Virtual Machines & the OS Kernel

Not in the book!
Power of Contexts: Sharing a CPU

1. TIMESHARING among several programs --
   - Programs alternate running in time slices called “Quanta”
   - 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.

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

What is this OS KERNEL thingy?
Building a Virtual Machine

Goal: give each program its own “VIRTUAL MACHINE”; programs don’t “know” about each other…

Abstraction: create a PROCESS, with its own
- machine state: $1, …, $31
- context (pagemap)
- stack
- program (w/ possibly shared code)
- virtual I/O devices (console…)

Diagram:

PROCESS #0
  virtual memory

PROCESS #1
  virtual memory

physical memory
PO
P1
PO
P1
shared
?
PO
P1
?

Context #0

Context #1
Multiplexing the CPU

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 $27 ($k1)
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

And, vice versa. Result: Both processes get executed, and no one is the wiser
Stack-Based Interrupt Handling

BASIC SEQUENCE:

• Program A is running when some EVENT happens.
• PROCESSOR STATE saved on stack (like a procedure 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”)
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”.
Interrupt Handler Coding

long TimeOfDay;
struct Mstate { int R1,R2,...,R31 } User;

/* Executed 60 times/sec */
Clock_Handler(){
    TimeOfDay = TimeOfDay + 1;
}

Clock_h:
    lui $k0,(User>>16)     # make $k0 point to
    ori $k0,$k0,User      # "User" struct
    sw $1,0($k0)          # Save registers of
    sw $2,4($k0)          # interrupted
    ...                   # application pgm...
    sw $31,124($k0)       # program
    add $sp,$0,KStack     # Use KERNEL stack
    jal Clock_Handler     # call handler
    lw $1,0($k0)          # Restore saved
    lw $2,4($k0)          # registers
    ...                   
    lw $31,124($k0)       # Return to app.
    jr $k1

Handler (written in C)

"Interrupt stub" (written in assy.)
Time-Sharing the CPU

We can make a small modification to our clock handler implement time sharing.

```c
long TimeOfDay;
struct Mstate { int R1, R2, ..., R31 } User;

/* Executed 60 times/sec */
Clock_Handler()
{
    TimeOfDay = TimeOfDay + 1;
    if (TimeOfDay % QUANTUM == 0) Scheduler();
}
```

A Quantum is that smallest time-interval that we allocate to a process, typically this might be 50 to 100 mS. (Actually, most OS Kernels vary this number based on the processes priority).
Simple Timesharing Scheduler

long TimeOfDay;
struct Mstate { int R1,R2,...,R31 } User;
.
.
.
(PCB = Process Control Block)

struct PCB {
    struct MState State; /* Processor state */
    Context PageMap; /* VM Map for proc */
    int DPYNum; /* Console number */
} ProcTbl[N]; /* one per process */

int Cur; /* “Active” process */

Scheduler() {
    ProcTbl[Cur].State = User; /* Save Cur state */
    Cur = (Cur+1)%N; /* Incr mod N */
    User = ProcTbl[Cur].State; /* Install for next User */
}
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()
{
  ...
  ...
  ...
}
```

**KERNEL mode**
(Op Sys)

- User (saved state)
- Page Fault Handler
- Interrupt Vector
- Kernel Stack
- SYSCALL Handlers
- Clock Handler
- Processor State K-Mode
  Flag: PC_{31} = 1 for Kernel Mode!

PC = 0...........

PC = 1...........

That’s where the rest of memory is!
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 (a “device driver”) is invoked, runs to completion;
- state of USER program restored; program resumes.

TRANSPARENT to USER program.

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

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

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

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. Iloop)

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);
    }
}

Problem: The process just wastes its time-slice waiting for someone to hit a key...
ReadKey SYSCALL: Attempt #3

BETTER: On I/O wait, YIELD remainder of time slot (quantum):

```c
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

```
ReadKEY_h() {
  ...
  if (BufferEmpty(kbdnum)) {
    User.R27 = User.R27 - 4;
    sleep(kbdnum);
    ...
  }
  ...
}

sleep(status s) {
  ProcTbl[Cur].status = s;
  Scheduler()
}

Scheduler() {
  ...
  while (ProcTbl[i].status != 0) {
    i = (i+1)%N;
  }
  ...
}

wakeup(status s) {
  for (i = 0; i < N; i += 1) {
    if (ProcTbl[i].status == s)
      PCB[i].status = 0;
  }
}

KEYhit_h() {
  ...
  WriteBuffer(kbdnum, key)
  wakeup(kbdnum);
  ...
}

INTERRUPT from Keyboard n
```
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