Operating Systems: What and Why

Agenda

- Recap
- What is an operating system
- Why is it important
- Summary of OS functionalities
- Basics of UNIX OS
- System calls
- Path of a system call
- User interfaces
- System boot sequence

Recap

- What do we know about computing systems
  - How to express algorithms in a programming language (ESC101)
  - How to design basic digital logic blocks (ESC102)
  - Design of algorithms (ESO207)
  - How to use the basic digital logic blocks to design simple computers (CS220)
    - For example, using adders, shifters, and other basic logic blocks, one can design an arithmetic logic unit (ALU)

What is an operating system

- A piece of software application
- Resource manager of any computing system
  - Schedules resources like CPU, memory, hard disks, and other I/O devices
  - Virtualizes the resources so that
    - a programmer does not have to worry about the low-level details of a physical resource
    - presents a virtual interface of the resource to the user

Why is it important

- Let’s assume that we have a minimal hardware implementing some abstract instruction set architecture
- Peripheral devices consist of memory, hard disk, keyboard, and a display
- We have a simple problem to solve: add the elements of a vector and print the result on the display device
- Let’s assume that somebody has written the program to solve it and prepared the binary image of the executable

Why is it important

Why is it important

- Q1: How do I store the binary image on hard disk? Why is it important to store it on hard disk?

Why is it important

- Q1: How do I store the binary image on hard disk? Why is it important to store it on hard disk?
  - Learn to program the disk controller manually and write a stream of bytes (how?)

Why is it important

- Q1: How do I store the binary image on hard disk? Why is it important to store it on hard disk?
  - Learn to program the disk controller manually and write a stream of bytes (how?)
- Q2: What if there isn't enough contiguous space on disk? Is contiguous space necessary?

Why is it important

- Q1: How do I store the binary image on hard disk? Why is it important to store it on hard disk?
  - Learn to program the disk controller manually and write a stream of bytes (how?)
- Q2: What if there isn't enough contiguous space on disk? Is contiguous space necessary?
  - Learn to create this space (de-fragment)

Why is it important

- Q1: How do I store the binary image on hard disk? Why is it important to store it on hard disk?
  - Learn to program the disk controller manually and write a stream of bytes (how?)
- Q2: What if there isn't enough contiguous space on disk? Is contiguous space necessary?
  - Learn to create this space (de-fragment)
- Q3: How do I load the binary image on memory? Why is it necessary? What if memory is small?

Why is it important

- Q1: How do I store the binary image on hard disk? Why is it important to store it on hard disk?
  - Learn to program the disk controller manually and write a stream of bytes (how?)
- Q2: What if there isn't enough contiguous space on disk? Is contiguous space necessary?
  - Learn to create this space (de-fragment)
- Q3: How do I load the binary image on memory? Why is it necessary? What if memory is small?
  - Learn to program the memory controller manually; read from disk and write to memory

Why is it important

Why is it important

- Q4: How to start executing the executable?

Why is it important

- Q4: How to start executing the executable?
  - Learn to load the program counter with the address of the first instruction of the executable
  - This is a sequence of instructions (how?)

Why is it important

- Q4: How to start executing the executable?
  - Learn to load the program counter with the address of the first instruction of the executable
  - This is a sequence of instructions (how?)
- Q5: How to read the vector size from keyboard?

Why is it important

- Q4: How to start executing the executable?
  - Learn to load the program counter with the address of the first instruction of the executable
  - This is a sequence of instructions (how?)
- Q5: How to read the vector size from keyboard?
  - The program must handle keyboard interrupts

Why is it important

Why is it important

- Q6: How to allocate memory for the vector?

Why is it important

- Q6: How to allocate memory for the vector?
  - The program must create enough memory space before the allocation

Why is it important

- Q6: How to allocate memory for the vector?
  - The program must create enough memory space before the allocation
- Q7: What if the vector size is larger than memory?

Why is it important

- Q6: How to allocate memory for the vector?
  - The program must create enough memory space before the allocation
- Q7: What if the vector size is larger than memory?
  - The program must check for this condition, divide the vector in portions, implement a swapping procedure, and operate only on portions that are in memory

Why is it important

Why is it important

- Q8: How is the final result displayed?

