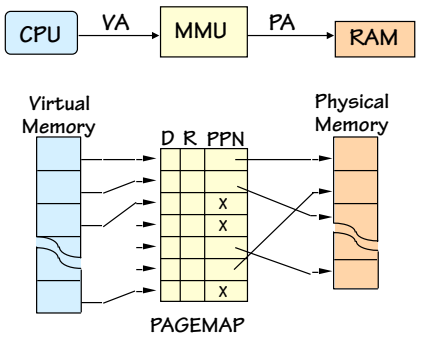
6.004 Computation Structures
Spring 2009

Virtual Machines

Lab 6 due Thursday!

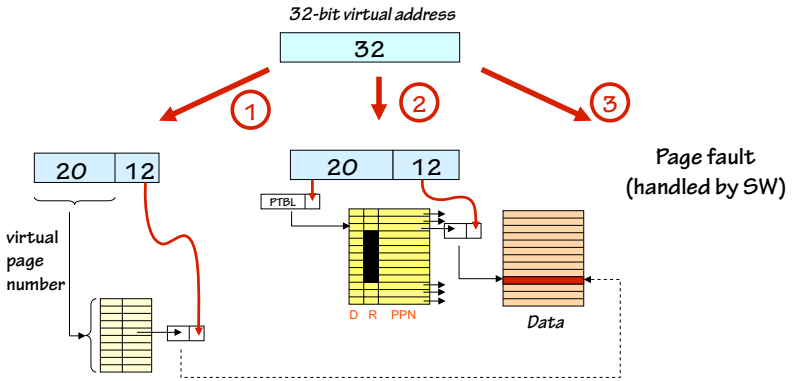
Review: Virtual Memory



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

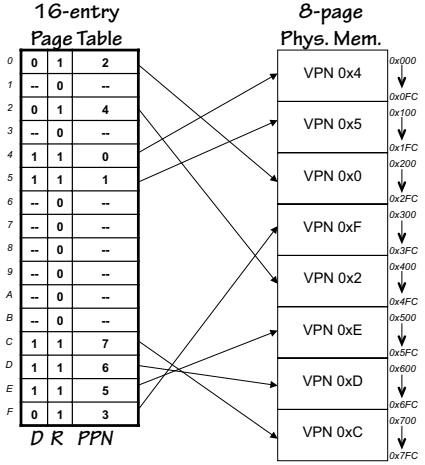
Typical Multi-level approach

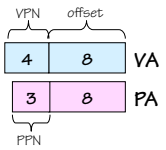


Page fault (handled by SW)

Look in TLB: VPN→PPN cache
Usually implemented as a small (16- to 64-entry) fully-associative cache

Example I



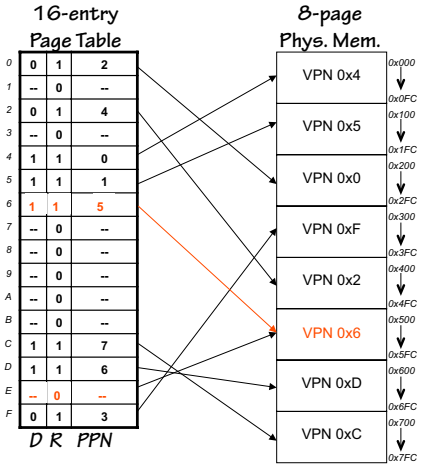


Setup:
- 256 bytes/page (2^8)
- 16 virtual pages (2^4)
- 8 physical pages (2^3)
- 12-bit VA (4 vpn, 8 offset)
- 11-bit PA (3 ppn, 8 offset)
- LRU page: VPN = 0xE

LD(R31,0x2C8,R0):
VA = 0x2C8, PA = 0x4C8

VPN = 0x2
→ PPN = 0x4

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

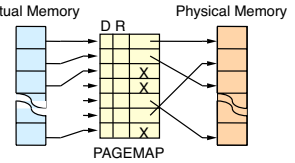
ST(BP,-4,SP), SP = 0x604
VA = 0x600, PA = 0x500

VPN = 0x6
→Not resident, it's on disk
→Choose page to replace (LRU = 0xE)
→D[0xE] = 1, so write 0x500-0x5FC to disk
→Mark VPN 0xE as no longer resident
→Read in page 0x6 from disk into 0x500-0x5FC
→Set up page map for VPN 0x6 = PPN 0x5
→PA = 0x500
→This is a write so set D[0x6] = 1

Contexts

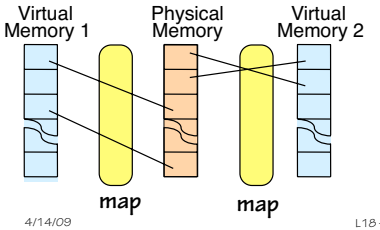
A *context* is an entire set of mappings from VIRTUAL to PHYSICAL page numbers as specified by the contents of the page map:

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

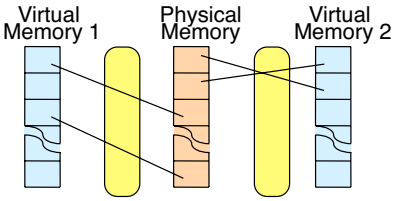


THE BIG IDEA: Several programs, each with their own context, may be simultaneously loaded into main memory!

