The Process Abstraction

Ken Birman

Operating System Structure

- An OS is just another kind of program running on the CPU – a process:
  - It has main() function that gets called only once (during boot)
  - Like any program, it consumes resources (such as memory)
  - Can do silly things (like generating an exception), etc.

Operating System Structure

- An OS is just another kind of program running on the CPU – a process… But it is a very sophisticated program:
  - "Entered" from different locations in response to external events
  - Does not have a single thread of control
    - can be invoked simultaneously by two different events
    - e.g. sys call & an interrupt
  - It is not supposed to terminate
  - It can execute any instruction in the machine

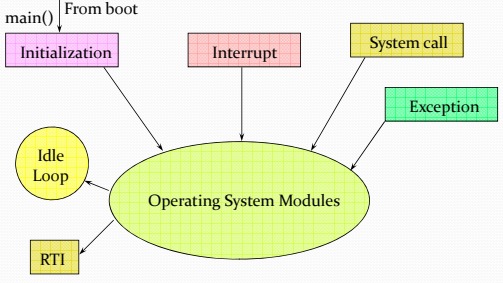
Booting an OS

- Your computer has a very simple program pre-loaded in a special read-only memory
  - The Basic Input/Output Subsystem, or BIOS
- When the machine boots, the CPU runs the BIOS
- The BIOS, in turn, loads a "small" O/S executable
  - From hard disk, CD-ROM, or whatever
  - Then transfers control to a standard start address in this image

Booting an OS

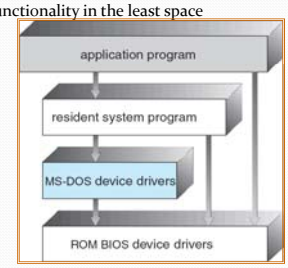
- The small version of the O/S loads and starts the "big" version.
  - The two stage mechanism is used so that BIOS won't need to understand the file system implemented by the "big" O/S kernel
  - File systems are complex data structures and different kernels implement them in different ways
  - The small version of the O/S is stored in a small, special-purpose file system that the BIOS does understand
- Some computers are set up to boot to one of several O/S images. In this case BIOS asks you to pick

OS Control Flow

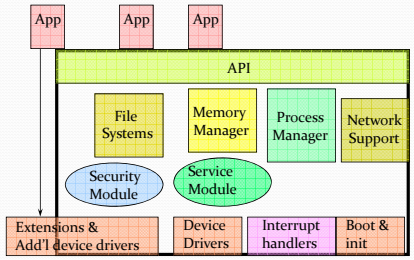


Operating System Structure

- Simple Structure: MS-DOS
  - Written to provide the most functionality in the least space
  - Applications have direct control of hardware
- Disadvantages:
  - Not modular
  - Inefficient
  - Low security



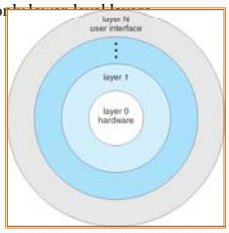
General OS Structure



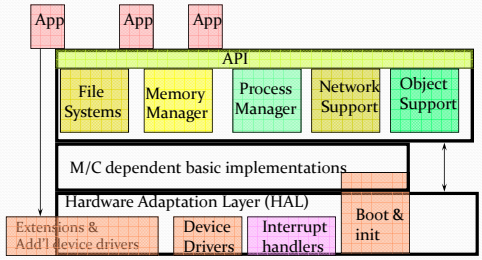
Monolithic Structure

Layered Structure

- OS divided into number of layers
  - bottom layer (layer 0), is the hardware
  - highest (layer N) is the user interface
  - each uses functions and services of only lower-level layers
- Advantages:
  - Simplicity of construction
  - Ease of debugging
  - Extensible
- Disadvantages:
  - Defining the layers
  - Each layer adds overhead



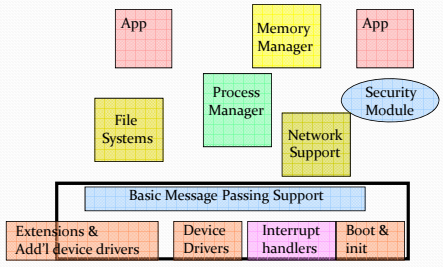
Layered Structure



Microkernel Structure

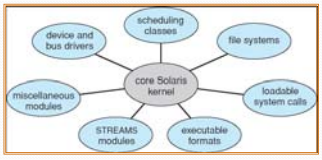
- Moves as much from kernel into "*user*" space
- User modules communicate using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
  - Example: Mach, QNX
- Detriments:
  - Performance overhead of user to kernel space communication
  - Example: Evolution of Windows NT to Windows XP

Microkernel Structure



Modules

- Most modern OSs 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
- Examples: Solaris, Linux, MAC OS X



