Virtual Machines

Lab 6 due Thursday!

Review: Virtual Memory

Goal: create illusion of large virtual address space
- divide address into (VPN, offset), map to (PFN, offset) or page fault
- use high address bits to select page: keep related data on same page
- use cache (TLB) to speed up mapping mechanism—works well
- long disk latencies: keep working set in physical memory, use write-back
**Example II**

Setup:
- 256 bytes/page ($2^8$)
- 16 virtual pages ($2^4$)
- 8 physical pages ($2^3$)
- 12-bit VA (4 vpn, 8 offset)
- 11-bit PA (3 ppn, 8 offset)

LRU page: VPN = 0xE

ST(BP, -4, SP), SP = 0x604

VA = 0x600, PA = _______

---

**Contexts**

A **context** is an entire set of mappings from VIRTUAL to PHYSICAL page numbers as specified by the contents of the page map:

- **Virtual Memory 1**
- **Virtual Memory 2**
- **Physical Memory**

THE BIG IDEA: Several programs, each with their own context, may be simultaneously loaded into main memory!

"Context switch": reload the page map!

---

**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 Operating System “Kernel” (eg, interrupt handlers)...
   - "Kernel" vs "User" contexts
   - Switch to Kernel context on interrupt;
   - Switch back on interrupt return.

**TYPICAL HARDWARE SUPPORT**: rapid context switch mechanism

---

**Building a Virtual Machine**

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

New abstraction: a **process** which has its own
- machine state: R0, …, R30
- context (virtual address space)
- PC, stack

"OS Kernel" is a special, privileged process that oversees the other processes and handles real I/O devices, emulating virtual I/O devices for each process
Each process has its own virtual machine

- OS Kernel (Specially privileged process)

Processes: 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 XP
3. First: save process #0 state (regs, context) Then: load process #1 state (regs, context)
4. "Return" to process #1: just like return from other trap handlers (ie., use address in XP) but we're returning from a different trap than happened in step 2!
5. Running in process #1

Key Technology: Interrupts.

Beta Interrupt Handling

Minimal Hardware Implementation:
- Check for Interrupt Requests (IRQs) before each instruction fetch.
- On IRQ:
  - copy PC into Reg[XP];
  - INSTALL j*4 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 Reg[XP]

WHERE to find handlers?
- BETA Scheme: WIRE IN a low-memory address for each exception handler entry point
- Common alternative: WIRE IN the address of a TABLE of handler addresses ("interrupt vectors")
External (Asynchronous) Interrupts

Example:
Operating System maintains current time of day (TOD) count. But..this value must be updated periodically in response to clock EVENTS, i.e. signal triggered by 60 Hz timer 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 “asking” OS.

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

```c
long TimeOfDay;
struct Mstate { int Regs[31];} User;

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

```assembly
Clock_h:
    ST(r0, User)        | Save state of
    ST(r1, User+4)      | interrupted
    ...                 | app pgm...
    ST(r30, User+30*4)  | Use KERNEL SP
    CMOVE(KStack, SP)   | BR(Clock_Handler,lp) | call handler
    LD(User, r0)        | Restore saved
    LD(User+4, r1)      | state.
    ...                 |
    LD(User+30*4, r30)  | execute interrupted inst
    SUBC(XP, 4, XP)  | JMP(XP)             | Return to app.
```

Simple Timesharing Scheduler

```c
struct Mstate {
    /* Structure to hold */
    int Regs[31];  /* processor state */
} User;

(PCB = Process Control Block)

```c
struct PCB {
    struct MState State;  /* Processor state */
    Context PageMap;  /* VM Map for proc */
    int DYNum;  /* 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 state for next User */
    LoadUserContext(ProcTbl[Cur].Context);  /* Install context */
}
```

Avoiding Re-entrance

Handlers which are interruptable are called RE-ENTRANT, and pose special problems... Beta, like many systems, disallows reentrant interrupts!
Mechanism: Uninterruptable “Kernel Mode” for OS:

- USER mode
  - main()
    - ...
    - ...
    - ...
  - Processor State K-Mode Flag: PC31 = 1 for Kernel Mode!