"Context switch": reload the page map!



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.

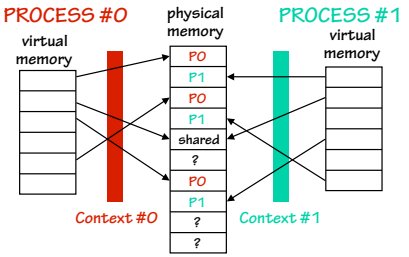
More than Virtual Memory: A VIRTUAL MACHINE

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

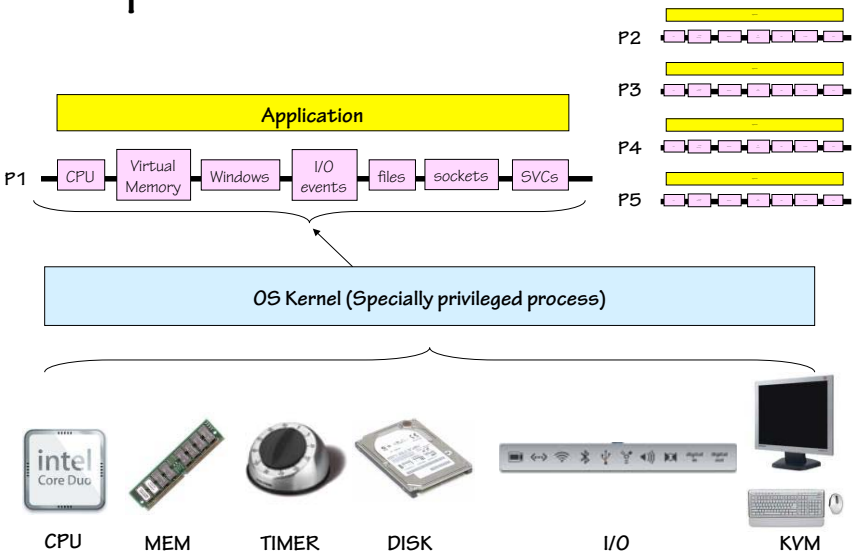


Goal: give each program its own "VIRTUAL MACHINE"; programs don't "know" about each other…

New abstraction: a *process* which has its own
- machine state: R0, …, R30
- context (virtual address space)
- PC, stack
- program (w/ shared code)
- virtual I/O devices (console…)

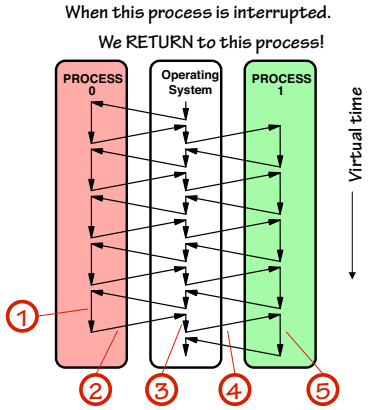
"OS Kernel" is a special, privileged process that oversees the other processes and handles real I/O devices, emulating virtual I/O devices for each process

Each process has its own virtual machine



Processes:

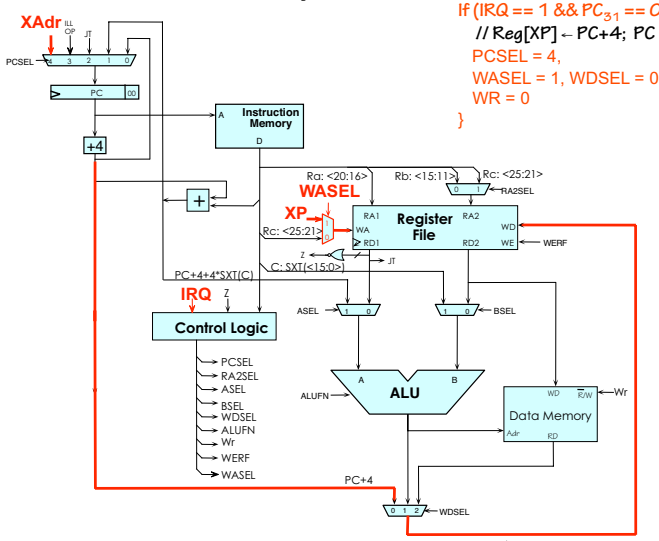
Multiplexing the CPU



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.

