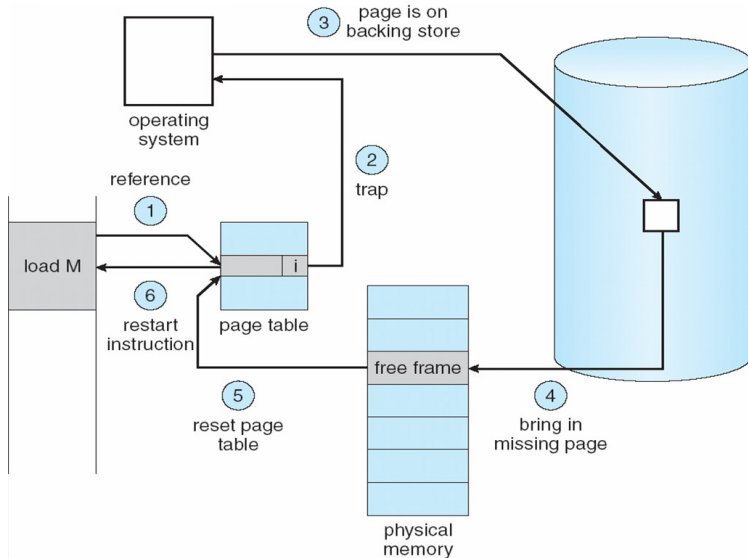


Outline

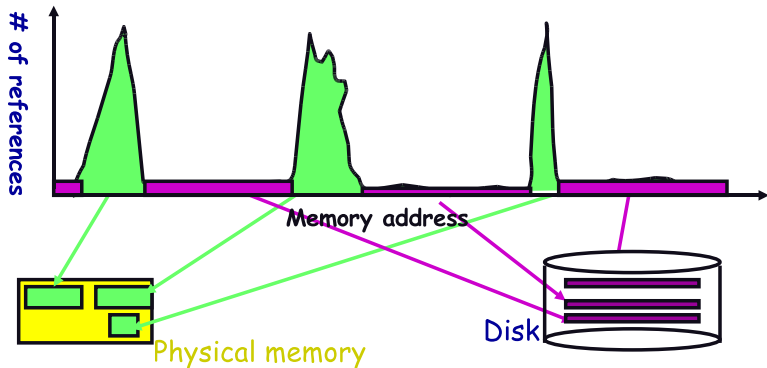
- ① Paging
- ② Eviction policies
- ③ Thrashing
- ④ Details of paging
- ⑤ The user-level perspective
- ⑥ Case study: 4.4 BSD

Paging



- Use disk to simulate larger virtual than physical mem

Working set model



- **Disk much, much slower than memory**
 - Goal: Run at memory, not disk speeds
- **90/10 rule: 10% of memory gets 90% of memory accesses**
 - So, keep that 10% in real memory, the other 90% on disk
 - How to identify the hot 10%?

Paging challenges

- **How to resume a process after a fault?**
 - Need to save state and resume
 - Process might have been in the middle of an instruction!
- **What to fetch from disk?**
 - Just needed page or more?
- **What to eject?**
 - How to allocate physical pages amongst processes?
 - Which of a particular process's pages to keep in memory?

Re-starting instructions

- **Hardware provides kernel w. info about page fault**
 - Faulting virtual address (In %cr2 reg on x86—may see it if you modify Pintos `page_fault` and use `fault_addr`)
 - Address of instruction that caused fault
 - Was the access a read or write? Was it an instruction fetch?
Was it caused by user access to kernel-only memory?
- **Hardware must allow resuming after a fault**
- **Idempotent instructions are easy**
 - E.g., simple load or store instruction can be restarted
 - Just re-execute any instruction that only accesses one address
- **Complex instructions must be re-started, too**
 - E.g., x86 move string instructions
 - Specify `src`, `dst`, `count` in `%esi`, `%edi`, `%ecx` registers
 - On fault, registers adjusted to resume where move left off

What to fetch

- **Bring in page that caused page fault**
- **Pre-fetch surrounding pages?**
 - Reading two disk blocks approximately as fast as reading one
 - As long as no track/head switch, seek time dominates
 - If application exhibits spacial locality, then big win to store and read multiple contiguous pages
- **Also pre-zero unused pages in idle loop**
 - Need 0-filled pages for stack, heap, anonymously mmaped memory
 - Zeroing them only on demand is slower
 - Hence, many OSes zero freed pages while CPU is idle

