Assignment 3 Design Doc

Team Longcat: Abby Lyons, Timothy Tamm

Introduction

For this assignment, our goal is to implement TLB handling and paging. We plan on implementing a two-level page table for every process for virtual-to-physical page translation. For physical-to-virtual translation, we will have a global coremap that contains the page table entry's location and other information about the page. We also need to keep track of swapped-out pages with a global swap space tracker that holds the highest used swap offset and a list of free offsets. Finally, we will implement a TLB shootdown struct to keep TLBs on different cores up-to-date. In addition to those data structures, we will also implement functions that handle address space operations, page faults, and swapping.

Data structures

Swap space tracker

We will keep track of swap space usage using a bitmap. The size of the bitmap will be the size of the swap device divided by 4k bytes. This is protected by a spinlock.

When a page needs to be written to swap space, then we will look through the bitmap and find the first free slot. All writes and reads from the swap space have to be protected by the spinlock.

Here are the structs for the swap space tracker:

```
/*
 * Data structure for tracking the swap space
 */
struct swap_tracker {
    struct bitmap *st_bitmap;  /* Bitmap to keep track of used blocks */
    struct spinlock st_lock;   /* Lock for this structure */
    struct vnode *st_vnode;    /* vnode of the swap device */
    int st_size;               /* number of blocks */
};
```

Address spaces

An address space consists of the following:

- A page directory
- The current heap size
- The start address of the heap
- A lock

The first 128 PDEs (starting at KERNEL_PT_START) of every process is going to be reserved for the kernel. The kernel pages don't have a pagetable.

Page directory/table

For each process, there will be two levels of page tables, such that a virtual address consists of a 10-bit page directory index, a 10-bit page table index, and a 12-bit offset. Each page directory is an array of 1024 entries.

PDE

Each page directory entry contains a 32-bit page table address.

PTE

Each page table entry contains a 20-bit physical page number, a valid bit, a writable bit and a present bit.
Here are the relevant structures.

```
/*
 * page table entry - data structure for keeping track of
 * virtual memory pages
 */
struct pt_entry {
    unsigned pte_ppn:20;        /* physical page number */
    unsigned pte_valid:1;       /* is page valid */
    unsigned pte_writeable:1;   /* is page writeable */
    unsigned pte_present:1;     /* is page in phys ram */
    unsigned pte_zeroed:1;      /* is page zeroed */
    unsigned pte_padding:8;     /* padding */
};

/*
 * pgtable - data structure for keeping track of virtual memory pages
 */
struct pgtable {
    struct pt_entry pt_ptes[PT_SIZE];    /* page table entries */
};


struct addrspace {
#if OPT_DUMBVM
    vaddr_t as_vbase1;
    paddr_t as_pbase1;
    size_t as_npages1;
    vaddr_t as_vbase2;
    paddr_t as_pbase2;
    size_t as_npages2;
    paddr_t as_stackpbase;
#else
    /* Put stuff here for your VM system */
    struct pgtable *as_pd[PD_SIZE]; /* the page directory of the addrspace */
    size_t as_heap_size;            /* the current heap size */
    vaddr_t as_heap_start;          /* start addess of the heap*/
    struct lock *as_lock;           /* lock to protect this struct */
#endif
};
```

Core map

The coremap contains an entry for each physical page. Each entry will contain:

- A pointer to the address space.
- A virtual address.
- Where in the swap device this page is backed.
- Which CPU the page belongs to.
- Whether the page is dirty or not.
- Whether the page is in the TLB.
- Whether the page is currently being accessed (busy bit).
- A kpage bit which indicates if this is a kernel page or not
- A kernel bit, which will be used to keep track of contiguous kernel memory blocks. The kernel bit should be set as 0 for most pages. If a block of more than 1 pages gets allocated using alloc_kpages, then the kernel bit will be set to 1 for every page after the first one in the contiguous block.
- A bit that indicates if the page exists in the device.

The core map will be protected by one spinlock. The core map will also have a wait channel. The core map will also keep track of the total number of physical pages that exist, the total number of kernel pages, the number of dirty pages and a clock head used for our paging algorithm.

