Fork - Creation of a New Process

- Reserve virtual address space for the child process
  - → Total virtual memory = physical memory available for page frames + total swap space
- Allocate process entry and a thread structure for the child and copy data from parent.
- Allocate a user structure and kernel stack, copying from parent to initialize them.
- Allocate a *vmspace* structure
- Create copies of the parent *vm_map_entry* structures:
  - → If it is read-only, use a copy of the parents vm_map_entry
  - → If the region is privately mapped, mark it COW in both child and parent
- Arrange for the child process to return 0, and for the parent to return the PID of the child.

Fork - Comments

The kernel should ensure that it do not promise to provide more virtual memory than it can deliver.

- A process should get an error from a system call (such as fork or mmap) if there is not enough virtual memory available.
- If the kernel promises more virtual memory than it can support, it can run into deadlock when trying to service a page fault.
  - → The problem arises when it has no free page frame and no swap space available to save an active page. Here, the kernel will have no other choice but to kill the process unfortunate enough to be page faulting. Unacceptable!

Exec - Execution of a File

- Validate that the file is executable
- Copy the arguments to a temporary area in the kernel
- Reserve virtual memory for the new areas
- Release the current code and memory areas
- Allocate a new *vmspace* structure *and four* *vm_map_entry* structures
  - → COW, fill-from-file entry for the code segment
    - ∗ COW (instead of RO) for debugging
  - → Private COW, fill-from-file entry for initialized data (.data)
  - → Anonymous zero-fill-on-demand entry for uninitialized data (.bss)
  - → Anonymous zero-fill-on-demand entry for stack
- Each *vm_map_entry* filled in with a pointer to an existing or new object.

Exec - How to find out if the program file is already in use by another process?

- If the file is already used by another process, the code segment should be shared with that process.
- The same program means that it is loaded from the same file, making the file *inode* a unique identifier of the program.
- When the file is opened, its path name is translated to an inode.
- If the file is used by another program, its inode is already present in the inode cache.
- All *cached inodes* point to a related *vnode*.
- If the vnode represents background memory for an object, it points to the object structure for the object.

Change of Process Size

- A process can increase the size of its .bss area (malloc area) by using the system call *sbrk*.
  - → Verify that virtual memory resources are available.
  - → Verify that the requested virtual addresses are free.
  - → If possible, increase the ending address of an already existing *vm_map_entry.*
    - ∗ If the swap object used by the *vm_map_entry* has more than one reference, a new vm_map_entry has to be allocated.

Mmap - Implementation

Using *mmap()* a specific virtual address (VA) can be given.

- If the new VA interval overlaps an existing mapping, the old mapping is deallocated.
- Create a new *vm_map_entry* structure to describe the region.
- Set *vm_map_entry* to reference an existing object, or create a new object.

Exit - Termination of a Process

When a process calls *exit*, its virtual memory resources must be freed both in main memory and on the swap area:

- Traverse the *vm_map_entry* list
  - → If the last reference to a shadow object, throw away
  - → If the last reference to an anonymous object, throw away.
  - → If the last reference to a file object , save the object in the vnode cache (sorted in LRU order)
- With all its resources free, the process becomes a zombie and wakes its parent.
- The parent collects the exit status with a *wait* system call.
  - → Process structure, user structure and kernel stack are returned to the zone allocator.

Pager Interface

- The *pager interface* provides the mechanism for moving data between the backing store and physical memory.
- Each object has an associated pager that is called to perform data transports.
- The pager is identified by its type.
- Each pager registers a set of functions that define its operations.
- Some information needed by the pager is stored in the *object* structure and information specific to a page is stored in the *vm_page* structure.

At page fault if the page is not resident:

- Allocate a *vm_page* structure
- Record the offset within the object and add the *vm_page* structure to the object's list of vm_page structures.
- Call the pager to fill the *page frame* with data.

Pager Interface, cont.

The pager is also used by the *pageout daemon* to save modified pages.

Seven operations exist:

pgo_init() Called at boot to initialize the pager.

pgo_alloc(handle,...) Create an “instance” of the pager. Handle is a pointer to a vnode for the vnode pager.

pgo_dealloc(object) Called when the objects reference counter has become zero to deallocate the pager.

pgo_getpages(object,marray[],...) Called by the *pagefault handler* to fill one or more page frames with data. marray is an array of pointers to *vm_page* structures.

pgo_putpages(object,marray[],...) Called by the *pageout daemon* to rewrite one or more pages to the backing store.

pgo_haspage() Check whether backing store has a page.

pgo_pageunswapped() Remove a page from the backing store (swap pager only).

