The Page Table, Our Faulty Friend

Computer Science 432 — Lecture 9 — Duane Bailey

March 2, 2022

Announcements

- Lab 4 (Page Tables) is out.
  - *You caught me*. I (unsuccessfully) edited the page table dump in the lab handout. Revised version on the web site.

- Small group discussions about Lab 3, today and tomorrow.

- Next two weeks: follow calendar:
  - No small groups next week (but *yes* to the week after)
  - Lab meeting next week, (but *no* to the week after)

Notes on Page Table Permission Bits (for Petros)

- Recall: there are 10 bits reserved for metadata associated with virtual address translations:
  - `PTE_V` — this is a valid entry. If other bits are zero, this is a pointer to another table node.
  - `PTE_R`, `_W`, `_X` — read/write/execute access to page. (`PTE_W` *requires* `PTE_R`)
  - `PTE_U` — (1) this page may be accessed in User mode — *but not Supervisor,* (0) *vise versa*
  - `PTE_G` — global definition
  - `PTE_A` — this page was *read* accessed.
  - `PTE_D` — this page was *write* accessed.
  - bits 8 & 9 — reserved for O/S use.

- Typically, `PTE_U` is set in process page tables, mainly used in User mode.
- Typically, `PTE_U` is cleared in kernel page tables, exclusively used in Supervisor mode.
- Thus: you can control access using at least *two* page table entries.

Handling Page Faults

- Recall that exceptions—including access violations—are handled by the trap system.
  - Different types of traps—system calls, exceptions, and interrupts are routed in `usertrap()`. (See `trap.c`)
    This routine is where *page faults* and their processes go to die.

  - There are three types of page faults:
    - Instruction-based faults (no `PTE_X`)
    - Load-based faults (no `PTE_R`)
    - Store-based faults (no `PTE_W`)

  - We can route page fault handling based on *cause*.

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2–3	*Reserved*
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6–7	*Reserved*
1	8	User external interrupt
1	9	Supervisor external interrupt
1	≥10	*Reserved*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	*Reserved*
0	5	Load access fault
0	6	AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call
0	9–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	≥16	*Reserved*

Table 4.2: Supervisor cause register (`scause`) values after trap.

An optimization: Copy-on-Write (COW) Pages

- Motivation: The fork process makes a copy of all the pages of the parent, in the new child.
  - Fork calls `uvmcopy`. (See `proc.c`)
  - uvmcopy walks the page table, `kalloc`-ing and `movmem`-ing (See `vm.c`)
  - Every byte of the parent memory is copied…
  - And then, typically, the child process is *replaced* by an `exec(prog, args)` system call.
- How might we avoid copying the parent memory into the child?
- Solution: Creating a copy of each page *only* if it is written.

Rough Outline of COW Optimizations

- In fork, page tables are prepared:
  - The child gets a new page table,
    but instead of referring to *copies* of physical pages
    the child's page tables *share* references with the parent
  - If these pages have read or execute access, *everything is good!*
  - If the pages are writable, the two processes may wish to diverge by writing different values
    This is the main concern of Copy-on-Write optimizations:
    - The `PTE_W` bit is cleared
    - One of the O/S bits is used to represent: "Copy this page if written to."

- The O/S can't inject itself *directly* in the writing process, but it can use *page faults*, indirectly.

Identifying the Need to Copy: Store Faults

- When a COW-optimized process wants to write to memory:
  - The write to virtual memory performs a translation through the table with no `PTE_W` bit
  - A *store-based page fault* (`scause` 15) occurs.
    - The fault arrives in the `usertrap()` routine with
      - `scause` CSR containing 15
      - `sepc` CSR containing the address of the instruction that tried a write
      - `stval` CSR containing the address that was the target of the write
    - `usertrap()` could hand off to a routine that
      - makes a *writable* copy of the target page (`kalloc`, memcpy, set `PTE_W`),
      - unmaps the shared page from the process' page table,
      - replaces it with a mapping that targets the new writable copy,
      - returning from the trap returns the sepc, the instruction that failed will now go through.

