Topics in computer architecture

SunOS on SPARC

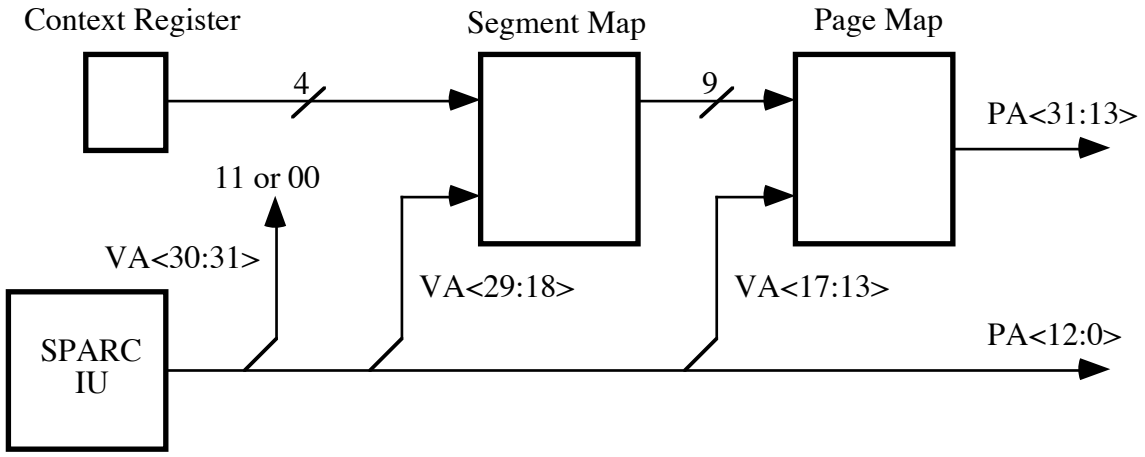
P.J. Drongowski
SandSoftwareSound.net

SunOS on SPARC

"SunOS on SPARC," S.R. Kleiman and D. Wiiliams, *Sun* Technology, Summer 1988, pg. 56 - 63

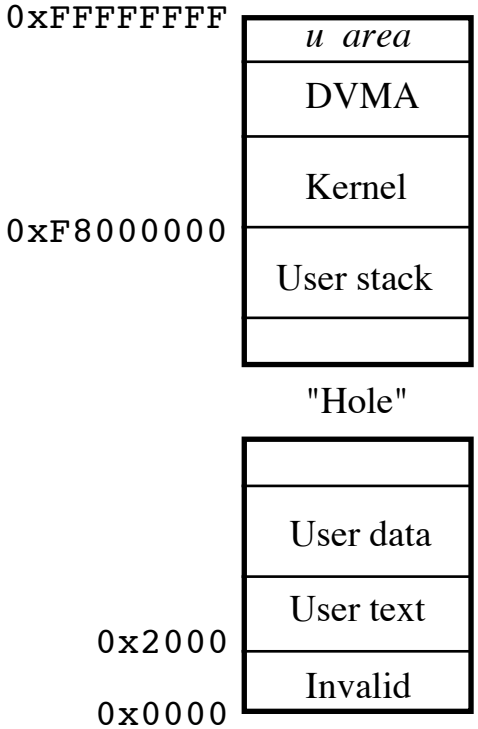
Virtual memory

- CPU does not have internal memory management
- CPU cache contains virtual addresses
- Memory management unit (MMU)
  - Provides protection and mapping
  - Two-level static-RAM based translation table
- "Segement map" (first-level table)
  - Index by 4-bit context and 12-bit segment number
  - Produce index into page map table
  - "Context"
    - ◊ Context is selected by Context Register
    - ◊ Memory map of virtual address space for a process
- "Page map" (second-level table)
  - 9-bit index from segment map and 5-bit VA page
  - Page table entry
    - ◊ Actual physical page address
    - ◊ Protection information and statistics bits
  - Page size is 8K bytes
  - "Page map entry group" (PMEG)
    - ◊ 32 page map entries
    - ◊ Each PMEG maps 256K bytes of memory



