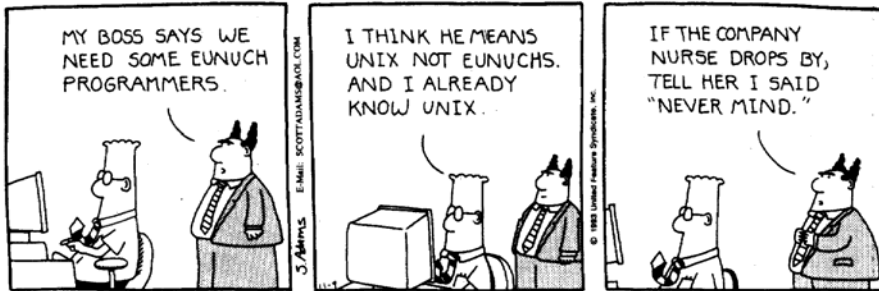


## Virtual Machines & the OS Kernel

DILBERT by Scott Adams



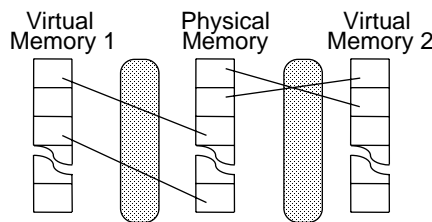
(not in the book)

Comp 120 - Spring 2005

04/21/05

L23 - Virtual Machines & the OS Kernel 1

## Power of Contexts: Sharing a CPU



Every application can be written as if it has access to all of memory, without considering where other applications reside.

More than Virtual Memory  
A VIRTUAL MACHINE

### 1. TIMESHARING among several programs --

- Separate context for each program
- OS loads appropriate context into pagemap when switching among pgms

### 2. Separate context for OS "Kernel" (eg, interrupt handlers)...

- "Kernel" vs "User" contexts
- Switch to Kernel context on interrupt;
- Switch back on interrupt return.



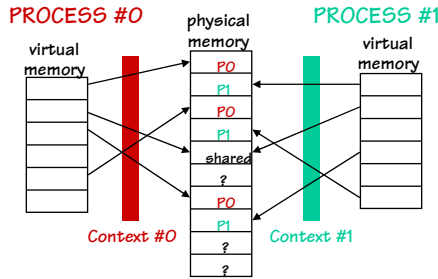
What is this OS KERNEL thingy?

Comp 120 - Spring 2005

04/21/05

L23 - Virtual Machines & the OS Kernel 2

## Building a Virtual Machine

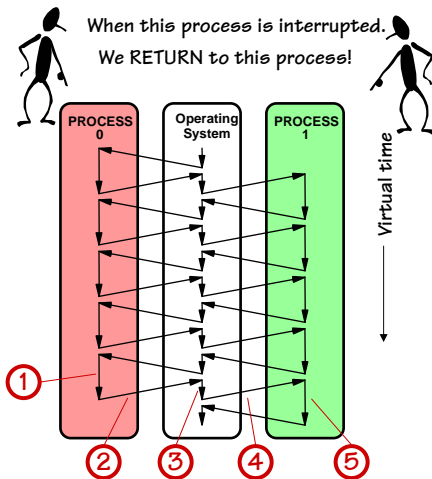


Goal: give each program its own "VIRTUAL MACHINE";  
 programs don't "know" about each other...

Abstraction: create a **PROCESS** which has its own

- machine state: \$1, ..., \$31
- context (pagemap)
- stack
- program (w/ possibly shared code)
- virtual I/O devices (console...)

## Multiplexing the CPU



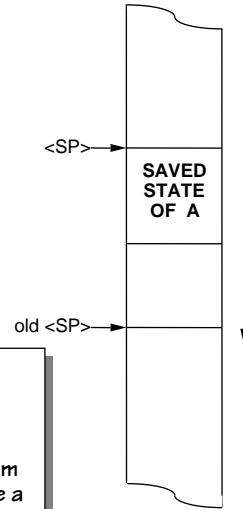
And, vice versa.  
 Result: Both processes get executed,  
 and no one is the wiser

1. Running in process #0
2. Stop execution of process #0 either because of explicit *yield* or some sort of timer *interrupt*; trap to handler code, saving current PC in \$27
3. First: save process #0 state (regs, context) Then: load process #1 state (regs, context)
4. "Return" to process #1: just like a return from other trap handlers (ex. jr \$27) but we're returning from a *different* trap than happened in step 2!
5. Running in process #1

## Stack-Based Interrupt Handling

### BASIC SEQUENCE:

- Program A is running when some EVENT happens.
- PROCESSOR STATE saved on stack (like CALL)
- The HANDLER program to be run is selected.
- HANDLER state (PC, etc) installed as new processor state.
- HANDLER runs to completion
- State of interrupted program A popped from stack and re-installed, JMP returns control to A
- A continues, unaware of interruption.



