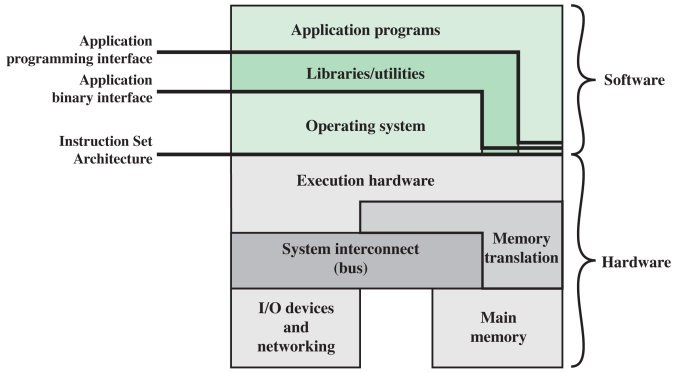
## Computing Infrastructure






## Operating System

- An OS functions in the same way as ordinary computer software: suite of programs, executed by the processor
  - Frequently relinquishes control and depends on the processor to regain control
- A program that controls the execution of application programs
  - Computer = set of resources for the movement, storage, and processing of data
  - The OS is responsible for managing these resources
- An interface between applications and hardware
  - Program development and then execution
  - Access I/O devices, including controlled access to files
  - System access
  - Error detection and response
  - Accounting
- Main objectives of an OS: convenience, efficiency, ability to evolve



## OS Evolution

Serial Processing $\longrightarrow$ Batch Processing $\longrightarrow$ Multitasking $\longrightarrow$ Time Sharing

- Earliest computers = serial processing
  - No OS – programmers interact directly with the computer hardware
  - Users have access to the computer in series
  - Scheduling: sign-up sheet (hardcopy!), wasted (computer and human) time
  - Considerable setup time
- Simple batch systems = jobs submitted to an operator who batches them and feed them to a computer
  - Uses a monitor to control the sequence of events
  - The resident monitor (always in memory) reads in job and gives control to them
  - Job returns control to monitor once finished
  - Sacrifices memory and CPU time (to the monitor), but nonetheless improves the utilization of the computer

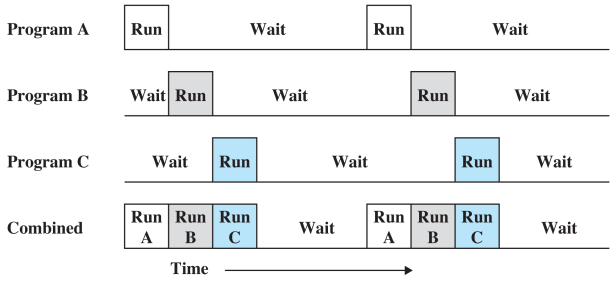


## Batch Systems (cont'd)

- Special programming language to provide instructions to the monitor: job control language (JCL) (what compiler and what data to use, etc.)
- Needed hardware features:
  - Memory protection for monitor (user program must not alter the memory area containing the monitor)
  - Timer (prevents a job from monopolizing the system)
  - Privileged instructions (can only be executed by the monitor)
  - Interrupts (more flexibility in controlling user programs)
- User mode for user programs
  - Certain areas of memory are protected from user access
  - Certain instructions may not be executed
- Kernel mode for the monitor
  - Privileged instructions may be executed
  - Protected areas of memory may be accessed

## MULTITASKING (MULTIPROGRAMMING)

- Batch jobs execute one after another
  - The processor simply waits whenever I/O happens = processor is mostly idle
- If we have more than one program in memory we can simply run the others while one waits for I/O ⟶ multitasking






## TIME-SHARING SYSTEMS

- Handle multiple (interactive) jobs and multiple users
  - Processor time is shared among multiple users and jobs
  - Multiple users access simultaneously the system through terminals, with the OS interleaving the execution of each user program in a short burst or quantum of computation

| | Batch multitasking | Time sharing |
|---|---|---|
| Principal objective | Maximize CPU use | Maximize response time |
| Source of directives to the OS | JCL commands (provided with the job) | Terminal commands (interactive) |

- Early time-sharing OS: Compatible Time-Sharing Systems or CTSS (MIT, 1961)
  - System clock generates interrupts approximately every 0.2 seconds
  - At each interrupt OS regains control and can assign CPU to another user
  - At regular intervals the current user is preempted and another user is loaded
  - Old user code and data are written out to disk
  - Code and data is restored in memory when that program is next given a turn