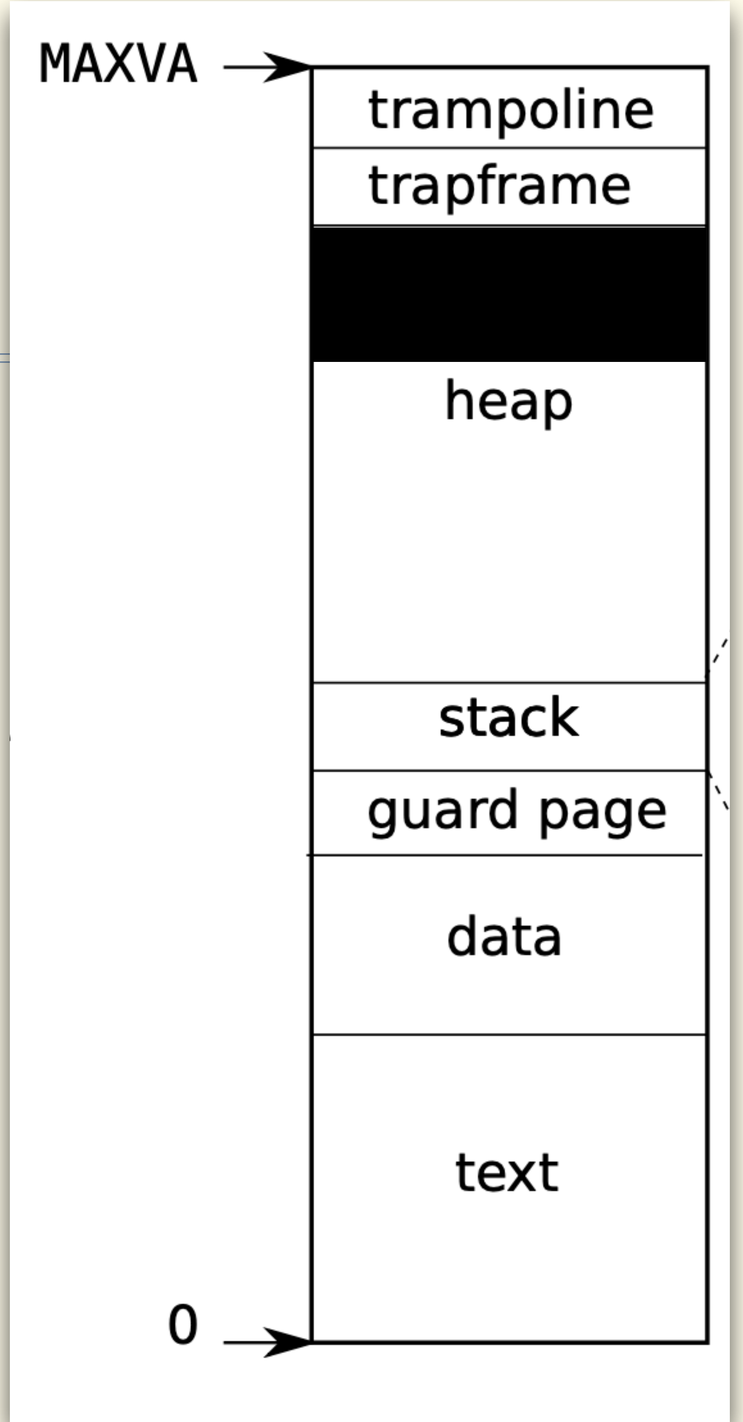
Reference Counting Shared COW Pages

- The process of sharing process pages requires some changes to kernel allocation.
  - For each physical page, we track the number of page table entries that reference that page.
  - This *reference count* is set to 1 in `kalloc()`.
  - The reference count is incremented by 1 when we *share* a page reference.
  - Each time we call `kfree()` we decrement the reference count.
  - Only when the reference count is reduced to zero, does the page get returned to the free list.

- Observation: writable pages will only have a reference count of 1.
- Observation: shared pages with reference count of 1 can be written to *directly*.

