Lab 3 Design Doc: Address Space Management

Names: Aadhithya Arun, Jacob Young

Overview

Lab 3 improves virtual memory management by implementing heap growth (sbrk), dynamic stack growth, and copy on write fork which optimizes memory allocation and process execution. We implement sbrk to enable dynamic heap allocation. Dynamic stack growth allows pages to be allocated on demand. And copy on write fork reduces memory duplications by sharing pages until they become modified.

Major Parts

Heap Growth

- Goals: Implement sbrk to allow user programs to allocate memory at runtime with malloc. We should track the size of the heap and update virtual page mappings after allocating the physical memory (page size allocation)
- Challenges: Making sure we safely update virtual memory that is backed with physical memory, making sure we dont collide with any other part of virtual memory

Stack Growth

- Goals: Create an On-Demand page allocator for the stack that can handle when the user want to grow the stack by handling page faults
- Challenges: Distinguishing types of page faults, making sure not to go over the limit

Copy-on-write Fork

- Goals: Create a more efficient representation of virtual memory after fork that does not copy every page
- Challenges: making sure to mark all pages and update metadata to track the COWpages

Interactions with Page Fault Handler

- Goals: make sure that the page fault handler can correctly distinguish between stack growth faults, cow faults, and illegal memory accesses. We also need to correctly update the page tables and mappings on the cause of the fault
- Challenges: make sure cow page faults allocate a new page. Properly manage locks and synchronization. Ensure the TLB is flushed. Correctly identify the page fault (cow violation or page access violation)

In-depth Analysis and Implementation

Heap Growth

Functions To Implement & Modify

- Sbrk():
  - Use myProc() to save the current proc's vspace
  - Check if the current request will collide with stack memory, if so return -1
  - Save the old top of the heap with vspace.regions[VR_HEAP].va_base + vspace.regions[VR_HEAP].size
  - Use vregionaddmap to extend the heap, then call vspaceupdate to update the hardware specific pagetable, then vspace install to install to the hardware
  - If vregionaddmap fails return -1
  - Return previous top

Design Questions

> Heap growth is not demand paged in xk, meaning that the kernel allocates frames and maps them (`vregionaddmap`) as needed upon `sbrk`. How does `vregionaddmap` allocates a physical frame? What structures are updated as a result?

- Just uses kalloc to allocate physical memory, and uses memset to zero out all of the page memory
- The page's vpage_info struct is updated as a result

> Does `vregionaddmap` update the machine dependent page table? If so, what line performs the update? If not, how should you update the machine dependent page table?

- No vregionaddmap does not update the machine-dependent page table, we must use vspaceupdate to update the machine-dependent pagetable

> Does `sbrk` always cause new page(s) to be mapped? Justify your answer.

- No the loop condition just checks if we need new page, if so then a new page is mapped

> `sbrk` cannot succeed if the resulting heap will collide with an already allocated region within the virtual address space. How do you determine whether growing the heap will cause a collision?

- We can check in sbrk whether the current top of the heap + the size > the bottom of the stack (vspace.regions[VR_USTACK].va_base - vspace.regions[VR_USTACK].size)

Corner Cases

- Negative sbrk argument - Treat negative values as 0; just return the current heap break without allocating or freeing any memory
- Sbrk(0) - refer to above
- To large allocation - when the requested alloc exceeds available virtual memory, or would cause collisions, just return -1 (make sure no partial allocs)
- Return -1 if system runs out of memory
- Alignment issues and Heap/stack collision

Test Plans

- Bad_mem_access: accessing memory outside of allowed ranges yields proper error handling
- Malloc_test: tests sbrk implementation for correct memory allocation
  - Verifies that multiple consecutive malloc operations work
  - Memory is allocated correctly and can be used
  - Free works properly
- Sbrk_small: Test byte level allocation of memory. It increments the heap 5k times, 1 byte at a time
  - Each time it tests if sbrk returns the previous heap break
  - Memory is allocated properly
  - The child process gets a parent's heap and it grows after fork
- Sbrk_large: Bigger allocations and out of memory conditions
  - 256 page allocation
  - System properly cleans up partial allocs if we run out of memory

Stack Growth

Functions To Implement & Modify

- In trap.c when a page fault is called
- Check whether we encountered a stack page fault or a permission page fault(we can use b0 to determine this)
  - If page fault and address <= stack_base - (10 * (4096))
    - exit()
  - Iteratively call vregionaddmap/vspaceupdate/vspaceinstall to add pages until the faulting address is in the new vregion (use vregion contains to check whether the vregion contains it)

Design Questions

Stack growth must be demand paged, meaning that the growth needs to be handled through a page fault. Upon a page fault, what value(s) do we expect as the present bit (b0) of the error code (`tf->err`)?

- We would expect this bit to be zero since this virtual address is not yet mapped to a physical frame

Upon a page fault, what value(s) do we expect as the read/write bit (b1) of the error code (`tf->err`)? Can a stack growth happen as a result of read and write?

- This value could be either, as the stack could try to access further memory for a read or write operation

Upon a page fault, what value(s) do we expect as the user/kernel bit (b2) of the error code (`tf->err`)? Can a stack growth occur within the kernel? If yes, explain a scenario that leads to it, if not, explain why not.

- We expect this bit to be a one since the kernel has its own separate stack that it uses, so only the user can extend past it for a page fault.

Upon a page fault, how do you tell whether the faulting address is within the the stack range?

- We can use the function vregioncontains to tell if the faulting address is with the stack range of virtual addresses

Is there any synchronization necessary when handling stack growth?

- No since there is no multi-threading and each process has its own virtual memory, there will only be one change to the vspace at a time.