Process address space

- Portions of 32-bit virtual address space can be unmapped
- Part of address space can be marked invalid
- Process has a "hole" that grows or shrinks
- Sun-4/200
  - 16 contexts
  - 1 gigabyte of mappable virtual address space
  - 512 Mbytes at the top; 512 Mbytes at the bottom
- Operating system kernel
  - Occupies top 128 Mbytes in all contexts
  - *u area*
    - Per-process, global structure
    - Can be accessed with short, absolute address
    - Contains the kernel stack
  - Direct Virtual Memory Access (DVMA)
    - Used by external I / O devices
    - Provides mapping for devices into memory
    - Translates device memory requests to physical ones
- User area
  - User text (code) is in low virtual memory
  - User data (or heap) is above text and grows up
  - User stack grows down toward the "hole"



Page table entry

- Physical Page Number (PPN)
- Cacheable ($)
- Modified (M)
- Referenced (R)
- Access permissions (ACC)
- Entry Type (ET)

MMU management

- MMU is managed as a cache of entries
- Process begins execution without a context
- Context is built on first page fault
- Context or PMEG is taken away if no free ones available

MMU access

- MMU registers accessed as peripheral device registers
- Registers
  - Control Register
  - Context Table Pointer and Context Registers
  - Fault Status and Fault Address Registers

Fault Status Register

- External Bus Error (EBE) - timeout, parity error
- Level (L) - segement or page mapping level
- Access Type (AT) - user/supervisor, read/write/execute
- Fault Type (FT)
  - Invalid address error
  - Protection error
  - Privilege violation
  - Translation error
  - Access bus error
  - Internal error

Cache consistency

- Memory is accessed through virtually addressed cache
- Synonyms
  - Two or more VA's that map to same physical address
  - Cache entries can become inconsistent
  - Sun-4 synonyms can be cached if in same cache line
  - Note that this condition is machine dependent!
- Solution
  - Turn off cache for pages that have synonyms
  - Kernel maintains list of all mappings to a physical page
  - Request for new mapping to a physical page
    - Check for cached synonyms ("condition" above)
    - If yes, then all mappings may be cached
    - Otherwise, flush cache and mark page uncachable

Trap sequence

- Trap Base Register (TBR) points to a trap table
- Trap Table is indexed by trap type (256 types possible)
- Sequence
  - Disable traps
  - Copy S field to PS field in PSR; Set S to "supervisor"
  - Decrement Current Window Pointer (CWP) by 1
  - Save PC and nPC in r17 and r18 of new window
  - Load PC and nPC from trap table according to type
- Trap types
  - Synchronous
    - Reset
    - Window overflow and underflow
    - Instruction access and data access exception
    - Illegal or privileged instruction
    - FP or co-processor disabled
    - Memory address not aligned
    - Trap instruction
  - Asynchronous (interrupts)
  - Floating point / coprocessor exception traps

Processor State Register (PSR)

- Current Window Pointer (CWP)
- Enable Traps (ET) bit
- Supervisor (S) bit
- Previous Supervisor (PS) bit
- Window Invalid Mask (WIM)

Window overflow and underflow

- Form LIFO stack
- `Save` instruction pushes a window; `Restore` pops
- Overflow and underflow detection
  - Mark at least one window invalid (in WIM)
  - Usually between least and most recently used window
  - `Save` or `restore` to invalid window causes trap
  - Must prevent overlap of *ins* and *outs*
  - Need one window as CWP is decremented on trap
  - Trap handler may only use *local* registers freely
- Windows are saved on normal program stack
  - *Out* register (normally SP) points to save area
  - Area will contain 8 *in* and 8 *local* registers for window
  - One window save/restore is the most effective
- Handler
  - Loads and stores SPARC doublewords
  - Stack pointer must be doubleword aligned
  - Must check for save area VM residency

```
window_overflow:
  save
  %wim = rotate_right(%wim)
  if (window to be saved is a user window)
    if (%sp & 7)
      save user window to intenal buffer
      goto user_alignment_trap
    if (stack is writeable)
      save window data to stack
    else
      save window data to internal buffer
      if (user_trap) goto user_page_fault
  else
     save system window data to stack
  restore
  return from trap
```

```
window_underflow:
  %wim = rotate_left(%wim)
  restore
  restore
  if (user_trap)
    if (%sp & 7)
      goto user_alignment_trap
    if (stack is not readable)
      goto user_page_fault
  restore window from stack
  save
  save
  return from trap
```