- KERNEL mode
  - (Op Sys)
    - User
      - Page Fault Handler
        - User (saved state)
          - PC = 0........
        - SVC Handlers
          - Interrupt Vector
          - Clock Handler
          - Kernel Stack
    - Other K-mode functions, e.g.
      - choosing Kernel/User context
      - Allowing "privileged" operations
Communicating with the OS

User-mode programs need to communicate with OS code:
- Access virtual I/O devices
- Communicate with other processes

... But if OS Kernel is in another context (i.e., not in user-mode address space) how do we get to it?

Solution:
- Abstraction: a supervisor call (SVC) with args in registers – result in RO or maybe user-mode memory
- Implementation: use illegal instructions to cause an exception -- OS code will recognize these particular illegal instructions as a user-mode SVCs

But if OS Kernel is in another context (i.e., not in user-mode address space) how do we get to it?

Okay… show me how it works!

Exception Hardware

Exception Handling

If (bad opcode) {
    // Reg[XP] = PC+4; PC = “Illop”
    PCSEL = 3,
    WASEL = 1, WDSEL = 0, WERF = 1, WR = 0
}

Illop Handler

I_IllOp:
    SAVESTATE()   | Save the machine state.
    LD(KStack, SP) | Install kernel stack pointer.
    LD(XP, -4, r0) | Fetch the illegal instruction
    SHRC(r0, 26, r0) | Extract the 6-bit OPCODE
    SHLC(r0, 2, r0) | Make it a WORD (4-byte) index
    LD(r0, UUOTbl, r0) | Fetch UUOTbl[OPCODE]
    JMP(r0) | and dispatch to the UUO handler.
    .macro UUO(ADR) LONG(ADR+0x80000000) | Auxiliary Macros
    .macro BAD() UUO(UUOError)  | (including SVCs)

This is a 64-entry dispatch table. Each entry is an address of a ‘handler’
Actual Illops

Here’s the handler for truly unused opcodes (not SVCs):

UUOError:
- CALL(KWMsg)
  - text "Illegal instruction "
  - LD(xp, -4, r0)
  - CALL(KHexPrt)
  - text " at location 0x"
  - MOVE(xp, r0)
  - CALL(KHexPrt)
  - CALL(KWrMsg)
  - text "! ....."
  - HALT()

These utility routines (Kxxx) don’t follow our usual calling convention—they take their args in registers or from words immediately following the procedure call. They adjust LP to skip past any args before returning.

Supervisor Call Handler

Sub-handler for SVCs, called from I_Illoop on SVC opcode:

SVC_UUO:
- LD(XP, -4, r0)
  - ANDC(r0,0x7,r0)
  - SHLC(r0,2,r0)
  - LD(r0,SVCTbl,r0)
  - JMP(r0)

SVCTbl:
- SVC(0): User-mode HALT instruction
- SVC(1): Write message
- SVC(2): Write Character
- SVC(3): Get Key
- SVC(4): Hex Print
- SVC(5): Wait(S), S in R3
- SVC(6): Signal(S), S in R3
- SVC(7): Yield()

Another dispatch table

Handler for HALT SVC

SVC(0): User-mode HALT instruction
- UOO(HaltH)
- UOO(WrMsgH)
  - SVC(1): Write message
- UOO(WrChH)
  - SVC(2): Write Character
- UOO(GetKeyH)
  - SVC(3): Get Key
- UOO(HexPrtH)
  - SVC(4): Hex Print
- UOO(WaitH)
  - SVC(5): Wait(S), S in R3
- UOO(SignalH)
  - SVC(6): Signal(S), S in R3
- UOO(YieldH)
  - SVC(7): Yield()

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

Applications can context-switch to simulation of an inactive process.