Virtual Memory

- Virtual Memory
- *Reading: Silbershatz, Chapter 9*
- Virtual Memory, MIPS-Style

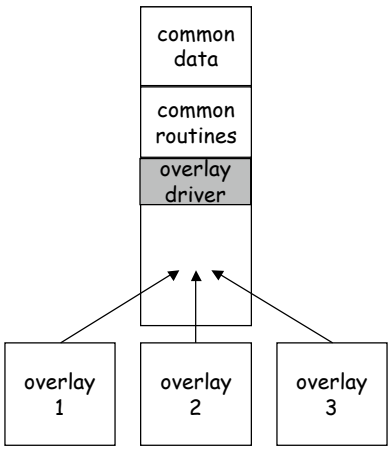
Virtual Memory

- Overview / Motivation
- Simple Approach: Overlays
- Locality of Reference
- Demand Paging
- Policies
  - Placement
  - Replacement
  - Allocation
- Case Studies: Unix SystemV
- *Reading: Silberschatz, Chapter 9*

Virtual Memory

- Allow execution of processes that may not be completely in memory.
  - 1990: Run dBaseIV on MS/DOS without expanded memory.
  - 1995: Run X and Netscape on a Sun with 12MB memory.
  - 2004: Run RiseOfNations on Windows box with 128MB memory.
- Benefits:
  - Program size not constrained by amount of physical memory available.
  - More programs can be run simultaneously
  - Less need for swapping

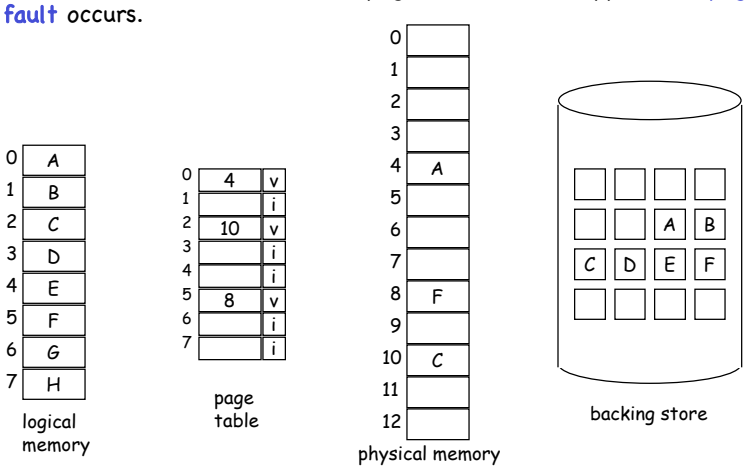
Virtual Memory at its Simplest: Overlays



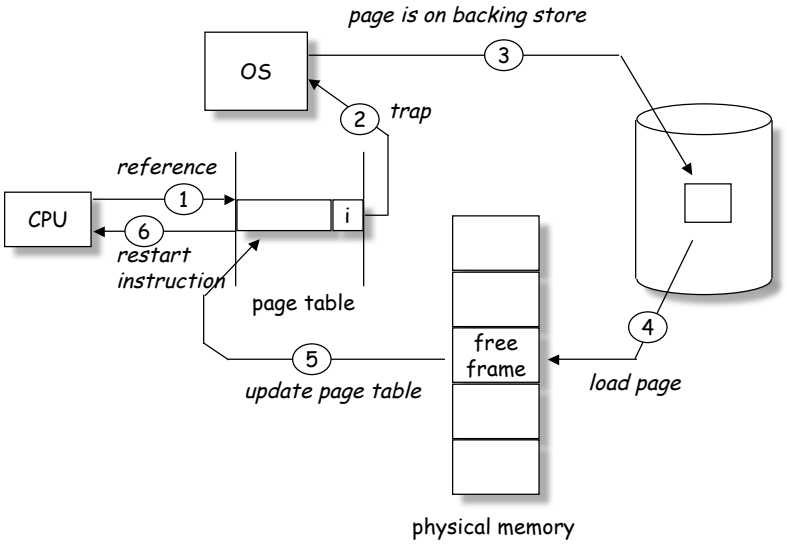
- Keep in memory only those instructions and data that are needed at any given time.
- Special relocation and linking needed to construct overlays.
- Don't need special support from OS.
- Require proper design of overlay structure.

