Virtualizing the Processor

• Page map example
• Virtual machines
• Processes
• Timesharing
• Communicating with OS

Lab 6 due Thursday
**Goal:** create illusion of large virtual address space
- divide address into (VPN, offset), map to (PPN, 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
MMU Address Translation

Look in TLB: VPN → PPN cache
Usually implemented as a small (16- to 64-entry) fully-associative cache
A context is an entire set of mappings from VIRTUAL to PHYSICAL page numbers as specified by the contents of the page map:

We would like to support multiple VIRTUAL to PHYSICAL Mappings and, thus, multiple Contexts.

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

“Context switch”: reload the page map!

We only have change Context# and PTBL Pointer
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 running in its own context. It manages the execution of other processes and handles real I/O devices, emulating virtual I/O devices for each process.
One VM For Each Process

Application

CPU
Virtual Memory
Windows
I/O events
files
sockets
SVCs

OS Kernel (Specially privileged process)

P1

P2

P3

P4

P5

CPU
MEM
TIMER
DISK
I/O
KVM

6.004 Computation Structures

L19: Virtualizing the Processor, Slide #6
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: Timer Interrupts

If (IRQ == 1 && PC[31] == 0) {
    // Reg[XP] ← PC+4;  PC ←“Xadr”
    PCSEL = 4,
    WASEL = 1, WDSEL = 0, WERF = 1,
    MWR = 0
}
Beta Interrupt Handling

Minimal Hardware Implementation:

- Check for Interrupt Requests (IRQs) before each instruction fetch.
- On IRQ j:
  - copy PC+4 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]-4

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”)
Example: Timer Interrupt Handler

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];} UserMState;

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

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

Handler
(written in C)

“Interrupt stub”
(written in assy.)
Simple Timesharing Scheduler

```
struct MState { int Regs[31]; } UserMState;

struct PCB { // Process Control Block
    struct MState State; // Processor state
    struct Context PageMap; // MMU state for proc
    int DPYNum; // Console number (and other I/O state)
} ProcTbl[N]; // one per process

int Cur; // index of “Active” process

Scheduler() {
    ProcTbl[Cur].State = UserMState; // Save Cur state
    Cur = (Cur+1)%N; // Incr mod N
    UserMState = ProcTbl[Cur].State; // Install state for next User
    LoadUserContext(ProcTbl[Cur].PageMap); // Install context
}
```
OS Organization: Processes

USER PROCESSES

P1:

loop: SVC(0)
  ...  SVC(1)
  ...  BR(loop)

P2:

loop: SVC(2)
  ...  SVC(3)
  ...  BR(loop)

PN:

loop: SVC(3)
  ...  SVC(1)
  ...  BR(loop)

KERNEL

P1: MState
    DPYNUM=0
    Scheduler
    Saved Regs (MState)
    Clock Handler

P2: MState
    DPYNUM=1

PN: MState
    DPYNUM=27

User Mode
PC[31]=0

Kernel Mode
PC[31]=1
One Interrupt at a Time!

User Processes:

P1:
- `loop:SVC(0)`
- `... SVC(1)`
- `... SVC(3)`
- `BR(loop)`

P2:
- `loop:SVC(2)`
- `... SVC(3)`
- `BR(loop)`

PN:
- `loop:SVC(3)`
- `... SVC(1)`
- `BR(loop)`

User Mode: PC[31]=0

Kernel Mode: PC[31]=1

Interrupts allowed!

Kernel code is not re-entrant!

No interrupts!
0x8000004

Look! The supervisor bit is on! So the processor enters kernel mode before first instruction of handler is fetched.

if (bad opcode) {
    // Reg[XP] ← PC+4;  PC ← “Illop”
    PCSEL = 3,
    WASEL = 1, WDSEL = 0, WERF = 1,
    MWR = 0
}
Exception Handling

// hardware interrupt vectors are in low memory
. = 0
BR(I_Reset) // when Beta first starts
BR(I_Il10p) // on Illegal Instruction (eg SVC)
BR(I_Clk) // on timer interrupt
BR(I_Kbd) // on keyboard interrupt, use RDCHAR() to get character
BR(I_Mouse) // on mouse interrupt, use CLICK() to get coords

This is where the HW sets the PC during an illegal opcode exception

// start of kernel-mode storage

KStack:
  LONG(.+4) // Pointer to ...
  STORAGE(256) // ... the kernel stack.

// Here's the SAVED STATE of the interrupted user-mode process
// filled by interrupt handlers
UserMState:
  STORAGE(32) // R0-R30... (PC is in XP/R30!)

