Architectural Support for OS

H. Levy

OS and Architectures

- **What an OS can do is dictated, at least in part, by the architecture.**
- **Architecture support can greatly simplify (or complicate) OS tasks**
- **Example: PC operating systems have been primitive, in part because PCs lacked hardware support (e.g., for VM)**

Architectural Features for OS

- **Features that directly support OS needs include:**
  1. timer (clock) operation
  2. sychronization (atomic instructions)
  3. memory protection
  4. I/O control and operation
  5. interrupts and exceptions
  6. OS protection (kernel/user mode)
  7. protected instructions
  8. system calls

Protected Instructions

- **some instructions are typically restricted to the O.S.**
  - users cannot be allowed direct access to I/O (disks, printers, etc) [can be done through either privileged instructions or through memory mapping]
  - must control instructions that manipulate memory management state (page table pointers, TLB load, etc)
  - setting of special mode bits (kernel mode)
  - halt instruction

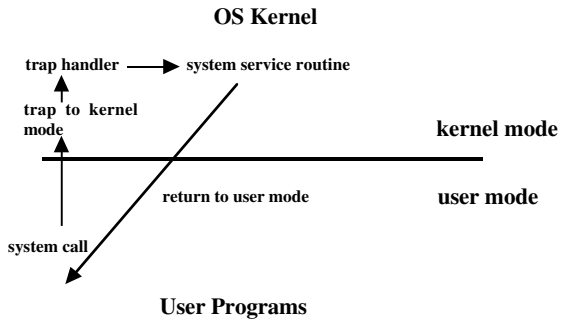
OS Protection

- **How do we know if we can execute a protected instruction?**
  - architecture must support (at least) two modes of operation: *kernel* mode and *user* mode
  - mode is indicated by a status bit in a protected processor register
  - user programs execute in user mode; the os executes in kernel mode
- **Protected instructions can only be executed in kernel mode.**

Crossing Protection Boundaries

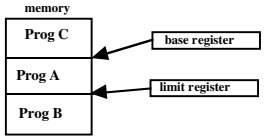
- **For a user to do something "privileged" (e.g., I/O) it must call an OS procedure.**
- **How does a user-mode program call a kernel-mode service?**
- **There must be a system call instruction that:**
  - causes an exception, which vectors to a kernel handler
  - passes a parameter, saying which system routine to call
  - saves caller's state (PC, mode bit) so it can be restored
  - arch must permit os to verify caller's parameters
  - must provide a way to return to user mode when done

Protection Modes and Crossing



Memory Protection

- must be able to protect user programs from each other
- must protect OS from user programs
- may or may not protect user programs from OS
- simplest scheme is base and limit registers:



- base and limit registers are loaded by the OS before starting a program
- virtual memory and segmentation are similar

Exceptions

- hardware must detect special conditions: page fault, write to a read-only page, overflow, trace trap, odd address trap, privileged instruction trap, syscall...
- must transfer control to handler within the O.S.
- hardware must save state on fault (PC, etc) so that the faulting process can be restarted afterwards
- modern operating systems use VM traps for many functions: debugging, distributed VM, garbage collection, copy-on-write...
- exceptions are a performance optimization, i.e., conditions could be detected by inserting extra instructions in the code (at high cost)

I/O Control

- I/O issues:
  - how to start an I/O (special instructions or memory-mapped I/O
  - I/O completion (interrupts)
- Interrupts are the basis for asynchronous I/O
  - device controller performs an operation asynch to CPU
  - device sends an interrupt signal on bus when done
  - in memory is a *vector table* containing a list of addresses of kernel routines to handle various events
  - CPU switches to address indicated by vector specified by the interrupt signal

I/O Control (cont)

device interrupts

↓

CPU stops current operation, switches to kernel mode, and saves current PC and other state on kernel stack

↓

CPU fetches proper vector from vector table and branches to that address (to routine to handle interrupt)

↓

interrupt routine examines device database and performs action required by interrupt

↓

handler completes operation, restores saved (interrupted state) and returns to user mode (or calls scheduler to switch to another program)

Timer Operation

- how does the OS prevent against runaway user programs (infinite loops)?
- a timer can be set to generate an interrupt in a given time
- before it transfers to a user program, the OS loads the timer with a time to interrupt
- when the time arrives, the executing program is interrupted and the OS regains control
- this ensures that the OS can get the CPU back even if a user program erroneously or purposely continues to execute past some allotted time.
- the timer is privileged: only the OS can load it

Synchronization

- **Interrupts cause potential problems because an interrupt can occur at any time -- causing code to execute that interferes with code that was interrupted**
- **OS must be able to synchronize concurrent processes.**
- **This involves guaranteeing that short instruction sequences (read-modify-write) execute atomically.**
- **One way to guarantee this is to turn off interrupts before the sequence, execute it, and re-enable interrupts; CPU must have a way to disable interrupts.**
- **Another is to have special instructions that can perform a read/modify/write in a single cycle, or can atomically test and conditionally set a bit, based on its previous value.**