Memory Management

CIS 657

Memory Hierarchy

- Multiple layers
  - On-chip, off-chip cache
  - Main memory (RAM)
  - Secondary storage (disk, network)
  - Tertiary storage (long-term off-line storage, e.g. tape)
- Distance from CPU:
  - inversely related to speed
  - proportional to size
- Each level can be viewed as a cache for the next

Managing the Hierarchy

- How much of a process’s address space must be in RAM for the process to run?
- Where else should it be in the hierarchy, and how will we move portions of the address space between levels of the hierarchy?
- How can we optimize memory usage to give maximum performance for processes?
Processes and Memory

- Each process has a virtual address space
  - Independent of physical address space (usually larger)
  - Not all of a process’s virtual address space need be in main memory to run
- Virtual addresses translated to physical addresses by hardware
  - Relocatable code
  - Fast context switching (don’t have to move process to fixed physical addresses)
- Address space can be contiguous or segmented

Memory Management Unit (MMU)

- Hardware that handles
  - Address translation
  - Memory protection
- Defines the structure of the page tables
- Usually considered part of the CPU architecture (whether it’s actually on the CPU or a separate chip)

Pages and Frames

- Address spaces composed of fixed-sized pieces
  - Virtual pages
  - Physical frames
- Pages either resident or nonresident
- CPU presents virtual address to MMU, which checks residency and protection, then translates
- Reference to nonresident page causes a page fault—makes VM work
The Three Policies

- **Fetch**
  - When pages are loaded into memory
    - Pure demand paging: only when touched
    - Prefetch: use locality of reference to pull in extra pages
- **Placement**
  - Where in memory the page is placed (relevant because of cache effects; see Page Coloring)
- **Replacement**
  - Which pages are removed from main memory when there are no (or few) free frames

Page Replacement

- Most critical aspect of the paging system
  - Good choices minimize page faults
  - Poor choices induce thrashing
- Page reference string: sequence of pages referenced over a time interval
- Page Fault Rate (PFR): page faults per time interval
- Algorithm choice depends on expected application behavior, information available from the system, and ease of implementation.

Page Replacement II

- Global vs. local page replacement
  - Global: any page can be a victim
  - Local: only my pages can be victim
- Thrashing:
  - Excessive page traffic between main memory and backing store
  - Caused by choosing a victim page that will be needed “too soon” or because a process doesn’t have enough memory for its working set
Working Sets

- Based on concept of locality of reference
  - processes will use a certain set of pages for a period of time, then change to new set
- The \( k \) unique pages used in the last \( n \) references in the page reference string
- If we keep a process’s working set in main memory, it won’t page fault until it changes phases
- High and Low watermark techniques for approximating the working set

Page Replacement Algorithms

- Belady’s MIN
  - Guaranteed optimal
  - “Replace the page that will next be needed farthest in the future.”
- Least Recently Used (LRU) [MRU]
  - “Replace the page that was last needed farthest in the past.”
- Least Frequently Used (LFU) [MFU]
  - “Replace the page that has been needed least over the last \( n \) page references.”
- Approximations to the above (e.g. clock)

Swapping

- Swapping moves an entire process between main memory and backing store
- Used to manage the page fault rate
  - if PFR is high for a single process, either the replacement algorithm is a poor match for its behavior or its working set doesn’t fit in its available memory
  - If the PFR is high for all processes, we need to decrease the degree of multiprogramming
Hardware Support for VM

- MMU (must have)
  - Address translation
  - Protection against users changing address mappings (and other processes)
  - Distinguish resident from non-resident pages
- Restartable or interruptible instructions
- Statistics gathering support is helpful, but not strictly necessary.

Conventional Address Space Layout

- Top of address space only accessible to kernel (up to \( \frac{1}{4} \) of the space on the old VAX)
- Keep first page empty (0-filled). Why?
- Stack grows down
- Heap grows up
- Why doesn’t the kernel have a general stack?

