Large Page Support:

1. Notes from reviews
   a. Population maps per process or global? Per process, but could be shared
   b. Why walk PTE 6 times? Need to modify each PTE on each promotion (3 times)

2. Presentation group

3. 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
   f. What is benefit?
      i. Cost of a TLB miss: ~30 cycles
      ii. How many TLB misses do you have to avoid with a single large page to be worth spending 1000 cycles managing it?
         1. A: 30

4. 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.
5. 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 contiguous 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
6. 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
d. Linux approach:
   i. Try to allocate 2 mb pages if allocate >2mb
   ii. Never break (on fork, on write, page out, etc)
   iii. Khugepaged scans for opportunity to promote by copying
   iv. Compacting thread tries to relocate pages to make huge pages available
e. Background: Demand paging
   i. Allocate virtual memory regions with mmap()
      1. Kernel just reserves address space, provides no memory
   ii. On touching a page:
      1. Kernel allocates physical memory
7. Issues:
   a. On allocation: what to do?
      i. Use large pages immediately (Linux)
      ii. Use small pages but reserve a large page if used (this paper)
      iii. Use small pages and later rearrange memory if useful
   b. Fragmentation:
      i. Using multiple page sizes increase fragmentation
      ii. Could reclaim some reserved pages for other uses
      iii. Could evict data to make contiguous pages available
   c. Promotion
      i. When have a number of base pages allocated, can promote (convert to superpage)
d. Demotion
   i. When break apart a super page?
      1. Reclaim pages
   ii. How know what is dirty? 1 dirty bit/superpage
   e. Eviction
   i. On swapping, what size to write out? Whole superpage (slow) or base page (no dirty bit)?
   ii. What pages to write out? LRU or least useful for contiguity?
f. Considerations:
   i. Do you expect to swap?
ii. Do you expect to have enough memory?

8. 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

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)
4. **SO**: use progressively smaller super pages at beginning and end (SHOW)

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
   4. **NOTE**: in current systems, only stacks grow. Heap is grown through mmap() calls, not extending heap region (brk)

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
         a. Do recursively around victim page
      2. PROBLEM: No referenced bit on individual pages
         a. Cannot tell if whole superpage is used or only parts
      3. 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
         d. Can do with a probability to reduce cost of demotions
   4. **QUESTION: What is impact on performance?**
      a. If not used, no increase in TLB misses, so not make things worse
      b. If get referenced, will quickly repromote
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
       3. PROBLEM: for large superpages (2mb), may not get as much chance to use large pages
   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 preemipping reservation
          a. Sort reservations by time of last allocation – used for preemipping
          b. Split reservation into largest-sized extents (not base pages) on allocaiton
             i. Keep contiguity as long as possible
             ii. Example: 256kb reservaton on 64kb list
                1. Take 64kb out, have two more 64kb pages to put back on list
          c. Depending on what used, a reservation goes on lists of page sizes smaller than it
   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 (next smaller sizes)
       2. 2MB reservations go on 4KB list (cannot make smaller superpages)
   iv. On allocation:
       1. Try buddy list for desired size (will find anything bigger and break)
       2. Try per-size list, if empty, try all larger sizes

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
### IMPLEMENTATION ISSUES:

#### a. Swapping: want contiguity awareness in swapping

- 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
   1. Put at end so use totally free pages first
   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 pages to cache to adding to contiguity (ones that not help not moved)
          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
   iv. Mark clean pages from files inactive when closed (still cached in memory)
       1. so more pages to take later for contiguity
   v. **OVERALL:** may reallocate pages sooner instead of retaining contents to get better 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
10. Issues on x86:
    a. Pages are much larger; chance of touching all pages is lower, cost of reserving too much is higher
11. Evaluation:
    a. What needed?
       i. Show contribution of each technique
       ii. Show cost of each technique (so predict benefit with different TLB)