
<tr>
<td>94</td>
<td>Linux 1.0.0</td>
<td>176K</td>
</tr>
<tr>
<td>12</td>
<td>Linux 3.3 kernel</td>
<td>15M</td>
</tr>
<tr>
<td>05</td>
<td>MacOS 10.4</td>
<td>86M</td>
</tr>
<tr>
<td>07</td>
<td>Debian 4.0</td>
<td>283M</td>
</tr>
<tr>
<td>09</td>
<td>Debian 5.0</td>
<td>324M</td>
</tr>
</tbody>
</table>
Chapter 2: Roadmap

2.1 OS objectives and functions
2.2 Evolution of the OS
2.3 Major achievements
2.4 Other developments
2.5 Virtual Machines
2.6 Multi-core considerations
2.7 MS Windows
2.6 Traditional UNIX
2.7 Modern UNIX
2.8 Linux
**Major Achievements**

- Impossible to deal with OS complexity without certain systematic ways of managing resources, jobs, and users

- Major advances in the development of operating systems (layout of the book):
  - Processes and threads (ch. 3-4)
  - IPC (inter-process communication) and synchronization mechanisms (ch. 5-6)
  - File systems (ch. 11-12)
  - Memory (RAM) management (ch. 7-8)
  - Scheduling and resource allocation (ch. 9-10)
  - Information protection and security (ch. 14-15)

covered in this class
Chapter 2: Roadmap

2.1 OS objectives and functions
2.2 Evolution of the OS
2.3 Major achievements
2.4 Other developments
2.5 Virtual Machines
2.6 Multi-core considerations
2.7 MS Windows
2.6 Traditional UNIX
2.7 Modern UNIX
2.8 Linux
MS Windows

Hardware abstraction layer (hal.dll)

Simple kernel (ntoskrnl.exe)

Native API (ntdll.dll)

Win32 API (kernel32.dll, user32.dll, gdi32.dll)

Wrappers and frameworks (MFC, .NET, msvcrtd.dll)

Session manager
Winlogon
Security policy
Service manager
Svchost.exe
Spooler
Explorer
Task manager

I/O manager
Cache manager
File system manager
Device drivers

Kernel system processes
Process and thread manager
Object manager
Virtual memory manager

System services

Window manager
GDI
Graphics drivers

Managed user processes

low-level user processes

managed user processes

user mode

kernel mode

these APIs studied in homework

Virtual memory manager

Object manager

Process and thread manager

Runtime libraries

Simple kernel (ntoskrnl.exe)

Hardware abstraction layer (hal.dll)

Hardware

Window manager

GDI

Graphics drivers

I/O manager

Cache manager

File system manager

Device drivers

Session manager

Winlogon

Security policy

Service manager

Svchost.exe

Spooler

Explorer

Task manager

Virtual memory manager

Object manager

Process and thread manager

Runtime libraries

Simple kernel (ntoskrnl.exe)

Hardware abstraction layer (hal.dll)

Hardware

System services

Kernel system processes
Homework #1

- When running A*
  - Incorrect # of nodes if weight is integer in $q = L + \frac{w}{(d+1)}$

- Basic BFS and DFS
  - Order of traversal on this graph?

Adjacent list
A: E, D, B
B: A, G
C: E, D, F
D: A, C
E: A, C
F: C, G
G: B, F
Homework #1

- Refresh the concept of search
  - Assume an undirected graph $G = (V,E)$
  - Start node $s \in V$

- Maintain two structures
  - Unexplored set $U$
  - Discovered set $D$

- Approach #1:

```
U.add(s)
while ( U.notEmpty() )
  x = U.removeNextNode() // node to explore
  if ( D.find(x) == true ) // if already explored, ignore
      continue
  N = G.getNeighbors(x) // N is a set of nodes
  if ( N.size() == 0 ) break // exit?
  for each y in N
      U.add(y)
```

Any problems?
Homework #1

• This code fails to actually insert anything into D
• Correct version:

```java
U.add(s)
while (U.notEmpty())
    x = U.removeNextNode()
    if (D.find(x) == true) // if already explored, ignore
        continue
    D.add(x)
    N = G.getNeighbors(x)
    if (N.size() == 0) break // exit? for each y in N
        U.add(y)
```

Any drawbacks?

• Requires huge storage as each node may be pushed into U as many times as there are links to it
  - Not advisable in practice
Homework #1

• Approach #2 inserts a single copy of each node in U:

```
U.add(s); D.add(s);  // s = source node
while ( U.notEmpty() )
    x = U.removeNextNode()
    N = G.getNeighbors(x)
    if ( N.size() == 0 ) break  // exit?
    for each y in N
        if ( D.find(y) == false ) // has been pushed in U?
            U.add(y)
            D.add(y)
U.add(s);
D.add(s);
```

Always use this version!

• For most types of non-trivial exploration, approach #2 is far superior to #1

• What if D has a function that combines find/add?
  - Can directly use STL set’s insert() function
Homework #1

- When you find the exit, how far is it from s?
- **Idea**: make U keep track of tuples (nodeID, distance)

```c
U.add (s, 0); D.add (s);
while ( U.notEmpty () )
    t = U.removeNextTuple () // t is a tuple
    N = G.getNeighbors (t.ID)
    if ( N.size() == 0 )
        printf ("Found at distance %d\n", t.distance)
        break
    for each y in N
        if ( D.find (y) == false ) // new node?
            U.add (y, t.distance + 1)
            D.add (y)
```

- Note that U.add() also needs light intensity for bFS/A*
  - See the handout for details
Homework #1

- Reusing the search algorithm
  - Create a base class

```cpp
class Ubase {
    virtual void Add (uint64 ID, int distance, float intensity) = 0;
    virtual UnexploredRoom RemoveNextTuple (void) = 0;
    ...
}
```

- Inherit four classes

```cpp
class Ubreadth : public Ubase {
    // implement a queue here
}
class Udepth : public Ubase {
    // implement a stack here
}
...
```

- Create base pointer to a specific class, then send it to search()

```cpp
Ubase *ptr;
if (searchType == BFS)
    ptr = new Ubreadth;
else if ...
    Search (ptr);

Search (Ubase *U)
{
    while (U->size() > 0)
        ...
}
```