Generic trap handlers

- Standard preamble and postamble for traps
- Trap handler can use register windows
- Generic preamble
  - Save the *global* registers
  - Save the PSR
  - If window overflow, save next window
  - If it's an interrupt, set the processor interrupt level
  - Enable traps
  - Dispatch to handler code
- Generic postamble
  - If window underflow, restore previous window
  - Restore the PSR
  - Restore the *global* registers
  - Return from trap
- Kernel must clean the kernel register windows

Floating-point trap handlers

- FPU is enabled by bit in the PSR
  - Processes start with the FPU disabled
  - Execution of first FP instruction produces trap
  - Handler enables the FPU
  - Initializes FPU registers to "Not a Number" (NaN)
  - Marks the process for FPU context switching
- Kernel can simulate unimplemented FP operations
- FPU operation
  - Concurrent with Integer Unit (IU)
  - Multiple FP operations may proceed in parallel
  - FPU saves state when exception occurs
  - All FP operations appear to complete in sequence
- FPU exceptions
  - Generated asynchronously, taken in synch by IU
  - FPU maintains queue of address/instruction pairs
  - Trap handler unloads queue and takes action

Context switching

- Main action is saving/restoring windows on stack
- Two less than number of windows must be moved
- Three active register windows are flushed (average)
- Kernel context switch
  - Flush current stack from virtual address cache
  - Flush is expensive (75% of context switch time)
  - Switch kernel stacks by switching MMU *u area* entries

```
cswitch:
  store stack pointer
  store PC (return address)
  save global registers (if required)
  save floating-point registers (if required)
  flush active register windows to the stack
  restore floating-point registers (if required)
  restore global registers (if required)
  load new return address
  load new stack pointer
  restore
  return
```

```
Sun OS cswitch:
  store stack pointer
  store PC (return address)
  save global registers (if required)
  save floating-point registers (if required)
  save; save; ... NWINDOWS-2 times
  restore; restore; ... NWINDOWS-2 times
  update u area MMU entries
  charge to new MMU context
  restore floating-point registers (if required)
  restore global registers (if required)
  load new return address
  load new stack pointer
  restore
```

OS related instructions

- Load from alternate space (`lda`)
  - Privileged instruction
  - (un)signed byte, halfword, word, double word
  - Must specify *address space identifier* (asi)
- Store into alternate space (`sta`)
  - Privileged instruction
  - (un)signed byte, halfword, word, double word
  - Must specify *address space identifier* (asi)
- Atomic load-store unsigned byte (`ldstub`)
  - Read byte from memory, write all ones back atomically
  - Multiprocessors are guaranteed to execute sequentially
  - Alternate space version (privileged)
- Swap register with memory (`swap`)
  - Swap register with contents of location atomically
  - Asynchronous traps are not allowed
  - Multiprocessor execute is sequential in some order
  - Alternate space version (privileged)
- Return from trap (`rett`)
  - Adds one to CWP (deallocates current window)
  - Can cause underflow trap
  - Delayed control transfer to target address
  - Privileged instruction
- Read Processor Status Register (`rdpsr`)
  - Privileged instruction
  - Corresponding write to PSR
- Read Window Invalid Mask Register (`rdwim`)
  - Privileged instruction
  - Corresponding write to WIM
- Read Trap Base Register (`rdtbr`)
  - Privileged instruction
  - Corresponding write to TBR
- Unimplemented instruction (`unimp`)
- Instruction cache flush (`iflush`)
  - Flushes specific word from internal instruction cache
  - Causes trap if instruction cache is not internal