### CHARACTERISTICS:

- TRANSPARENT to interrupted program!
- Handler runs to completion before returning
- Obeys stack discipline: handler can "borrow" stack from interrupted program (and return it unchanged) or use a special handler stack.

## miniMIPS Interrupt Handling

### Minimal Implementation:

- Check for EVENTS before each instruction fetch.
- On EVENT j:
  - save PC into \$27, (\$k1);
  - INSTALL  $0x80000000 + j*40$  as new PC.

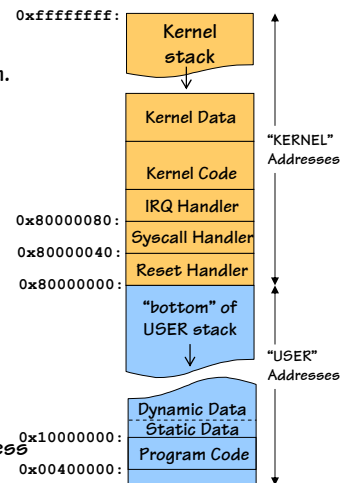
### Handler Coding:

- Save state in "User" structure
- Call C procedure to handle the exception
- re-install saved state from "User"
- Return to \$27, (\$k1)

### WHERE to find handlers?

miniMIPS Scheme: WIRE IN a high-memory address for each exception handler entry point

Real MIPS alternative: WIRE IN the address of a TABLE of handler addresses ("interrupt vectors")



## PC interrupt vector example

On PCs, the interrupt vector table consists of 256 4-byte pointers, and resides in the first 1 K of addressable memory. Each interrupt number is reserved for a specific purpose. For example, 16 of the vectors are reserved for the 16 IRQ lines.

IRQ Number	Typical Use
IRQ 0	System timer
IRQ 1	Keyboard
IRQ 2	Cascade interrupt for IRQs 8-15
IRQ 3	Second serial port (COM2)
IRQ 4	First serial port (COM1)
IRQ 5	Sound card
IRQ 6	Floppy disk controller
IRQ 7	First parallel port
IRQ 8	Real-time clock
IRQ 9	Open interrupt
IRQ 10	Open interrupt
IRQ 11	Open interrupt
IRQ 12	PS/2 mouse
IRQ 13	Floating point unit/coprocessor
IRQ 14	Primary IDE channel
IRQ 15	Secondary IDE channel

## External (Asynchronous) Interrupts

### Example:

System maintains current time of day (TOD) count at a well-known memory location that can be accessed by programs. But...this value must be updated periodically in response to clock EVENTS, i.e. signal triggered by 60 Hz clock hardware.

### Program A (Application)

- Executes instructions of the user program.
- Doesn't want to know about clock hardware, interrupts, etc!!
- Can incorporate TOD into results by examining well-known memory location.



### Clock Handler

- GUTS: Sequence of instructions that increments TOD. Written in C.
- Entry/Exit sequences save & restore interrupted state, call the C handler. Written as assembler "stubs".



## Avoiding Re-Entrance

Handlers which are interruptable are called *RE-ENTRANT*, and pose special problems... miniMIPs, like many systems, disallows reentrant interrupts!  
 Mechanism: Interrupts are disabled in "Kernel Mode" (PC = 0x80000000):

USER mode  
(Application)

```
main()
{ ...
  ...
  ...
}
```

Processor State K-Mode  
Flag: PC<sub>31</sub> = 1 for Kernel Mode!

PC = 0.....

KERNEL mode  
(Op Sys)

User  
(saved  
state)

Page  
Fault  
Handler

PC = 1.....

Kernel  
Stack

Interrupt  
Vector

SYSCALL  
Handlers

Clock  
Handler

That's where the  
rest of memory is!