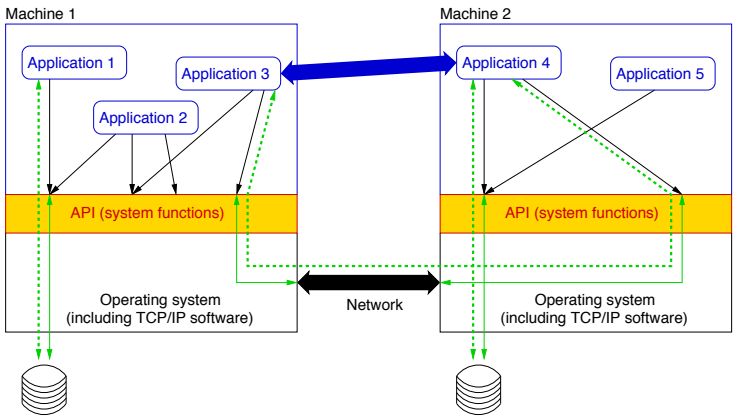


## THE REST IS HISTORY

- Operating Systems are among the most complex pieces of software ever developed
- Major advances in development include:
  - Processes
  - Memory management
  - Information protection and security
  - Scheduling and resource management
  - System structure



## A USER PERSPECTIVE: SYSTEM CALLS



## System Calls (cont'd)

- Programming interface to the services provided by the OS, forms an Application Binary Interface (ABI)
- Typically written in a C-like language (C, C++, Objective-C)
- Accessed by programmers via an Application Programming Interface (API)
- Most common APIs: Win32 (Windows), POSIX (virtually all versions of UNIX, Linux, and Mac OS X), Java (Java VM)
- In Unix, the description of all the API calls are in Section 2 of the manual pages: `man -S2 write`
  - Some other, useful functions found in Section 3 (Standard C library but not OS-related): `man -S3 printf`
- interesting APIs: processes, terminal I/O, file I/O, networking (or Berkeley sockets)



## API Example: File I/O

| Operation | Meaning |
|---|---|
| `open` | prepares a file for I/O operations<br>returns a file descriptor (`int`) used by all the other operations |
| `close` | terminates the use of a previously opened file/device |
| `read` | obtain data from a file/device |
| `write` | write data to a file/device |
| `lseek` | move to some position in the file/device<br>(not applicable to all devices) |

- For example,

```
char result[256];
int file = open("echo",O_RDONLY);
if (file == -1)
  return 1;
read(file,result,255);
close(file);
```



## File I/O (cont'd)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

int main (int argc, char** argv) {
  int file = open(argv[1], O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
  int i = 0; char prefix[12]; char message[256] = "something";
  if (file == -1) return 1;
  while (true) {
    i++;
    printf("Message: ");
    fgets(message,255,stdin);
    if (message[strlen(message)-1] == '\n') message[strlen(message)-1] == '\0';
    if (strlen(message) == 0) break;
    snprintf(prefix, 12, "%d: ", i);
    write(file, prefix, strlen(prefix));
    write(file, message, strlen(message));
    write(file, "\n", 1);
  }
  close(file);
}
```



## Text Files

```
int readline(int fd, char* buf_str, size_t max) {
  size_t i;
  for (i = 0; i < max; i++) {
    char  tmp;
    int what = read(fd,&tmp,1);
    if (what == 0 || tmp == '\n') {
      buf_str[i] = '\0';
      return i;
    }
    buf_str[i] = tmp;
  }
  buf_str[i] = '\0';
  return i;
}
```

```
int dbf = open("myfile",O_RDONLY);
if (dbf == -1) {
  perror(myfile);
  exit(1);
}
char message[256];
int nc = readline(dbf,message,255);
```

## TEXT FILES (CONT'D)

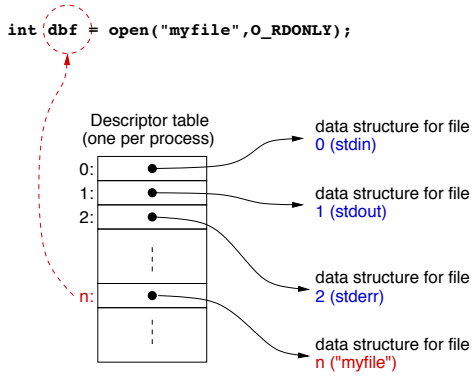
```
const int recv_nodata = -2;

int readline(const int fd, char* buf, const size_t max) {
    size_t i;    int begin = 1;