After handling the stack growth, is it necessary to flush the TLB? Why or why not?

- No we don't change existing mappings or change the permissions of the page

Corner Cases

Describe any special/edge cases and how your program logic handles these cases.

- Max stack size: if page fault happens but stack is already at max, terminate
- Stack grow up: invalid address if fault address is above STACKBASE
- Non contiguous stack access: user cant access super far below the current stack
- Consecutive Faults: properly handle race conditions if page faults happen one after the other

Test Plans

- Stack_growth_basic: tests basic stack growth functionality with write and read triggered growth
- stack_growth_bad_access:
  - Cant grow more than 10 page limit
  - Cant grow stack above STACKBASE

Copy-on-write Fork

Functions To Implement & Modify

- vspacecowcopy():
  - Copy the vspace and set the regions pointer values the same addresse as the parent
  - Iterate through the vpage_infos, if the current writable == 1, set cowpage_flag to 1, then for all vpage)info set writable to 0, (if cowpage_flag is already 1)
- Page Fault Handler:
  - Triggered by a write to a read-only page
    - If the cowpage flag
      - We find which page it is in the vspace(va2page)
      - We allocate a new page (perhaps using kalloc)
      - We copy the information from one page to the other (using memove)
      - Make sure to update vpi_page_info iteratively like in copy_vpi_page
      - Change the writable section to true
      - Kalloc will already handle the refcount for us
      - The vregions->pages field will now be changed, we call vspace update to update the pagetable
    - Otherwise fault like normal
- Modification to kfree, kalloc, core_map_entry
  - Add a refcount, and lock to safely increment the ref count
  - Change kalloc to initialize the refcount to 1
  - Change kfree to only free if the refcount before freeing == 1
- Modification to vpage_info:
  - Add a cowpage_flag that is set when the original frames permission were writable

Design Questions

At fork time, what needs to be changed in parent's vspace to reflect the copy-on-write?

- In the parent's vsapce modify the page table entries to read only, by updating the writable field and set the COW flag. Also update the ref count. This occurs in the vpi_page stored in the vregion

After a cow fork, is it necessary to flush the TLB? Why or why not?

- Yes because the cow fork updates page table entries and since the TLB caches page table entries, this means that the read/write permissions are also cached. There may be cases where the CPU may continue to use pages that used to be marked as writable but are read only now. Also, if the child's TLB is not flushed, it may use stale mappings that were marked as writable that were inherited from the parent before the fork.

Upon a cow page fault, what value(s) do we expect as the present bit (b0) of the error code (`tf->err`)?

- We are trying to access a page that is already mapped so we expect this bye to be 0.

Upon a cow page fault, what value(s) do we expect as the read/write bit (b1) of the error code (`tf->err`)?

- We expect this to be set since cow page faults will always be triggered by a write access on a read_only page

Upon a cow page fault, what value(s) do we expect as the user/kernel bit (b2) of the error code (`tf->err`)? Can a cow occur within the kernel? If yes, explain a scenario that leads to it, if not, explain why not.

- We expect either to be possible, if we are performing a system call such as read or write, where we change a buffer from the user stack, we could be violating a COW page

How do you distinguish a cow page fault from an actual permission violation?

- We can create a flag in vpipage called is_cowpage and set it to true to help determine if its a cowpage or a readonly apge

Is there any synchronization necessary when handling a cow fault? How do you plan to handle multiple cow page faults backed by the same frame?

- Yes we can use kalloc to make sure the same frames are not allocated for two different writes
- We also use locks in the core_map_entry objects to make sure updating the ref_count is stable

After handling the cow, is it necessary to flush the TLB? Why or why not?

- Yes, we should flush the TLB as the vspace that tried to write to a frame will now have a new frame that it should referencing and writing to. The old mapings will still point to the other process's frame that does not have the permissions to write.

A parent may fork a child who then forks another child, how do you handle the nested fork case? Please explain what needs to be considered at `fork` time and at page fault time.

- In the nested fork case, we just increase the number of references to each frame. With our implementation it doesn't matter if the process trying to write to the COW frame is a parent or child, they still will be allocated a new frame and copied over.

Corner Cases

- Out of memory - this is not a cowfork issue, we apply the stack growth criteria here
- Concurrent COW faults on the same page if two processes attempt to write to the same COW page - kalloc ensures each process will not try to write to the same frame
- Race conditions - we use a lock to protect the refcount

Test Plans

- Cow_fork:
  - Tests memory efficiency by allocating 200 pages and checks that it does use less memory than traditional full copy
  - Tests read only sharing behavior by testing reads from 200 allocated pages after fork and verifies it doesn't trigger cow
  - Kernel - mode COW handling - cow works when a write is performed by the kernel
  - Write COW bheavior - tries to write to all 200 pages and verifies proper cow allocation
  - Proper memory cleanup

Risk Analysis

Unanswered Questions

- What will happen to performance if multiple processes share a large amount of COW pages
- What is the optimal stack size? Why did we choose 10?

Staging of Work

- Sbrk
  - Sys_sbrk
  - Test sbrk_small and large test
  - Verify malloc works with malloc_test
- Stack growth
  - Modify trap.c to detect page faults
  - Implement stack growth with vregionaddmap/vspaceudpate/vspaceinstall
  - 10 page stack limit
  - Test with outlined tests
  - Ref count - modify kalloc and kfree to handle ref count
- COW
  - Implement vpsacecowcopy
  - Modify the vpage_info struct
  - Change fork to use vspacecowcopy
  - Test with outlined tests
  - Fault handling in trap.c to handle COW page faults

Time Estimation

- Sbrk (4-5 hrs)
- Stack growth (4-5)
- COW (5)
- Debugging (8)