Here are the relevant structs:

```
/*
 *  Struct for a core map entry
 */
struct cm_entry {
    struct addrspace *cme_as;   /* pointer to the address space that owns this page */
    vaddr_t cme_vaddr;          /* the virtual address in the address space */
    int cme_swap_location;      /* location of this page in the swap device */
    struct cpu *cme_owner_cpu;  /* which cpu the thread which has this PTE runs on */
    unsigned cme_dirty:1;       /* whether page has been written to */
    unsigned cme_tlb:1;         /* whether page is in tlb */
    unsigned cme_busy:1;        /* whether page is busy */
    unsigned cme_kernel:1;      /* whether page is in a contiguous kernel block */
    unsigned cme_kpage:1;       /* whether page belongs to the kernel */
    unsigned cme_exists:1;      /* whether page exists in ram */
};

/*
 *  Struct of the core map.
 *  The core map is responsible for keeping track of physical pages
 *  ram pages will be 512MB / 4KB
 */
```

```
struct coremap {
    struct cm_entry cm_entries[RAM_PAGES];  /* the core map entries */
    struct spinlock cm_lock;                /* lock protecting this struct */
    struct wchan *cm_wchan;                 /* wait channel for core map busy bits */
    struct wchan *cm_tlb_wchan;             /* wait channel for core map tlb bits */
    int cm_num_pages;                       /* number of existing pages */
    int cm_num_kpages;                      /* number of existing kernel pages */vim
    int cm_num_dirty;                       /* number of dirty paged */
    int cm_clock_head;                      /* clock head for paging algo */
};

/* This is the structure for the kernel coremap*/
extern struct coremap *k_coremap;
```

Struct tlbshootdown

The tlb shootdown struct needs to contain the following information:

- Which cpu's TLB to shootdown
- A virtual address to shootdown.
- A flush_all boolean variable. If this is true, then flush the entire TLB.

VM Functions

These are functions declared in vm.h, initially implemented in dumbvm.c, and newly implemented in a new file, vm.c.

Swap_init (new function)

- Initialize the swap space tracker
- Initialize the swap spaces spinlock.
- Call **vfs_swapon**, passing in "**lhd0:**" as the device name to initialize the swap device
- Store a pointer to the swap devices vnode in the tracker.
- Get the size of the swap space and store it in the tracker.

Swap_find_free

- Use bitmap alloc to find a free slot and return it

Swap_read/write

- Check if swapping is allowed

- Make a kernel facing uio
- Use VOP_READ/WRITE to read/write between the passed in physical page and the swap device

Vm_bootstrap

This is called in main.c. We want to initialize all of the global data structures described above.

- Initialize the metrics storing struct
- Swap space usage tracker initialization (call **vm_swap_init**)
- Core map initialization
  - Figure out how many physical pages there are.
  - Calculate the size of the coremap, and call **ram_stealmem** with enough pages.
  - Use **ram_getfirstfree** to find the first free address
  - Create a spinlock.
  - For each entry below the first free index, set all bits low, except for kpage and exists. Set the kernel bit 0 for the first page and 1 for each other one.
  - For each following entry until last_existing_entry set all bits low except for exists
  - For each other entry set all bits low.
  - Set the number of pages in the coremap
  - Create the 2 coremap wait channels.

Vm_fault

When a TLB fault occurs, then a handler will get called in exception-MIPS1.s, which in turn calls **mips_trap** in trap.c, which finally calls **vm_fault**. In **vm_fault** we have to do the following:

- If there's no current process or address space, then the fault must have happened early in the boot process and there's probably nothing we can do about it. Panic.
- Check to make sure the arguments are valid.
- Acquire the AS's lock, and make sure there's a pagetable corresponding to the faultaddress. If there isn't create one.
- Acquire the core map lock, and call **pte_acquire** on the PTE.
- Check if the PTE is a valid and present entry; if not, raise a page fault (see **page_fault**) to allocate space in memory for it.
  - If **page_fault** fails, release all locks, call **pte_release**, and return the error code.
- Disable interrupts after this point.
- Get the CME of the potentially new page in memory, and set its busy bit high.
- Make sure the page is writable if a **VM_FAULT_WRITE** or **VM_FAULT_READONLY** fault happened. If not, release all locks, enable interrupts, and return **EFAULT**.
- Create a new TLB entry!
  - Look for an existing TLB entry for this page using **tlb_probe**.
  - If there isn't one, pick a random index. Find the existing entry using **tlb_read** if there is one, setting its **cme_tlb** bit to 0 and waking up anything waiting on the tlb waitchannel.