## I/O Device Notifying the OS

The OS needs to know when:

The I/O device has completed an operation

The I/O operation has encountered an error

This can be accomplished in two different ways:

Polling:

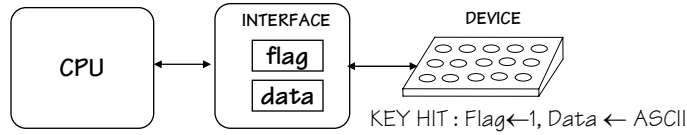
The I/O device put information in a status register

The OS periodically check the status register

I/O Interrupt:

Whenever an I/O device needs attention from the processor,  
it interrupts the processor from what it is currently doing.

## Polled I/O



Application code deals directly with I/O (eg, by busy-waiting):

```
loop: lw  $t0, flag($t1) # $t1 points to
      beq $t0,$0,loop   # device structure
      lw  $t0, data($t1) # process keystroke
```

### PROBLEMS:

- Wastes (physical) CPU while busy-waiting  
(FIX: Multiprocessing, codestripping, etc)
- Poor system modularity: running pgm MUST know about ALL devices.
- Uses up CPU cycles even when device is idle!

## Interrupt-driven I/O

OPERATION: NO attention to Keyboard during normal operation

- on key strike: hardware asserts IRQ to request interrupt
- USER program interrupted, PC+4 saved in \$k1
- state of USER program saved on KERNEL stack;
- KeyboardHandler invoked, runs to completion;
- state of USER program restored; program resumes.

TRANSPARENT to USER program.

Keyboard Interrupt Handler (in O.S. KERNEL):

```
struct Device {
    char flag, data;
} Keyboard;

KeyboardHandler(struct Mstate *s) {
    Buffer[inptr] = Keyboard.data;
    inptr = (inptr + 1) % 100;
}
```

Each keyboard has an associated buffer



That's how data gets into the buffer. How does it get out?

## ReadKey SYSCALL: Attempt #1

A *system call* (syscall) is an instruction that transfers control to the kernel so it can satisfy some user request. Kernel returns to user program when request is complete.


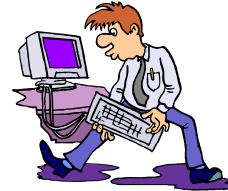
(Can be implemented as a "synchronous" interrupt, a.k.a. illop)

First draft of a ReadKey syscall handler: returns next keystroke to user

Each process has an index to a keyboard

```

ReadKEY_h ()
{
    int kbdnum = ProcTbl[Cur].DPYNum;
    while (BufferEmpty(kbdnum)) {
        /* busy wait loop */
    }
    User.R2 = ReadInputBuffer(kbdnum);
}
  
```

Problem: Can't interrupt code running in the supervisor mode...  
so the buffer never gets filled.

## ReadKey SYSCALL: Attempt #2

A keyboard SYSCALL handler

(slightly modified, eg to support a Virtual Keyboard):

```

ReadKEY_h ()
{
    int kbdnum = ProcTbl[Cur].DPYNum;
    if (BufferEmpty(kbdnum)) {
        User.R27 = User.R27 - 4;
    } else
        User.R2 = ReadInputBuffer(kbdnum);
}
  
```

That's a  
funny way  
to write  
a loop



Problem: The process just wastes its time-slice waiting for some one to hit a key...

## ReadKey SYSCALL: Attempt #3

BETTER: On I/O wait, YIELD remainder of quantum:

```

ReadKEY_h ()
{
    int kbdnum = ProcTbl[Cur].DPYNum;
    if (BufferEmpty(kbdnum)) {
        User.R27 = User.R27 - 4;
        Scheduler( );
    } else
        User.R2 = ReadInputBuffer(kbdnum);
}

```

RESULT: Better CPU utilization!!

FALLACY:

Timesharing causes a CPUs to be less efficient

## Sophisticated Scheduling

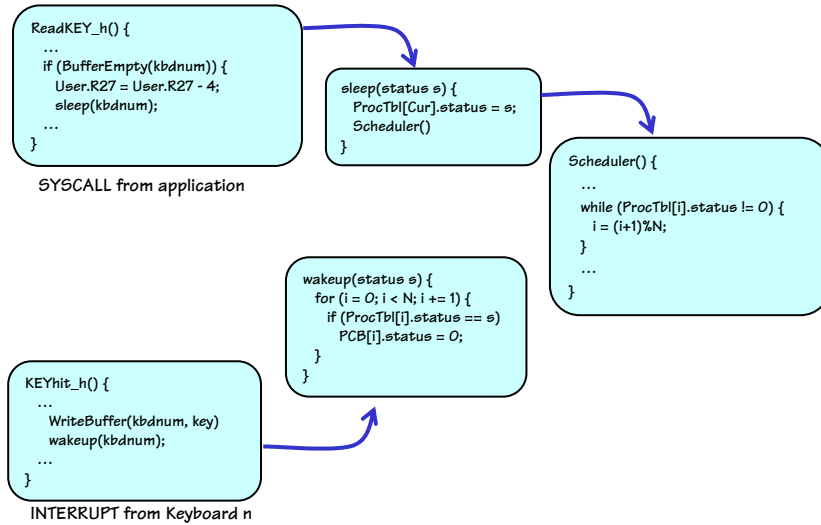
To improve efficiency further, we can avoid scheduling processes in prolonged I/O wait:

