CS5460/6460: Operating Systems

Lecture 18: Midterm Discussion

Anton Burtsev

February, 2014

Question 1. Page tables

1.a. Flat page table

- 4K pages, 4GB address space
- You need 1M entries
- How big is each entry?
  - Technically you need 20bits to address a page
  - But lets say similar to x86 you use some space for flags, so each entry is 32bits (4bytes)
- Page table size
  - 4x1M = 4MB

1.a. 2-level x86 page table

- 4K pages, 4GB address space
- You need 1 page for PTD (4K)
  - PTD has 1024 entries
  - Each PTD points to another 4K page
- How big is the page table:
  - 4K + 1024*4K = 4M + 4K

1.b. 1K pages

- You need to address individual bytes in each 1K page
    - 10 bits for offset inside each page

1.b. 1K pages

- How many pages can be addressed by a page of a page table?
- Pages are 1K aligned
  - Each entry needs 22 bits
  - Note, 2 bits more than x86
  - 1K pages, not 4K
- Where do we take 2 extra bits?
  - Take some unused flags
- Each entry is 32 bits

1.b. 1K pages

- Each entry is 32 bits
  - 256 entries per page
  - 8 bits to address them
- Final topology
  - 6 + 8 + 8 + 10

1.b. 1K pages

- Can I do
  - 8 + 6 + 8 + 10
  - 8 + 8 + 6 + 10
  - Yes!

1.b. Advantages

- Advantages
  - Fine-grained memory management
- What does it mean?
  -

1.b. Disadvantages

- Longer page walk
  - 3 memory reads from page tables
  - 1st level (1 page) – always cached
  - 2nd level (64 pages or 64KB) – likely cached
  - 3rd level (64x256 pages or 16MB) – likely a cache miss
  - Remember 3 Level caches are up to 37.5MB (high-end IvyBridge server)
- Compare to 2-level paging
  - 2nd level 1024x4K or 4MB
- Well, example is artificial, servers run 64bit systems, 32GB RAMS
  - 64bit has 3-level page hierarchy anyway
  - But with 1KB pages the problem will be even worse

1.b. Disadvantages (contd)

- TLB pressure
  - To read 4K of RAM you need 4 TLB entries now
  - Instead of 1 entry
- Example: TLBs on IvyBridge
  - 2 level hierarchy:
    - 64 entries first level + 512 second level
  - Enough to cache 0.5MB of memory

Question 2. Synchronization

```
// Allocate one 4096byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

Performance goes up for up to 4CPUs, but then stops, why?

2.b. Does RCU make sense?

- Can we rewrite this code with RCU?
  - No
  - With RCU concurrent updaters still need use locks or some other synchronization primitive to synchronize among themselves
- RCU helps when there is a
  - Large number of readers
  - Small number of updaters
- Main advantage of RCU
  - Readers don't need a lock

```c
// Allocate one 4096byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
char*
kalloc(void)
{
  struct run *r;

retry:
  xbegin();
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  xend(retry);
  return (char*)r;
}
```

Hardware transactions

2.c. Do hardware transactions help?

- Hardware transactions do not help in this case
  - Multiple CPUs content on a single variable
  - kmem.freelist

```
retry:
  xbegin();
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  xend(retry);
```

2.c. Hardware transactions

- Hardware transactions will conflict and abort
- Depending on implementation performance will remain unchanged or will go down
  - All conflicting transactions abort
    - Performance goes down
    - Livelock
  - HTM chooses one winner
    - Performance remains about the same

2.d. Making it fast

- Scalable spin-locks, aka MCS?
- Your experiment runs on 16 CPUs
  - You are not limited by performance of the cache coherence protocol (I think)
  - Your bottleneck is the critical section
    - MCS might help a bit, but not a lot

2.d. Making it fast

- You need to remove the bottleneck
- Per-CPU page pools
  - Each CPU has a pool of pages
  - Similar to sloppy counters
  - Allocates and deallocates pages without contention
  - If pool is empty takes more pages from the global pool
    - Of course global pool is under a lock
  - If pool grows too large, free pages to the global pool

Question 3. Segmentation

Why do you really need paging?

- Isolation?
  - No, for isolation you just need a way to say that part of memory is accessible/inaccessible
  - Segments do work for this
- Paging
  - Allows you to build address spaces with holes, due to
    - Lazy allocation of memory, i.e. the address space can grow
    - Swapping out pages to disk, i.e. the address space can shrink
  - Other useful things
    - Sharing with other address spaces, i.e. one physical page appears in multiple address spaces (mmap(), shared libraries)
    - Copy-on-write – sharing identical pages read only, until they are written (fork())
    -

3.a. Address spaces with segments

- Each process has it's private segment for
  - Stack, data, and code
  - Segments map logical addresses to physical, e.g.
    - Logical data 0 – 8MB can be mapped anywhere in the physical memory
    - Segment base + segment limit
- Kernel needs access to every process
  - Initialization, fork, copy in and out of system call arguments
  - Kernel data segment spawns entire physical memory

3.a. Context switching

- Two ways
  - Reload GDT (each process has a private GDT)
    - Alternatively update segments in the GDT
  - Reload LDT (each process has a private LDT)
    - Similar to GDT
- Interrupt path
  - IDTR and GDTR hold linear addresses of IDT and GDT
    - Linear == physical (we don't have page tables)
    - Everything just works
    - Of course physical memory of IDT and GDT should not be mapped into any process

3.b. sbrk()

- Sbrk grows heap of the process
    - i.e., data segment
- Trivial if more physical memory is available right after the data segment
    - Just change the DS limit
- If no memory available need to relocate segments
    - Note, you can move content of physical memory
    - Then change segment base
    -

3.c. Sharing a region of memory

- Sharing means two or more processes access the same physical memory simultaneously
  - Useful for communicating information, e.g. inter-process communication mechanism
    - Send messages from one process to another
- Create a segment and map (add to the GDT) of all sharing processes
  - Segment defines a region of physical memory that is shared

3.d. Disadvantages

- Address spaces can grow up, but not down
  - E.g. you can increase segment limit for the data segment
  - But what about stack?
    - Stack grown down
    - Well, stacks can grow up too (there is a flag somewhere)
- No holes in address spaces
  - Can't unmap and swap out a couple of rarely used pages
  - Swapping is possible but only at granularity of segments
- No copy-on-write sharing, e.g. fork()
  - Sharing schemes are much more restrictive

3.d. Advantages

- No overheads of page translation
  - No TLB misses
  - No cache misses due to page table walks
  - No page walks at all (even in case of cache hits, page walk still adds overhead, but small)
- Faster context switch
  - No need to flush and reload TLB
    - Remember on context switch page table is reloaded, TLB needs to be flushed
  - Wait what about tagged TLBs
    - True, tagged TLBs avoid TLB flush, but increase TLB pressure

Conclusions

Thank you!