Selecting physical pages

- **May need to eject some pages**
 - More on eviction policy in two slides
- **May also have a choice of physical pages**
- **Direct-mapped physical caches**
 - Virtual \rightarrow Physical mapping can affect performance
 - In old days: Physical address A conflicts with $kC + A$ (where k is any integer, C is cache size)
 - Applications can conflict with each other or themselves
 - Scientific applications benefit if consecutive virtual pages do not conflict in the cache
 - Many other applications do better with random mapping
 - These days: CPUs more sophisticated than $kC + A$

Superpages

- **How should OS make use of “large” mappings**
 - x86 has 2/4MB pages that might be useful
 - Alpha has even more choices: 8KB, 64KB, 512KB, 4MB
- **Sometimes more pages in L2 cache than TLB entries**
 - Don't want costly TLB misses going to main memory
- **Or have two-level TLBs**
 - Want to maximize hit rate in faster L1 TLB
- **OS can transparently support superpages [Navarro]**
 - “Reserve” appropriate physical pages if possible
 - Promote contiguous pages to superpages
 - Does complicate evicting (esp. dirty pages) – demote

Outline

- 1 Paging
- 2 Eviction policies
- 3 Thrashing
- 4 Details of paging
- 5 The user-level perspective
- 6 Case study: 4.4 BSD

Straw man: FIFO eviction

- Evict oldest fetched page in system
- Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 physical pages: 9 page faults

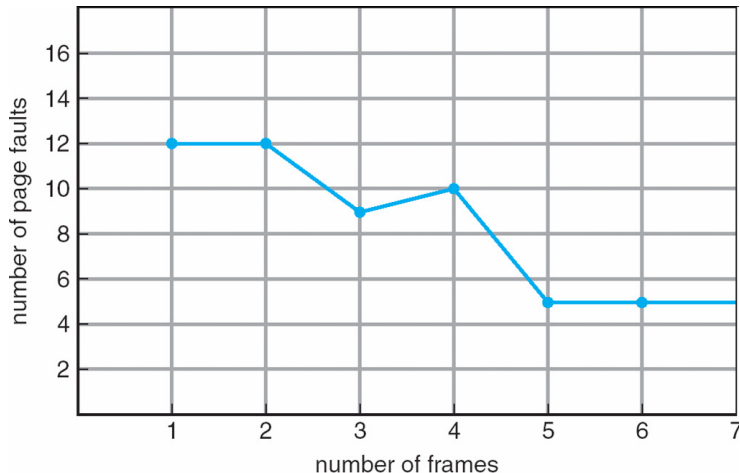
| | | | | |
|---|---|---|---|---------------|
| 1 | 1 | 4 | 5 | 9 page faults |
| 2 | 2 | 1 | 3 | |
| 3 | 3 | 2 | 4 | |

Straw man: FIFO eviction

- Evict oldest fetched page in system
- Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 physical pages: 9 page faults
- 4 physical pages: 10 page faults

| | | | | |
|---|---|---|---|----------------|
| 1 | 1 | 5 | 4 | |
| 2 | 2 | 1 | 5 | 10 page faults |
| 3 | 3 | 2 | | |
| 4 | 4 | 3 | | |

Belady's Anomaly



- More physical memory doesn't always mean fewer faults

Optimal page replacement

- What is optimal (if you knew the future)?

Optimal page replacement

- What is optimal (if you knew the future)?
 - Replace page that will not be used for longest period of time
- Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- With 4 physical pages:

| |
|---|
| 1 |
| 2 |
| 3 |
| 4 |

4

6 page faults

5

LRU page replacement

- **Approximate optimal with least recently used**
 - Because past often predicts the future
- **Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**
- **With 4 physical pages: 8 page faults**

| | |
|---|-----|
| 1 | 5 |
| 2 | |
| 3 | 5 4 |
| 4 | 3 |

- **Problem 1: Can be pessimal – example?**
- **Problem 2: How to implement?**

LRU page replacement

- **Approximate optimal with least recently used**
 - Because past often predicts the future
- **Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**
- **With 4 physical pages: 8 page faults**

| | |
|---|-----|
| 1 | 5 |
| 2 | |
| 3 | 5 4 |
| 4 | 3 |

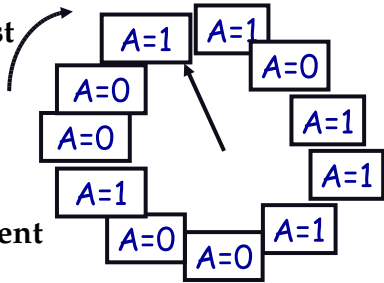
- **Problem 1: Can be pessimal – example?**
 - Looping over memory (then want MRU eviction)
- **Problem 2: How to implement?**