Demand Paging

- "Lazy Swapper": only swap in pages that are needed.
- Whenever CPU tries to access a page that is not swapped in, a page fault occurs.



Mechanics of a Page Fault



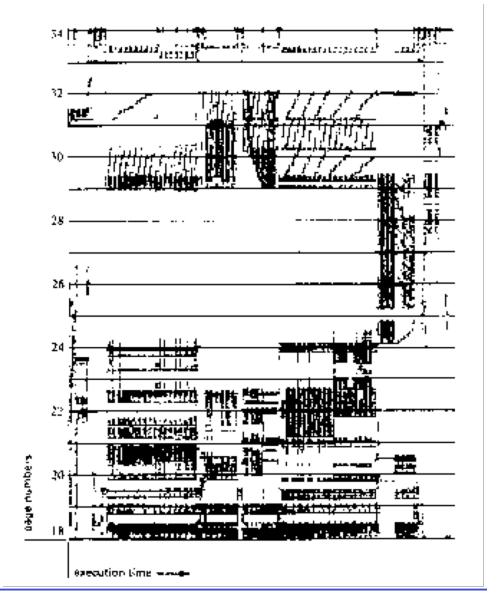
Locality of Reference

- Page faults are expensive!
- Thrashing: Process spends most of the time paging in and out instead of executing code.
- Most programs display a pattern of behavior called the principle of locality of reference.

Locality of Reference

A program that references a location *n* at some point in time is likely to reference the same location *n* and locations in the immediate vicinity of *n* in the near future.

Memory Access Trace



Architectural Considerations

- Must be able to restart any instruction after a page fault.
- e.g.

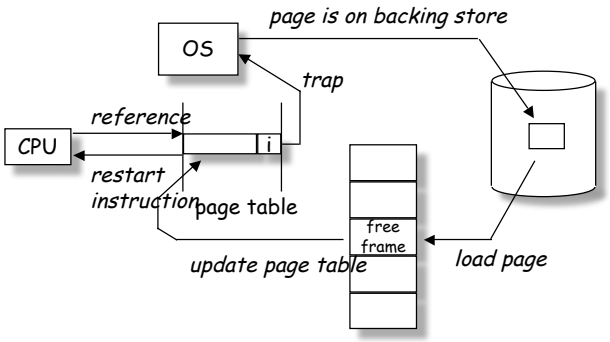
```
ADD A,B TO C
```

- What about operations that modify several locations in memory?
  - e.g. block copy operations?
- What about operations with side effects?
  - e.g. PDP-11, 80x86 auto-decrement, auto-increment operations?
  - Add mechanism for OS to "undo" instructions.

Performance of Demand Paging

- Effective Memory Access time **ema**:

  *ema = (1-p) * ma + p * "page fault time"*
- where
  - *p* = probability of a page fault
  - *ma* = memory access time
- Operations during Page Fault:



1. service page fault interrupt
2. swap in page
3. restart process

OS Policies for Virtual Memory

- Fetch Policy
  - How/when to get pages into physical memory.
  - demand paging *vs.* prepaging.
- Placement Policy
  - Where in physical memory to put pages.
  - Only relevant in NUMA machines.
- Replacement Policy
  - Physical memory is full. Which frame to page out?
- Resident Set Management Policy
  - How many frames to allocate to process?
  - Replace someone elses frame?
- Cleaning Policy
  - When to write a modified page to disk.
- Load Control

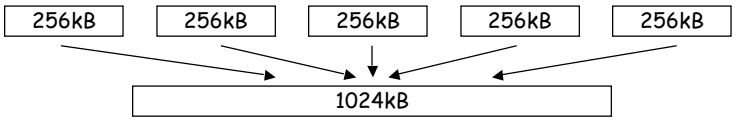
Configuring the Win2k Memory Manager

- Registry Values that Affect the Memory Manager:

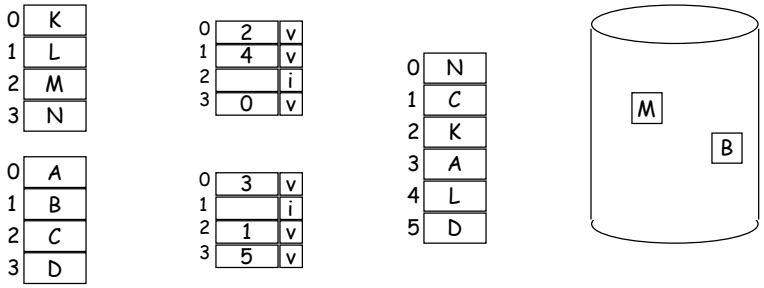
```
ClearPageFileAtShutdown
DisablePagingExecutive
IoPageLockLimit
LargePageMinimum
LargeSystemCache
NonPagedPoolQuota
NonPagedPoolSize
PagedPoolQuota
PagedPoolSize
SystemPages
```

Page Replacement

- Virtual memory allows higher degrees of multiprogramming by over-allocating memory.

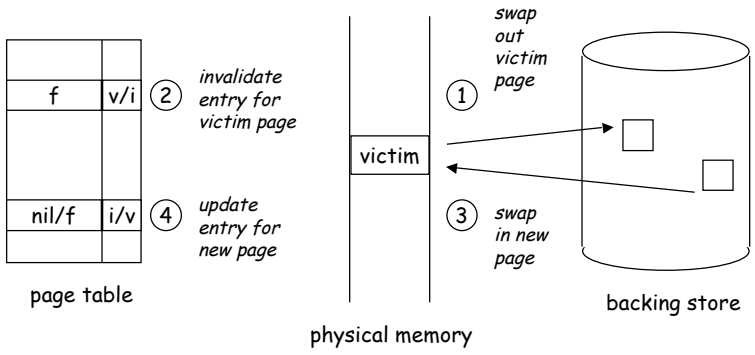


- Example:



Mechanics of Page Replacement

- Invoked whenever no free frame can be found.



- Problem: Need two page transfers!
  Solution: **Dirty bit**.

Page Replacement Algorithms

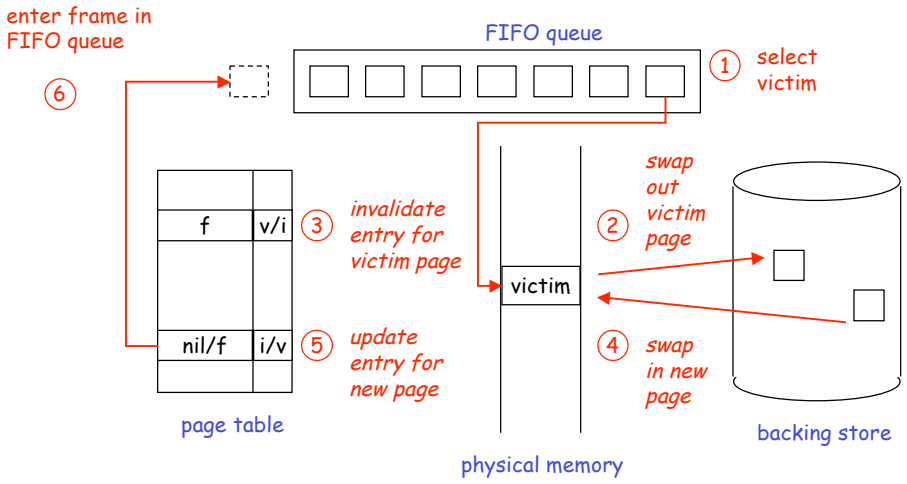
- Objective: Minimize page fault rate.
- Why bother?

- Example

```
for(int i=0; i<10; i++) {
  a = x * a;
}
```

- Evaluation: Sequence of memory references: **reference string**.