5.2 FreeBSD Memory Management Data Structures
VM Data Structures

- Vmspace holds all the data structures
- Vm_map: machine-independent representation
- Vm_map_entry: describes contiguous range of virtual addresses with same protection
- Vm_object pointers are the objects that are mapped into the address ranges
  - Shadow objects are copies of the original data
- Vm_page structures represent the physical memory cache of the object

Kernel Memory Management

- Where should we map the kernel?
  - Answer 1: Map it into the top of every process’s address space
    - Changing processes doesn’t affect the kernel mapping
    - Neither kernel nor user processes have the entire address space to themselves
  - Answer 2: Give the kernel its own address space and switch back and forth between kernel and processes
    - Makes copying data MUCH more expensive (1/3 of kernel time is copying, even using efficient instructions)

Two Kernel Mapping Possibilities
Kernel Maps and Submaps

- FreeBSD maps the kernel into each process’s address space
- First thing the kernel does is set up address maps
- Submaps
  - kernel-only constructs
  - used to reserve contiguous blocks (e.g. mbufs)
  - specific addresses
  - specific alignments

Kernel Address-Space Maps

Kernel Address Space Allocation

- Allocation calls take address map (may be submap) and size – no explicit location
- Page-aligned, page-rounded
- Non-pageable (wired) and pageable ranges
  - Use wired pages when we can’t take a page fault
Kernel Address Space
Allocation II

- Allocating wired memory
  - `kmem_malloc()`
    - Called by `malloc()`
    - Has non-blocking option
  - `kmem_alloc()`
    - Returns 0-filled memory
    - May block if insufficient physical memory avail.

Kernel Address Space
Allocation III

- Allocating pageable space
  - FreeBSD: used only for arguments to `exec` and for kernel stacks of swapped out processes
  - `kmem_alloc_pageable()`
    - Returns error if insufficient address space is available
  - `kmem_alloc_wait()`
    - Blocks until space is available

Freeing Kernel Memory

- `kmem_free()`
  - Deallocates kernel wired memory and pageable memory from `kmem_alloc_pageable()`
- `kmem_free_wakeup()`
  - Deallocates memory and wakes up any process waiting for address space in the specified map
Kernel Malloc() and Free()

- The previous routines are actually for special cases
- malloc()/free() are the general allocator and deallocator
  - Nonpageable memory
  - No restrictions on alignment or size
  - Can allocate memory at interrupt time
  - Interface just like the C library

Why use malloc()?

- In a user program, we’d probably just throw a buffer on the stack
- Risky practice in the kernel – why?
- Allocate temporary variable space in the heap (and use pointers on the stack)
- Also allocate memory for persistent objects

Homework Question 
(due Monday, 22 October)

- Which kernel memory routine(s) would/could I use for:
  - Allocating memory in an interrupt handler?
  - Allocating memory at a fixed address?
  - Freeing memory when a process might be blocked waiting for address space?
  - Allocating memory at an arbitrary address?
  - Allocating memory of a fixed size?
  - Allocating aligned memory?
Memory Utilization

\[
\text{utilization} = \frac{\text{requested}}{\text{required}}
\]

- Requested is the total of requests made
- Required is the total allocated
- Utilization \( \leq 1 \) (.5 is considered good)
  - Why =?
  - Why <?
  - Why is high utilization critical in the kernel?

Kernel Memory Allocation Efficiency

- “Speed wins.” – Chris Kantjeriev
  - Frequent allocation: kernel must allocate things that are on stacks in user space
  - Psychological damage: if it’s perceived as slow, programmers will roll their own
- Hybrid strategy
  - Manage small, fixed-sized blocks through one interface
  - Call allocator on large requests

4.4BSD Small Memory-Block Allocator

- Keep power-of-2-sorted list of small blocks; quickly check the list on small requests
- Why not just use power of 2 strategy?
- Large vs. small threshold: 2 pages
- Round large allocations to next page size
Kernel Memory Allocation Efficiency

- When allocating a new PO2 block, allocate a whole page and divide it into those blocks.
- Problem: how do we know how big the block is when free() is called?
  - User-level solution: store block size just before block
  - E.g., on free(p):

Power of 2 Allocation

- Will this work for the PO2 scheme?
- Will it be efficient?
- 4.4 BSD stores the block size allocated within a page in a separate table