Straw man LRU implementations

- **Stamp PTEs with timer value**
 - E.g., CPU has cycle counter
 - Automatically writes value to PTE on each page access
 - Scan page table to find oldest counter value = LRU page
 - Problem: Would double memory traffic!
- **Keep doubly-linked list of pages**
 - On access remove page, place at tail of list
 - Problem: again, very expensive
- **What to do?**
 - Just approximate LRU, don't try to do it exactly

Clock algorithm

- Use accessed bit supported by most hardware
 - E.g., Pentium will write 1 to A bit in PTE on first access
 - Software managed TLBs like MIPS can do the same
- Do FIFO but skip accessed pages
- Keep pages in circular FIFO list
- Scan:
 - page's A bit = 1, set to 0 & skip
 - else if A = 0, evict
- A.k.a. second-chance replacement



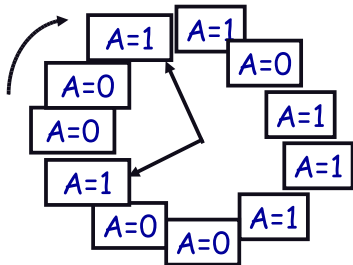
Clock alg. (continued)

- **Large memory may be a problem**

- Most pages referenced in long interval

- **Add a second clock hand**

- Two hands move in lockstep
- Leading hand clears A bits
- Trailing hand evicts pages with A=0



- **Can also take advantage of hardware Dirty bit**

- Each page can be (Unaccessed, Clean), (Unaccessed, Dirty), (Accessed, Clean), or (Accessed, Dirty)
- Consider clean pages for eviction before dirty

- **Or use n -bit accessed *count* instead just A bit**

- On sweep: $count = (A \ll (n - 1)) \mid (count \gg 1)$
- Evict page with lowest *count*

Other replacement algorithms

- **Random eviction**

- Dirt simple to implement
- Not overly horrible (avoids Belady & pathological cases)

- ***LFU* (least frequently used) eviction**

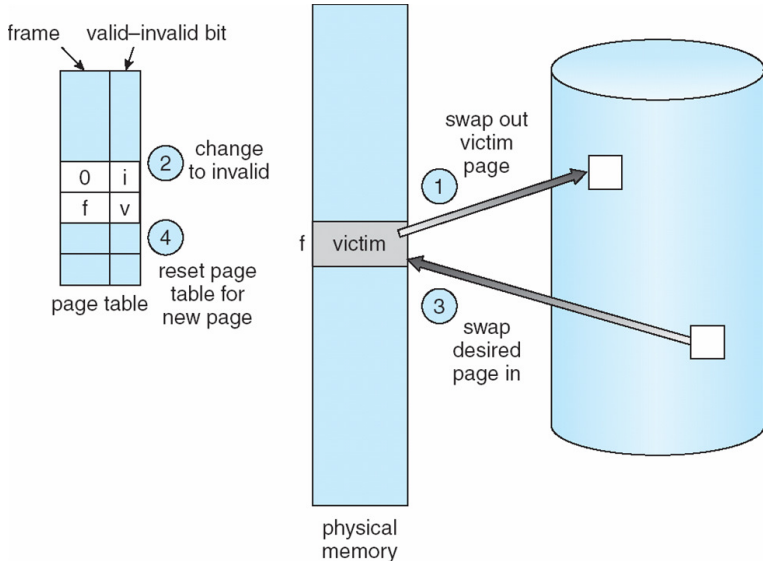
- Instead of just A bit, count # times each page accessed
- Least frequently accessed must not be very useful
(or maybe was just brought in and is about to be used)
- Decay usage counts over time (for pages that fall out of usage)

- ***MFU* (most frequently used) algorithm**

- Because page with the smallest count was probably just brought in and has yet to be used