FIFO Page Replacement



FIFO Page Replacement (cont.)

- Example:

time	1	2	3	4	5	6	7	8	9	10
reference string	c	a	d	b	e	b	a	b	c	d
frames	a	a	a	a	a	e	e	e	e	e	d
	b	b	b	b	b	b	b	a	a	a	a
	c	c	c	c	c	c	c	c	b	b	b
	d	d	d	d	d	d	d	d	d	c	c
						!		!	!	!	!

- Advantage: simplicity
- Disadvantage: Assumes that pages residing the longest in memory are the least likely to be referenced in the future (does not exploit *principle of locality*).

Optimal Replacement Algorithm

- Algorithm with lowest page fault rate of all algorithms:

> Replace that page which will not be used for the longest period of time.

- Example:

time	1	2	3	4	5	6	7	8	9	10
reference string	c	a	d	b	e	b	a	b	c	d
frames	a	a	a	a	a	a	a	a	a	a	d
	b	b	b	b	b	b	b	b	b	b	b
	c	c	c	c	c	c	c	c	c	c	c
	d	d	d	d	d	e	e	e	e	e	e
						!				!	

Approximation to Optimal: LRU

- **Least Recently Used**: replace the page that has not been accessed for longest period of time.
- Example:

time	1	2	3	4	5	6	7	8	9	10
reference string	c	a	d	b	e	b	a	b	c	d
frames	a	a	a	a	a	a	a	a	a	a
	b	b	b	b	b	b	b	b	b	b
	c	c	c	c	c	e	e	e	e	d
	d	d	d	d	d	d	d	d	d	c
						!			!	!

LRU: Implementation

- Need to keep chronological history of page references; need to be reordered upon each reference.
- **Stack**:

stack	?	c	a	d	b	e	b	a	b	c	d
	?	?	c	a	d	b	e	b	a	b	c
	?	?	?	c	a	d	d	e	e	a	b
	?	?	?	?	c	a	a	d	d	e	a

- **Capacitors**: Associate a capacitor with each memory frame. Capacitor is charged with every reference to the frame. The subsequent exponential decay of the charge can be directly converted into a time interval.
- **Aging registers**: Associate aging register of n bits (R_{n-1}, ..., R_0) with each frame in memory. Set R_{n-1} to 1 for each reference. Periodically shift registers to the right.

Approximation to LRU: Clock Algorithm

- Associate a *use_bit* with every frame in memory.
  - Upon each reference, set *use_bit* to 1.
  - Keep a pointer to first "victim candidate" page.
  - To select victim: If current frame's *use_bit* is 0, select frame and increment pointer. Otherwise delete *use_bit* and increment pointer.

time	1	2	3	4	5	6	7	8	9	10
reference string	c	a	d	b	e	b	a	b	c	d
frames	a/1	a/1	a/1	a/1	a/1	e/1	e/1	e/1	e/1	e/1	d/1
	b/1	b/1	b/1	b/1	b/1	b/0	b/1	b/0	b/1	b/1	b/0
	c/1	c/1	c/1	c/1	c/1	c/0	c/0	a/1	a/1	a/1	a/0
	d/1	d/1	d/1	d/1	d/1	d/0	d/0	d/0	d/0	c/1	c/0
						!	!		!	!	

Improvement on Clock Algorithm (Second Chance Algorithm)

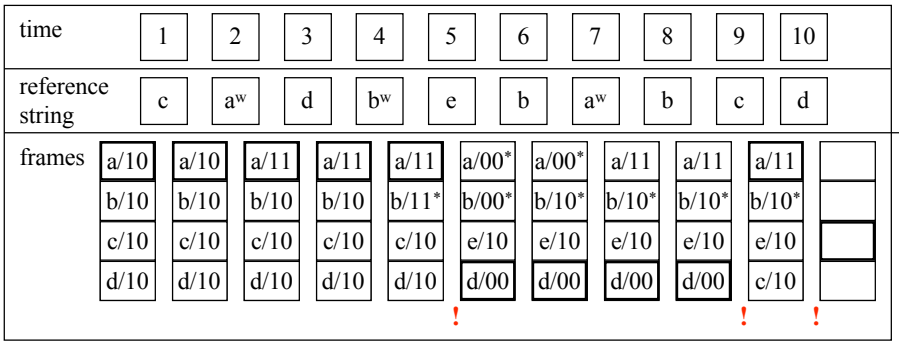
- Consider read/write activity of page: *dirty_bit* (or *modify_bit*)
- Algorithm same as clock algorithm, except that we scan for frame with both *use_bit* and *dirty_bit* equal to 0.
- Each time the pointer advances, the *use_bit* and *dirty_bit* are updated as follows:

	u d	u d	u d	u d
before	1 1	1 0	0 1	0 0
after	0 1	0 0	0 0*	(select)

- Called *Second Chance* because a frame that has been written to is not removed until <u>two</u> full scans of the list later.
- <u>Note</u>: Stallings describes a slightly different algorithm!

Improved Clock (cont)

- Example:

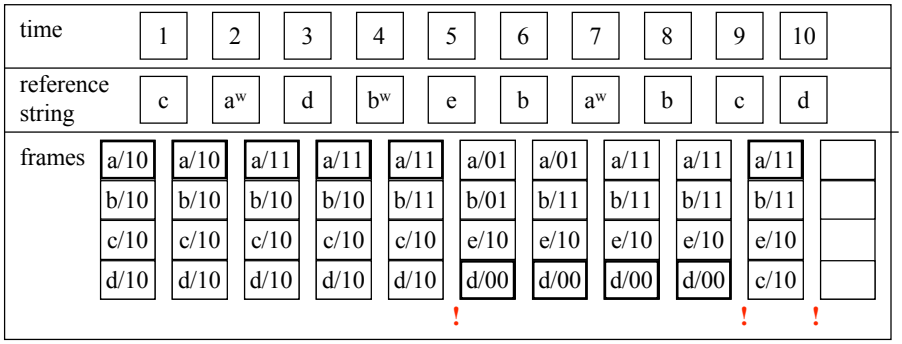


The Macintosh VM Scheme (see Stallings)

- Uses use_bit and modify_bit.
- **Step 1:** Scan the frame buffer. Select first frame with use_bit and modify_bit cleared.
- **Step 2:** If Step 1 fails, scan frame buffer for frame with use_bit cleared and modify_bit set. During scan, clear use_bit on each bypassed frame.
- Now all use_bit's are cleared. Repeat Step 1 and, if necessary, Step 2.

The Macintosh Scheme (cont)

- Example:



Resident Set Management

- Local vs. Global replacement policy:
  - The page to be replaced is selected from the resident set of pages of the faulting process. (local)
  - The page to be replaced may belong to any of the processes in memory.
- Each program requires a certain minimum set of pages to be resident in memory to run efficiently.
- The size of this set changes dynamically as a program executes.
- This leads to algorithms that attempt to maintain an optimal resident set for each active program. (Page replacement with variable number of frames.)

The Working Set Model

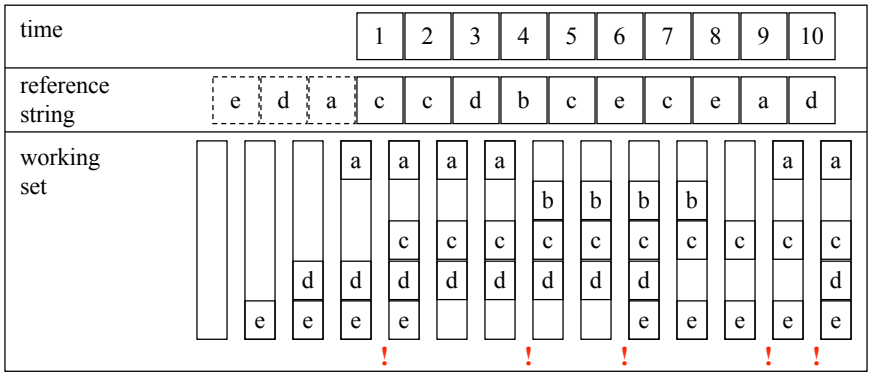
- **Working Set** *W(t,Δ)*: set of pages referenced by process during time interval *(t-Δ, t)*

$$\|W(t,1)\| = 1 \qquad 1 \leq \|W(t,\Delta)\| \leq min(\Delta, N)$$

- The storage management strategy follows two rules:
  - At each reference, the current working set is determined and only those pages belonging to the working set are retained in memory.
  - A program may run only if its entire current working set is in memory.
- Underlying Assumption: cardinality of working set remains constant over small time intervals.

