CS162 Textbook Cliff Notes

*Ch 2. The Kernel Abstraction*

2.0

* Protection is essential to achieving several of the operating system goals:

1. **Reliability**: system must be bullet proof to operate correctly regardless of what an application or user might do
2. **Security**: system must prevent untrusted code from modifying system state
3. **Privacy**: prevent untrusted code from accessing unauthorized data
4. **Fair** resource allocation: limit the amount of resources assigned to each user

* Implementing protection is the job of the operating system kernel
  1. Kernel is lowest-level and has access to everything
  2. **Process**: execution of an application program with restricted rights

2.1 The process abstraction: vocab overview

* **Executable image**: code compiled into machine instructions, also includes static data/initial values
* **Execution stack**: holds state of local variables during procedure calls
* **Heap**: memory region set aside for any dynamically allocated data structures the program might need
  1. Running separate instances of the same program: while the code is reused to save memory, separate instances of the program’s heap, stack, and data are made
* Thus, a process is an instance of a program
* **Process control block (PCB**): OS uses to keep track of various processes on the computer and stores info about those processes like:
  1. Where stored in memory
  2. Where executable image resides on disk
  3. Which user asked it to execute
  4. Privileges

2.2 Dual Mode operation

* Represented by a single bit in the processor status register that signifies the current mode of the processor
  1. **User Mode**: checks each instruction before execution to verify it is permitted to be performed by that process
  2. **Kernel Mode**: operated without protections
* Hardware needs to support:
  1. **Privileged instructions**
     + Processes change privilege level by executing a system call, now can access privileged instructions
     + If process tries to change its privilege level or access restricted memory w/o permission, causes a processor exception. Kernel will halt the processor after such privilege violation.
  2. **Memory Protection**
     + Each application process can only read/write to its own memory
       - Read: is important for privacy
       - Write: is important for security
     + Base and bound memory protection: base register signifies start of process’ memory region; bound signifies end.
       - Only changed by privileged instructions
       - Every time processor fetches instruction, checked to be sure within base/bound
       - Kernel itself does not use these (already assumed to be safe)
     + Base/bound does not provide for these features:
       - Expandable heap/stack (growing from opposite sides of memory)
       - Memory sharing (between different processes)
       - Physical memory addresses: program can be loaded at different locations every time depending on what other processes are currently running. Code would have to change every time
       - Memory fragmentation (processes start/end at different times so memory is filled up randomly, soon not enough continuous space is left for a new process)
     + Virtual addresses used to address (ha) this problem
       - Every process’ memory starts at 0, thinks it has the whole memory, etc
  3. **Timer Interrupts**
     + Helps prevent infinite loops from making computer unresponsive
     + Hardware timer (one for each processor) interrupts processor after predetermined delay
  4. **Safely transfer** from user 🡪kernel mode and back

2.3 Types of Mode transfer

1. User to Kernel Mode

* Interrupt: asynchronous signal sent to processor that some external event has occurred that requires its attention
  + If it happens, processor handles interrupt and stalls current processes. On multiprocessor, other processors can continue with executions.
  + Different handlers for different interrupts: timer, I/O, etc.
  + Interprocessor interrupts for multiprocessor machines. This is how they communicate
  + Polling is an alternative to interrupts: kernel checks I/O device periodically to see if an event has occurred that requires handling.
* Processor Exceptions: hardware event caused by user program behavior that hands control to kernel.
  + Can happen when user program tries to run a privileged instruction, divides by zero, tries to write to read-only memory, etc. (In these cases OS halts request and returns error code to user)
  + Can also happen in debugging when you want to set a breakpoint
* System calls: a procedure provided by the kernel that can be invoked at the user level. To protect the system, changes over to kernel and starts executing code at predefined place.
  1. Kernel to User Mode
     + Start a new process: basically initializes everything (copy prog into memory, PC=first instruction’s address, SP= base of user stack, etc.) and jumps into user mode
     + Resume after interrupts, syscall, exception: restores its registers PC, an jumps into user mode
     + Switch to a different process: loads other process’ state into PCB
     + User-level upcall: user programs can receive asynchronous notifications of events
  2. Implementing Safe mode transfer

