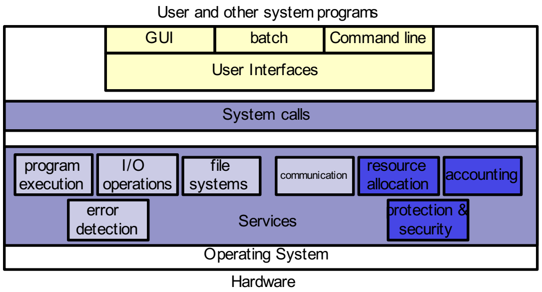
User Space ⇔ Kernel Interaction

ECE 650
Systems Programming & Engineering
Duke University, Spring 2016

Operating System Services



- Picture adapted from "Operating System Concepts", 8th edition

Operating System Services

- User interface: GUI, batch, or command line
- Program execution: load program into memory and execute it
- I/O operations: I/O device interaction (e.g. DVD drive, display)
- File system: create, read & write files & directories
- Communications: shared memory or message-based IPC
- Resource allocation: for multiple users or multiple jobs; allocate and manage CPU cycles, main memory, file storage, etc.
- Accounting: track use of resources by users or processes
- Protection & Security: protect independent processes; user security

Invoking OS Services

- These OS services can be invoked actively or passively
- Active: System Calls
- Passive: Variety of ways these can occur for an executing process
  – Exceptions
  – Interrupts
  – Signals

Interrupts

- Event external to an executing process that changes the normal flow of instruction execution (e.g. the event is generated by HW devices)
- Basic mechanism
  – CPU has a wire called Interrupt-Request Line
  – CPU senses it after executing every instruction
  – If wire is asserted, CPU performs state save (context switch)
  – CPU jumps to an interrupt handler routing at fixed address
  – Interrupt handler executes and ends w/ "return from interrupt"
    - E.g. IRET instruction in x86
- Raise => Catch => Dispatch => Clear flow

Interrupt Controller

- Hardware to enable:
  – Deferring interrupt handling during critical processing
  – Efficiently transfer control to appropriate interrupt handler
  – Multi-level interrupts (e.g. priorities across interrupts)
- 2 Interrupt Request Lines
  – Non-Maskable Interrupts (NMI): e.g. memory errors (ECC)
  – Maskable Interrupts: CPU can temporarily disable
- Interrupt mechanism receives an address
  – Selects a specific interrupt handling routine
  – From a table in memory: interrupt vector
  – Contains direct jumps to the interrupt vector code routines
  – Interrupt chaining is often used in implementations

More on Interrupts

- Interrupt priority levels
  - CPU can defer handling of low-priority interrupts
  - Doesn't mask of all interrupts
  - Allows handling of high-priority interrupts
- At boot time:
  - OS probes hardware devices
  - Determines devices present and installs interrupt handles in interrupt vector
- Similar process (save state, jump to pre-defined handler) used for other operation as well:
  - Exceptions (e.g. page fault)
  - Signal handling
  - System calls

Signals

- Used in UNIX systems to notify a process of an event
- Essentially a way for Software to mimic the Interrupt mechanism
- Behavior
  - Signal generated due to an event occurrence
  - Signal is delivered to a process
  - The signal must be handled once delivered
- Synchronous
  - Caused by an event within an executing process
    - E.g. divide by zero, illegal memory access
  - Delivered to same process that caused the signal
- Asynchronous
  - Generated by an event external to the running process
    - E.g. kill signal (Ctrl-C) or OS timer for scheduling

Signal Handling

- Every signal has a default signal handler
  - Run by the kernel to handle the signal
- Can also override with a user-defined signal handler
- E.g.
  - Ignore signal, terminate all threads, stop or resume all threads
- Where to deliver a signal in multi-threaded process?
  - The thread to which signal applies
  - Every thread in the process
  - Certain threads in the process
  - Specific thread to receive all signals for the process
- Synchronous: deliver to causing thread; Asynchronous: many options
  - UNIX allows threads to specify which signals to accept or block
  - Typically delivered only to first thread that is not blocking it
  - UNIX mechanism: `kill(pid_t pid, int signal)`

System Calls

- Used to actively invoke OS services
- System calls usually wrapped in library API functions
  - E.g. C standard library
  - 'man 1' (general commands)
  - 'man 2' (system calls)
  - 'man 3' (library functions, esp. C standard library)
- Library routines:
  - Check & validate arguments
  - Build data structure to convey arguments to the kernel
  - Execute special instruction (SW interrupt or trap)
    - Operand identifies desired kernel service

System Call Types

- Process Control (e.g. fork, exit, wait)
  - load, execute, end, abort, wait, allocate & free memory
- File Management (e.g. open, close, read, write)
  - create & delete, open & close, read & write, get & set attributes
- Device Manipulation (e.g. ioctl, read, write)
  - request & release device, read & write device
- Information maintenance (getpid, alarm, sleep)
  - get time or date, get & set system data
- Communication (pipe, shmget, mmap)
  - create & delete communication channels; send & receive msgs
- Protection (chmod, umask, chown)
  - set file security & permissions

System Call Process

- Similar to the mechanism for an interrupt
  - System call results in execution of a 'trap' instruction
  - Trap transfers control to a location in the interrupt vector
    - Based on the 'trap code' which indicate the specific system call
  - Interrupt vector location jumps to trap handler code
  - Trap handler code changes to supervisor execution mode & saves process state (e.g. registers, pc) just as a context switch
  - Parameters typically passed via indirection
    - E.g. a register stores a memory address to a block of memory which contains parameter values
  - Kernel executes the system call
  - User execution mode is resumed
  - 'Return from Interrupt' executed to resume user process

Example System Call - sbrk

```
BRK(2)            Linux Programmer's Manual            BRK(2)

NAME
    brk, sbrk - change data segment size

SYNOPSIS
    #include <unistd.h>
    void *sbrk(intptr_t increment);

DESCRIPTION
    brk() and sbrk() change the location of the program break, which defines the end of the process's
    data segment (i.e., the program break is the first location after the end of the uninitialized
    data segment). Increasing the program break has the effect of allocating memory to the process;
    decreasing the break deallocates memory.

    sbrk() increments the program's data space by increment bytes. Calling sbrk() with an increment
    of 0 can be used to find the current location of the program break.

RETURN VALUE
    On success, sbrk() returns the previous program break. (If the break was increased, then this
    value is a pointer to the start of the newly allocated memory). On error, (void *) -1 is
    returned, and errno is set to ENOMEM.
```


malloc (not a system call)

- Uses (possibly) sbrk to service an allocation request
- malloc is used to obtain address space storage for dynamically allocated data, e.g. an array of 100 ints:
  `int *integer_array = malloc(100 * sizeof(int));`
- If malloc does not find a region of free space on the heap large enough, then it invokes sbrk to grow the data size
- Returns a pointer to an unused memory region
- Marks that region as allocated

How does malloc work?

- Typically maintains a "free list"
  - E.g. a linked list of free memory regions
  - Each free region has a header:
    - Pointer to next free region header
    - Size of free region
- malloc() will search free list for a large enough region
  - If no match, then use sbrk() to grow heap
  - Return pointer to free region & update free list
- free() will return a memory region to the free list
  - Or merge into an existing free list entry
- There are many design choices, optimizations, etc.