CS 3204
Operating Systems

Lecture 19

Godmar Back

Virginia Tech

Announcements

- Midterm March 29
  - See announcement + sample midterms on class website
- Fix remaining project 2 bugs (if any)
  - Reminder: by end of semester, passing students will have provided a deliverable that achieves >= 90% test score on project 2 (or a 100% test score on project 3 or 4's regression tests.) – multi-oom not part of these
- Project 3 Design Milestone
  - Will return before midterm

Virtual Memory

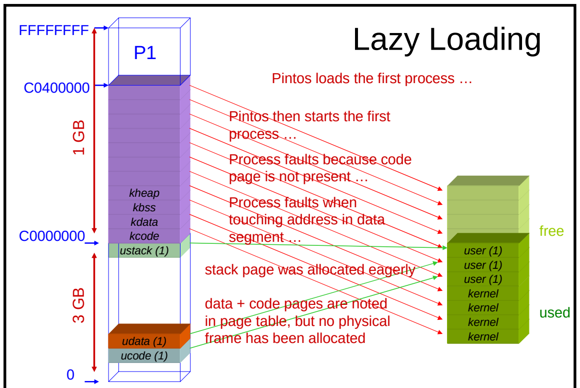
Paging Techniques (cont'd)

Virginia Tech

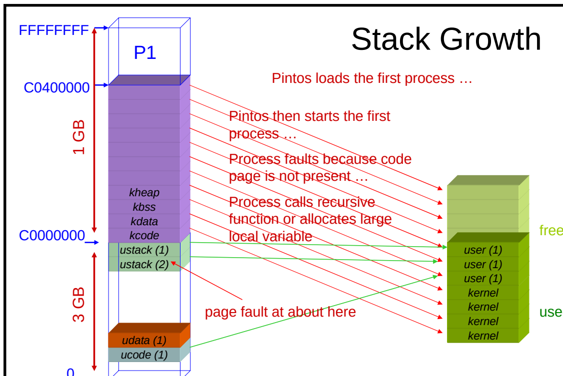
Demand paging

- Idea: only keep data in memory that's being used
  - Needed for virtualization – don't use up physical memory for data processes don't access
- Requires that actual allocation of physical page frames be delayed until first access
- Many variations
  - Lazy loading of text & data, mmapped pages & newly allocated heap pages
  - Copy-on-write

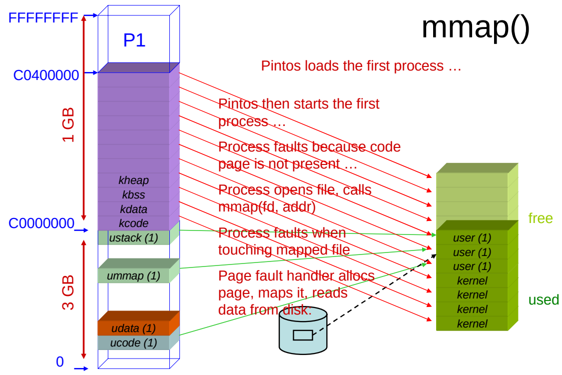
Lazy Loading



Stack Growth



mmap()



Lazy Loading & Prefetching

- Typically want to do some prefetching when faulting in page
  - Reduces latency on subsequent faults
- Q.: how many pages? which pages?
  - Too much: waste time & space fetching unused pages
  - Too little: pay (relatively large) page fault latency too often
- Predict which pages the program will access next (how?)
- Let applications give hints to OS
  - If applications knows
  - Example: madvise(2)
  - Usual conflict: what's best for application vs what's best for system as a whole

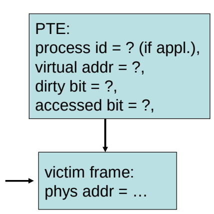
Copy-On-Write

- Sometimes, want to create a copy of a page:
  - Example: Unix fork() creates copies of all parent's pages in the child
- Optimization:
  - Don't copy pages, copy PTEs – now have 2 PTEs pointing to frame
  - Set all PTEs read-only
  - Read accesses succeed
  - On Write access, copy the page into new frame, update PTEs to point to new & old frame
- Looks like each have their own copy, but postpone actual copying until one is writing the data
  - Hope is at most one will ever touch the data – never have to make actual copy

Page Eviction

- Suppose page fault occurs, but no free physical frame is there to allocate
- Must evict frame
  - Find victim frame (how – later)
  - Find & change old page table entry pointing to the victim frame
  - If data in it isn't already somewhere on disk, write to special area on disk ("swap space")
  - Install in new page table entry
  - Resume
- Requires check on page fault if page has been swapped out – fault in if so
- Some subtleties with locking:
  - How do you prevent a process from writing to a page some other process has chosen to evict from its frame?
  - What do you do if a process faults on a page that another process is in the middle of paging out?

Page Eviction Example



Process A needs a frame
decides it wants this frame
Q.: how will it find the PTE,
if any, that points to it?