Interrupt Hardware



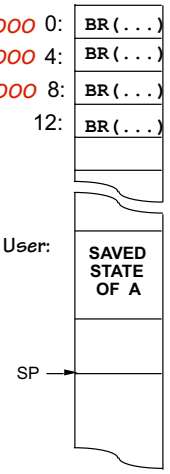

```
If (IRQ == 1 && PC31 == 0) {
   // Reg[XP] ← PC+4; PC ←"Xadr"
   PCSEL = 4,
   WASEL = 1, WDSEL = 0, WERF = 1,
   WR = 0
}
```


Beta Interrupt Handling

Minimal Hardware Implementation:

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

RESET → 0x80000000 0:
ILLOP → 0x80000000 4:
X_ADR → 0x80000000 8:
12:



Handler Coding:

- Save state in "User" structure
- Call C procedure to handle the exception
- re-install saved state from "User"
- Return to Reg[XP]

TRANSPARENT to interrupted program!

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

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

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

Handler (written in C)

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

"Interrupt stub" (written in assy.)

Simple Timesharing Scheduler

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

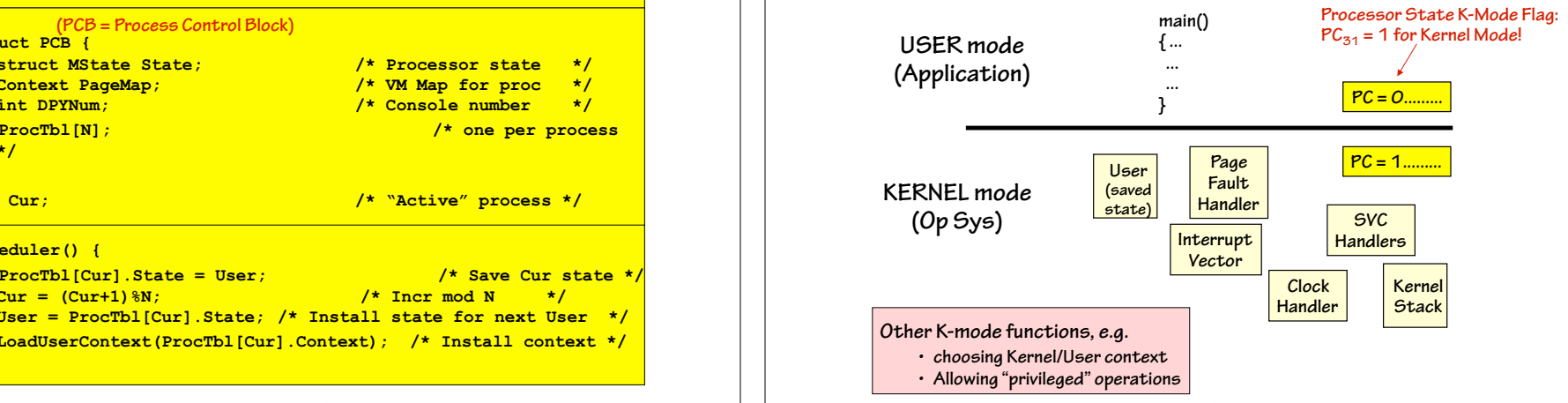
(PCB = Process Control Block)

```
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 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:



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 (ie, not in user-mode address space) how do we get to it?

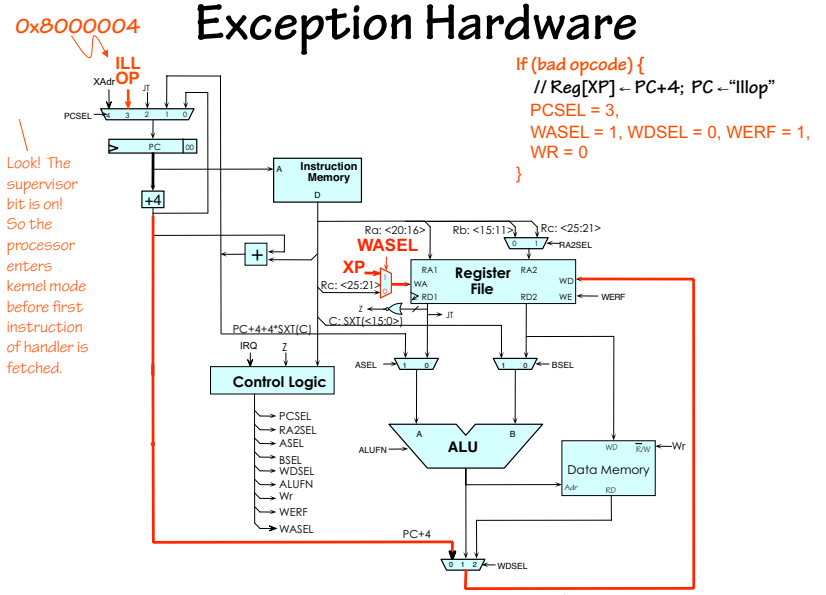
Solution:

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

Okay… show me how it works!

Exception Hardware



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,
WR = 0
}

Code is from lab8.uasm

Exception Handling

This is where the HW sets the PC during an exception

```
. = 0x00000004
BR(I_IllOp)       | on Illegal Instruction (eg SVC)

| Here's the SAVED STATE of the interrupted process, while we're
| processing an interrupt.
UserMState: STORAGE(32)   | R0-R31... (PC is in XP!)

| Here are macros to SAVE and RESTORE state -- 31 registers -- from
|   the above storage.

