CS 736

Lecture 8

Large Page Support:

1. Motivation:
    a. TLB size limited by being in the middle of a processor, accessed on every cycle
    b. Memory size limited by the amount you are willing to spend, # of physical address bits
    c. You can now buy 4TB servers,
    d. PROBLEM:
        i. Amount of memory a TLB can reference is small
            1. 4kb * 64 entries, = 256 kb
    e. QUESTION: What are some solutions?
        i. Turn off virtual memory
            1. Singularity, uclinux
                a. need new protection mechanism
        ii. Make TLBs larger and slower
            1. makes common case slower
        iii. Add a second level TLB
            1. Still performance/size sensitive
                a. With 1024 entries, still only 4MB
        iv. Share a TLB with multiple cores
        v. Prefetch into the TLB
2. Standard solution: multiple page sizes
    a. RISC machines:
        i. SPARC: 8kb, 64kb, 512kb, 4mb
        ii. ARM: 4kb, 64kb, 1mb, 16mb
            1. Typically not hardware walked, so flexible SW structures
        iii. How do page table?
            1. Often copy PTEs: entries for all the pages in a large page indicate the size
    b. Intel
        i. 4kb (64 entries) , 2mb (32 entries), 1 gb (4 entries)
    c. AMD
        i. Same sizes, 32 entry L1 (fully associative), 1024-entry 8-way associative L2
        ii. Follows radix tree of page table
            1. Easy for hardware walker – a level points to a page or the next level
    d. TLB design
        i. Fully associative (Sun Niagara, AMD)
            1. Can put any page size anywhere in TLB
        ii. Split set-associative TLB

        1. Have a separate TLB for each page size
        2. Each TLB is set associative
    iii. QUESTION: Why?
        1. Not know page size, so now know which set to access in set associative
  e. QUESTION: DO LARGE PAGES ALWAYS HELP
    i. Can waste memory if you don't use all the data
    ii. If have fewer TLB entries (see 1GB pages on Intel) may have more TLB misses
    iii. Expensive, inaccurate to swap.
3. OS Support possibilities
  a. Use for compile/install-time known data:
    i. kernel code, data
        1. Linux maps physical memory into its address space using arithmetic
        2. Map whole kernel, heap on large pages
    ii. Program segments in executable
        1. Mark segments (code, data, etc.) with a page size
        2. Must know at compile time what to do, how many pages available on the machine in the TLB
  b. Program request
    i. Windows: VirtualAlloc(MEM_LARGE_PAGES)
    ii. Linux: mmap(libhugetlbfs)
        1. Create "virtual file" /mnt/hugepagefile
        2. mmap(virtual file, memory size)
            a. Reserve contriguous memory for large pages
            b. Allocate and fill in on access
        3. PROBLEM: What happens if a process forks()?
    iii. QUESTION: Is this enough?
        1. Lets big-memory programs that suffer "do the right thing"
        2. Doesn't help most programs (lost opportunity)
  c. Transparent super pages/huge pages
    i. Programs do the normal thing
    ii. OS tries to use superpages if possible
4. INTERNAL OS Memory management
  a. GOAL: Need to have contiguous memory
    i. Overall: always merge contiguous blocks into "extents"
    ii. Have constant-time operations via efficient data structures
        1. Easily find whether neighbor is available for merging
  b. PROBLEM:
    i. Frequent allocation/deallocation creates fragmentation
    ii. Pinned pages cannot be moved – e.g. for DMA
  c. DATA STRUCTURE: Buddy heap
    i. Array of lists of powers-of-2 regions

    ii. Each list is sorted
    iii. Coalesce neighboring buddies into next power-of-2 list

5. Implementing Transparent Super Pages
    a. Reservations: on every use of a page, reserve pages around it to form a large page
        i. a reservation is a data structure referencing all the extra pages, taking them out of kernel allocator
        ii. Can reclaim an unused reservation for someone else