Problems with the Po2 Approach

- Commonly allocated structures waste space
  - Process, thread, vnode, control-block structures
  - E.g., process structure is 550 bytes; rounded up to 1024 bytes.
- Structures contain list pointers
  - Same offset from beginning of page
  - Walking the list causes cache misses
- Structures contain items that require initialization
  - Locks, lists
  - If allocated as a pool, init can be done once
FreeBSD 5.2 Zone Allocator

- Group together like items (size, type)
- Allocate a pool of memory (a zone) dedicated to those items
- Not used to hold anything else
- Has disadvantage of segregating memory (decreases utilization)
  - Benefits perceived to outweigh disadvantage

Per-Process Resources

Initial Memory Map
Initial Process Memory Map

- First `vm_map_entry` is for read-only program text
- Second `vm_map_entry` is for initialized data (copy-on-write)
- Third `vm_map_entry` is for uninitialized data
  - Anonymous object
- Fourth `vm_map_entry` is for stack.
  - Anonymous object

Anonymous Objects

- 0-filled pages on allocation
- Store modified pages in swap area if memory space is tight
- Abandoned when no longer needed
- More on these object types later

Page Fault Handling

- Page-fault handler is given virtual address that caused fault, and type of fault (r/w)
  - Find the vmspace structure for the faulting process, and access the head of the `vm_map_entry` list.
  - Walk the `vm_map_entry` list, checking whether the faulting address appears within the start/stop
    - If we reach the end without finding it, this is an invalid memory reference, so seg fault the process
Page Fault Handling II

- Translate from the VM address to an offset within the object
  - \(\text{object_offset} = \text{fault_address} - \text{vm_map_entry->start_address} + \text{vm_map_entry->object_offset}\)
- Present the absolute object offset to the object
  - Object allocates a \(\text{vm_page}\) structure
  - Uses pager to fill the page
  - Object returns pointer to \(\text{vm_page}\) structure, which is added into the map
  - Return and execute faulting instruction

Mapping to Objects

- Objects hold information about a file or area of anonymous memory
  - Text, data map to file
  - Stack, heap map to anonymous memory
- Object, not the process(es) that map it, is responsible for maintaining map into physical memory
- Why?

Contents of an Object

- List of currently resident pages
- Reference count
- Size of file/anonymous area described by the object
- Number of pages in memory
- Pointers to shadow objects
- Pointer to the pager for the object
Three Types of Objects

- Named objects
  - Represent files
  - Memory-mapped devices (frame buff)
- Anonymous objects
  - Zero-filled on use
  - Impermanent: abandoned when no longer needed
- Shadow objects
  - Private copies of pages that have been modified
  - Abandoned when no longer referenced

The Fourth Object in 4.4BSD

- Copy objects
  - Hold old pages of files modified after private mapping
  - Abandoned when private mapping abandoned
- Deemed too complex, so removed from FreeBSD
- Processes must manually reproduce this effect (more under Private Snapshots)

Object Pagers

- Pager must handle page faults
- Device pagers
  - Only page-in
- vnode pager
  - For files in the file system
  - Page to file
  - Writes may require shadow object
- swap pager
  - Handles anonymous, shadow objects
  - Page out to swap area
  - Anonymous: zero-fill on first page-in
  - Shadow: copy existing page on first page-in
  - Later page-in requests read from swap area
### Vnode Caching
- Objects are reclaimed when their reference counts drop to 0
  - Anonymous objects: freed
  - File-based objects: LRU cache of vnode
- Vnode object only release when vnode is reclaimed for another file
- Object associated with unreclaimed vnode may still have pages
  - Can save time on page fault if vnode is reactivated before pageout

### Objects and Pages
- System describes pages of (non-kernel) memory with vm_page structures
- Pages allocated to objects on page faults
- Pages belong to at most 1 object at a time (shared mappings share objects)

### Shared Memory (mmap)
- Files are the basis for shared memory
- mmap call causes vm_map_entry from multiple processes to point to the same object
  ```c
  caddr_t addr = mmap(...
```
More Sharing Calls

- `munmap(address, length)`
  - Removes a shared mapping
  - Doesn’t have to exactly match an earlier `mmap()` call
- `Mprotect(address, length, protection)`
  - Control RWX on memory regions
  - Used by debuggers