N = 16 // max number of processes
Cur:
  LONG(0) // index (0 to N-1) into ProcTbl for current process
ProcTbl:
  STORAGE(N*PCB_Size) // PCB_Size = # bytes to hold complete state
Useful Macros

// Macro to extract and right-adjust a bit field from RA, and leave it
// in RB. The bit field M:N, where M >= N.
.macro extract_field (RA, M, N, RB) {
    SHLC(RA, 31-M, RB)       // Shift left, to mask out high bits
    SHRC(RB, 31-(M-N), RB)   // Shift right, to mask out low bits.
}

.macro save_all_regs(WHERE) save_all_regs(WHERE, r31)
    .macro save_all_regs(WHERE, base_reg) {
        ST(r0, WHERE, base_reg)
        ...
        ST(r30, WHERE+120, base_reg)
    }

.macro restore_all_regs(WHERE) restore_all_regs(WHERE, r31)
    .macro restore_all_regs(WHERE, base_reg) {
        LD(base_reg, WHERE, r0)
        ...
        LD(base_reg, WHERE+120, r30)
    }

Code is from beta.uasm
Illop Handler

/// Handler for Illegal Instructions
I_IlIlOp:
  save_all_regs(UserMState)    // Save the machine state.
  LD(KStack, SP)               // Install kernel stack pointer.
  ADDC(XP, -4, r0)            // Fetch the illegal instruction
  BR(ReadUserMem,LP)          // interpret addr in user context
  SHRC(r0, 26, r1)            // Extract the 6-bit OPCODE
  MULC(r1, 4, r1)             // Make it a WORD (4-byte) index
  LD(r1, UUOTbl, r1)          // Fetch UUOTbl[OPCODE]
  JMP(r1)                     // and dispatch to the UUO handler.

.macro UUO(ADR)  LONG(ADR+0x80000000)    // Auxiliary Macros
.macro BAD()    UUO(UUOError)

UUOTbl: BAD()    UUO(SVC_UUO)    UUO(swapreg)    BAD()
BAD()           BAD()           BAD()           BAD()
BAD()           BAD()           BAD()           BAD()

... more table follows ...
Accessing User Locations

We’ll need to use the VtoP routine from the previous lecture to translate a user-mode virtual address into the appropriate physical address. VtoP will have to be modified slightly to find the correct context now that we have multiple processes.

```assembly
// expects user-mode virtual address in R0,
// returns contents of that location in user’s virtual memory
ReadUserMem:
    PUSH(LP) // save registers we use below
    PUSH(r1)
    ANDC(r0,0xFFF,r1) // extract page offset
    PUSH(r1)
    SHRC(r0,12,r1) // extract virtual page number
    PUSH(r1)
    BR(VtoP,LP) // returns physical address in R0
    DEALLOCATE(2)
    LD(r0,0,r0) // load the contents
    POP(r1) // restore regs, return to caller
    POP(LP)
    JMP(LP)
```
Handler for Actual Illops

// Here's the handler for truly unused opcodes (not SVCs or swapreg):
// Illegal instruction is in R0, it's address is Reg[XP]-4
UuoError:
    CALL(KWrMsg)   // Type out an error msg,
    .text "Illegal instruction"

ADDC(XP, -4, r0)    // Fetch the illegal instruction
BR(ReadUserMem,LP)  // interpret addr in user context
CALL(KHexPrt)
CALL(KWrMsg)
    .text " at location 0x"

MOVE(xp,r0)
CALL(KHexPrt)
CALL(KWrMsg)
    .text "! ....."

HALT()             // Then crash system.

These kernel 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.
Emulated Instruction: swapreg(Ra,Rc)

// swapreg(RA,RC) swaps the contents of the two named registers.
.macro swapreg(RA,RC) betaopc(0x02,RA,0,RC)

// swapreg instruction is in R0, it’s address is Reg[XP]-4
swapreg:
  extract_field(r0, 25, 21, r1) // extract rc field
  MULC(r1, 4, r1) // convert to byte offset into regs array
  extract_field(r0, 20, 16, r2) // extract ra field
  MULC(r2, 4, r2) // convert to byte offset into regs array
  LD(r1, UserMState, r3) // r3 <- regs[rc]
  LD(r2, UserMState, r4) // r4 <- regs[ra]
  ST(r4, UserMState, r1) // regs[rc] <- old regs[ra]
  ST(r3, UserMState, r2) // regs[ra] <- old regs[rc]

// all done!  Resume execution of user-mode program
BR(I_Rtn) // defined in the next section!
Communicating with the OS

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

Solution:

*Abstraction*: a supervisor call (SVC) with args in registers – result in R0 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 (ie, not in user-mode address space) how do we get to it?