- **Neither LFU nor MFU used very commonly**

Naïve paging



- Naïve page replacement: 2 disk I/Os per page fault

Page buffering

- **Idea: reduce # of I/Os on the critical path**
- **Keep pool of free page frames**
 - On fault, still select victim page to evict
 - But read fetched page into already free page
 - Can resume execution while writing out victim page
 - Then add victim page to free pool
- **Can also yank pages back from free pool**
 - Contains only clean pages, but may still have data
 - If page fault on page still in free pool, recycle

Page allocation

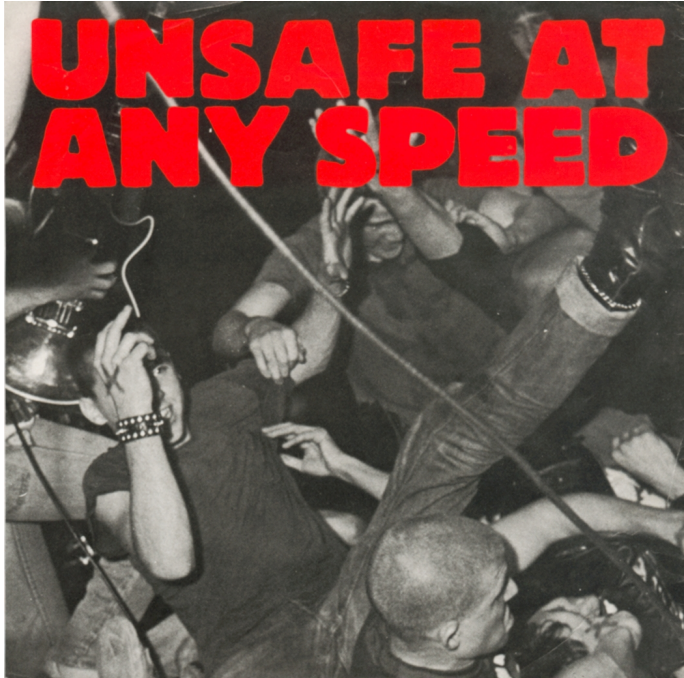
- Allocation can be *global* or *local*
- **Global allocation doesn't consider page ownership**
 - E.g., with LRU, evict least recently used page of any proc
 - Works well if P1 needs 20% of memory and P2 needs 70%:



- Doesn't protect you from memory pigs
(imagine P2 keeps looping through array that is size of mem)
- **Local allocation isolates processes (or users)**
 - Separately determine how much mem. each process should have
 - Then use LRU/clock/etc. to determine which pages to evict within each process

Outline

- ① Paging
- ② Eviction policies
- ③ Thrashing
- ④ Details of paging
- ⑤ The user-level perspective
- ⑥ Case study: 4.4 BSD

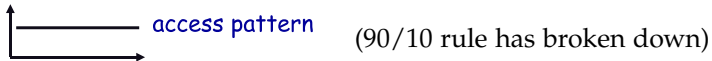


Thrashing

- **Processes require more memory than system has**
 - Each time one page is brought in, another page, whose contents will soon be referenced, is thrown out
 - Processes will spend all of their time blocked, waiting for pages to be fetched from disk
 - I/O devs at 100% utilization but system not getting much useful work done
- **What we wanted: virtual memory the size of disk with access time the speed of physical memory**
- **What we have: memory with access time of disk**

Reasons for thrashing

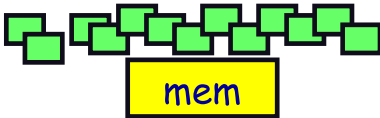
- Access pattern has no temporal locality (past \neq future)



