Introduction to OS (cs1550)

• Why take this class? Why with Mosse?
  – it’s mandatory
  – it’s a great class
  – it’s a great prof
  – it’s easy (NOT!!!! do not fool thyself!)  
  – it’s good for you

• Life is not life anymore while this class is going on.  
  Be careful!  Specially if you’re taking also compilers or some other hard programming class…
Class Outline

- Book: Tanenbaum’s Modern OSs
- Intro to OSs (including Real-Time OSs)
- Processes (definition, synchronization, management)
- Memory (virtual memory, memory allocation)
- IO (disks, sensors, actuators, keyboards, etc)
- InterProcess Communication (networking, data transmission, etc)
- Fault Tolerance, Real-Time and Security (time permitting)
Schedule and Grading

- Pop quizzes **10% of grade: about every 2.5 weeks**
- Programming assignment (NACHOS on UNIX) **30% of grade**
- Midterm **30% of grade: around March 1**
- Second Exam **30% of grade: April 19**
- Class participation may carry **5% of grade** for extra credit

- Project 0 is self-test!!! For you to test whether you will die or not taking this class… Add/drop period ends Tuesday Jan 17th
Operating Systems

• Manages different resources (CPU, mem, disk, etc)
• Improves performance (response time, throughput, etc)
• Allows portability, enables easier programming (no need to know what the underlying hardware)
• Interface between the hardware and the rest of the machine… editors, compilers, user programs, etc
• Standard interface is typically done in two ways:
  – system calls: control goes to the Operating System
  – library calls: control remains with the User
Brief History

• First Generation of computers had no OS: **single-user**. All coding done directly in machine language, memory resident code (no other resources to manage)

• Second Generation has basic OS: **batch processing**. Read input (tape/cards), process, output to tape or print

• Third Generation improved life: **multiprogramming**! Careful partitioning of memory space (4-12KB), drums and disks added for reading cards and spooling outputs (**Simultaneous Peripherals Operations On-Line**)  

• **Time-sharing** created several **virtual machines**
History (cont)

• Fourth Generation: **PCs** and **workstations**. Cheaper, faster, more user-friendly (Thank Macs for interfaces!)

• UNIX precursor MULTICS (MULTIplexed Information and Computing Services) was the first “modern” OS. Bell+MIT+GE (MULTICS --> units --> Unix)

• Berkeley improved on it: paging, virtual memory, file systems, signals (interrupts), networking!
Networking!

- **Networked OSs** are connected through a network, but user needs to know the name/type/location of everything.
- **Distributed OSs** (e.g., Amoeba, Mach, Locus) provide transparency to user, yielding one huge virtual machine!
- **Specialized OSs** are built for specific purposes: routing engines (Networking), lisp machines (AI), time constrained applications (Real-Time), Internet (WWW servers), massively parallel uses (supercomputers), etc.
- All these are coming together, hard to identify boundaries anymore.
Microsoft World

- Excellent marketing, some good products
- OSs started with DOS (Disk OS), no nothing, just very simple commands!
- Windows 3.1 was a huge jump (based on decades-old technology initially developed at Xerox then Macs)
- Windows 95 (released in 96) improved tremendously the state-of-the-affairs for MS, but still unreliable
- Windows NT approaches Unix distributions, with more user-friendly interface.
Unix World

- Created at AT&T, re-written/improved by Berkeley
- ATT had majority control and good support (reliable OS)
- OSF (Open SW Foundation, now Open Group) is a consortium of several companies to standardize UNIX
- Different subgroups (syscalls, shells, RT, etc)
- Standardization is with respect to interfaces and not implementation of primitives. Impln is left to the implr
- Modern applications are time constrained (tel, video, etc)
- Real-Time playing an increasingly important role
OS Structure

- Interface can be done at any level (depends on level of security of OS)
- Interface with the lower level layer gets translated
- Machine dependent language used for accessing hardware
- Main advantage of direct resource access is efficiency
- Main advantage of indirect access is portability
- Completely layered OS? Why or why not?
OS Functions