    for (i = 0; i < max; i++) {
        char tmp;
        int what = read(fd,&tmp,1);
        if (what == -1) return -1;
        if (begin) {
            if (what == 0)
                return recv_nodata;
            begin = 0;
        }
        if (what == 0 || tmp == '\n') {
            buf[i] = '\0';
            return i;
        }
        buf[i] = tmp;
    }
    buf[i] = '\0';
    return i;
}
```



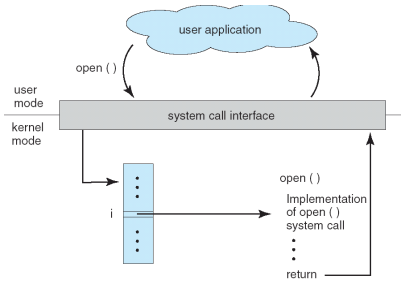
## FILE DESCRIPTORS






## SYSTEM CALL IMPLEMENTATION

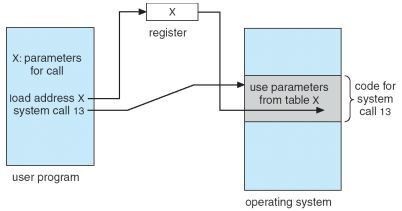
- Typically, a number (index) is associated with each system call
  - The system-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in kernel and returns status and any return values upon completion
- The caller need know nothing about how the system call is implemented
  - Details of OS interface hidden from programmer, managed by run-time libraries






## SYSTEM CALL IMPLEMENTATION (CONT'D)

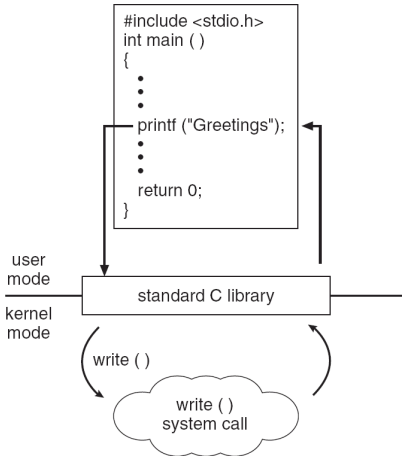
- Often, more information is required than simply identity of desired system call (parameters); how to pass that data?
  - Pass the parameters in registers (but what if we have more parameters than registers??)
  - Parameters stored in a block or table in memory, address of block passed as a parameter in a register (Linux, Solaris)




  - Parameters pushed on the stack by the program and popped off by the OS

## Other Functions in the Standard C Library

- C program calls: `printf()` (library call)
- `printf()` in turn will call `write()` (system call)






## Even More Wrappers: System Programs

- Provide a convenient environment for program development and execution
  - File manipulation
  - Status information
  - Programming language support
  - Program loading and execution
  - Communication
- Most users' view of the operation system is defined by system programs, not the actual system calls
- Some are simply interfaces to system calls; others are considerably more complex
- Example: File management: `ls`, `cp`, `mv`, `rm`, `cat`, `more`, ...
- Example: Status information
  - Some commands ask the system for info - date, time, amount of available memory, ...
  - Others provide detailed performance, logging, and debugging information



## OS Overview: Processes

- Processes are fundamental to the structure of operating systems
- A process is defined as:
  - A program in execution



## OS Overview: Processes

- Processes are fundamental to the structure of operating systems
- A process is defined as:
  - A program in execution
  - An instance of a running program

## OS Overview: Processes

- Processes are fundamental to the structure of operating systems
- A process is defined as:
  - A program in execution
  - An instance of a running program
  - The entity that can be assigned to, and executed on, a processor



## OS Overview: Processes

- Processes are fundamental to the structure of operating systems
- A process is defined as:
  - A program in execution
  - An instance of a running program
  - The entity that can be assigned to, and executed on, a processor
  - A unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources
- Concept refined by various issues found in three major lines of computer system development
  - multitasking batch operation (processor is switched among the various programs residing in main memory)
  - time sharing (be responsive to the individual user but be able to support many users simultaneously)
  - real-time transaction systems (a number of users are entering queries or updates against a database)



## Processes Are Tricky

A multi-process environment can fail in new, wonderful ways

- Improper synchronization
  - a program must wait until the data are available in a buffer
  - improper design of the signaling mechanism can result in loss or duplication
- Failed mutual exclusion
  - many users or programs attempt to use a shared resource at the same time
  - only one routine at at time allowed can manipulate the resource
- Nondeterminate program operation
  - program execution is interleaved by the processor
  - the order in which programs are scheduled may affect their outcome
- Deadlocks
  - it is possible for two or more programs to be hung up waiting for each other
  - may depend on the chance timing of resource allocation and release



## Process Components

- An executable program
- The associated data needed by the program (variables, work space, buffers, etc.)
- The execution context (or process state) of the program
  - Essential resources in time-sharing systems
  - It is the internal data by which the OS is able to supervise and control the process
  - Includes the contents of the various process registers
  - Includes information such as the priority of the process and whether the process is waiting for the completion of a particular I/O event
  - The entire state of the process at any instant is contained in its context
  - New features can be designed and incorporated into the OS by expanding the context to include any new information needed to support the feature

## MULTITHREADING

- Thread = dispatchable unit of work
  - Includes a processor context and its own data area to enable subroutine branching
  - Executes sequentially and is interruptible
- Process = a collection of one or more threads and associated system resources
  - A unit of activity characterized by ~~a single sequential thread of execution,~~ a current state, and an associated set of system resources
  - Programmer has greater control over the modularity of the application and the timing of application related events



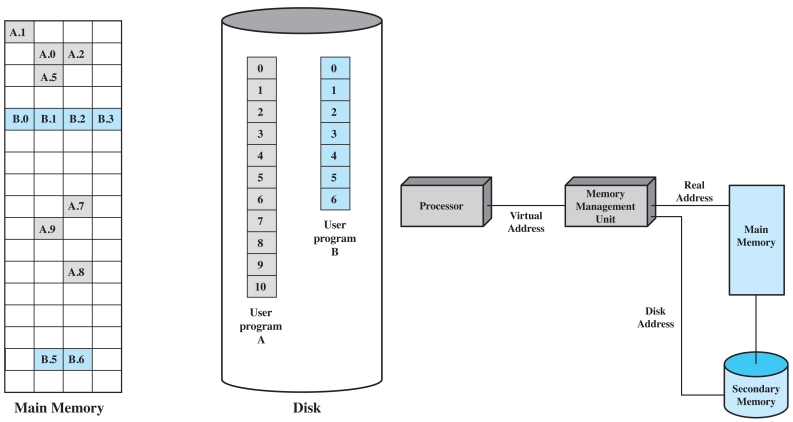
## MEMORY MANAGEMENT SOLUTIONS

- The OS has five principal storage management responsibilities:

  Process isolation
  Automatic allocation and management
  Support for modular programming
  Protection and access control
  Long-term storage
- Virtual memory
  - A facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available
  - Conceived to meet the requirement of having multiple user jobs reside in main memory concurrently
- Paging
  - Allows processes to be comprised of a number of fixed-size blocks, called pages
  - Program references a word by means of a virtual address = page number + offset within the page (each page may be located anywhere in main memory)