.macro SS(R) ST(R, UserMState+(4*R))       | (Auxiliary macro)
.macro SAVESTATE() {
SS(0)  SS(1)  SS(2)  SS(3)  SS(4)  SS(5)  SS(6)  SS(7)
SS(8)  SS(9)  SS(10) SS(11) SS(12) SS(13) SS(14) SS(15)
SS(16) SS(17) SS(18) SS(19) SS(20) SS(21) SS(22) SS(23)
SS(24) SS(25) SS(26) SS(27) SS(28) SS(29) SS(30) }

.macro RS(R) LD(UserMState+(4*R), R)       | (Auxiliary macro)
.macro RESTORESTATE() {
RS(0)  RS(1)  RS(2)  RS(3)  RS(4)  RS(5)  RS(6) RS(7)
RS(8)  RS(9)  RS(10) RS(11) RS(12) RS(13) RS(14) RS(15)
RS(16) RS(17) RS(18) RS(19) RS(20) RS(21) RS(22) RS(23)
RS(24) RS(25) RS(26) RS(27) RS(28) RS(29) RS(30) }
```

Macros can be used like an in-lined procedure call

Illop Handler

```
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||| Handler for Illegal Instructions
|||  (including SVCs)
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

I_IllOp:                                          Don't trust the user's stack!
        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)
                                  supervisor bit…

UUOTbl: BAD()    UUO(SVC_UUO)      BAD()   BAD()
        BAD()       BAD()             BAD()             BAD()
        BAD()       BAD()             BAD()             BAD()
        … more table follows…
```

This is a 64-entry dispatch table. Each entry is an address of a "handler"

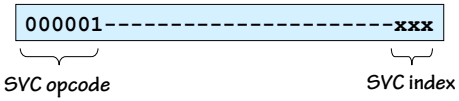
Actual Illops

```
||| Here's the handler for truly unused opcodes (not SVCs):
UUOError:
        CALL(KWrMsg)                    | Type out an error msg,
        .text "Illegal instruction "
        LD(xp, -4, r0)                  | giving hex instr and location;
        CALL(KHexPrt)
        CALL(KWrMsg)
        .text " at location 0x"
        MOVE(xp,r0)
        CALL(KHexPrt)
        CALL(KWrMsg)
        .text "! ....."
        HALT()                          | Then crash system.
```

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

SVC Instruction format




```
||| Sub-handler for SVCs, called from I_IllOp on SVC opcode:

SVC_UUO:
        LD(XP, -4, r0)          | The faulting instruction.
        ANDC(r0,0x7,r0)         | Pick out low bits,
        SHLC(r0,2,r0)           | make a word index,
        LD(r0,SVCTbl,r0)                | and fetch the table entry.
        JMP(r0)

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

Another dispatch table!

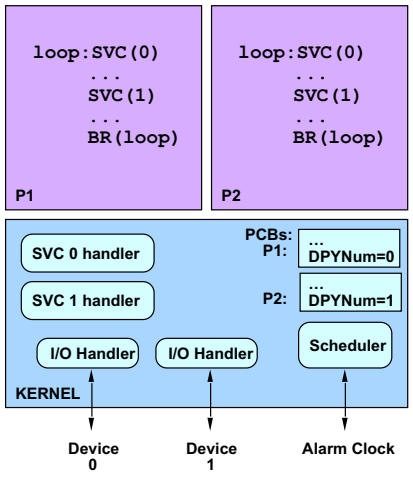
Handler for HALT SVC

```
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||| SVC Sub-handler for user-mode HALTs
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

HaltH:  BR(I_Wait)              | SVC(0): User-mode HALT SVC

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

I_Rtn:  RESTORESTATE()
kexit:  JMP(XP)                 | Good place for debugging breakpoint!

||| 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...

I_Wait: LD(UserMState+(4*30), r0)       | Grab XP from saved MState,
        SUBC(r0, 4, r0)         | back it up to point to
        ST(r0, UserMState+(4*30))       | SVC instruction

        CALL(Scheduler)         | Switch current process,
        BR(I_Rtn)               | and return to (some) user.
```

Looks like HALT should really be called LOOP!

Fills UserMState from PCB of next process to run

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