• Controls and manages resources (disks, memory, CPU, …); sends/receives control commands and data
• Allows multiprogramming (several programs “at the same time” in the same resource)
• Carries out communication between processes (inter and intra processor)
• Manages interrupt handlers for HW and SW interrupts
• Provides protection and security to processes
• Prioritizes requests and manages multiple resources in a single machine (eg multiprocessors or CPU IO reqs)
OS Functions

• OS manages resources, including management of
  – processes (creation, deletion, suspension, comm, synch)
  – main memory (usage, alloc/de-alloc, which processes get it)
  – 2ary storage (disk scheduling, alloc/de-alloc, swapping, files)
  – IO interfaces and devices (eg, keyboard, caching, memory)
  – protection (authorization, file and memory protection, etc)
  – InterProcess Communication (intra- and inter-machines)
  – Command interpretation (shells to Xlate user to OS).
    Typically includes the user interface that the OS uses.
OS Structure

- Hardware at the bottom layer
- Accessing the lower layer thru the higher layers
- DOS programs can access HW
- Unix has controllers and dev drivers (DD) controlling devices
- *system calls* are the interface between user and OS (DDs)
- *libraries* and *system programs* invoke *sys_calls*
OS Structure

- Interface can be done at any level (depends on security)
- Machine dependent language used for accessing HW
- Main advantage of direct resource access is efficiency (less layers means less overhead, ie, better performance)
- Main advantage of indirect access (syscall) is portability
- Modular approaches (ind access) have less flexibility, since apps only access HW thru libraries and sys_calls
- Layering means that one level is defined in terms of the level below (level 0 is the HW, level n is the user apps)
Modular Approach

- Create well-defined interfaces between any two layers
- Create well-defined properties of each layer
- Attempt to decrease the number of layers to improve efficiency and performance
- The final goal is to make the OS flexible and efficient
- Create the layers such that each user perceives the machine as belonging solely to himself or herself
- This is the concept of a virtual machine, which allows each user to avoid thinking about others’ processes
Language

• System calls are the interface between user and OS
• Access to the resources is done through privileged instructions (for protection)
• User applications cannot execute in kernel mode
• User applications use libraries that invoke sys_calls
• System procedures are executed to access resources, via privileged instructions (called from sys_calls)
• This way, no process can influence other executions, on purpose or by accident: resource protection
• Example: accounting, priority information
Language (cont)

• System calls can be divided into 5 categories:
  – process control
  – file manipulation
  – device manipulation
  – information maintenance
  – communication

• Special purpose OSs can also have special primitives:
  – specification of deadlines, priorities, periodicity of processes
  – specification of precedence constraints and/or synchronization among processes
Language (cont)

- Examples of libraries are language constructs to carry out formatted printing
- Examples of sys_calls are primitives to create a process
- For example, the reading of 10 bytes of a file:
  - The user does `fscanf`, the kernel requests a block of bytes from the device driver (DD), which talks to the controller of the disk to obtain a block of data. The block is transferred into a buffer, in the kernel address space. The kernel then picks the 10 bytes and copies them into the user-specified location. This way, the kernel accesses kernel and user space, but the user only accesses user space!
System Programs

- System programs do not interact directly with running user programs, but define a better environment for the development of application programs.
- Sys programs include: compilers, file manipulation and modification, editors, linker/loaders, etc
- An important one is the command interpreter (or shell), which parses user input, interprets it, and executes it
- Shells can either execute the command, or invoke other system programs or system calls to do it.
- Trade-offs: performance, increasing/updating # of commands
More on Languages

• Different process types have different requirements
• Different requirements beg for different languages
• Assembly, Lisp, Prolog, Java, RT-C, etc.
• Real-time languages inform the OS about its needs in order to enhance the predictability of its execution
  – deadline of a thread (by when do I need this done)
  – period of a thread (what is the frequency of this task?)
  – resources to be used (amount of memory or semaphores)
  – precedence constraints (door must be open for a robot to exit)