Okay... show me how it works!
OS Organization: Supervisor Calls

User Mode PC[31]=0

Kernel Mode PC[31]=1

P1: MState DPYNUM=0

Saved Regs (Mstate)

Scheduler

P2: MState DPYNUM=1

Illegal Op Handler

PN: MState DPYNUM=27

Scheduler

Field

WrCh

GetKey

User Mode

P1

P1: loop:SVC(0)

... SVC(1)...

BR(loop)

P1

P2

P2: loop:SVC(2)

... SVC(3)...

BR(loop)

P2

PN

PN: loop:SVC(3)

... SVC(1)...

BR(loop)

PN

Scheduler

Field

WrCh

GetKey

Illegal Op Handler

Saved Regs (Mstate)

User Mode PC[31]=0

Kernel Mode PC[31]=1

L19: Virtualizing the Processor, Slide #23
// Sub-handler for SVCs, called from I_Illop on SVC opcode:
// SVC instruction is in R0, it’s address is Reg[XP]-4
SVC_UU0:

\[
\begin{align*}
&\text{ANDC(r0,0x7,r1)} \quad \text{// Pick out low bits,} \\
&\text{SHLC(r1,2,r1)} \quad \text{// make a word index,} \\
&\text{LD(r1,SVCTbl,r1)} \quad \text{// and fetch the table entry.} \\
&\text{JMP(r1)}
\end{align*}
\]

SVCTbl:  UU0(HaltH) \quad \text{// SVC(0): User-mode HALT instruction}  \\
UU0(WrMsgH) \quad \text{// SVC(1): Write message}  \\
UU0(WrChH) \quad \text{// SVC(2): Write Character}  \\
UU0(GetKeyH) \quad \text{// SVC(3): Get Key}  \\
UU0(HexPrtH) \quad \text{// SVC(4): Hex Print}  \\
UU0(WaitH) \quad \text{// SVC(5): Wait(S), S in R3}  \\
UU0(SignalH) \quad \text{// SVC(6): Signal(S), S in R3}  \\
UU0(YieldH) \quad \text{// SVC(7): Yield()}

Another dispatch table!
Returning to User-mode

// Alternate return from interrupt handler which BACKS UP PC,
// and calls the scheduler prior to returning. This causes
// the trapped SVC to be re-executed when the process is
// eventually rescheduled...

HaltH:
I_Wait:
    LD(UserMState+(4*XP), r0) // Grab XP from saved MState,
    SUBC(r0, 4, r0) // back it up to point to
    ST(r0, UserMState+(4*XP)) // SVC instruction
YieldH:
    CALL(Scheduler) // Switch current process,
    BR(I_Rtn)

// Here's the common exit sequence from Kernel interrupt handlers:
// Restore registers, and jump back to the interrupted user-mode
// program.

I_Rtn:
    restore_all_regs(UserMState)
    JMP(XP) // Good place for debugging breakpoint!
Adding New SVCs

.macro GetTOD() SVC(8)  // return time of today in R0
.macro SetTOD() SVC(9)  // set time of day to value in R0

// Sub-handler for SVCs, called from I_Il1Op on SVC opcode:
// SVC instruction is in R0, its address is Reg[XP]-4
SVC_UU0:
    ANDC(r0, 0xF, r1)  // Pick out low bits,
    SHLC(r1, 2, r1)   // make a word index,
    LD(r1, SVCTbl, r1) // and fetch the table entry.
    JMP(r1)

SVCTbl:  UU0(HaltH)  // SVC(0): User-mode HALT instruction
         UU0(WrMsgH)  // SVC(1): Write message
         UU0(WrChH)  // SVC(2): Write Character
         UU0(GetKeyH) // SVC(3): Get Key
         UU0(HexPrtH) // SVC(4): Hex Print
         UU0(WaitH)  // SVC(5): Wait(S), S in R3
         UU0(SignalH) // SVC(6): Signal(S), S in R3
         UU0(YieldH) // SVC(7): Yield()
         UU0(GetTOD) // SVC(8): return time of day
         UU0(SetTOD) // SVC(9): set time of day
New SVC Handlers

// return the current time of day in R0
GetTOD:
  LD(TimeOfDay,r0)       // load OS time of day value
  ST(r0,UserMState+4*0)  // store into user’s R0
  BR(I_Rtn)              // resume execution with updated R0 value

// set the current time of day from the value in user’s R0
SetTOD:
  LD(UserMState+4*0,r0)  // load value in (saved) user’s R0
  ST(r0,TimeOfCay)      // store to OS time of day value
  BR(I_Rtn)              // resume execution

SVCs provide controlled access to OS services and data values and offer “atomic” (uninterrupted) execution of instruction sequences.