Vnode Pager

The vnode pager handles objects that map files in the filesystem.

When a file is opened by open, mmap or exec, the *pgo_alloc()* routine is called.

- If the file is not already mapped, an object structure is allocated. The *vnode* and *object* structures point to each other.

Pagein/pageout is performed by the pgo_getpages()/pgo_putpages() routines.

The I/O is done using a physical-I/O buffer:

- Map the pages into the kernel address space.
- Call the device-driver strategy routine to read/write the pages.
- Unmap the pages from the kernel address space.

Device Pager

- The device pager handles memory mapped hardware devices like for example a frame buffer for a terminal.
- The device pager do not fill the memory pages with data.
- The alloc routine allocates an object to handle the address interval.
- The first reference to a device page will cause a page fault. A *vm_page* structure is allocated in the objects private memory and marked as *fictitious*. The page frame pointer will point to a physical address in the device memory.
- The device pager *pgo_putpages()* routine may never be called (panic).

Swap Pager

The swap pager is used by anonymous objects and shadow objects for temporary storage of modified pages.

- The term *swap pager* refers to two different pagers; from the beginning the default *default pager* is used.
- The *default pager* provides no backing storage.
- Anonymous areas are filled with zeros by the page fault handler at the first page fault.
- The first time the default pager's *pgo_putpages()* routine is called, the *default pager* replaces itself with the *swap pager*.

The swap pager must be able to do non-blocking disk operations because it is called from the pageout daemon, which is not allowed to block.

Swap Pager, cont.

For performance reasons, the swap pager use a specially formated swap area on the disk.

- Swap space is allocated when needed in blocks with space for 32 contiguous pages.
- To locate a page in the swap area, a global hash table is used.
- Free space in the swap area is managed by a bitmap with one bit for each page-sized block of swap space.

  → The bitmap is organized as a radix-tree.

Swap Pager, cont.

- When a pageout operation is needed, the swap pager *pgo_putpages()* routine is called.
- It allocates a *buf structure* and maps the page frames to be written into the buffer.
- Because the swap pager may not wait, it marks the buffer with a callback to the routine *swp_pager_async_iodone()*.
- When the write operation completes, the interrupt handler calls *swp_pager_async_iodone().*

  → Each written page is marked as clean and *vm_page_io_finish()* is called to notify the pager.

  → The swap pager unmappes the pages from the *buf structure* and releases it.

Swap Pager, cont.

Unfortunately, a bug has crept into the design that makes it possible for the *pageout daemon* to block, despite the fact that this is not allowed.

The mechanism is as follows:

- Because the number of swap buffers is constant, a limit is set on the number of buffers the swap pager may use.
- Once this limit is reached, the *pgo_putpages()* routine blocks until one of its outstanding writes completes.
- This will block the pageout daemon that called *pgo_putpages()*.
- Under unlucky circumstances, this may deadlock the system.

Page Fault Handler

```
vm_fault(map, addr, type) {
   lookup addr in map, returning object/offset/prot
   Loop down object chain looking for page {
     lookup page at object/offset
      if page found in memory {
         if (busy) { block until avail; retry }
         else { mark page busy; break; }
      } else if object has pager {
         allocate a page_frame;
         call pager to fill page_frame;
       if pager has page { break; }
      }
      if (no next object) {  //Must be in anonymous area
         allocate a page_frame in first object;
         fill page_frame with zeros;
         break;
      } else { continue };
   }
   if (not first object in chain) {
      if (WRITE fault)
         copy page to first object; // COW
      else if (READ fault) //Referenced new page in COW obj
         mark page COW; disable WRITE;
   }
   if (WRITE enabled)
      mark page *not* COW;
   enter mapping for page in pagetable;
   activate and unbusy page;
}
```

Virtual Memory and Caches

- Two variants of caches:
  - → The cache uses virtual addresses
  - → The cache uses physical addresses
- With a physically addressed cache, a virtual address must be translated by the MMU (TLB) before it can be looked up in the cache.
- A virtually addressed cache uses the virtual address for lookup and is therefore faster.
- However, the virtual address cache must be flushed completely after each context switch.
- In systems with many short-running processes, a virtual-address cache gets flushed so frequent that it is seldom useful.