Why is it important

- Q8: How is the final result displayed?
  - The program must send the final result to the display device and command it to display the communicated characters

Why is it important

- Q8: How is the final result displayed?
  - The program must send the final result to the display device and command it to display the communicated characters
- Q9: How does the program terminate?

Why is it important

- Q8: How is the final result displayed?
  - The program must send the final result to the display device and command it to display the communicated characters
- Q9: How does the program terminate?
  - A special instruction at the end can lead the CPU to an “idle” state

Why is it important

- The example shows several interactions with the hardware that the user program has to take care of: write to disk, read from disk, memory and disk management, read from keyboard, write to display, control start/stop sequence on CPU
  - Shows the importance virtualizing disk and keyboard (as files?), memory, and CPU
- Very poor utilization of the CPU
  - CPU is idle when swapping between disk and memory or reading from keyboard or writing to display

Why is it important

Why is it important

- Q10: Is it possible to run multiple programs in a time-shared fashion? Also known as multi-tasking or multi-programming

Why is it important

- Q10: Is it possible to run multiple programs in a time-shared fashion? Also known as multi-tasking or multi-programming
  - A program before going on a long-latency event must save state in memory, restore saved state of another program, load it from disk, and start it from where it left off

Why is it important

- Q10: Is it possible to run multiple programs in a time-shared fashion? Also known as multi-tasking or multi-programming
  - A program before going on a long-latency event must save state in memory, restore saved state of another program, load it from disk, and start it from where it left off
  - Each program sees a virtual CPU

Why is it important

- Q10: Is it possible to run multiple programs in a time-shared fashion? Also known as multi-tasking or multi-programming
  - A program before going on a long-latency event must save state in memory, restore saved state of another program, load it from disk, and start it from where it left off
  - Each program sees a virtual CPU
  - How to manage memory of multiple programs?

Why is it important

- Q10: Is it possible to run multiple programs in a time-shared fashion? Also known as multi-tasking or multi-programming
  - A program before going on a long-latency event must save state in memory, restore saved state of another program, load it from disk, and start it from where it left off
  - Each program sees a virtual CPU
  - How to manage memory of multiple programs?
    - Each program sees a virtual view of the memory

Why is it important

- Q10: Is it possible to run multiple programs in a time-shared fashion? Also known as multi-tasking or multi-programming
  - A program before going on a long-latency event must save state in memory, restore saved state of another program, load it from disk, and start it from where it left off
  - Each program sees a virtual CPU
  - How to manage memory of multiple programs?
    - Each program sees a virtual view of the memory
- Q11: What if there are many programs to select from?

Why is it important

- Q10: Is it possible to run multiple programs in a time-shared fashion? Also known as multi-tasking or multi-programming
  - A program before going on a long-latency event must save state in memory, restore saved state of another program, load it from disk, and start it from where it left off
  - Each program sees a virtual CPU
  - How to manage memory of multiple programs?
    - Each program sees a virtual view of the memory
- Q11: What if there are many programs to select from?
  - Must invoke a “scheduling algorithm”

Why is it important

- The addition of vector elements now begins to look really complicated
  - The programmer needs to take care of too many things that have no relation to the actual problem
- After working with this system for a while, you realize that the disk contents should be organized in a better way
  - Related information should be grouped in a class (call it a directory) and each class may have subclasses

Why is it important

- Q12: How to represent directories and subdirectories on a disk?
  - Call each independent entity on the disk a file. Directories are also files. Need to implement all necessary functionalities of a "file system" e.g., creating a file, reading from a file, writing to file, etc.
  - Notice that all these have to be handled by the end-user's program which wishes to use the capabilities of the file system

Why is it important

- One day you find that one program has overwritten the data belonging to another program in memory due to a wrong address in the former program
- Q13: How to protect a program's code and data from unintentional/intentional bugs/attacks?
  - Every program must incorporate an elaborate security solution
  - Every program's memory must be isolated