    - Set **entryhi** to the **faultaddress** and **entrylo** to **PADDR | TLBLO_VALID**.
    - If the fault is **VM_FAULT_WRITE** or **VM_FAULT_READONLY**, set the coremap entry so the page is dirty, and **entrylo |= TLBLO_DIRTY**.
    - Call **tlb_write**.
- Clean up everything by setting the CME's busy bit to 0, waking up anything waiting on busy entries, calling **pte_release**, releasing the coremap and address space locks, and enabling interrupts.

Alloc_kpages

- Acquire the coremap's spinlock.
- Check to see that the number of pages asked for does not over-allocate kernel pages, since we want to make sure that userland still gets some pages. If too many pages will be allocated, release the spinlock and return 0.
- Set **start_of_block** to -1 and **cur_block** to 0.
- Attempt to acquire **npages** contiguous free entries **NUM_TRIES** times:
    - Loop through the entire coremap, looking for the contiguous free entries.
    - If we found **npages**, break out of this loop.
    - Otherwise, if we only asked for 1 page, directly call **page_get** to evict one user page. If it returns a valid page number, break out of this loop.
    - Finally, attempt to find a free block on the next pass by evicting **npages - pages_found** pages using **page_get**, and unsetting their busy bits.
    - Reset **start_of_block** before continuing.
- If we couldn't find a block of pages (**start_of_block** is -1), release the coremap lock and return 0.
- Loop through the entire block starting at **start_of_block**, and set their CME entries so the address space is NULL, the virtual address is the kernel virtual address of the block, the swap location is 0, and the owner CPU is 0. Also make sure that the dirty, tlb, and busy bits are all unset. Lastly, the kpage bit should be set to 1.
    - Make sure the kernel (NOT kpage) bit is set to 0 if it's first in the block.
- Call as_zero_region on the block.
- Increase the number of kernel pages (**k_coremap->cm_num_kpages**).
- Release the coremap lock.
- Return the virtual address of the first block.

Free_kpages

- Assert that the virtual address is in the kernel.
- Acquire the coremap lock.
- Find the physical page number of the virtual address, and assert that it's a valid kernel page and the first in the block.
- Unset the kpage bit.
- For each subsequent page in the block (**cme_kernel** == 1):

    - Unset the kpage and kernel bits.
    - Decrease **k_coremap->cm_num_pages**.
- Release the coremap lock.

Vm_tlbshootdown

This function should get called whenever a page gets evicted, or if **sbrk** gets called with a negative argument. Here's what has to happen:

- If the target cpu is not the current cpu, then call **ipi_tlbshootdown** with the given tlbshootdown struct and target cpu.
- Otherwise:
    - Turn off interrupts.
    - Check the **flush_all** variable. If it's true:
        - Acquire the coremap lock if you don't already hold it.
        - Loop through all **NUM_TLB** entries, reading each one.
        - If the read TLB entry was valid, set its coremap entry's **cme_tlb** to 0.
        - Use **tlb_write** to invalidate each entry.
    - Else:
        - Call **tlb_probe** to look for an existing entry. If it's not there, go to cleanup.
        - Call **tlb_read** on the found index, and use the resulting **entrylo** to set the **cme_tlb** bit to 0.
        - Use **tlb_write** to invalidate the entry at that index.
    - Wake up everything waiting on the TLB waitchannel.
    - Cleanup:
        - Release the coremap spinlock if it was acquired.
        - Turn on interrupts.

Pagetable Functions

pgt_init

- For each entry in the given pagetable, set **pte_valid** to 0 and **pte_padding** to 0.

pgt_destroy

- For each entry in the pagetable:
    - If it's valid and present:
        - Call **pte_acquire**. If it returns -1, call **swap_destroy_block** and continue.
        - Acquire the coremap spinlock.

        - For the corresponding CME, set **cme_tlb** to 0, **cme_dirty** to 0, **cme_vaddr** to 0, and **cme_as** to NULL.
        - Release the coremap spinlock and call **pte_release**
        - If it has a swap location, call **swap_destroy_block**.
    - If it's just valid:
        - Call **swap_destroy_block**.
- Free the pagetable.

AS Functions

As_create

- Create a new address space that contains an empty page directory.
- Initialize the address space lock.
- Set the current heap size to 0.
- Return the address of the new address space.