Working Set Model (cont.)

- Example: (Δ = 4)



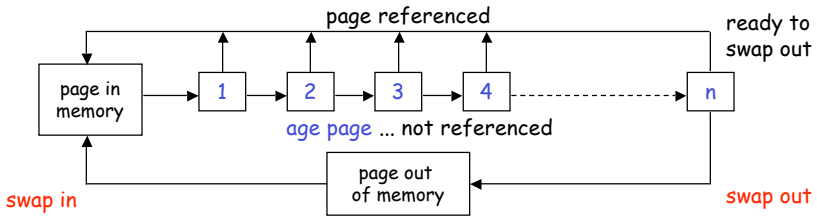
- Problems:
  - Difficulty in keeping track of working set.
  - Estimation of appropriate window size Δ.

Improve Paging Performance: Page Buffering

- Victim frames are not overwritten directly, but are removed from page table of process, and put into:
  - free frame list (clean frames)
  - modified frame list (modified frames)
- Victims are picked from the free frame list in FIFO order.
- If referenced page is in free or modified list, simply reclaim it.
- Periodically (or when running out of free frames) write modified frame list to disk.

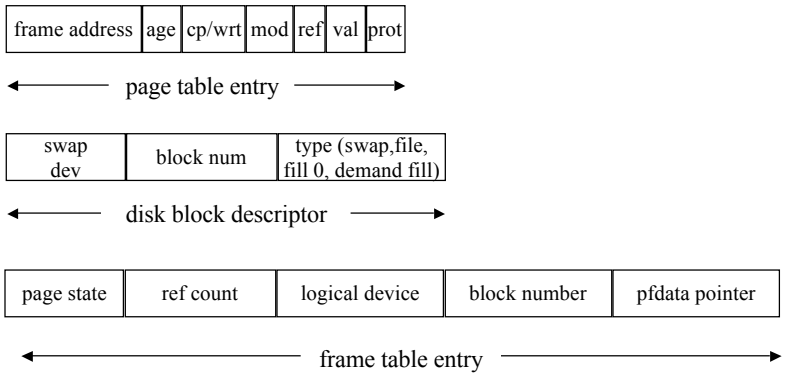
Page Buffering and Page Stealer

- Kernel process (e.g. `pageout` in Solaris) that swaps out memory frames that are no longer part of a working set of a process.
- Periodically increments age field in valid pages.



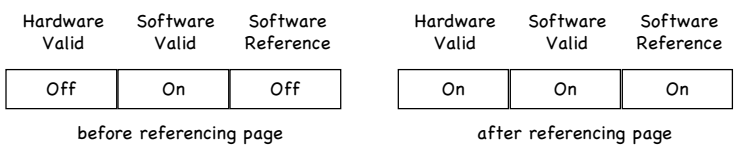
- Page stealer wakes up when available free memory is below *low-water mark*. Swaps out frames until available free memory exceeds *high-water mark*.
- Page stealer collects frames to swap and swaps them out in a single run. Until then, frames still available for reference.

Implementation of Demand Paging in UNIX SVR4



Demand Paging on Less-Sophisticated Hardware

- Demand paging most efficient if hardware sets the *reference* and *dirty* bits and causes a protection fault when a process writes a page whose *copy_on_write* bit is set.
- Can duplicate *valid* bit by a *software-valid* bit and have the kernel turn off the *valid* bit. The other bits can then be simulated in software.
- Example: Reference Bit:
  - If process references a page, it incurs a page fault because *valid* bit is off. Page fault handler then checks *software-valid* bit.
  - If set, kernel knows that page is really valid and can set *software-reference* bit.