Sharing Calls III

- `mlock(address, length)`
  - Process wires portions of address space
  - Resource limits on how many pages/proc
- `munlock(address, length)`
  - Make address range pageable again
- `msync(address, length)`
  - Ensure pages written to backing store
  - No effect on anonymous objects

Private and Shared Mappings

- Shared mappings point to the real file object
  ![Diagram of shared mapping]
- Private mappings interpose a shadow object:
  ![Diagram of private mapping]
Shadow Chains

Proc A (parent)

vm_map_entry
shadow object 2
vm_page 0
(Mod by A before fork)

Proc B (child)

vm_map_entry
shadow object 3
vm_page 0
(Mod by B after fork)

vm_page 1
(Mod by A after fork)

(file object)

Shadow Chains:
A Takes a Private Mapping

Proc A (parent)

vm_map_entry
shadow object
(file object)

vm_page 0
vm_page 1
(unmod)

Shadow Chains:
A Modifies Page 0

Proc A (parent)

vm_map_entry
shadow object
(file object)

vm_page 0
vm_page 1
(Mod by A)

(unmod)
Shadow Chains:
A Forks B

Proc A (parent) → \(\text{vm\_map\_entry}\) → \(\text{shadow\ object\ 2}\) → \(\text{shadow\ object\ 1}\) → \(\text{file\ object}\) → \(\text{vm\_page\ 0}\) → \(\text{vm\_page\ 1}\) (unmod)

Proc B (child) → \(\text{vm\_map\_entry}\) → \(\text{shadow\ object\ 3}\) → \(\text{shadow\ object\ 1}\) → \(\text{file\ object}\) → \(\text{vm\_page\ 0}\) (Mod by A before fork)

Shadow Chains:
A Modifies Page 1

Proc A (parent) → \(\text{vm\_map\_entry}\) → \(\text{shadow\ object\ 2}\) → \(\text{shadow\ object\ 1}\) → \(\text{file\ object}\) → \(\text{vm\_page\ 0}\) → \(\text{vm\_page\ 1}\) (unmod)

Proc B (child) → \(\text{vm\_map\_entry}\) → \(\text{shadow\ object\ 3}\) → \(\text{shadow\ object\ 1}\) → \(\text{file\ object}\) → \(\text{vm\_page\ 0}\) (Mod by A before fork)

Shadow Chains:
B Modifies Page 0

Proc A (parent) → \(\text{vm\_map\_entry}\) → \(\text{shadow\ object\ 2}\) → \(\text{shadow\ object\ 1}\) → \(\text{file\ object}\) → \(\text{vm\_page\ 0}\) (Mod by A after fork)

Proc B (child) → \(\text{vm\_map\_entry}\) → \(\text{shadow\ object\ 2}\) → \(\text{shadow\ object\ 1}\) → \(\text{file\ object}\) → \(\text{vm\_page\ 0}\) (Mod by B after fork)
Shadow Chains: A Exits

Key observation: objects 3 and 1 form a chain with no intervening references.

Proc B (child)

Shadow Chains: The Shadow Chain Collapses

Private Snapshots (NOT in FreeBSD 5.2)

- Use copy objects to capture an entire file so that we don’t see others’ modifications
Creating a New Process

- Reserve virtual address space for child
- Allocate and fill in process entry/thread struct
- Copy parent's pgroup, credentials, file descriptors, limits, and signal actions
- Allocate a new user area, and copy parent's
- Allocate a vmspace structure
- Copy parent vm_map_entry structures marked copy-on-write (duplicate address space)
- Arrange for child process to return 0.