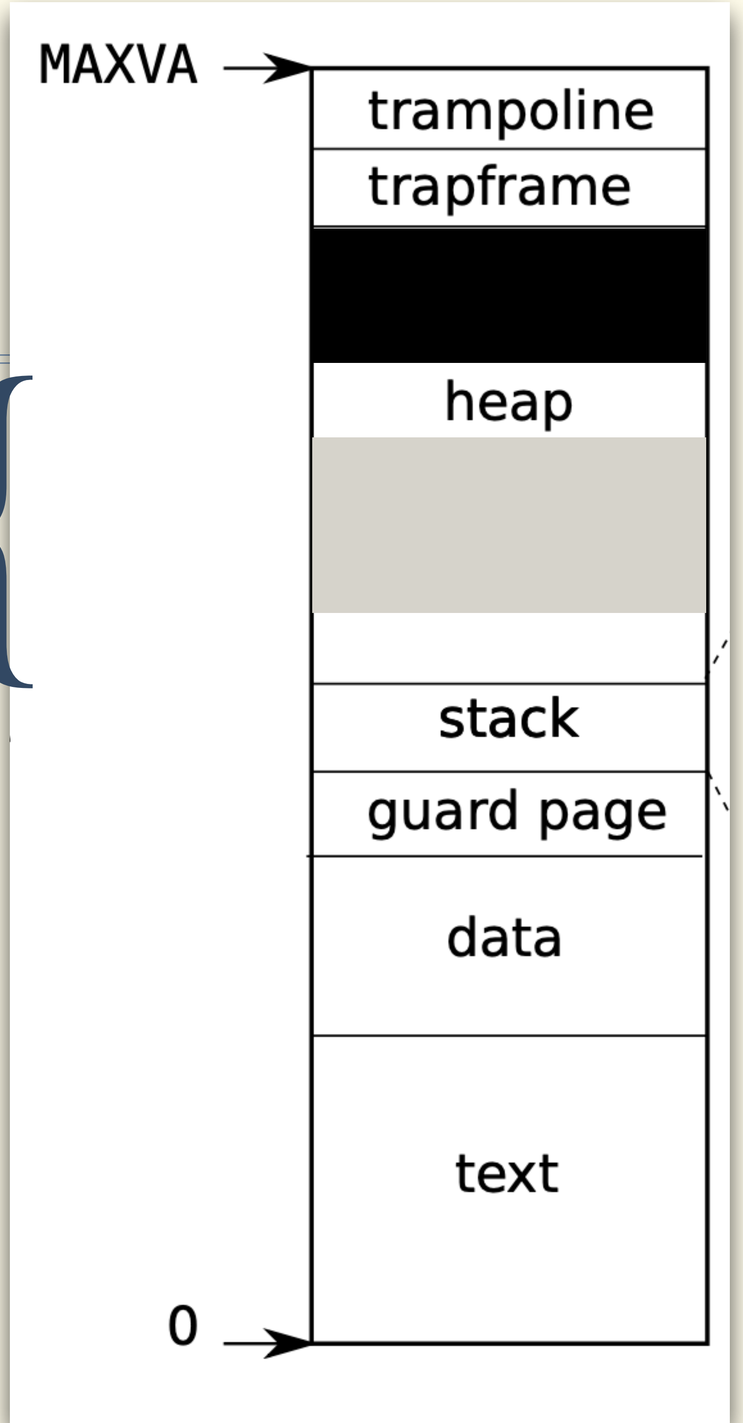
An Optimization: Lazy Allocation

- *Lazy allocation* allows processes to extend process memory only when a page is actually accessed.
  - Process memory is extended by calling `sbrk()`. This services adds more pages to the heap.
  - Traditionally, that memory is allocated and targeted by the page table mapping.



