ECE 650
Systems Programming & Engineering

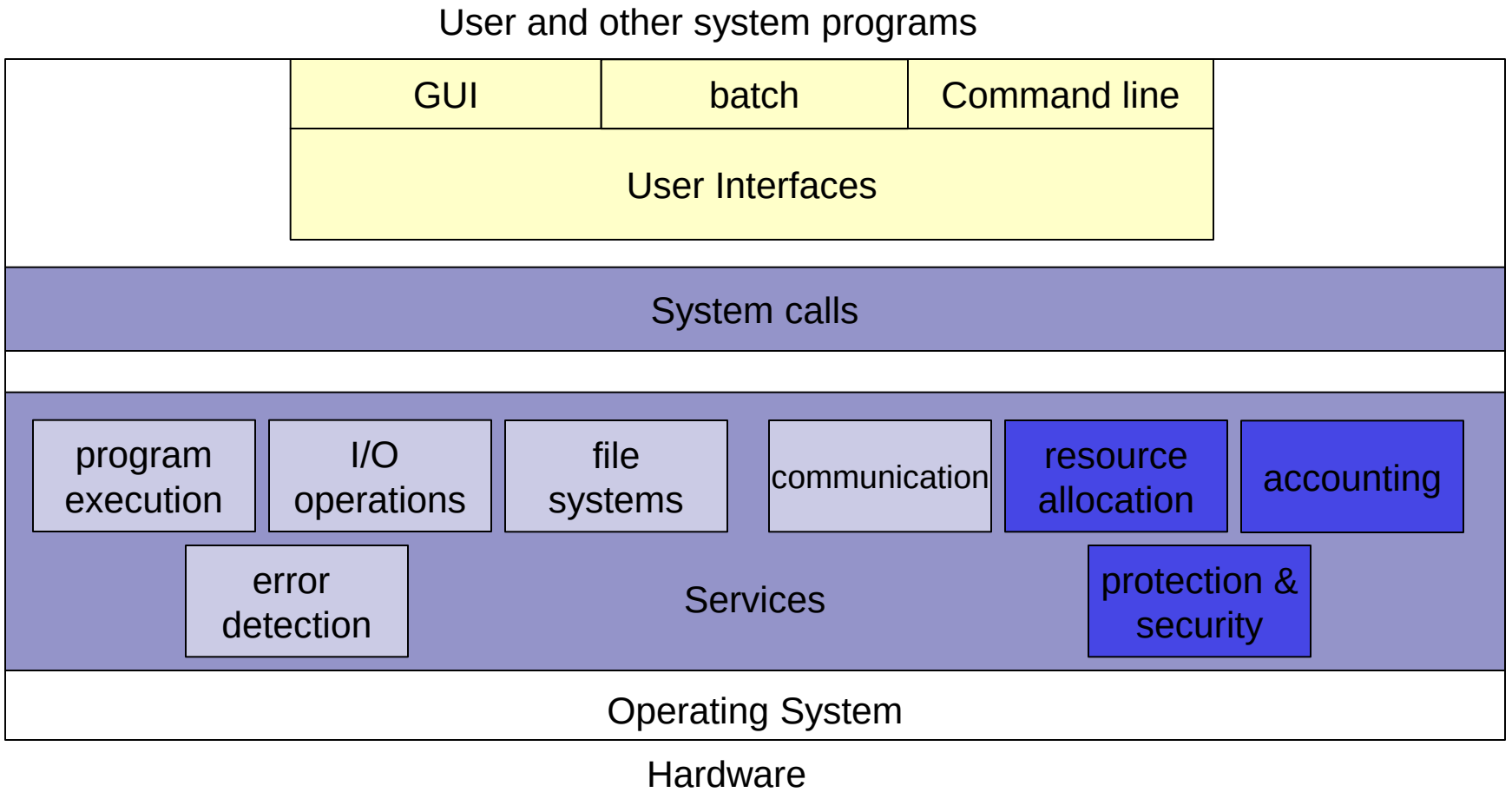
Spring 2018

User Space / Kernel Interaction

Tyler Bletsch
Duke University

Slides are adapted from Brian Rogers (Duke)

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 (IRQ) 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 off 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)` or `kill` command

Yes, you send signals with the '`kill`' function, which is badly named, but many signals do usually end the process.

Common Unix Signals

Signal	#	Default action	Description
SIGHUP	1	Exit	**Hangup**: terminal disconnects
SIGINT	2	Exit	**Interrupt**: Ctrl+C
SIGQUIT	3	Core	**Quit**: Ctrl+\
SIGILL	4	Core	**Illegal Instruction**
SIGTRAP	5	Core	**Trace/Breakpoint Trap**
SIGABRT	6	Core	**Abort**
SIGFPE	8	Core	**Arithmetic Exception** in floating point
SIGKILL	9	Exit	**Killed**: Unmaskable way to kill a process (`kill -9` on terminal)
SIGBUS	10	Core	**Bus Error**: Low level IO failure
SIGSEGV	11	Core	**Segmentation Fault**: You know this one!
SIGPIPE	13	Exit	**Broken Pipe**: Tried to read/write to a pipe with other end closed
SIGALRM	14	Exit	**Alarm**: User-controlled timers; you can override signal to do general timekeeping!
SIGTERM	15	Exit	**Terminated**: Default signal from `kill` command
SIGUSR1	16	Exit	**User Signal 1**: Use for whatever you want!
SIGUSR2	17	Exit	**User Signal 2**: Use for whatever you want!
SIGSTOP	23	Stop	**Stopped**: Ctrl+Z
SIGCONT	25	Ignore	**Continued**: On resume from stop

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