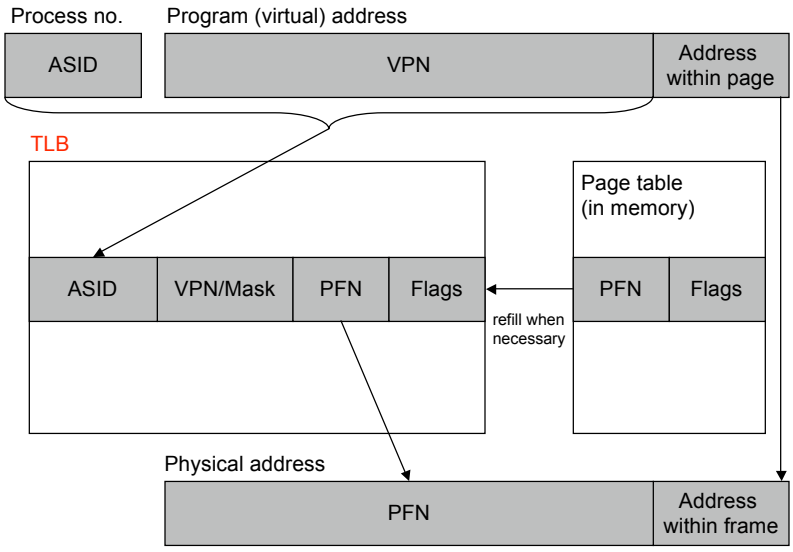
`fork()` System Call in Paging Systems

- Naive: `fork()` makes a physical copy of parent address space. However, `fork()` mostly followed by an `exec()` call, which overwrites the address space.
- System V: Use `copy_on_write` bit:
  - During `fork()` system call, all `copy_on_write` bits of pages of process are turned on. If either process writes to the page, incurs protection fault, and, in handling the fault, kernel makes a new copy of the page for the faulting process.
- BSD: Offers `vfork()` system call, which does not copy address space. Tricky! (May corrupt process memory.)

Virtual Memory

- Virtual Memory
- *Reading: Silberschatz, Chapter 9*
- Virtual Memory, MIPS-Style

Virtual Memory - MIPS Style



Memory Translation -- VAX style

- Split virtual address
- Concatenate more-significant bits with Process ASID to form page address.
- Look in the TLB to see if we find translation entry for page.
- If YES, take high-order physical address bits.
  - (Extra bits stored with PFN control access to frame.)
- If NO, system must locate page entry in main-memory-resident page table, load it into TLB, and start again.

Memory Translation -- MIPS Style

- In principle: Do the same as VAX, but with as little hardware as possible.
- Apart from register with ASID, the MMU is just a TLB.
- The rest is all implemented in software!
- When TLB cannot translate an address, a special exception (TLB refill) is raised.
- Question: This is easy in principle, but tricky to do efficiently.

MIPS TLB Entry Fields

- VPN: higher order bits of virtual address
- ASID: identifies the address space
- G: if set, disables the matching with the ASID

- PFN: Physical frame number
- N: 0 - cacheable, 1 - noncacheable
- D: write-control bit (set to 1 if writeable)
- V: valid bit

MIPS Translation Process

- CPU generates a program (virtual) address on a instruction fetch, a load, or a store.
- The 12 low-end bits are separated off.
- TLB matches key:
  - Matching entry is selected, and PFN is glued to low-order bits of the program address.
  - Valid?: The V and D bits are checked. If problem, raise exception, and set BadVAddr register with offending program address.
  - Cached?: IF C bit is set, the CPU looks in the cache for a copy of the physical location's data. If C bit is cleared, it neither looks in nor refills the cache.

TLB Refill Exception

- Figure out if this was a correct translation. If not, trap to handling of address errors.
- If translation correct, construct TLB entry.
- If TLB already full, select an entry to discard.
- Write the new entry into the TLB.