As_copy

- Call **as_create** to make a new address space. If this returns NULL, return **ENOMEM**.
- Lock the old address space.
- Iterate through all page directory entries in the old address space.
    - If the pagetable belongs to the kernel, continue.
    - If the pagetable is NULL, continue.
    - Iterate through all PTEs in the table.
        - If **pte->pte_valid** is 0, continue.
        - Call **pte_acquire**.
        - Acquire the coremap lock.
        - If the page is only in swap (**pte_present == 0**), bring it into memory by calling **page_swapin**. If there was an error, call **pte_release**, and release the coremap and address space locks before returning the error.
        - If **pte_zeroed** is false:
            - Call **page_get** to find a new spot in memory for copying the contents.
            - Set **cme_as** of the new page to the new address space, and set **cme_owner_cpu** to the current cpu.
            - Set **cme_busy** to 1, and set **cme_exists** to 1.
            - Release the coremap lock.
            - Call **memmove** to copy over the contents.
            - Reacquire the coremap lock.
        - Release the coremap lock.

        - Allocate a new pagetable in the new address space if necessary, and call **pgt_init** on it.
        - Make a new PTE for the page that was just copied over.
        - Call **pte_release** on the old address space's PTE.
- Acquire the coremap lock again.
- For every new page in the coremap (check that **cme_as** == newas):
    - Set the busy bit low.
- Set the new address space's heap size and heap start to equal the old address space's.
- Release the old address space's lock.
- Set ret to the new address space
- Return 0--we're done!

As_destroy

- Acquire the AS lock.
- Call **pgt_destroy** on every valid pagetable.
- Release the AS lock.
- Destroy the lock.
- Free the address space.

As_activate

- Shoot down the current CPU's TLB completely (by setting **flush_all** true in the struct).
- Set the current active AS pointer to point to the given AS using **proc_setas**.

As_deactivate

- Don't do anything.

As_define_region

- Check the arguments, and return if they're invalid.
- Acquire the address space lock.
- If L2 page tables do not exist at the given addresses, allocate them and update their respective PDEs. Error with **ENOMEM** and release the lock if this fails.
- For each newly created L2 page table, call **pgt_init**.
- For each already existing page in the given range:
    - Call **pte_acquire**.
    - Set the writable bit high if the argument is nonzero; otherwise, set the writable bit low.
    - If you changed the writable bit in the PTE:
        - Acquire the coremap lock
        - Check if the page is in the tlb

        - If if is then shoot down that tlb entry and lower the **in_tlb** bit
        - Release the coremap lock
    - Call **pte_release**.
- Release the address space lock.

As_prepare_load

- This function doesn't need to do anything.

As_complete_load

- This function doesn't need to do anything.

As_define_stack

- Set the stackptr to **USERSTACK** and return 0.

As_zero_region

- Call **bzero** on the given vaddr and npages * **PAGE_SIZE**.

Pte_acquire

- Make sure we're holding the address space lock.
- If the given PTE is present:
    - acquire the coremap lock
    - Find its coremap entry
    - If it's busy, sleep until it's not busy.
    - If the page was evicted in the meantime, release the coremap lock and return -1.
    - Else, set the busy bit to 1, release the coremap lock, and return the physical page number.
- Else, return -1.

Pte_release

- Make sure we're holding the address space lock.
- If the given ppn is valid:
    - Acquire the coremap lock.
    - Unset the busy bit on the CME.
    - Wake up anything waiting on the coremap's waitchannel.
    - Release the coremap spinlock.

TLB replacement algorithm

We will use a random TLB replacement algorithm. This means that if a TLB entry needs to be evicted, then out of the clean pages we just randomly pick one.

Paging

The following functions will be in a new file, paging.c (and paging.h).

page_fault

This function mostly handles argument checking and stats.

- Assert that we are holding the address space lock and coremap spinlock.
- Handle invalid addresses by calling kern__exit.
- Call **page_swapin** and return whatever it returns.

page_swapin

This function handles updating the coremap entries and pagetable entries of pages swapped into memory.

- Assert that we are holding the address space lock and coremap spinlock.
- Call **page_get** to find a free location in memory. If this fails, return **ENOMEM**.
- Get the CME of the new page, and make extra sure that it's not a kernel page.
- If the PTE indicated by the virtual address says that the page is in swap and not zeroed, release the coremap lock, call **swap_read** to bring it into the new page, and reacquire the coremap lock.
- Else, call **as_zero_region** on the new page.
- Update the coremap entry so **cme_as** is the current address space, **cme_vaddr** is the virtual address, **cme_busy** is 0, and **cme_swap_location** is the swap location from the PTE entry.
- Update the PTE by setting **pte_ppn** to the new page number, **pte_dirty** to 0, and **pte_present** to 1.
- Wake up anything waiting on the coremap's waitchannel.
- Return 0.

page_get

This function looks for a free or clean page, and evicts a page if it can't find one.