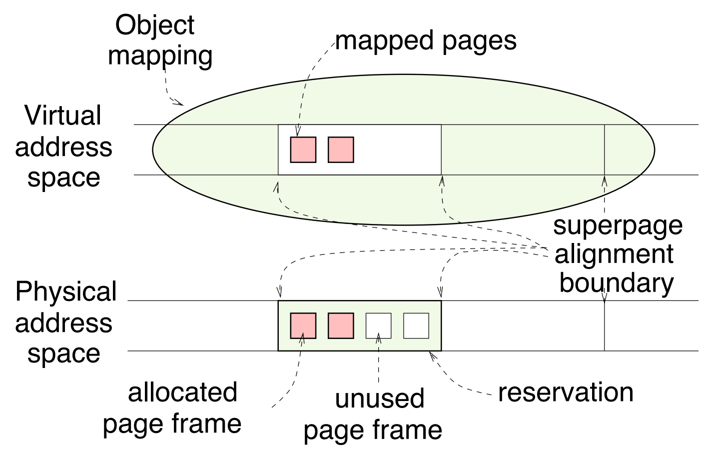

Figure 2: Reservation-based allocation.

        iii.
    b. Options:
        i. Decide at allocation time on a page size
            1. promote allocations
            2. Like static approach – but statically predicted by OS
            3. PROBLEMS:
                a. Can get it wrong and it costs a lot
        ii. Decide based on references to "upgrade" or "downgrade" a page
            1. If all of a large page is used, should upgrade to a large page
            2. HOW?
                a. Find a large page and move existing data
                b. Pick pages already in the right place and get rid of existing data and move new data in
            3. VERY EXPENSIVE
        iii. Prepare for upgrade on all allocations
            1. **Reserve** adjacent pages making a large page
            2. Use reserved pages on nearby faults
            3. At some threshold, upgrade to a large page
    c. POLICY: What page size should be **reserved** (if there are multiple)
        i. Fixed-size objects (code, global data): pick:
            1. largest aligned superpage that contains faulting page,
            2. doesn't overlap with other pages,

        3. Fits within the object (no waste)
    ii. Dynamically growing objects (stack, heap)
        1. Largest aligned superpage containing faulting page
        2. Not overlapping other superpages
        3. Can reach beyond object, but not larger than object
            a. Doesn't waste large pages on small objects
d. PREEMPTING RESERVATIONS:
    i. If not contiguous pages for new reservation:
        1. can not reserve for new allocation
        2. preempt existing reservation that has many unallocated frames
    ii. Preferred POLICY:
        1. Preempt existing reservation to create new one.
        2. Pick oldest reservation (least recently allocated a page from the reservation)
            a. Give away un-used pages, but don't remove valid data?
        3. WHY?
            a. Useful reservations likely to be used quickly
e. Promotions/promotions
    i. Incremental: grow mapping to next available page size
        1. QUESTION: Do you promote early, when 80% of base pages used, or wait for all 100%?
        2. ANSWER: Promote only at 100%
            a. Common case is programs use memory early and completely
            b. Makes sense for small super pages (8!)
    ii. Demotions: on page replacement
        1. Replace large page with next-size smaller
        2. Do recursively around victim page
        3. PROBLEM: No referenced bit on individual pages
            a. Cannot tell if whole superpage is used or only parts
        4. SOLUTION: demote to smaller pages & get more precise information
            a. Demote superpages under pressure but NOT swap out
            b. Occurs on clock hand sweep
            c. Re-promote if ALL pages around base page are re-referenced
f. Swapping dirty pages
    i. QUESTION: Do you need to swap large pages?
        1. ANSWER: Yes, because transparent!
    ii. No dirty bit for base pages – not know what changed
        1. Treat all base pages as dirty, must write **all** back
    iii. SOLUTION:
        1. Demote clean super pages on write
            a. set read only, trap, demote, re-map, set dirty bit