Why is it important

- Every program needs to do a common set of things
  - Read from keyboard
  - Write to display
  - Read from and write to disk
  - Manage disk
  - Manage memory
  - Switch between programs or "jobs"
  - Select jobs for scheduling
  - Protect program code and data

Why is it important

- Back to the basic: pack commonly used functionalities in a library
  - Call the appropriate function from the library whenever needed
- Still looks cumbersome: why should a programmer implement these calls?
- Observation: every program is a marriage of computation and these “system library calls”
- An operating system implements these system calls and takes over whenever such a service is needed by a running job
  - The C standard library is a wrapper around these system calls

Design goals of OS

- Efficient virtualization of physical resources
- Efficient support for correct handling of concurrent execution
- Efficient file system for persistent storage
- Appropriate abstraction for each resource to make it easy to use
- Efficient means high-performance and low overhead in terms of space, time, energy
- Protection, isolation, security
- Reliability and failure-freedom
- Support for mobile devices

Summary of OS functionalities

- Two broad categories
  - Functionalities to improve performance
    - Job scheduling, context switch, memory management
  - Functionalities to improve ease of use
    - File systems, I/O, security
- Two modes of operations
  - User mode and kernel mode
  - A system call from a user program leads to a switch to kernel mode
  - Kernel mode allows unrestricted access to hardware including execution of privileged instructions

Summary of OS functionalities

- Two modes of operations
  - User mode avoids catastrophic failures
    - Isolated virtual address space for each process in user mode
    - Isolated execution of each process
    - No direct access to any hardware device
  - CPU needs to support a mode bit as part of the machine status word or processor status word
  - Switching modes is expensive
    - Needs to save and restore user and kernel register states

Summary of OS functionalities

- System boots up in kernel mode
  - Kernel of the OS is loaded by the bootstrap loader from a fixed location in the disk
  - Only when a user program is scheduled to run on the CPU, does the mode bit switch to user mode
- System calls invoke system call handlers
  - Locations of the handlers are found in an interrupt vector table residing in the low memory
  - Hardware interrupts are handled in the same way
  - System call arguments are passed in registers and/or in memory (by passing a pointer in a register); what are these arguments?

Summary of OS functionalities

- OS does several book-keeping tasks periodically
  - Job scheduling is an important example
  - Implemented by setting up a timer register which is decremented on every processor clock tick
  - When the timer register expires, a hardware interrupt is generated
  - The interrupt handler services the periodic tasks one by one and sets up the timer register again

Summary of OS functionalities

- Four basic OS modules
  - Process management, memory management, storage management, protection
  - Process management involves
    - Creation, deletion, scheduling of processes
    - Offering support for communication between processes
    - Synchronizing communicating processes
    - Handling deadlocks
  - Memory management involves
    - Handling memory allocation and de-allocation requests from user and kernel mode processes

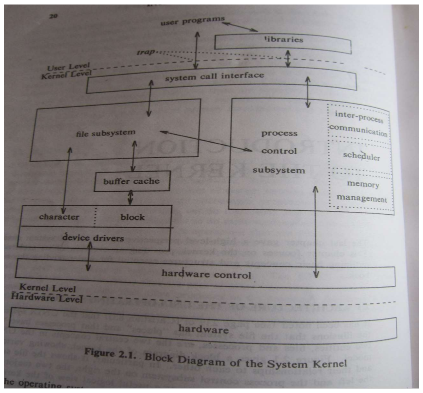
Summary of OS functionalities

- Four basic OS modules
  - Storage management involves
    - Implementing a virtual environment called file system
  - Protection cross-cuts all the three modules
    - Controls accesses to the resources managed by the OS
    - Usually the OS kernel is assumed to be trusted
    - A stricter protection model requires hardware-supported security

Basics of the UNIX OS

- Bit of history
  - 1965: Bell Telephone Labs, General Electric Company, and MIT join hands to build Multics (multiplexed information and computing service)
    - http://www.multicians.org/
  - 1969: Early version of Multics runs on GE 645; Bell Labs terminate participation
  - 1970: Honeywell takes over GE along with Multics
  - 1969-1971: Ken Thompson and Dennis Ritchie of Bell Labs implement a minimal OS in Fortran on a PDP-7 machine, which was later migrated to a PDP-11
    - Brian Kernighan names it UNIX, a pun on the name Multics
  - 1973: UNIX gets rewritten in C (which grew out of B)