- Assert that we are holding the coremap lock. (Not the address space lock, since this is also called in **alloc_kpages**.)
- Find a physical page using our paging algorithm. This will call **page_write_out** if it decides that a page needs to be evicted.
- Find the CME and PTE of the clean page.
- Call **vm_tlbshootdown** on its virtual address and owner CPU.
- Wait for its **cme_tlb** bit to be set to 0.
- Update the PTE so it's not present and **pte_ppn** is equal to its swap location.
- Remove the CME from the coremap by setting everything (except **cme_exists**) to 0.
- Return the clean page's page number.

page_write_out

This function writes a page out to swap, but doesn't evict it. This is done on purpose, because it's also used by the swap daemon.

- Assert that we are holding the coremap lock.
- Figure out where to write out the page.
    - If its CME has a swap location, use that.
    - If not, call **swap_find_free** to find an unused location. This will mark the swap location as used. If no location is found, return ENOMEM.
    - Set **pte_zeroed** to 0.
- Release the coremap lock.
- Call **swap_write**.
- Reacquire the coremap lock.
- Set **cme_dirty** to 0, and set **cme_swap_location** to the swap location.
- Call **vm_tlbshootdown**, and wait for the **cme_tlb** bit to be unset.
- Return 0.

Swapping

The following functions will be in a new file, swap.c (and swap.h).

Swap_init

This is called during bootup only.

- Allocate space for the swap tracker.
- Initialize a spinlock for it.
- Figure out the size of the swap disk. Set its size to that divided by the **PAGE_SIZE**.
- Call **bitmap_create**, and **bitmap_mark** the 0th index so it's never used.

Swap_find_free

This finds a free swap location and marks it as used.

- Acquire the spinlock for the swap space.
- Call **bitmap_alloc**.
- Release the spinlock.
- Panic if it returns an error.
- Return bitmap_alloc's found index.

Swap_read

This reads data from a location in swap to a physical page.

- Create a new uio.
- Assert that **bitmap_isset** for the given swap location.
- Return **VOP_READ**'s return value.

Swap_write

This writes data from a physical page to a location in swap.

- Create a new uio.
- Assert that **bitmap_isset** for the given swap location.
- Return **VOP_WRITE**'s return value.

Swap_destroy_block

This sounds impressive, but really it just unmarks something in the bitmap so the swap location can be reused.

- Acquire the spinlock.
- Check **bitmap_isset**, and if it is, call **bitmap_unmark**.
- Release the spinlock.

Swap daemon

- Check that we've reached our limit for the swap daemon to run.
- Acquire the coremap spinlock.
- Check all physical pages and find the dirty ones.
- Call **page_write_out** if the page is dirty, and then mark it as not busy.
- Wake up everything on the coremap's waitchannel.
- Release the coremap spinlock..
- Call **thread_yield**.

Thread migrations

In **thread_consider_migration** in thread.c, add the following:

- After adding a thread to the tail of the other CPU, iterate through the coremap, find every entry that belongs to that thread, and update which CPU that page belongs to.

Paging algorithms

Question

Q: How will you structure your code (where will you place your paging algorithms) so that others can be added trivially, and switching between them only requires reconfiguring your kernel?
A: Most of our code for paging will be in a file called paging.c. In paging.c we will have a function called **find_page_to_evict**, which use kernel macros to call one of multiple paging algorithms
Ex:

```
#ifdef USE_PAGING_ALGO_1
Page_to_evict = paging_algo1();
#endif
#ifdef USE_PAGING_ALGO_2
Page_to_evict = paging_algo2();
#endif
```

This way the paging algorithms can be written in separate files.
Finally, at the top of paging.h, we will have a macro for each of these paging algorithms, but all except for one should be commented out.
Ex:

```
//#define USE_PAGING_ALGO_1 1
//#define USE_PAGING_ALGO_2 1
#define USE_PAGING_ALGO_3 1
//#define USE_PAGING_ALGO_4 1
```

In this example, the third paging algorithm will be used.