Page Coloring

- The Intel architecture uses a physical-address cache referred to as the L1 cache.
- If page frames are randomly assigned to virtual addresses, two consecutive virtual pages could be mapped to the same location in the L1 cache, causing frequent cache misses.
- The role of the page-coloring algorithm is to ensure that consecutive pages in virtual memory will be consecutive also from the view of the L1 cache.
- At system startup, each *vm_page* records its color in the L1 cache so that consecutive page frames always have different colors (fig. 5.13).
- Each color have its own free list of page frames.
- When an object is created, it is assigned a starting color.
- When a page fault occurs, a page is taken from the free list with the preferred color if it is available.

Page Replacement

- The page replacement algorithm used in FreeBSD is an approximation of the *Least Actively Used*.
- The algorithm uses a reference counter instead of the single bit used in LRU.
- The algorithm is similar to the one used in 4.4BSD but its implementation is considerably different.
- The memory is divided in five lists:

Wired: Locked in memory (kernel pages, pages locked with mlock() and thread stacks of loaded (i.e. not swapped out) processes.

Active: Actively used by one or more processes.

Inactive: Have content that is still known, but not part of any active region. May be dirty (modified).

Cache: Have content that is still known, but not part of any active region. Not dirty, so they may be moved to the free list when needed.

Free: Have no useful content.

The goal for the paging system is to keep a certain amount of pages in the *free, cache and inactive* lists.

Page Replacement, cont.

- The replacement algorithm is executed by the *pageout daemon*.
- The pageout daemon is a part of the kernel, but runs as a separate process with its own process structure and kernel stack to be able to use synchronization mechanisms such as *sleep()*.
- When the page allocation routine *vm_page_alloc*() discovers that more memory is needed, it starts the pageout daemon.
- When a page is first brought into memory it is given an *active count* of three.
- As each page at the active list is scanned, its reference bit is checked and if set the *active count* for the page is incremented by the number of references to the page.
- If the reference bit is clear, the *active count* is decremented.
- Pages that are repeatedly used build up a large *active count* that will cause them to remain at the active list much longer than pages used just once.

Page Replacement, cont.

The pageout daemon use the following memory limits for the lists:

Free min 0.7% target 3%

Cache min 3% target 6%

Inactive min 0% target 4.5%

If the limits are not fulfilled, the pageout daemon moves pages between the lists.

1. If ***free_count+cache_count*** $<$ ***free_target + cache_min***, pageout daemon moves pages from *inactive* list to *cache* list.
2. If ***free_count+cache_count+inactive_count*** $<$ ***free_target + cache_min + inactive_target*****,** pageout daemon moves pages from *active* list to *inactive* list.
3. If ***free_count*** $<$ ***free_min***, pageout daemon moves pages from *cache* list to *free* list.

Page Replacement, cont.

- When the pageout daemon moves pages between different lists, it scans the lists beginning with the page that has spent most time at the list (fig. 5.14).
- For each page on the *inactive* list, the following actions are possible:
  - $\rightarrow$ If the page is referenced, update the reference counter and move it back to the *active* list.
  - $\rightarrow$ If the page is invalid, move it to the front of the *free* list.
  - $\rightarrow$ If the page is clean and unreferenced, move it to the *cache* list.
  - $\rightarrow$ If the page is seen dirty for the first time, mark it *seen dirt*y and circulate it another time on the *inactive* list.
  - $\rightarrow$ If page is *seen dirty*, start asynchronous write and move to the end of *inactive* list.
- The activity is interrupted when enough pages are found or the end of the list is reached.
- Pages that are moved to the inactive and cache lists, still have a valid translation in the page table; it is only the referenced bit that is cleared.

Swapping

- Swapping is used if the pageout daemon is unable to free pages fast enough. This may happen if several large processes are run on a machine lacking enough memory for the minimum working sets of the processes.
- If the swap-out daemon can find a process that have been sleeping for more than 10 seconds it may be swapped out. If no such process is available a process that have been sleeping for as briefly as 2 seconds may be swapped out.
- Active processes are not swapped in FreeBSD5.
- When a process is swapped out all its pages are marked as invalid, including the page tables, the kernel stack and the user structure.
- Swap out operations are done by the *vmdaemon* (process 3) and swap-in operations are done by the *swappper* (process 0).