- Hot memory (10% absorbing most accesses) does not fit

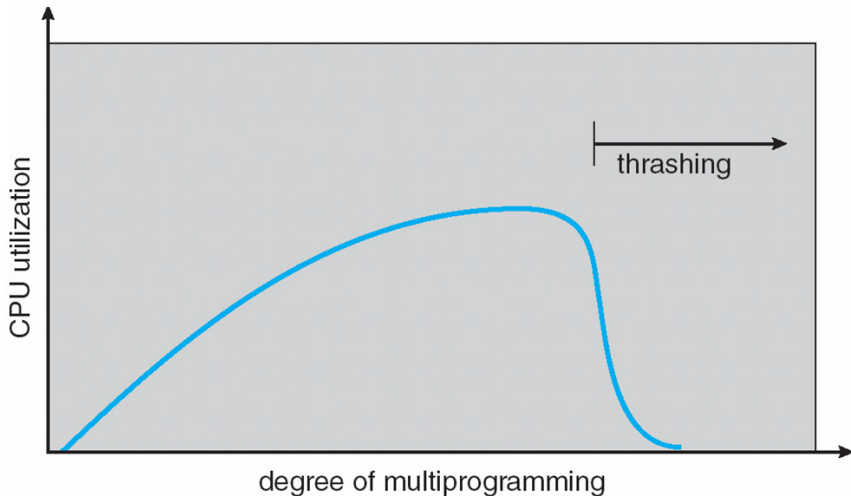


- Each process fits individually, but too many for system



- At least this case is possible to address

Multiprogramming & Thrashing



- **Must shed load when thrashing**

Dealing with thrashing

- **Approach 1: working set**

- Thrashing viewed from a caching perspective: given locality of reference, how big a cache does the process need?
- Or: how much memory does the process need in order to make reasonable progress (its working set)?
- Only run processes whose memory requirements can be satisfied

- **Approach 2: page fault frequency**

- Thrashing viewed as poor ratio of fetch to work
- $PFF = \text{page faults} / \text{instructions executed}$
- If PFF rises above threshold, process needs more memory.
Not enough memory on the system? Swap out.
- If PFF sinks below threshold, memory can be taken away

Calculating the working set

- **Working set: all pages process will access in next T time**
 - Can't calculate without predicting future
- **Approximate by assuming past predicts future**
 - So working set \approx pages accessed in last T time
- **Keep idle time for each page**
- **Periodically scan all resident pages in system**
 - A bit set? Clear it and clear the page's idle time
 - A bit clear? Add CPU consumed since last scan to idle time
 - Working set is pages with idle time $< T$

Two-level scheduler

- **Divide processes into *active* & *inactive***
 - Active – means working set resident in memory
 - Inactive – working set intentionally not loaded
- **Balance set: union of all active working sets**
 - Must keep balance set smaller than physical memory
- **Use long-term scheduler [recall from lecture 4]**
 - Moves procs active \rightarrow inactive until balance set small enough
 - Periodically allows inactive to become active
 - As working set changes, must update balance set
- **Complications**
 - How to choose idle time threshold T ?
 - How to pick processes for active set
 - How to count shared memory (e.g., libc.so)

Outline

- 1 Paging
- 2 Eviction policies
- 3 Thrashing
- 4 Details of paging
- 5 The user-level perspective
- 6 Case study: 4.4 BSD

Some complications of paging

- **What happens to available memory?**
 - Some physical memory tied up by kernel VM structures
- **What happens to user/kernel crossings?**
 - More crossings into kernel
 - Pointers in syscall arguments must be checked
(can't just kill process if page not present—might need to page in)
- **What happens to IPC?**
 - Must change hardware address space
 - Increases TLB misses
 - Context switch flushes TLB entirely on old x86 machines
(But not on MIPS... Why?)

Some complications of paging

- **What happens to available memory?**
 - Some physical memory tied up by kernel VM structures
- **What happens to user/kernel crossings?**
 - More crossings into kernel
 - Pointers in syscall arguments must be checked
(can't just kill process if page not present—might need to page in)
- **What happens to IPC?**
 - Must change hardware address space
 - Increases TLB misses
 - Context switch flushes TLB entirely on old x86 machines
(But not on MIPS... Why? MIPS tags TLB entries with PID)

