Lecture - II

OS Structures

Tevfik Koşar

Louisiana State University
January 18th, 2007

Roadmap

- OS Operations
  - Kernel vs User Mode
- OS Structures
  - Multiprogramming and Multitasking
  - Storage Structure
  - System Calls
Operating System Operations

- **Interrupt driven** by hardware
- Unexpected errors can happen anytime
  - Software error or request creates *exception* or *trap*
    - eg. division by zero, invalid memory access
  - Other process problems include infinite loop, processes modifying each other or the operating system
- OS needs to protects itself
  ➔ **Dual-mode** operation

Dual-Mode Operation

- **Dual-mode** operation allows OS to protect itself and other system components
  - *User mode* and *kernel mode*
  - **Mode bit** provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Protects OS from errant users, and errant users from each other
    - Some instructions designated as *privileged*, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user
**Transition from User to Kernel Mode**

- How to prevent user program getting stuck in an infinite loop / process hogging resources
  - **Timer**: Set interrupt after specific period (1ms to 1sec)
    - Operating system decrements counter
    - When counter zero generate an interrupt
    - Set up before scheduling process to regain control or terminate program that exceeds allotted time

**Operating System Structure**

- **Multiprogramming** needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - How it works:
    - A subset of total jobs in system is kept in memory simultaneously
    - One job selected and run via job **scheduling**
    - When it has to wait (for I/O for example), OS switches to another job
Operating System Structure

- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive computing**
  - **Response time** should be < 1 second
  - Each user has at least one program loaded in memory and executing ⇒ **process**
  - If several jobs ready to be brought into memory ⇒ **job scheduling**
  - If several jobs ready to run at the same time ⇒ **CPU scheduling**
  - If processes don’t fit in memory, **swapping** moves them in and out to run

Storage Structure

- **Main memory** - only large storage media that the CPU can access directly.
- **Secondary storage** - extension of main memory that provides large nonvolatile storage capacity.
- **Magnetic disks** - rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into **tracks**, which are subdivided into **sectors**.
  - The **disk controller** determines the logical interaction between the device and the computer.
Tertiary Storage: low cost, high capacity storage
- eg. tape libraries, CD, DVD, floppy disks

Tape is an economical medium for purposes that do not require fast random access, e.g., backup copies of disk data, holding huge volumes of data.

Large tape installations typically use robotic tape changers that move tapes between tape drives and storage slots in a tape library.
- stacker - library that holds a few tapes
- silo - library that holds thousands of tapes
Storage Hierarchy

- Storage systems organized in hierarchy.
  - Speed
  - Cost
  - Volatility*
- **Caching** - copying information into faster storage system; main memory can be viewed as a last cache for secondary storage.

*volatile: loses its content when the power is off.
Performance of Various Levels of Storage

- Movement between levels of storage hierarchy can be explicit or implicit

<table>
<thead>
<tr>
<th>Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>registers</td>
<td>cache</td>
<td>main memory</td>
<td>disk storage</td>
</tr>
<tr>
<td>Typical size</td>
<td>&lt; 1 KB</td>
<td>&gt; 16 MB</td>
<td>&gt; 16 GB</td>
<td>&gt; 100 GB</td>
</tr>
<tr>
<td>Implementation technology</td>
<td>custom memory with multiple ports, CMOS</td>
<td>on-chip or off-chip CMOS SRAM</td>
<td>CMOS DRAM</td>
<td>magnetic disk</td>
</tr>
<tr>
<td>Access time (ns)</td>
<td>0.25 – 0.5</td>
<td>0.5 – 25</td>
<td>80 – 250</td>
<td>5,000,000</td>
</tr>
<tr>
<td>Bandwidth (MB/sec)</td>
<td>20,000 – 100,000</td>
<td>5000 – 10,000</td>
<td>1000 – 5000</td>
<td>20 – 150</td>
</tr>
<tr>
<td>Managed by</td>
<td>compiler</td>
<td>hardware</td>
<td>operating system</td>
<td>operating system</td>
</tr>
<tr>
<td>Backed by</td>
<td>cache</td>
<td>main memory</td>
<td>disk</td>
<td>CD or tape</td>
</tr>
</tbody>
</table>

Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy
Migration of Integer A from Disk to Register

- Multitasking environments must be careful to use most recent value, not matter where it is stored in the storage hierarchy

![Diagram: Migration of Integer A from Disk to Register]

- Multiprocessor environment must provide cache coherency in hardware such that all CPUs have the most recent value in their cache
- Distributed environment situation even more complex
  - Several copies of a datum can exist
  - Various solutions covered in Chapter 17

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use
  - Ease of programming
  - Portability
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
Example of System Calls

- System call sequence to copy the contents of one file to another file

Example System Call Sequence
- Acquire input file name
- Write prompt to screen
- Accept input
- Acquire output file name
- Write prompt to screen
- Accept input
- Open the input file
  - if file doesn’t exist, abort
- Create output file
  - if file exists, abort
- Loop
  - Read from input file
  - Write to output file
  - Until read fails
- Close output file
- Write completion message to screen
- Terminate normally

Example of Standard API

- Consider the ReadFile() function in the Win32 API—a function for reading from a file

```c
BOOL ReadFile (HANDLE file, LPVOID buffer, DWORD bytesToRead, LPDWORD bytesRead, LPOVERLAPPED ovl);
```

- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - LPOVERLAPPED ovl—indicates if overlapped I/O is being used
System Call Implementation

- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

API - System Call - OS Relationship
Standard C Library Example

- C program invoking printf() library call, which calls write() system call

```c
#include <stdio.h>
int main ()
{
    printf("Greetings");
    return 0;
}
```

System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - eg. source device, address and length of memory buffer
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in **registers**
    - In some cases, may be more parameters than registers
  - Parameters stored in a **block**, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
    - Block and stack methods do not limit the number or length of parameters being passed
Parameter Passing via Table

Solaris System Call Tracing

```
$ ./all.d 'pgrep xclock' XEventsQueued
   dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued   U
  0 -> _XEventsQueued  U
  0 -> _XIlTransBytesReadable U
  0 <- _XIlTransBytesReadable U
  0 -> _XIlTransSocketBytesReadable U
  0 <- _XIlTransSocketBytesReadable U
  0 -> ioctl  U
  0  -> ioctl
  0  -> getfd
  0  -> set_active_fd
  0  <- set_active_fd
  0  <- getfd
  0  -> get_userdata
  0  <- get_userdata
  ...
  0  -> release
  0  -> clear_active_fd
  0  <- clear_active_fd
  0  <- cv_broadcast
  0  <- cv_broadcast
  0  <- release
  0  <- ioctl  U
  0  <- ioctl
  0  -> XEventsQueued  U
  0  <- XEventsQueued  U
```
Any Questions?

Hmm..

Reading Assignment

- Read chapter 2 from Silberschatz.
Acknowledgements