The Swap-In Process

- A process to swap in is selected based on:
- The time it has been swapped out
  - → Its *nice* value
  - → The amount of time it has been asleep since it last ran
  - → At swap-in the user area and the kernel stack of each of its threads are read back from the swap area.
- The process is marked as resident and its runnable threads are inserted in the run queue.
- The other pages are not read back until they generate a page fault.

Machine Dependent VM - PMAP

- The *pmap* module exports interface routines that are used by the machine-independent levels to manipulate the memory management hardware (MMU).
- Two types of MMU:
  - → Multilevel forward-mapped page table (I386)
    - ∗ Indexed by virtual addresses
  - → Inverted (reverse-mapped) page table (common in 64-bit architectures)
    - ∗ Indexed by physical addresses
- Page tables for 32-bit addresses may be several Mbyte big. It is inconvenient to allocate contiguous memory areas of this size.
- A way to solve the problem is to split the page table into smaller tables that look like a contiguous area by being referenced via an extra table (called directory table or segment table).
- In this case the page number field in the virtual address is split in two parts (see fig. 5.15)

Pmap

- The interface to the *pmap* module deals with machine-independent pages and machine-independent protections.
- The machine-independent page size may be a multiple of the architecture-supported page size. Thus, *pmap* operations must be able to affect more than one physical page per logical page.
- The *pmap* routines may act on either a virtual address range or on all mappings for a physical address.
- Mapping information maintained by the *pmap* module must be easily found by both virtual and physical addresses:
  - → Physical-to-virtual lookup uses a list of *pv_entry* structures, pointed to from the *vm_page* structure, to find all the page table entries referencing a page.
  - → For architectures such as the PC that support memory resident page tables, the virtual-to-physical lookup may be a simple emulation of the hardware page-table traversal.

Physical-to-virtual Address lookup

- For all page frames (*vm_page*) that is part of a process address space there is a corresponding *pv_entry*.
  → In FreeBSD there also exist page frames that are not part of a process, because the page frames are also used for the file system buffer cache.
- The purpose of the *pv_entry* structures is to identify the address space that has the page mapped.
- Fig. 5.16 shows the *pv_entry* references for a set of pages that have a single mapping.
- when an object is shared between two or more processes, each physical page get mapped into two or more sets of page tables.
- To track these multiple references, a chain of *pv_entry* structures is used as shown in fig. 5.17.
- The vm_page structures contains a list head that points to this chain of pv_entry structures.

Pmap interface routines

System initialization routines

`pmap_bootstrap()`

- Set up kernel *pmap* data structures

`pmap_init()`

- Allocates a minimal amount of wired memory for kernel page tables. The page table space is expanded dynamically by *pmap_growkernel()* as it is needed.

Mapping between virtual addresses (VA) and physical addresses (PA):

`pmap_enter(pmap, va, pa, prot, wired)`

- Called from the page-fault handler to initialize a new mapping in the page table.

`pmap_remove(pmap, start_va, end_va)`

- Remove all mappings for the specified address interval.

Pmap Interface Routines, cont.

Change of Access attributes

`pmap_protect(pmap, start_va, end_va, prot)`

- Change the protection for a region of process address space.

`pmap_page_protect(pa, prot)`

- Change the protection for a physical page in all *pmaps*.

Initialization of physical pages:

`pmap_zero_page(pa)`

`pmap_copy_page(src_pa, dst_pa)`

Management of Page-Usage Information:

`pmap_is_referenced(pa)`

`pmap_clear_reference(pa)`

`pmap_is_modified(pa)`

`pmap_clear_modify(pa)`

Pmap_enter()

- The *pmap_enter()* routine is called from the page-fault handler to set up a VA/PA mapping.
- The pmap_enter routine is also responsible for side effects such as flushing TLB or cache entries.
- On the PC, pmap_enter() must first check whether a page-table entry exist for the requested address.
- If there is no page table for the address, allocate a zeroed page table and add the address to the directory table.
- After ensuring that all page-table resources exist for the mapping:

1. If a mapping exist for the same address - change of protection or wiring attributes.
2. If a mapping exists but references a different physical address - remove old mapping.
3. A page_table entry is created and set valid, with cache and TLB entries flushed as necessary.
4. A pv_entry structure is created.