2. Re-promote only if **all** base pages written

iv. ALTERNATIVE:

1. Store hash of clean page; assume if hash matches than is clean.
    a. PROBLEM: Is possibility of being wrong (see VMware)
    b. PROBLEM: costly to do (when under memory pressure)

g. TRACKING RESERVATIONS:

i. Problem: Lots of reservations around all pages allocated

1. Solution: keep a list per page size
2. Reservation goes on the list of **the page size that can be gained** by preempting reservation
    a. Sort reservations by time of last allocation – used for preempting
    b. Split reservation into largest-sized extents (not base pages)
        i. Keep contiguity as long as possible

ii. SO: if need 64KB, can go to 64KB list and preempt a reservation

iii. QUESTION: What about Intel, with 1GB, 2MB, 4KB pages

1. 1GB reservations go on 2MB or 4KB list
2. 2MB reservations go on 4KB list (cannot make smaller superpages)

h. FINDING MEMORY

i. WHY?

1. Need to find reservations on page fault
2. Detect overlapping regions
3. Have information on whether page promotions are possible
    a. if all neighboring pages exist
4. Identify un-used regions for preemption

ii. Population Map:

1. Data structure: radix tree like a page table, each level is a page size

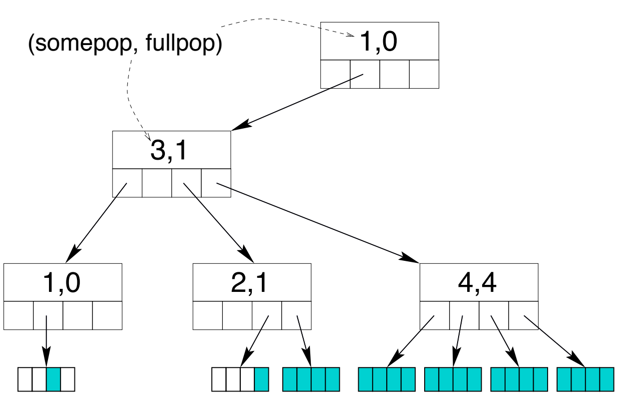

Figure 3: A population map. At the base page level, the actual allocation of pages is shown.

2.
3. Contents:

            a. Number of entries at next level that are full (fullpop)
            b. Number of entries at next level that are non-zero but not full (2 levels smaller at least)
        4. USE:
            a. Find allocation for a page:
                i. walk down tree to find reserved frame
            b. Overlap avoidance:
                i. walk down until "somepop" is zero
            c. Promotion:
                i. If fullpop goes from R-1 to R (fill)

6. IMPLEMENTATION ISSUES:
    a. Swapping: want contiguity awareness in swapping
        i. FreeBSD background:
            1. Cache pages = valid data, but can be immediately reclaimed. Not in any page table
            2. Inactive pages = valid data, but need some work to reclaim – swap data out
            3. Active pages = data used by a process, in a page table
        ii. keep cache pages (have data but not mapped) in a buddy allocator with free (totally unused) pages
        iii. Page daemon runs when contiguity is low
            1. Failure to allocate region of requested size
            2. Traverse inactive list (pages with ref bit clear) add moves caches adding to contiguity
                a. Inactive list is valid pages but not in page table; easier to reclaim. Can be dirty, though
                b. Make invalid but remember still exists in memory
                c.
        iv. Mark clean pages from files inactive when closed (still cached in memory)
            1. so more pages to take later for contiguity
    b. Wired pages: stuck in memory and cannot be moved/evicted
        i. If in the middle of a page, cannot reclaim to form page
        ii. SOLUTION: cluster in one place
            1. Coalesce to one large page – could relocate before wiring/pinning
    c. Multiple mappings: map a file in two processes
        i. Try to use same alignment for mappings – largest superpage smaller than mapping itself
7. Issues on x86:
    a. Pages are much larger; chance of touching all pages is lower, cost of reserving too much is higher
8.