Extensions

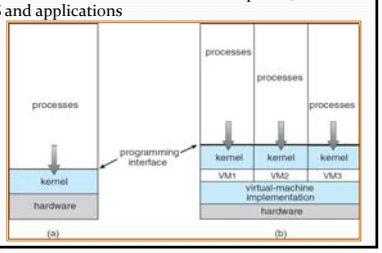
- Most modern kernels allow the user to add new kernel functions (if you have the right permissions)
  - Idea is that sometimes, the set of existing system calls isn't adequate
  - A good example: Modern data centers, like Google, need applications that "inspect" network packets
    - Traffic arrives over the Internet at incredibly high speed: 10Gbits/second
    - Need to pass them to one of perhaps 20,000 "first line" web servers
    - But need to look at them to decide which packet goes to which server
    - No time to pass them up to a user-mode program
- Extension: *user-coded module that runs in the kernel* (only) for situations where speed is key to success

A collection of virtual machines

- A good way to think of the O/S is as a creator of virtual machine environments
  - Your program sees what it thinks of as the O/S
  - The O/S runs on the raw hardware and creates the environment for your program to run in
- Even kernel modules live in a kind of virtual machine
  - Of course, the environment and operations available are very different than for a user program
  - Can do things users can't… and need to obey rules that user programs aren't subjected to

Revisit: Virtual Machines

- Implements an observation that dates to Turing
  - One computer can "emulate" another computer
  - One OS can implement abstraction of a cluster of computers, each running its own OS and applications
- Incredibly useful!
  - System building
  - Protection
- Cons
  - implementation
- Examples
  - VMWare, JVM



OS "Process" in Action

- OS runs user programs, if available, else enters idle loop
- In the idle loop:
  - OS executes an infinite loop (UNIX)
  - OS performs some system management & profiling
  - OS halts the processor and enter in low-power mode (notebooks)
  - OS computes some function (DEC's VMS on VAX computed Pi)
- OS wakes up on:
  - interrupts from hardware devices
  - traps from user programs
  - exceptions from user programs

UNIX structure

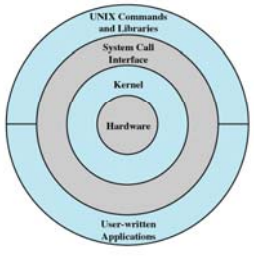


Figure 2.14 General UNIX Architecture

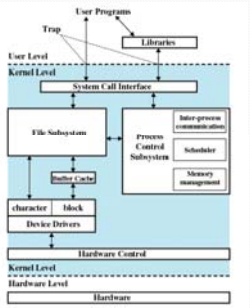


Figure 2.15 Traditional UNIX Kernel [BACH86]

Windows Structure

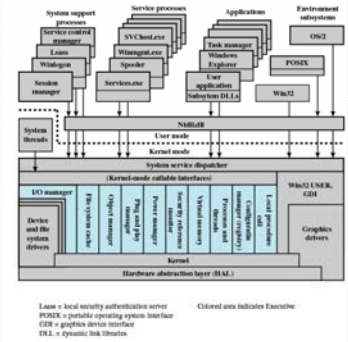


Figure 2.13 Windows 2000 Architecture [SOLO00]

Modern UNIX Systems

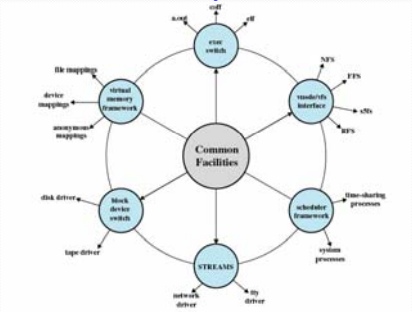
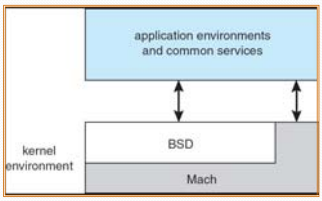
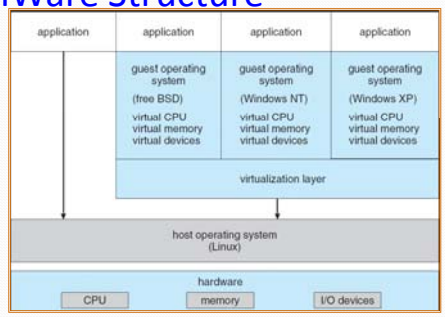


Figure 2.16 Modern UNIX Kernel [VAHA96]

MAC OS X



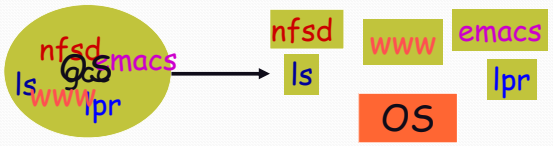
VMWare Structure



User-Mode Processes

Why Processes? Simplicity + Speed

- Hundreds of things going on in the system



- How to make things simple?
  - Separate each in an isolated process
  - Decomposition
- How to speed-up?
  - Overlap I/O bursts of one process with CPU bursts of another

What is a process?