1. Most OS have a common sequence of instructions when switching that include:
   * Limited entry into the kernel: entry point in to the kernel is set up by the kernel.
   * Atomic changes to processor state: mode, PC, stack, memory protection all changed at same time
   * Transparent, restartable execution: must be able to restore previous user program state before the transfer.
   1. Interrupt Vector Table
      * Area of kernel memory with an array of pointers each pointing to the first instruction of a different handler procedure in the kernel
      * Layout is processor-specific
   2. Interrupt Stack:
      * hardware automatically saves some of the interrupted process’ registers by pushing them here before calling kernel handler
      * needed for reliability: user-level stack may not be valid memory pointer
      * needed for security: on multiprocessor, other processors can still be running user programs and write to user-level stack, w/ a buffer overflow they can modify the kernel’s return address and take control
   3. Two stacks per process: user stack and kernel stack (for each one!)
      * Makes it easier to switch to a new process inside an interrupt/syscall handler
   4. Interrupt Masking: instruction delivery can be deferred if it is not safe at the moment
      * Interrupt disable: not disabled, merely ignored and put into a buffer
      * Interrupt enable: pending interrupts delivered to processor
   5. Hardware support for saving/restoring registers
      * State must be saved before handler runs
      * X86: when trap occurs:
        + X86 pushes interrupted process’ SP onto kernel interrupt stack, switches to kernel stack
        + X86 pushes interrupted process’ instruction pointer
        + X86 pushes processor status word (control bits, etc)
        + pushad (push all double) saves remaining registers
      * does not return to instruction that called the exception (infinite loop). Instead, returns one instruction later.

2.6 Implementing Secure Syscalls

1. OS must define a calling convention: - how to name syscalls, pass arguments, so program doesn’t need to care about how kernel implements the calls
2. Kernel must check parameters of syscalls very thoroughly (always assume it is malicious)
3. Diagram of a syscall:
   * User program calls stub in normal way, oblivious the implementation is actually in the kernel
   * Stub fills in code and executes trap
   * Hardware transfers control to kernel
   * Syscall completes, returns to handler
   * Handler returns to user level; stub returns to caller
4. Kernel Stub: pair of stubs is pair of procedures that mediate between two environments (user program and kernel, for example). Four tasks:
   * Locate syscall arguments: stored in user memory, so stub must verify it is a legal address on the stack, converts it to physical address the kernel can safely use
   * Validate parameters: make sure the file name is valid/what it should be
   * Copy before check: prevents application from modifying after the stub checks value but before parameter is used in execution. This attack is TOCTOU (time of check vs. time of use).
   * Copy back any results
   1. Starting a new process: kernel must
   2. Allocate/initialize PCB
   3. Allocate memory for process
   4. Copy program from disk to newly allocated memory
   5. Allocate user-level stack for user-level execution
   6. Allocate kernel stack for syscalls, interrupts, processor exceptions
   7. Copy arguments into user memory
   8. Transfer control to user mode
   9. Implementing Upcalls
5. There is a need to virtualize a part of the kernel so that applications can behave like OS
6. **Upcalls**: virtualized exceptions/interrupts. signals (UNIX), asynchronous events (Windows)
7. Several uses for immediate event delivery:
   * preemptive user-level threads
   * asynchronous I/O notification
   * interprocess communication
   * user-level exception handling
   * user-level resource allocation

Other notes:

*Kernel vs. rest of OS*: why isn’t entire OS in kernel? A: easier to debug user-level code than kernel since kernel can operate a debugger with its low level hardware. This represents principle of least privilege.

*Process status register*: bunch of flags set/reset as a byproduct of executing instructions.

1. Kernel/user mode bit
2. condition codes: compact coding of conditional branch instructions
3. Some processors have possibility of more than 2 privilege levels

*MS/DOS and memory protection*: early Microsoft user program that could read/modify anywhere in memory. Was seen as ok for one person’s PC, but it soon caused lots of programs to crash, and with the introduction of viruses was seen as a threat. Also in terms of defining an interface between OS and applications

*Memory Mapped Devices*: OS controls I/O devices by reading/writing to special memory locations. Device itself monitors its corresponding bus and acts accordingly when something is written. With memory protection added, now more protection: user programs can’t mess up the mouse, etc.

*Address Randomization*: combat viruses that prey on hidden vulnerabilities to cause things like buffer overflows. Most OS randomize virtual addresses; some go beyond this and randomize procedure/static variables, and offset between adjacent procedures on the stack.

*Buffer Descriptors and high-performance I/O*: OS sets up circular queue of requests for each device to handle.