Reserving Virtual Address Space

- The system can only hold a subset of all its processes' address spaces
  - Typical process address space: 4 Gig
  - Typical RAM in system: <= 4 Gig
  - Typical disk drive: 120-200 Gig
- Limited amount of space to hold pages
  - In RAM
  - Swap area
  - Files for mapped named objects
- Kernel must not oversubscribe available memory

The Danger of Oversubscribed VM

- A process makes a request for VM (e.g. sbrk() or mmap())
- Kernel says yes (oversubscription)
- Page fault
  - The kernel cannot allocate a free frame (no where to put victim page)
- It must then signal the process about the failure—hard to deal with this model
- Better is to not oversubscribe and return an error code from sbrk()/mmap().
Problem With This

- Some processes sparsely use a large address space
- The “better” model forces them to ask for all of it, and the kernel assumes that they will use all of it
- Some of these large processes will never even be created
- Solution: allow large processes to specify that they’ll take asynchronous signals
- After receiving signal, process must munmap

FreeBSD’s Solution

- 5.2 FreeBSD allows the total allocated VM to exceed the resources to manage it
  - Most procs use < 1/2 their VM
- If things go bad, kill a process
  - Book says “favor processes with large VM”
    - Does it mean “favor to kill” or “favor to not kill?”
    - Look through the code to answer

Duplication of User Address Space

- Create new process structure to hold copy of parent’s
- Lock parent against swapping (we need to copy that info)
- Put child in NEW state (ignored by scheduler)
- Copy vm_map_entry data structures and page tables (see next slide)
- Reset child process state put on run queue
### Copying vm_map_entry list

- For read-only region, just copy the entry
- For privately mapped regions, make a copy, marking both as copy-on-write and turning off write rights in page tables
  - Page fault handler notes c-o-w and makes a copy of the page (shadow object), resets write rights, and lets process continue.
- For shared regions, use shared mappings.

### File Execution

- First allocate new VM space before releasing old
  - Need to be able to return from exec() in case of bad arguments or other error
- Allocate a new vm_space structure with 4 vm_map_entry structures

### File Execution II

- Copy-on-write, fill-from-file map for the text segment
  - C-o-w allows debugging (read-only wouldn’t)
- Private (copy-on-write) fill-from-file entry maps initialized data
- Anonymous zero-fill-on-demand entry for uninitialized data
- Anonymous zero-fill-on-demand for stack
**Process Changing its Address Space—sbrk()**

- Round size up to multiple of page size
- Check if request would exceed limit on segment size
- Verify that VM resources are available
- Verify that address space immediately following segment is unmapped
- Expand existing `vm_map_entry` if not yet swapped; else add new entry.

**File Mapping**

- `mmap()` requests that a file be mapped into an address space
  - Leave it up to kernel where to put it, or
  - Particular address
    - Kernel checks if not in use
    - If in use, kernel does an `munmap()` first

**Five Possibilities for Overlap**

- Direct: exact match
- Subset: new mapping is entirely contained within old mapping
- Superset: new mapping entirely contains old mapping
- Extend Past: new mapping starts part way into and extends past old mapping
- Extend Into: new mapping starts before and extends into old mapping
Five Possibilities II

![Diagram showing different possibility combinations]

Changing Protection

- The mprotect() system call changes protections of a region
- Smallest granularity: single page
- Can set a region for read, write, execute permissions (as supported by underlying architecture)
- Compare new rights to maximum allowed by underlying object
- Create new vm_map_entry structure(s) to describe new protection(s)

Process Exit

- First step: free user portions of address space
  - Done in exit()
- Second step: Free user area
  - An “inverse bootstrapping” problem—we need to use a portion of the address space until the process goes away
  - Done in wait()
Freeing User Address Space

- Walk list of `vm_map_entry` structures
  - Walk list of shadow and copy objects
    - If last reference, release swap space, call machine-dependent routines to unmap and free object resources
    - Call machine-dependent routines to free map resources
  - If last reference to underlying object, it might be freed (e.g. anonymous object); named objects saved in object cache (LRU)

Pagers

- Move pages between RAM and backing store
- All requests in multiples of the software page size
- Pagers are internal to the kernel (unlike Mach), just called as functions
- `vm_page` structures passed as descriptors to pagers
- `(*pagertab[object->type]->pgo_putpages)(object, vmpage, count, flags, rtvals);`

Pager Instances

- Per-object pager structure
  - Pointers to read/write routines
  - Reference to storage for this instance
  - Created at time of mapping into address space
  - Exists until object is deallocated
  - Page fault handler allocates `vm_page` structure, calculates offset of faulting address in object, records information in pager structure, and calls pager routine
Page Faults