- A task created by the OS, running in a restricted virtual machine environment –a virtual CPU, virtual memory environment, interface to the OS via system calls
- The unit of execution
- The unit of scheduling
- Thread of execution + address space
- Is a program in execution
  - Sequential, instruction-at-a-time execution of a program.

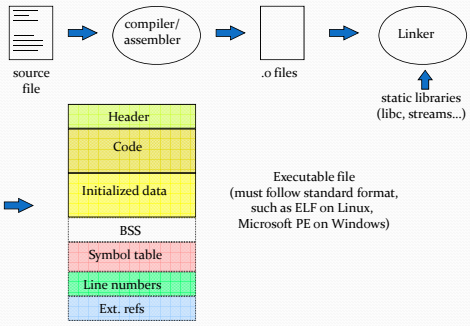
The same as "job" or "task" or "sequential process"

What is a program?

A program consists of:

- **Code:** machine instructions
- **Data:** variables stored and manipulated in memory
  - initialized variables (globals)
  - dynamically allocated variables (malloc, new)
  - stack variables (C automatic variables, function arguments)
- **DLLs:** libraries that were not compiled or linked with the program
  - containing code & data, possibly shared with other programs
- **mapped files:** memory segments containing variables (mmap())
  - used frequently in database programs
- ***A process is a executing program***

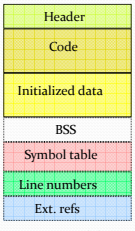
Preparing a Program



Running a program

- OS creates a "process" and allocates memory for it
- The loader:
  - reads and interprets the executable file
  - sets process's memory to contain code & data from executable
  - pushes "argc", "argv", "envp" on the stack
  - sets the CPU registers properly & calls "__start()" [Part of CRTo]
- Program start running at __start(), which calls main()
  - we say "process" is running, and no longer think of "program"
- When main() returns, CRTo calls "exit()"
  - destroys the process and returns all resources

Process != Program

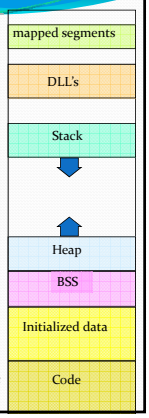


Executable

Program is passive
• Code + initial values for data

Process is running program
• stack, regs, program counter
• private copy of the data
• shared copy of the code

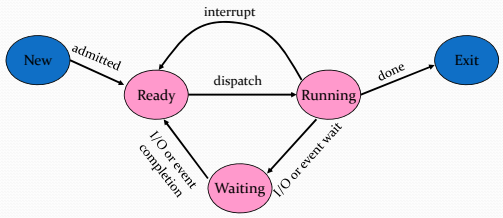
Example:
We both run IE on same PC:
- Same program
- Same machine
- *Different processes* **Process address space**



Process States

- Many processes in system, only one on CPU
- "Execution State" of a process:
  - Indicates what it is doing
  - Basically 3 states:
    - Ready: waiting to be assigned to the CPU
    - Running: executing instructions on the CPU
    - Waiting: waiting for an event, e.g. I/O completion
- Process moves across different states

Process State Transitions



Processes hop across states as a result of:
- Actions they perform, e.g. system calls
- Actions performed by OS, e.g. rescheduling
- External actions, e.g. I/O

Process Data Structures

- OS represents a process using a *PCB*
  - Process Control Block
  - Has all the details of a process

Process Id	Security Credentials
Process State	Username of owner
General Purpose Registers	Queue Pointers
Stack Pointer	Signal Masks
Program Counter	Memory Management
Accounting Info	…

Context Switch

- For a running process
  - All registers are loaded in CPU and modified
    - E.g. Program Counter, Stack Pointer, General Purpose Registers
- When process relinquishes the CPU, the OS
  - Saves register values to the PCB of that process
- To execute another process, the OS
  - Loads register values from PCB of that process

⇒**Context Switch**
- Process of switching CPU from one process to another
- Very machine dependent for types of registers

Details of Context Switching

- Very tricky to implement
  - OS must save state without changing state
  - Should run without touching any registers
    - CISC: single instruction saves all state
    - RISC: reserve registers for kernel
      - Or way to save a register and then continue
- Overheads: CPU is idle during a context switch
  - Explicit:
    - direct cost of loading/storing registers to/from main memory
  - Implicit:
    - Opportunity cost of flushing useful caches (cache, TLB, etc.)
    - Wait for pipeline to drain in pipelined processors

Context switching is costly!

