CS 3204
Operating Systems

Lecture 17

Godmar Back

Virginia Tech

Announcements

- Project 3 Milestone due Friday Oct 24, 11:59pm
  - No extensions
- Will return feedback by Monday
- Project 3 Help Session next Monday 6-8pm
  - Room McB 209
- Read book chapters 8 and 9 on memory management
- Reminder: need to pass 90% of tests of project 2 by the end of the semester to pass the class
  - All project 2 tests except multi-oom will appear as regression tests in project 3 and 4
- Tag and branch your CVS for projects 3 and 4 (after commit, do cvs rtag –b –r working_project2 ….)

Virtual Memory

Paging Techniques

Virginia Tech

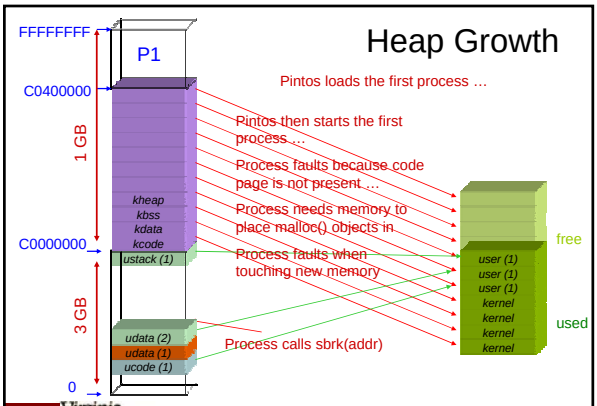
Fault Resumption

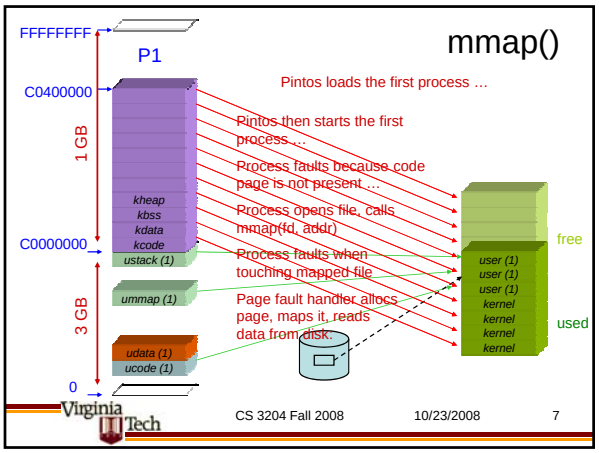
- Requires that faulting CPU instruction be restartable
  - Most CPUs are designed this way
- Very powerful technique
  - Entirely transparent to user program: user program is frozen in time until OS decides what to do
- Can be used to emulate lots of things
  - Programs that just ignore segmentation violations (!?) (here: resume with next instruction – retrying would fault again)
  - Subpage protection (protect entire page, take fault on access, check if address was to an valid subpage region)
  - Virtual machines (original IBM/360 design was fully virtualizable; vmware, qemu – run entire OS on top of another OS)
  - Garbage collection (detect how recently objects have been accessed)
  - Distributed Shared Memory

Distributed Shared Memory

- Idea: allows accessing other machine's memory as if it were local
- Augment page table to be able to keep track of network locations:
  - local virtual address → (remote machine, remote address)
- On page fault, send request for data to owning machine, receive data, allocate & write to local page, map local page, and resume
  - Process will be able to just use pointers to access all memory distributed across machines – fully transparent
- Q.: how do you guarantee consistency?
  - Lots of options

Heap Growth





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

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

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

PTE:
process id = ? (if appl.),
virtual addr = ?,
dirty bit = ?,
accessed bit = ?,

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

→ victim frame:
phys addr = …

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
- (Alternative is to handle page faults on user addresses in kernel mode)

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