- Traps to the system caused by hardware detecting an invalid memory reference
  - Page might not be in memory
  - Page might not be allocated yet (but does appear in the map)
  - Page might have the wrong permissions (e.g. write right)

Causes of Page Faults

- Program text or initialized data referenced for the first time (demand paging)
- Anonymous areas (e.g., uninitialized data) referenced for the first time
- Previously resident pages that have been swapped out

vm_fault()

- Routine that services all page faults
- Provided virtual address that caused fault
- Walks list of vm_map_entry structures looking for one holding the address
- Traverses list of objects (shadow, copy, etc.) to find or create the needed page
- Call machine-dependent layer to validate the faulted page
A

- Loop that walks the list of shadow, copy, anonymous, and file objects to find the page
- If it reaches the final object without finding the page, it requests that the final object produce the page

B

- An object has been found that has the page
- If the page is busy, which might happen if another process is faulting it in or it’s being paged out…
  - we must block.
  - After being awakened, restart the fault handler.
- If the page was not busy, break out of the loop (go to G)

C

- Check whether there is a pager allocated for the object. If so, allocate a page.
  - Named objects already have pagers
  - Anonymous objects get pagers the first time they write a page to backing store.
- We have the “first object” check here to avoid a race condition when two processes have faulted at the same time.
  - The first object through will create the page
  - The second one will block on it in B.
D & E

- D: Check to see if other nearby pages can be brought in at the same time
  - If the page already exists in the file or swap, have the pager bring it in.
    - If I/O error, abort
    - If pagein succeeds, break
    - If pagein fails, then free the page unless we’re the first object (remember the test in part C?)
- E: Remember that we created a page in the first object (we’ll need it later, see J)

F

- If we still haven’t found the page, and we’ve checked all the objects, then this must be an anonymous object chain
  - Why (answer this yourself for the future)?
- Zero fill the page
- Set first_page to NULL because we won’t be freeing it later
- Exit the loop

G

- At this point the page has been found or (allocated and initialized)
  - Either we paged it in on a pager
  - Or we hit the end of the loop and allocated a zero-filled page from an anonymous object
- “object” is the owner of the page
- Page has the correct data
If the object that gave us the page isn’t the first object, then the first object is a shadow object.

If it’s a write fault, we need to update mappings and make a copy of the page so that the page will be found in the shadow object next time.

If it’s a read fault, mark it copy-on-write.

---

If this was a write fault, we’ve handled it, so turn off C-O-W.

Mark any clustered pages brought in as read-only (defer analysis to later page fault).

Release page and (possibly) first_page, giving blocked processes chance to run.

---

Approximates global LAU

- Global: pages belonging to any process may be a victim
  - Single pool of pages competed for by all processes
  - Other systems (e.g. VMS) divide memory pages into multiple independent areas and have a group of processes use one area

LAU: Least Actively Used
Kernel Page Lists

- Wired: pages that may not be paged out
  - Pages used by the kernel
  - Pages of user areas of loaded processes
  - I/O pages
- Active: used by at least one region of virtual memory
  - If paged out, will probably be needed soon
  - Could lead to thrashing

Kernel Page Lists II

- Inactive: contents are known, usually not being used by any active memory region
  - When the system is short of memory, active pages are moved to the inactive list
  - Dirty pages; clean ones moved to Cache list
- Cache: like Inactive pages, but clean
- Free: frames with no useful contents
  - Used to fulfill new page fault request
- The kernel checks periodically and runs the pageout daemon to keep the free list above a minimum threshold. Why do that?

Paging Parameters

- vm_page_alloc() awakens the pageout daemon when more memory is needed (pagedaemon == pageout daemon)
- The page daemon examines parameters, which change over time as the system runs:
  - Free: 0.7% min, 3% max
  - Cache: 3% min, 6% max
  - Inactive: 0% min, 4.5% max
Pageout Daemon

- Handles page replacement
- Must write dirty pages when reclaimed
  - It must use normal kernel synchronization, such as sleep
  - So we run it as a separate process, with user/process structures and stack
  - It never leaves the kernel
  - Uses asynchronous disk I/O to continue scanning while writing. Why?