Basics of the UNIX OS



Reproduced from "The Design of the UNIX OS" by M J Bach

Basics of the UNIX OS

- Two major parts
  - The OS kernel
  - Auxiliary system applications such as the shell command interpreter, the editors such as "vi", etc.
- UNIX kernel talks to the system and user applications through the system call layer
  - Two major sets of system calls: process control and file system
  - The process control layer implements inter-process communication, scheduling, and memory management

Basics of the UNIX OS: File system

- Every I/O device is treated as a file
  - Keyboard is treated as a special file called the standard input
  - Display device is standard output
- File system interacts with two types of I/O devices
  - Block device: transfers a block of data between the device and the kernel e.g., hard disk
  - Character device: communicates through a stream of bytes e.g., keyboard, display, etc.

Basics of the UNIX OS: File system

- Block devices
  - Accessed through a buffer cache to exploit spatial locality
  - Seen by the kernel as random access storage devices even though physically they may not be
- Character devices
  - Directly talk to the kernel without any caching
- Even though the OS abstraction for a device is a file, every device needs a device driver to implement the hardware protocols

Basics of the UNIX OS: File system

- Files are maintained using a data structure called index node (inode for short) table
  - Each file has one inode
  - Each inode stores the disk layout of the file data and other information such as file owner, access permissions, access times, etc.
  - On a file system call specifying a file name, the file name is translated to the corresponding inode
    - Maintains two more tables to do this: user file descriptor table (UFDT) and kernel file table (FT)

Basics of the UNIX OS: File system

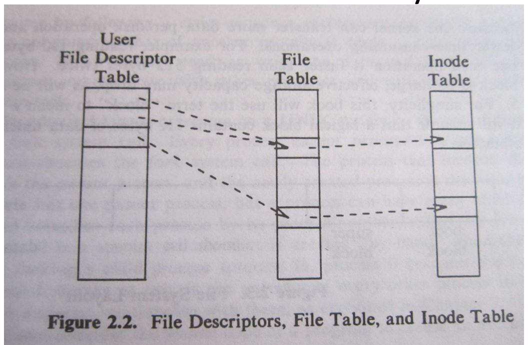


Figure 2.2. File Descriptors, File Table, and Inode Table

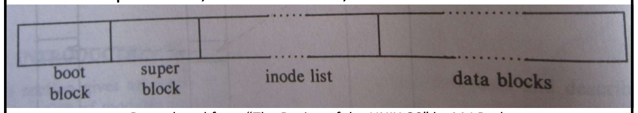
Reproduced from "The Design of the UNIX OS" by M J Bach

Basics of the UNIX OS: File system

- UFDT and FT
  - The UFDT maintains one entry for each open file of a process; each process has its own UFDT
  - The UFDT entry holds a pointer to the corresponding FT entry; two UFDT entries may point to the same FT entry (same file opened twice)
  - The open or creat call returns the index of the UFDT slot allocated to the file
    - Referred to as a file descriptor
  - Each FT entry stores the read/write byte offset into a file, access permissions, a unique pointer to the file inode; the FT is shared across all processes

Basics of the UNIX OS: File system

- File systems are kept on the block devices
  - Kernel remains oblivious to the physical block device addresses
  - The translation from the logical file system layout to the physical addresses is done by the device driver
  - Kernel treats the file system as a sequence of logical blocks, each of size multiple of 512 bytes
  - A file system starts with a boot block followed by a superblock, the inode list, and the data blocks



Reproduced from "The Design of the UNIX OS" by M J Bach

Basics of the UNIX OS: File system

- File system
  - Boot block resides in the first sector and contains the bootstrap loader
  - Superblock describes the state of the file system: how large it is, how many files it can store, free list information, etc.
  - Inode list size is configured by the admin when building the kernel
  - Inode table contains the indices into the inode list
  - Root inode is used as the root of the directory system
  - An allocated data block can belong to exactly one file