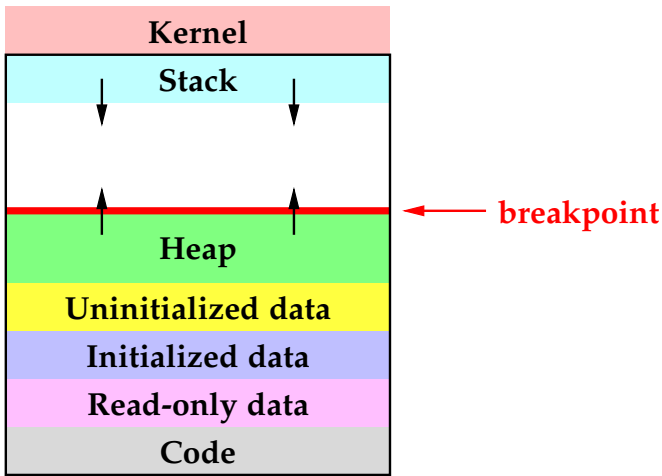
64-bit address spaces

- Recall x86-64 only has 48-bit virtual address space
- What if you want a 64-bit virtual address space?
 - Straight hierarchical page tables not efficient
 - But software TLBs (like MIPS) allow other possibilities
- **Solution 1: Hashed page tables**
 - Store Virtual \rightarrow Physical translations in hash table
 - Table size proportional to physical memory
 - Clustering makes this more efficient [Talluri]
- **Solution 2: Guarded page tables [Liedtke]**
 - Omit intermediary tables with only one entry
 - Add predicate in high level tables, stating the only virtual address range mapped underneath + # bits to skip

Outline

- 1 Paging
- 2 Eviction policies
- 3 Thrashing
- 4 Details of paging
- 5 The user-level perspective
- 6 Case study: 4.4 BSD

Recall typical virtual address space

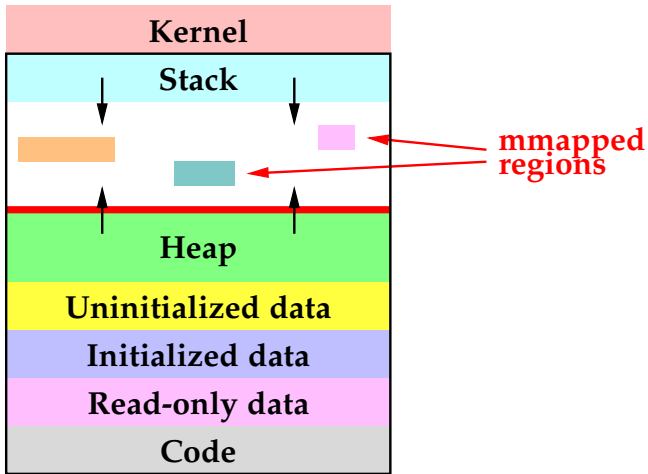


- Dynamically allocated memory goes in heap
- Top of heap called *breakpoint*
 - Addresses between breakpoint and stack all invalid

Early VM system calls

- **OS keeps “Breakpoint” – top of heap**
 - Memory regions between breakpoint & stack fault on access
- `char *brk (const char *addr);`
 - Set and return new value of breakpoint
- `char *sbrk (int incr);`
 - Increment value of the breakpoint & return old value
- **Can implement malloc in terms of sbrk**
 - But hard to “give back” physical memory to system

Memory mapped files



- Other memory objects between heap and stack

mmap system call

- `void *mmap (void *addr, size_t len, int prot, int flags, int fd, off_t offset)`
 - Map file specified by `fd` at virtual address `addr`
 - If `addr` is `NULL`, let kernel choose the address
- `prot` – **protection of region**
 - OR of `PROT_EXEC`, `PROT_READ`, `PROT_WRITE`, `PROT_NONE`
- `flags`
 - `MAP_ANON` – anonymous memory (`fd` should be `-1`)
 - `MAP_PRIVATE` – modifications are private
 - `MAP_SHARED` – modifications seen by everyone

More VM system calls

- `int msync(void *addr, size_t len, int flags);`
 - Flush changes of mmaped file to backing store
- `int munmap(void *addr, size_t len)`
 - Removes memory-mapped object
- `int mprotect(void *addr, size_t len, int prot)`
 - Changes protection on pages to or of PROT...
- `int mincore(void *addr, size_t len, char *vec)`
 - Returns in vec which pages present

Exposing page faults

```
struct sigaction {  
    union {                                /* signal handler */  
        void (*sa_handler)(int);  
        void (*sa_sigaction)(int, siginfo_t *, void *);  
    };  
    sigset_t sa_mask;    /* signal mask to apply */  
    int sa_flags;  
};  
  
int sigaction (int sig, const struct sigaction *act,  
               struct sigaction *oact)
```

- **Can specify function to run on SIGSEGV**
(Unix signal raised on invalid memory access)