- Processes can be in **ACTIVE** or **WAITING** ("sleeping") states;
- Scheduler cycles among **ACTIVE PROCESSES** only;
- Active process moves to **WAITING** status when it tries to read a character and buffer is empty;
- Waiting processes each contain a code (eg, in PCB) designating what they are waiting for (eg, keyboard N);
- Device interrupts (eg, on keyboard N) move any processes waiting on that device to **ACTIVE** state.

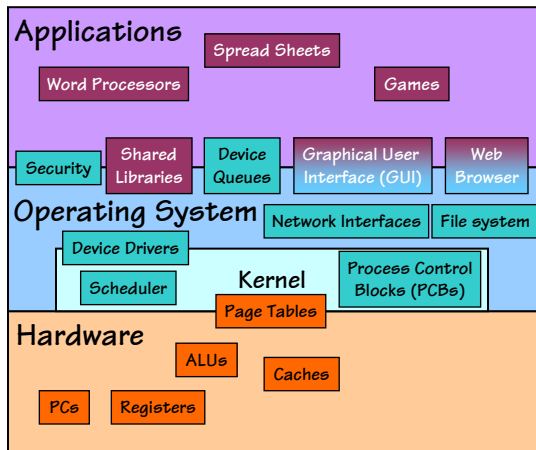
UNIX kernel utilities:

- **sleep(reason)** - Puts CurProc to sleep. "Reason" is an arbitrary binary value giving a condition for reactivation.
- **wakeup(reason)** - Makes active any process in sleep(reason).

### ReadKey SYSCALL: Attempt #4



### A “Typical” OS layer cake

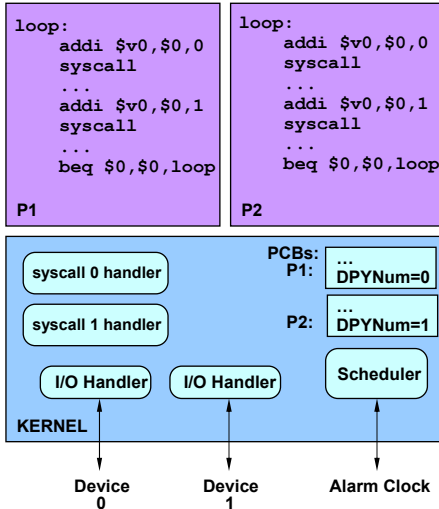


An OS is the Glue that holds a computer together.

- Mediates between competing requests
- Resolves names/bindings
- Maintains order/fairness

KERNEL - a RESIDENT portion of the O/S that handles the most common and fundamental service requests.

## A "Thin Slice" of OS organization



"Applications" are quasi-parallel "PROCESSES" on "VIRTUAL MACHINES",

each with:

- CONTEXT (virtual address space)
- Virtual I/O devices

O.S. KERNEL has:

- Interrupt handlers
- SYSCALL (trap) handlers
- Scheduler
- PCB structures containing the state of inactive processes

## Responsibilities of the Operating System

The operating system acts as the interface between:

The I/O hardware and the program that requests I/O

Three characteristics of the I/O systems:

The I/O system is shared by multiple program using the processor

I/O systems often use interrupts (external generated exceptions) to communicate information about I/O operations.

Interrupts must be handled by the OS because they cause a transfer to supervisor mode

The low-level control of an I/O device is complex:

Managing a set of concurrent events

The requirements for correct device control are very detailed

## Operating System Requirements

Provide protection to shared I/O resources

Guarantees that a user's program can only access the portions of an I/O device to which the user has rights

Provides abstraction for accessing devices:

Supply routines that handle low-level device operation

Handles the interrupts generated by I/O devices

Provide equitable access to the shared I/O resources

All user programs must have equal access to the I/O resources

Schedule accesses in order to enhance system throughput