Basics of the UNIX OS: Process control

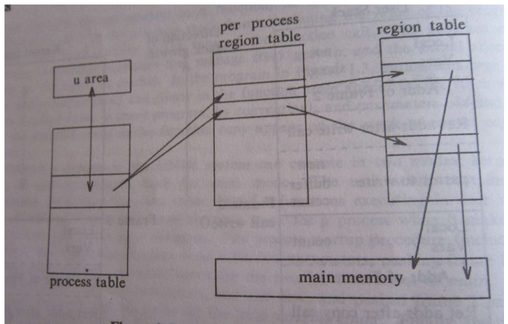


Figure 2.5. Data Structures for Processes

Reproduced from "The Design of the UNIX OS" by M J Bach

Basics of the UNIX OS: Process control

- Kernel process table entry for each process and u area to store process information manipulated by kernel only
  - Each process table entry points to a per process region table and each per process region table entry points to a global region table
  - A region is a contiguous memory segment containing process text, data, stack
  - A global region table entry contains region attributes such as text/data, private/shared, and the starting address of the region

Basics of the UNIX OS: Process control

- The u area contains information necessary to completely describe a process; kernel can directly access the u area of the currently executing process only; the u area contains
  - A pointer to the kernel process table slot of the currently executing process
  - Parameters, return values, and error codes of the last executed system call of the currently executing process
  - File descriptors of all open files of the currently executing process
  - Current directory and current root of the file system
  - Process and file size limits

Basics of the UNIX OS: Process control

- The context of a process is its complete state including
  - Text (code)
  - Values of global user variables and data structures
  - Values of processor registers
  - Contents of its kernel process table slot and u area
  - Contents of its user mode stack and kernel mode stack

Basics of the UNIX OS: Process control

- A process undergoes several state transitions from creation to termination
  - The kernel process table entry maintains the process state, user id of the owner, and an event descriptor if the process is in sleep state
  - A process undergoes a context switch on I/O or hardware interrupt or timer interrupt
    - Not all system calls cause a context switch
    - All system calls cause a mode switch
    - Context switch code or system call code executes in the context of the currently scheduled process
    - A mode switch changes the stack frame top pointer to user mode or kernel mode stack top depending on the direction of the switch

Basics of the UNIX OS: Process control

- Three important system calls for process management
  - execv, fork, wait
  - An execv call from a user program executes a new program, starting address of which is passed to the call; does not create a new process
    - Overlays new text, data, stack regions on top of the old regions, sets up the region table pointers correctly
  - A fork call creates a new process (called the child process)
    - Kernel duplicates the address space for the child from the parent by copying or sharing depending on the situation

Basics of the UNIX OS: Process control

- A process is always created by another process
  - Created process is the child of the creating process
- The system boots up as a process
  - *init* process in UNIX
  - This is the root of all processes
  - Every process gets a unique integer ID known as the process ID or pid. The root process has pid zero.
    - In UNIX, the system boot process has pid zero and *init* has pid one.

Basics of the UNIX OS: Process control

- The processes form a tree
  - The root of this tree in UNIX is *init*
  - The system boot process is killed after *init* is created
- The *pstree* command shows the process tree in UNIX (*pstree –p* shows the pid also)
- The *pgrep processname* command shows the pid of a process
  - *pgrep init* returns 1
- To find the parent pid of a process with pid x
  - Check the fourth entry in file /proc/x/stat on UNIX
  - In a C program, you can use getppid()

Basics of the UNIX OS: Process control

- A process can be created by calling fork()
  - Child pid is returned to parent, zero is returned to child. A negative return value indicates error in UNIX.
- When the fork() call returns
  - The child process has been created
  - Its text and global data are loaded in memory
  - The process will start executing when it is scheduled
  - Since the child process gets an exact copy of the address space and registers of the parent, it starts executing at the instruction that returns from fork()
    - This is the value of the program counter in the parent
    - The only difference between the parent and the child is in the return value of the fork() call