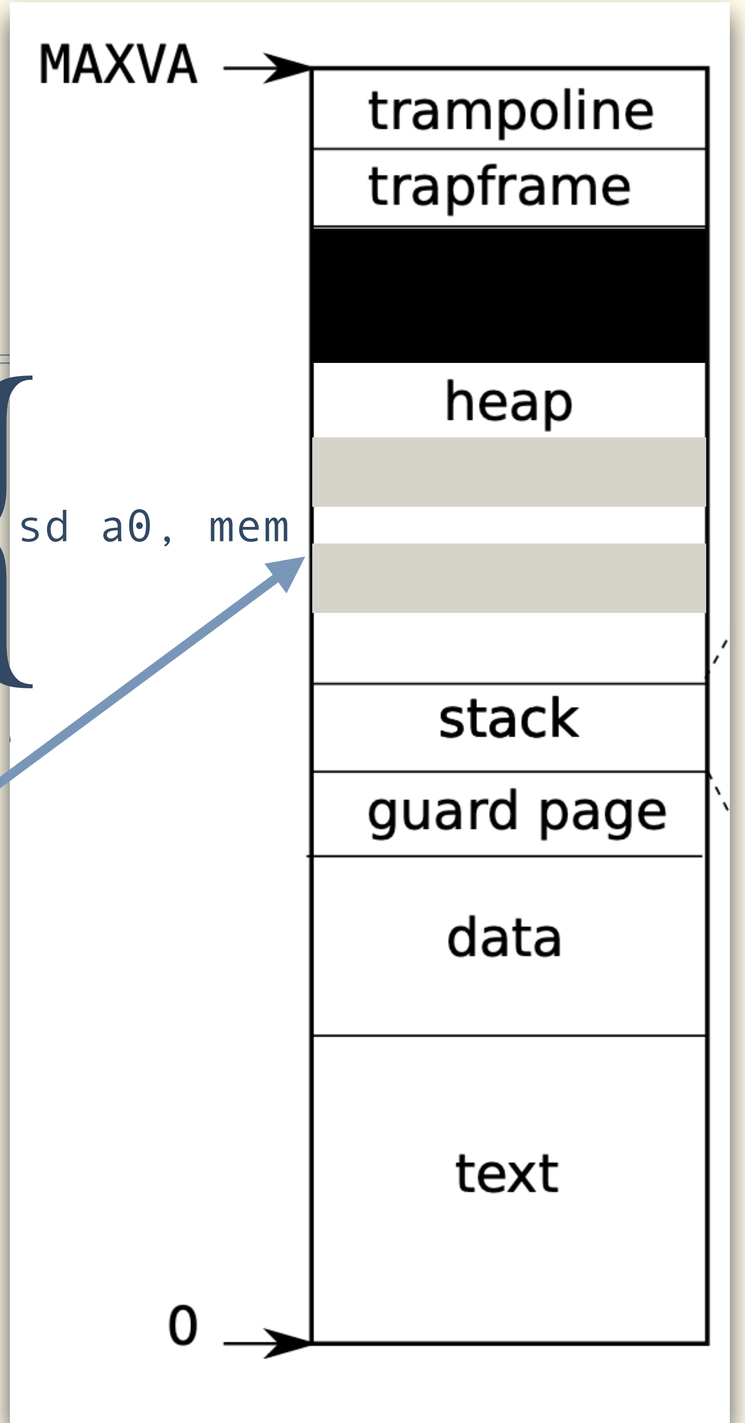
An Optimization: Lazy Allocation

- *Lazy allocation* allows processes to extend process memory only when a page is actually accessed.
  - Process memory is extended by calling `sbrk()`. This services adds more pages to the heap.
  - Traditionally, that memory is allocated and targeted by the page table mapping.
  - With lazy allocation, the kernel keeps track of the size of the process' address space *but does not allocate or map that memory.*



An Optimization: Lazy Allocation

- *Lazy allocation* allows processes to extend process memory only when a page is actually accessed.
  - Process memory is extended by calling `sbrk()`. This services adds more pages to the heap.
  - Traditionally, that memory is allocated and targeted by the page table mapping.
  - With lazy allocation, the kernel keeps track of the size of the process' address space *but does not allocate or map that memory.*
  - If a page fault occurs within the *logical* extent of the process, *that one page* is allocated and mapped and the instruction is restarted.



Other optimizations: Paging Strategies

- Optimization: Demand paging.
  - Traditionally all code and data pages are loaded into memory at the start.
  - Demand paging only loads code and data pages when they're first referenced.

- Optimization: Paging to secondary store.
  - Currently, all pages of the virtual address space are found somewhere in physical memory.
  - In a *paging* system, only a *working set* is kept in physical memory.
    The rest is saved to backing store.
  - The model is to think of physical memory as a *cache* for the virtual image saved on disk.