Linux uses a so-called "rmap" for that that links frames to PTE

Managing Swap Space

- Continuous region on disk
  - Preferably on separate disk, but typically a partition on same disk
- Different allocation strategies are possible
  - Simplest: when page must be evicted, allocate swap space for page; deallocate when page is paged back in
  - Or: allocate swap space upfront
  - Should page's position in swap space change? What if same page is paged out multiple times?
- Can be managed via bitmap 0100100000001
  - Free/used bits for each page that can be stored
  - Pintos: note 1 page == 8 sectors

Locking Frames

- Aka “pinned” or “wired” pages or frames
- If another device outside the CPU (e.g., DMA by network controller) accesses a frame, it cannot be paged out
  - Device driver must tell VM subsystem about this
- Also useful if you want to avoid a page fault while kernel code is accessing a user address, such as during a system call.

Accessing User Pointers & Paging

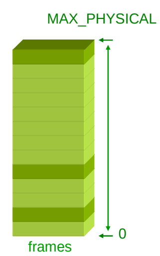
- Kernel must check that user pointers are valid
  - P2: easy, just check range & page table
- Harder when swapping:
  - validity of a pointer may change between check & access (if another process sneaks in and selects frame mapped to an already checked page for eviction)
- Possible solution:
  - verify & lock, then access, then unlock

```
if (verify_user(addr))
   process_terminate();
// what if addr's frame is just now
// swapped out by another process?
*addr = value;
```

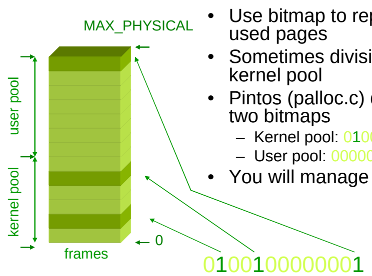

Physical Memory Management

Physical Memory Management

- Aka frame table management
- Task: keep efficiently track of which physical frames are used
- Allocate a frame when paging in, or eager loading
- Deallocate a frame when process exits or when page is evicted (later)



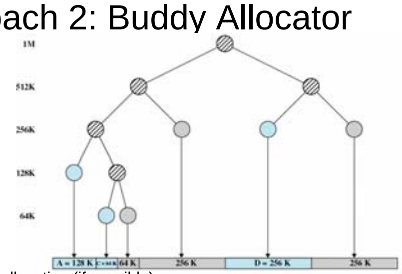
Approach 1: Bitmaps



- Use bitmap to represent free, used pages
- Sometimes division in user & kernel pool
- Pintos (palloc.c) does that: keeps two bitmaps
  - Kernel pool: 010010
  - User pool: 0000001
- You will manage user pool only

Approach 2: Buddy Allocator

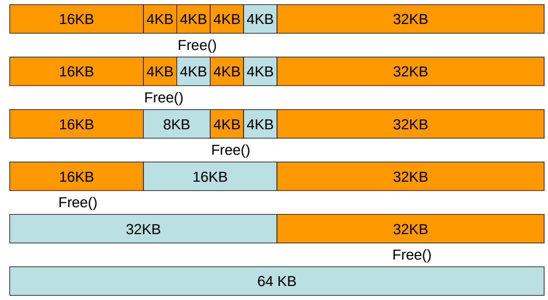
- Logically subdivide memory in power-of-two blocks
- Round up on allocation to next power of 2
- Split block on allocation (if necessary)



- Coalesce on deallocation (if possible)
  - Coalescing can be delayed
- Used in Linux: allocation requests are always multiple of pages, max blocksize is 4MB

Buddy Example - Allocation

Buddy Example - Deallocation

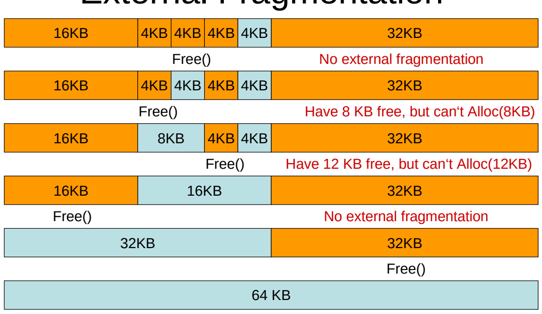


Fragmentation

- Def: *The inability to use memory that is unused.*
- Internal fragmentation:
  - Not all memory inside an allocated unit is used; rest can't be allocated to other users
- External fragmentation:
  - Impossible to satisfy allocation request even though total amount of memory > size requested

Internal Fragmentation

External Fragmentation



Buddy Allocator & Fragmentation

- Q.: what is the average internal fragmentation (per allocated object) for
  - buddy allocator with size 2^n?
  - in bitmap allocator for objects of size n*s, where each bit represents a unit of size s?
  - in first-fit allocator from project 0?
- Q.: what external fragmentation can you expect from buddy allocator scheme?
- Q.: what's a good way to measure fragmentation in general?

Page Size & Fragmentation

- How should a system's architect choose the page size? – Trade-Off
- Large pages:
  - Larger internal fragmentation
  - (not an issue if most pages are full…)
  - Page-in & write-back cost larger
- Small pages:
  - Higher overhead to store page table (more entries to maintain)
- Modern architectures provide support for "super pages" – 2MB or 4MB