Basics of the UNIX OS: Process control

- The wait() system call makes a process wait for any one of its children to complete
- It is a good practice to wait for all the children that a process has created
  - Can be programmed using a loop which runs number of times equal to the number of children
  - Each loop iteration makes one call to wait()
  - The wait() call takes one argument which is a pointer to the exit code of a completing child process
    - Can be passed NULL if we do not need the exit code of a child process

Basics of the UNIX OS: Process control

- The waitpid() system call makes a process wait for the child with a specific pid
  - Takes three arguments: pid, exit code pointer, an option
  - First argument can be -1 if any child is waited for
  - The last argument is usually 0
  - waitpid(-1, &status, 0) is equivalent to wait(&status)
  - A useful value for the last argument of waitpid is WNOHANG
    - Allows the parent to return immediately irrespective of whether the child waited for has completed or not

Basics of the UNIX OS: Process control

- The pipe() system call creates a communication channel with a write end and a read end
  - The ends of a pipe are implemented using a pair of file descriptors corresponding to a "common" file
  - A process can write to one end and read what it has written from the other end
    - Not very interesting for a process to do so
    - Gets interesting when one process writes to one end of the pipe and another process reads from the other end
  - The pipe() call takes an array of two file descriptors as argument
    - On return, the array is filled up appropriately by OS

Basics of the UNIX OS

- External hardware interrupts
  - OS may delay handling these if currently in the kernel mode depending on the criticality of the code being executed
  - Every interrupt has a priority level
  - To mask an interrupt, the processor execution priority level is raised above the priority level of the interrupt
    - For example, before manipulating the inode list, the processor priority level is raised above the disk interrupt priority level
  - Typical interrupt priority (low to high): software interrupts (system calls), character devices, network, disk, timer, machine error

System calls

- General mechanism of virtualizing a physical resource
  - Map virtual entities on to the physical resource in a time-shared manner
    - Swap out a virtual entity to make room for another virtual entity
      - Needed only if the physical resource is exhausted
    - Swap in the new virtual entity to occupy (portion of) the physical resource
  - Example: physical resource CPU is time-shared between virtual entities named "processes"
  - Example: physical resource memory is time-shared between virtual entities named "address spaces"

System calls

- General goals of CPU virtualization
  - Performance: a process should be able to run at native CPU speed with minimum OS interference
  - OS control: periodically/occasionally the OS should be able to get control over what the CPU does
  - The two goals have somewhat contradictory demands
    - A middle ground is possible meaning that some OS interference must be tolerated
- System calls play an important role in letting the OS control what other resources the CPU can access

System calls

- OS defines a set of restricted operations
  - Usually involves accessing some other resource such as an I/O device, or allocating some resource such as memory, etc.
  - OS would like to check whether such an operation is legitimate before letting the currently running process do it
- A process normally runs in user mode
  - Can do everything except any of the restricted operations
- A process must switch to kernel mode to do a restricted operation

System calls

- How does a process switch between user and kernel modes?
  - By executing a system call instruction (known as ecall in RISC-V ISA) a user mode to kernel mode switch can be achieved
  - Sometimes it is wrongly referred to as a trap instruction
    - Actually, there is no trap instruction; instead, trap is a mechanism for the CPU to transfer control to the OS
    - Trapping the process in a small cage to do the required restricted operation only and nothing else
  - A special return instruction (sret in RISC-V) switches the mode from kernel to user

System calls

- Steps involved in handling a trap
  - Must be transparent to the currently running process
  - Switch mode to kernel
  - Trap handler saves registers and other necessary states on kernel stack so that the running process can be resumed correctly after returning from trap
  - Handle system call (invokes a system call handler routine)
  - Restore registers and other states from kernel stack
  - Return from trap (switches mode and stack to user)
- A newly created process also switches to user mode through the return from trap instruction

System calls

- How to locate the trap handler entry point?
  - The system call instruction jumps to a location stored in a privileged register (written using privileged inst.)
  - In RISC-V, this is the stvec register
    - Supervisor trap vector base address register
  - In other architectures, this may come in the form of a table or a vector
    - Depending on the cause of the trap, a particular entry in the vector is used as the jump address
  - The stvec register or the trap table is set up by the OS kernel as part of the boot code
    - Recall that a machine boots up in kernel mode and therefore, the boot process can do privileged operations