Example: OpenBSD/i386 siginfo

```
struct sigcontext {  
    int sc_gs; int sc_fs; int sc_es; int sc_ds;  
    int sc_edi; int sc_esi; int sc_ebp; int sc_ebx;  
    int sc_edx; int sc_ecx; int sc_eax;  
  
    int sc_eip; int sc_cs; /* instruction pointer */  
    int sc_eflags;        /* condition codes, etc. */  
    int sc_esp; int sc_ss; /* stack pointer */  
  
    int sc_onstack;        /* sigstack state to restore */  
    int sc_mask;           /* signal mask to restore */  
  
    int sc_trapno;  
    int sc_err;  
};
```

VM tricks at user level

- **Combination of mprotect/sigaction very powerful**
 - Can use OS VM tricks in user-level programs [Appel]
 - E.g., fault, unprotect page, return from signal handler
- **Technique used in object-oriented databases**
 - Bring in objects on demand
 - Keep track of which objects may be dirty
 - Manage memory as a cache for much larger object DB
- **Other interesting applications**
 - Useful for some garbage collection algorithms
 - Snapshot processes (copy on write)

Outline

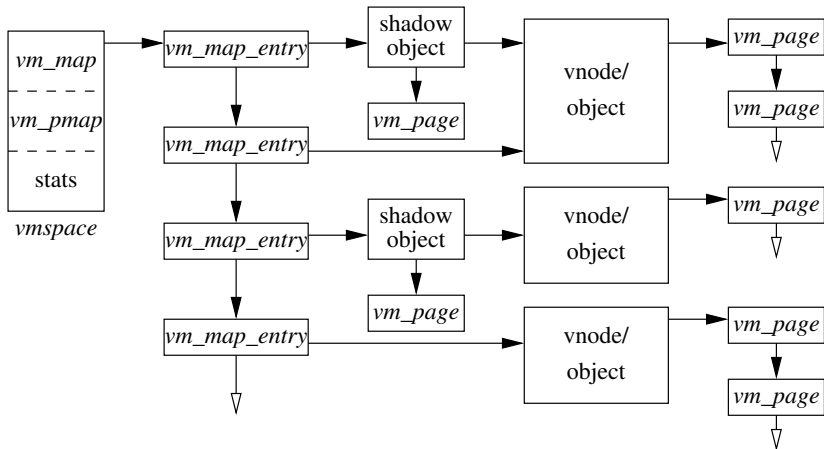
- ① Paging
- ② Eviction policies
- ③ Thrashing
- ④ Details of paging
- ⑤ The user-level perspective
- ⑥ Case study: 4.4 BSD

4.4 BSD VM system [McKusick]¹

- Each process has a *vm_space* structure containing
 - *vm_map* – machine-independent virtual address space
 - *vm_pmap* – machine-dependent data structures
 - statistics – e.g. for syscalls like *getrusage* ()
- *vm_map* is a linked list of *vm_map_entry* structs
 - *vm_map_entry* covers contiguous virtual memory
 - points to *vm_object* struct
- *vm_object* is source of data
 - e.g. vnode object for memory mapped file
 - points to list of *vm_page* structs (one per mapped page)
 - *shadow objects* point to other objects for copy on write

¹See library.stanford.edu for off-campus access

4.4 BSD VM data structures



Pmap (machine-dependent) layer

- **Pmap layer holds architecture-specific VM code**
- **VM layer invokes pmap layer**
 - On page faults to install mappings
 - To protect or unmap pages
 - To ask for dirty/accessed bits
- **Pmap layer is lazy and can discard mappings**
 - No need to notify VM layer
 - Process will fault and VM layer must reinstall mapping
- **Pmap handles restrictions imposed by cache**

Example uses

- ***vm_map_entry* structs for a process**
 - r/o text segment → file object
 - r/w data segment → shadow object → file object
 - r/w stack → anonymous object
- **New *vm_map_entry* objects after a fork:**
 - Share text segment directly (read-only)
 - Share data through two new shadow objects (must share pre-fork but not post-fork changes)
 - Share stack through two new shadow objects
- **Must discard/collapse superfluous shadows**
 - E.g., when child process exits

What happens on a fault?

- **Traverse *vm_map_entry* list to get appropriate entry**
 - No entry? Protection violation? Send process a SIGSEGV
- **Traverse list of [shadow] objects**
- **For each object, traverse *vm_page* structs**
- **Found a *vm_page* for this object?**
 - If first *vm_object* in chain, map page
 - If read fault, install page read only
 - Else if write fault, install copy of page
- **Else get page from object**
 - Page in from file, zero-fill new page, etc.