  - Provides for a dynamic mapping between the virtual address used in the program and a real (or physical) address in main memory



## VIRTUAL MEMORY






## RESOURCE MANAGEMENT

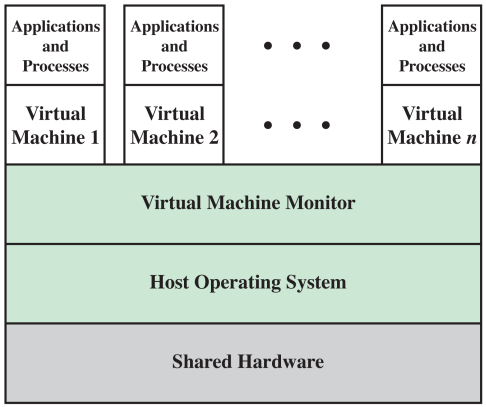
- Key responsibility of an OS is managing resources
- Allocation policies must consider efficiency, fairness, differential responsiveness
- On top of all of this one must consider information protection and security
  - Main issues: availability, authenticity, data integrity, confidentiality
  - The nature of the threat that concerns an organization will vary greatly depending on the circumstances
  - The problem involves controlling access to computer systems and the information stored in them
  - No perfect solution!

## Architectural Approaches

- Monolithic kernel versus microkernel
- Symmetric multiprocessing
  - Multi-core processing
- Distributed OS
  - Provides the illusion of a single main memory space, single secondary memory space, unified access facilities
- Modular design
  - Used for adding modular extensions to a kernel
  - Enables programmers to customize an operating system without disrupting system integrity
  - Eases the development of distributed tools and full-blown distributed operating systems
- Virtualization – "host" operating system can support a number of virtual machines

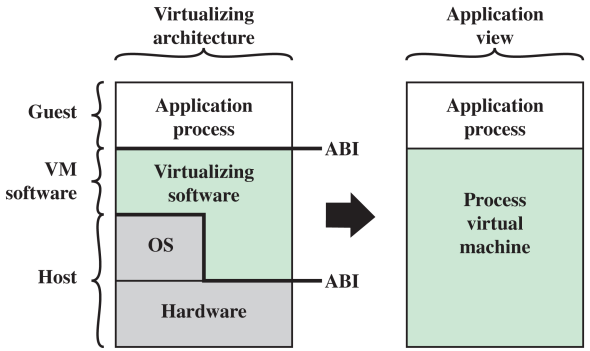


## Virtualization



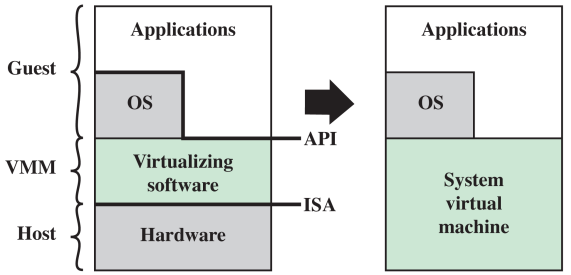



## Process Virtual Machine



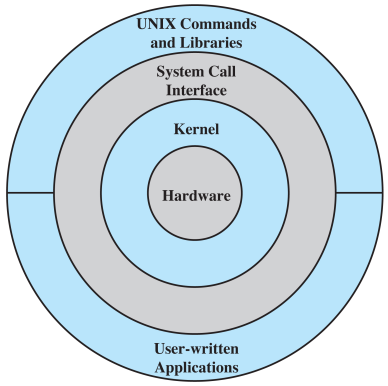



## System Virtual Machine



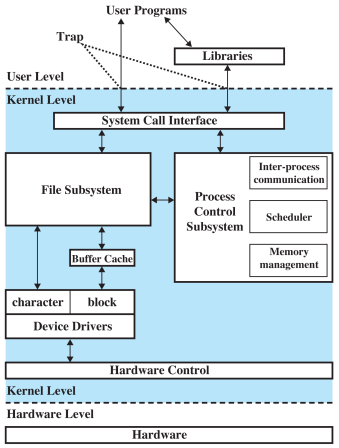
## TRADITIONAL UNIX

- Developed at Bell Labs and became operational on a PDP-7 in 1970
  - PDP-11 was then a milestone because it first showed that UNIX would be an OS for all computers
- Next milestone was rewriting UNIX in the programming language C
  - Demonstrated the advantages of using a high-level language for system code
- Was described in a technical journal for the first time in 1974
- First widely available version outside Bell Labs was Version 6 (1976)
- Version 7 (1978) is the ancestor of most modern UNIX systems
- Most important of the non-AT&T systems was UNIX BSD (Berkeley Software Distribution)



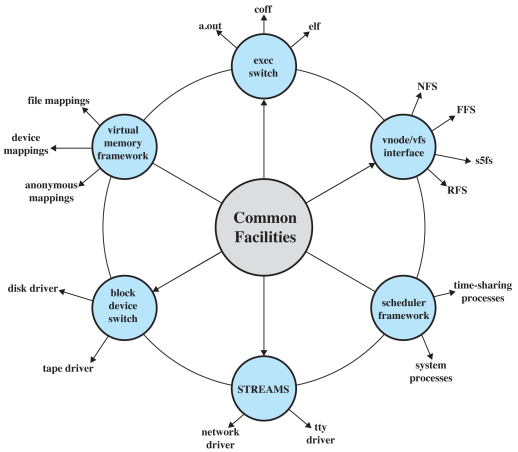



## TRADITIONAL UNIX KERNEL






## MODERN UNIX KERNEL






## LINUX

- Started out as a UNIX variant for the IBM PC; today a full-featured UNIX system that runs on several platforms
- Linus Torvalds wrote the initial version; first posted on the Internet in 1991
- Is free and the source code is available
- Highly modular and easily configured
- Modular, monolithic kernel
  - Includes virtually all of the OS functionality in one large block of code that runs as a single process with a single address space
  - All the components have access to all of its internal data and routines
  - Structured as a collection of modules
- Loadable modules
  - Relatively independent block, an object file whose code can be linked to and unlinked from the kernel at run time
  - Executed in kernel mode on behalf of the current process

## LINUX KERNEL COMPONENTS