System calls

- System calls have a lot of resemblance with function calls
  - The arguments of a system call are usually passed in the same registers used to pass function arguments
  - The return value of a system call uses the same register used for returning value from functions
  - There is a reason behind this resemblance
    - System calls are usually wrapped around by library functions e.g., system call to write to a file would be part of the fprintf function's code
    - Keeping the same calling convention for syscalls and functions makes it easy to pass arguments and return results

System calls

- How to distinguish between different system calls?
  - Memory allocation, file read, file write, sleep, exit, fork, execv, wait, etc.
  - To distinguish among these, a system call instruction has a special argument called the system call number
  - Different system calls have different numbers
  - The basic xv6 kernel has 21 system calls exposed to user programs
  - Each system call is exposed to user programs through one or a group of wrapper functions
    - For example, fopen() is a wrapper around the open syscall

Path of a system call

- Assume the RISC-V UNIX like platform
- RISC-V system call convention (integer args)
  - System call arguments are passed in registers 10, 11, …, 16 (aka a0 to a6; same as function call args)
  - System call number is passed in register 17 (aka a7)
  - No system call has more than six args; so a6 is not used
  - System call return value is in registers a0 and a1
    - Only a0 will be used in xv6
  - System call instruction in RISC-V is *ecall*
- Consider the read system call for reading from a file
  - C library calls read, fscanf, scanf, etc. all lead to this system call

Path of a system call

- The read system call
  - Three arguments: file descriptor (in register a0), destination memory buffer address (in register a1), number of bytes to read (in register a2)
  - These registers should be set up before the syscall instruction executes
  - The ecall instruction stops instruction fetching, waits for all pending instructions to complete, does a mode switch, invokes the appropriate system call handler after examining register a7

Path of a system call

- The read system call handler
  - If the file descriptor is bigger than two, various permission checks are done in the GFT and the UFDT
  - The inode is accessed and the data bytes are read from either the buffer cache or the disk if the file descriptor is bigger than 2; otherwise the bytes are read from the character device
  - The data bytes are written to the memory buffer by setting up a DMA depending on the device driver interface
  - After initiating the DMA, a context switch can take place
  - The system call returns the number of bytes read

Path of another system call

- Consider the open system call
  - Used to open a file
  - Takes two arguments: file name (pointer in register a0), access flags and permission mode flags if the file is to be created (in register a1)
  - The system call handler allocates a free file descriptor by looking up the UFDT and GFT, sets the permission bits in the allocated entry according to the flags, retrieves the inode and puts it in the inode cache
  - The normal return value of the system call is the allocated file descriptor id

Path of yet another system call

- Consider the exit system call
  - Every program on termination invokes this system call
    - In C programs after the main function returns, the exit system call gets invoked
  - Takes one argument: the exit code (in register a0)
  - The system call handler deletes the calling process
  - An example of a system call belonging to the process management subsystem

User interfaces

- Two types: command line interface and graphical user interface (GUI)
  - Command line interface is provided by a shell program or a command line interpreter
  - The shell can be implemented in two ways
    - A command line parser interprets the user's command and executes it (MS-DOS style)
    - A command line interface program only checks if the command exists, forks a child, and passes the command line to the child for execution (mostly used today)
    - The fork model allows background and concurrent execution of commands
  - A GUI usually involves a large number of system calls

A skeleton shell

```
while (1) {
  // read command into character array buf[][]
  if (buf[0] is "exit") break;
  else if (program with name equal to buf[0] exists)
{    if (fork() == 0) execv (buf[0], buf);
     else {
        if (last character in buf is not `&') wait(NULL);
      }
  } else printf("%s: command not found.\n",buf[0]);
}
```

System boot sequence

- Boot sequence involves the following steps
  - Run diagnostic codes from a ROM or EEPROM
  - Load the bootstrap loader from the boot block
  - The bootstrap loader loads the necessary parts of the kernel
- The bootstrap loader can be changed easily by modifying the boot block only
- A boot disk or a system disk contains the bootstrap loader