Operating System Design and Implementation

- Start by defining goals and specifications
- Affected by choice of hardware, type of system
  - Batch, time shared, single user, multi user, distributed
- 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
- No unique solution for defining the requirements of an OS
  \( \rightarrow \) Large variety of solutions
  \( \rightarrow \) Large variety of OS

Operating System Design and Implementation (Cont.)

- Important principle: to separate policies and mechanisms
  - Policy: What will be done?
  - Mechanism: How to do something?
- Eg. to ensure CPU protection
  - Use Timer construct (mechanism)
  - How long to set the timer (policy)
- The separation of policy from mechanism is allows maximum flexibility if policy decisions are to be changed later

OS Design Approaches

- Simple Structure
- Layered Approach
- Microkernels
- Modules

Simple Structure

- No well defined structure
- Start as small, simple, limited systems, and then grow
- MS-DOS - written to provide the most functionality in the least space
  - Not divided into modules
  - Its interfaces and levels of functionality are not well separated
  - e.g. application programs can access low level drivers directly
  \( \rightarrow \) Vulnerable to errant (malicious) programs
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

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
  - GLUnix: Global Layered Unix

Microkernel System Structure

• Move all non-essential components from the kernel into “user” space
• Main function of microkernel: Communication between client programs and various services which are run in user space
  - Uses message passing (never direct interaction)
• Benefits:
  - Easier to extend the OS
  - Easier to port the OS to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
• Detriments:
  - Performance overhead of user space to kernel space communication
• Examples: QNX, Tru64 UNIX
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 more flexible
  - Any module can call any other module

Solaris Modular Approach

Mac OS X Structure - Hybrid

- BSD: provides support for command line interface, networking, file system, POSIX API and threads
- Mach: memory management, RPC, IPC, message passing

Virtual Machines

- A virtual machine takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware
- A virtual machine provides an interface identical to the underlying bare hardware
- The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory

Virtual Machines (Cont.)

- The resources of the physical computer are shared to create the virtual machines
  - CPU scheduling can create the appearance that users have their own processor
  - Spooling and a file system can provide virtual card readers and virtual line printers
  - A normal user time-sharing terminal serves as the virtual machine operator’s console

Virtual Machines (Cont.)
Virtual Machines (Cont.)

• The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.
• A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
• The virtual machine concept is difficult to implement due to the effort required to provide an exact duplicate to the underlying machine.

VMware Architecture

The Java Virtual Machine

The Java Virtual Machine

Processes

Process Concept

• An operating system executes a variety of programs:
  - Batch system - jobs
  - Time-shared systems - user programs or tasks
• Process - a program in execution; process execution must progress in sequential fashion
• A process includes:
  - program counter
  - stack: temporary data
  - heap: dynamic memory
  - data section: global variables

Process State

• As a process executes, it changes state
  - new: The process is being created
  - running: Instructions are being executed
  - waiting: The process is waiting for some event to occur
  - ready: The process is waiting to be assigned to a process
  - terminated: The process has finished execution
Process Control Block (PCB)

Information associated with each process
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

CPU Switch From Process to Process

Process Scheduling Queues

- Job queue - set of all processes in the system
- Ready queue - set of all processes residing in main memory, ready and waiting to execute
- Device queues - set of processes waiting for an I/O device
- Processes migrate among the various queues

Ready Queue And Various I/O Device Queues

Representation of Process Scheduling
Schedulers

- **Long-term scheduler** (or job scheduler) - selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) - selects which process should be executed next and allocates CPU

Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
- The long-term scheduler controls the degree of multiprogramming
- Processes can be described as either:
  - I/O-bound process - spends more time doing I/O than computations, many short CPU bursts
  - CPU-bound process - spends more time doing computations; few very long CPU bursts
⇒ long-term schedulers need to make careful decision

Addition of Medium Term Scheduling

- In time-sharing systems: remove processes from memory “temporarily” to reduce degree of multiprogramming.
- Later, these processes are resumed ⇒ **Swaping**

Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - fork system call creates new process
  - exec system call used after a fork to replace the process’ memory space with a new program

C Program Forking Separate Process

```c
int main()
{
    Pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) {
        /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) {
        /* child process */
        execvp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit(0);
    }
}
```
A tree of processes on a typical Solaris

- sched: root process for OS
- pageout: manages memory
- fsflush: manages file system
- init: root for user processes
- inetd: Networking
- dtlogin: user login screen
- ...

⇒ Unique process id's

Any Questions?

Hmm...

Reading Assignment

- Read chapter 3 from Silberschatz.

Acknowledgements