- In systems that do excessive amounts of context switching, it balloons into a big overhead
  - This is often ignored by application developers
  - But if you split an application into multiple processes need to keep it in mind
    - Make sure that each process does big chunks of work
    - Think about conditions under which context switching could occur and make sure they are reasonably rare

How to create a process?

- Double click on a icon?
- After boot OS starts the first process
  - E.g. sched for Solaris, ntoskrnel.exe for XP
- The first process creates other processes:
  - the creator is called the parent process
  - the created is called the child process
  - the parent/child relationships is expressed by a process tree
- For example, in UNIX the second process is called *init*
  - it creates all the gettys (login processes) and daemons
  - it should never die
  - it controls the system configuration (#processes, priorities…)
- Explorer.exe in Windows for graphical interface

Processes Under UNIX

- Fork() system call is only way to create a new process
- int fork() does many things at once:
  - creates a new address space (called the child)
  - copies the parent's address space into the child's
  - starts a new thread of control in the child's address space
  - parent and child are equivalent -- almost
    - in parent, fork() returns a non-zero integer
    - in child, fork() returns a zero.
    - difference allows parent and child to distinguish
- int fork() returns TWICE!

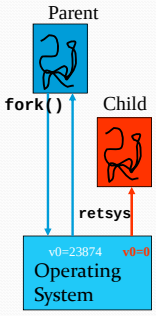
Example

```
main(int argc, char **argv)
{
   char *myName = argv[1];
   int cpid = fork();
   if (cpid == 0) {
      printf("The child of %s is %d\n", myName, getpid());
      exit(0);
   } else {
      printf("My child is %d\n", cpid);
      exit(0);
   }
}
```

What does this program print?

Bizarre But Real

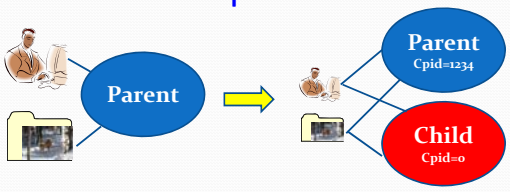
```
lace: <15> cc a.c
lace: <16> ./a.out foobar
The child of foobar is 23874
My child is 23874
```



Shared memory: For efficiency

- fork() actually *shares memory* between parent, child!
  - ***But if a page is modified, by either, fork duplicates it***
    - ***Called "copy on write" sharing***
  - Also shares open files, pipes, even stack and registers
  - In fact, duplicates everything except the fork() return value
  - For code, data pages uses a concept called "copy on write"
  - If you call exec() this page-level sharing ends
- Unix (Linux) and Windows also provide system calls to let processes share memory by "mapping" a file into memory
  - You tell it where, or let it pick an address range
  - Mapped files are limited to one writer. Can have many readers

Fork creates parallelism…



- Initially, child is a clone of parent except for "pid"
- Even share file descriptors for files parent had open!
  - Linux: includes stdin, stdout, stderr
  - Plus files the parent explicitly opened.
  - Confusing: these shared files have a single "seek pointer". If parent and child both do I/O, they "contend" for access.

Weird (but real) race condition

- Suppose that both do
  - lseek(fileptr, addr, SEEK_SET);
  - read(fileptr, buffer, somebytes)
- If both issue these system calls concurrently, they can interleave, for example this way:
  - lseek(fileptr, chld-addr, SEEK_SET);
  - lseek(fileptr, parnt-addr, SEEK_SET);
  - read(fileptr, parent-buffer, somebytes)
  - read(fileptr, chld-buffer, somebytes)

Fork is half the story

- Fork() gets us a new address space,
  - but parent and child share EVERYTHING
    - memory, operating system state
- int exec(char *programName) completes the picture
  - throws away the contents of the calling address space
  - replaces it with the program named by programName
  - starts executing at header.startPC
  - Does not return
- Pros: Clean, simple
- Con: duplicate operations

Fork+exec to start a new program

```
main(int argc, char **argv)
{
   char *myName = argv[1];
   char *progName = argv[2];

   int cpid = fork();
   if (cpid == 0) {
      printf("The child of %s is %d\n", myName, getpid());
      execlp("/bin/ls",   // executable name
             "ls", NULL); // null terminated argv
      printf("Error:/bin/ls cannot be executed\n");
 } else {
      printf("My child is %d\n", cpid);
      exit(0);
   }
}
```

Process Termination

- Process executes last statement and OS decides(**exit**)
  - Child: OS keeps some data for the parent to collect (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of child process (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some OSes don't allow child to continue if parent terminates
      - All children terminated - *cascading termination*

ProcExp Demo

- Windows process hierarchy
- explorer.exe and the system idle process
- Windows base priority mechanism
  - 0, 4, 8, 13, 24
  - What is procexp's priority?
- Creating a new process
- Terminating a process