Page Usage Counts

- Initial value of 3 when brought in
- Each time pagedaemon scans a page, check reference bit (look for PG_REFERENCED in the kernel)
  - If set, increment count (max 64)
  - If not set, decrement count
  - If count is 0, move from Active list to Inactive list

Vm_pageout_scan

- Calculate number of pages that need to be moved between lists
- Scan the inactive list until the desired number of pages are moved
- Decide where to put pages
  - Pages in use go back to Active
  - Pages with no object go to Free
  - Clean pages go to Cache
Vm_pageout_scan()

- If the page is dirty, and it's the first time we've seen it
  - Note we've seen it
  - Move to back of inactive list (second chance)
- Calculate how many pages to move from Active -> Inactive

Swapping Out

- The vmdaemon process (proc 3) handles this, see vm_daemon()
- High overhead—only do in case of serious resource shortage
  - Pagedaemon can't free memory fast enough to keep up—THRASHING
- Look for inactive processes that have been asleep a long time, then a short time
  - Processes have slept for more than swap_idle_threshold2 (default 10) seconds
  - Then processes have slept more than swap_idle_threshold1 (default 2) seconds
Swapping Out II

- Clear P_INMEM flag to show process is not in memory
- Set PS_SWAPPINGOUT flag
- Mark the user area as pageable (includes the kernel stack of threads)

4.4BSD vs. FreeBSD 5.2

- 4.4BSD used to swap the longest-resident running process as a last resort
- FreeBSD 5.2 won’t do this
  - Risks deadlock of system if runnable procs won’t fit in memory
  - Believed this won’t occur with multi-gig RAM machines, so overhead is eliminated by disallowing this case

Swapping In

- Handled by the swapper (proc 0), see the scheduler() routine.
- One of three states:
  - Idle: no swapped-out processes are ready to run. This is the expected case.
  - Swapping In: at least one runnable process is swapped out
  - Swapping out: scheduler() wakes up the pagedaemon to make space
Swapping In

- Allocate memory to hold the user structure and kernel stack for threads
- Read them from swap space
- Mark process as resident and return to run queue if runnable (not stopped or sleeping).

Swapped Process Selection

- When there are multiple processes that could be swapped in, the swapper must choose which to do first
  - Time swapped out
  - Its nice value
  - Amount of sleep time since it last ran

Portability

- Thus far, we've focused on the machine-independent portions of the VM subsystem
- We still haven't talked about page tables
- The machine-dependent part of the VM subsystem controls the MMU
- MMU does address translation and enforces access control
Page Tables

- **Forward-mapped**
  - One entry per virtual page
  - Structure
    - Simple array of entries
    - Hierarchical mappings (larger address spaces)
- **Inverse-mapped**
  - One entry per physical frame
- **Translation Look-Aside Buffer (TLB)**
  - Fast cache of mappings

Cache

- **On-chip, L1, L2 cache**
- Holds data recently loaded from memory (a “cache line”)
- Helps to close the gap between CPU speed and main memory speed
- Might need to be managed by the kernel for best effect

Cache Attributes

- **Virtual vs. Physical addressing**
  - On which address is lookup done?
  - **Physical**
    - No conflict between distinct processes’ address spaces
    - Need to map through MMU to find address
  - **Virtual**
    - No MMU mapping needed
    - How to distinguish Process A’s virtual address from process B’s?
### Cache Attributes II

- **Tagged addresses**
  - Associate an identifier (usually a small number, say < 16) with the cache entry
  - Kernel must manage these

- **Untagged addresses:**
  - No way for the MMU to tell which process caused a cache line to be loaded
  - Must flush cache on every context switch
  - VERY expensive

---

### Cache Attributes III

- **Write-through vs. write-back**
  - Write-through cache goes directly to memory
  - Write-back cache delays write for a while
  - Tradeoffs?

---

### The pmap Module

- The `pmap` module manages the machine-dependent translation and access tables
  - E.g., the page tables
- Interface is in machine-independent, page-aligned addresses and in machine-independent protections
  - Maps these onto the underlying system
  - E.g., MI page size can be a multiple of actual frame size
  - Protection maps from rwx to whatever the hardware supports.