Algorithm

We will use a clock based algorithm to approximate a LRU paging algorithm
The coremap will have a 32bit integer acting as a clock. It will be initialised to 0 and it will be incremented every time anything touches the coremap. Each core map entry will also have a

last_clock variable. Whenever a coremap entry is touched, set its last_clock value to the current coremap clock value.

To choose an entry to evict, do the following If at any step there is only one page left, use that one:

1. Go through the coremap, and find all the clean pages
2. Go through all the clean pages and find all the non-busy ones
3. Among the pages left over, choose the one with the lowest last_clock value.

Performance analysis

We have updated some of our functions to increment certain statistics whenever they are called; see **vmstats.h** for more details. We implemented two paging algorithms: one attempts to find a free or clean page (returning the last clean page) and picks a random page if it can't, and the other is the LRU paging algorithm described above. The paging algorithm can be changed by commenting or uncommenting the macros at the top of **paging.h**.

For the following statistics, we ran 5 trials with 2M of ram and 10M of swap space, rounding the averages to the nearest whole number. Generally, the LRU-like paging algorithm performs better in at least one statistic; when running sort, it incurs significantly fewer VM faults and TLB shootdowns, while in matmult, it has significantly fewer VM faults and only slightly more TLB shootdowns, and in parallelvm, it has significantly fewer TLB shootdowns and about the same VM faults. Unfortunately, there aren't any parameters to change in this algorithm, so this is the best it can do.

There are a number of variables in our random algorithm; for example there is a 1 in 10 chance that it will write out a dirty page instead of picking the clean one that it found. If we raise this probability, then there are more synchronous writes; however, lowering it might increase runtime due to the same two clean pages being evicted back and forth. We can also change the algorithm to return the first clean one instead of the last one, but this doesn't affect the performance at all.

Matmult:

	LAST_CLEAN_PAGING	CLOCK_PAGING
Page faults	383	383
Page faults w/synchronous write	0	0
VM faults	1382	941

TLB shootdowns	150	171
Daemon runs	1	1

Sort:

	LAST_CLEAN_PAGING	CLOCK_PAGING
Page faults	293	293
Page faults w/synchronous write	0	0
VM faults	7263	5151
TLB shootdowns	717	532
Daemon runs	3	3

Parallelvm:

	LAST_CLEAN_PAGING	CLOCK_PAGING
Page faults	69	49
Page faults w/synchronous write	15	15
VM faults	8502	8781
TLB shootdowns	7465	5211
Daemon runs	1	1

Questions

TLB Questions

Problem 1

- TLB miss, page fault: a page was swapped out to disk and evicted from the TLB.
- TLB miss, no page fault: a page is in physical memory, but was evicted from the TLB.
- TLB hit, page fault: cannot happen
- TLB hit, no page fault: a page is in physical memory and the TLB.

Problem 2

- An instruction that references something in memory gets executed. First the TLB entry references the place in memory where the code for the instruction is stored. Then a TLB Fault occurs, because the piece of memory that the instruction references is not in the TLB. It gets pulled into the TLB (and the code gets evicted). Now the instruction gets executed again, but now that is no longer in the TLB, so you get another TLB fault. So it gets pulled into the TLB, the memory that the instruction references gets evicted, the instruction gets re-executed and we're right back where we started.

Problem 3

- Accessing this instruction can cause:
  - TLB miss, if the virtual page containing the instruction is not in the TLB.
  - Page fault, if the physical page is not in physical memory.
  - Permission fault, if the read and execute bits are not set.
- Accessing the data at 0x0120 can also cause:
  - TLB miss, if the virtual page is not in the TLB.
  - Page fault, if the physical page is not in physical memory.
  - Permission fault, if the read bit is not set.
  - Invalid page fault, if the address is not in the virtual address space or is otherwise an illegal access.

Malloc Questions

- Question 1
  1. Depending on the state of the system before, sbrk might get called 0 times or once. If free blocks can't be found in the current state of the system, then sbrk will get called once with the page size (4K). That would be more than enough memory for the 10 malloc calls (each malloc call asks for 10 bytes, but because memory is given in 8 byte blocks, each call actually needs 16 bytes of memory, so a total of 160 bytes).
  2. If sbrk got called, then the value is 4096 bytes. If not, then it's 0 bytes.
- Question 2

    - If sbrk was called before section 2, then it will not be called again. However, if it was not, then it will be called once with the page size (4K) if there are no free sections of 64 bytes remaining.
    - We can say that x must be greater than res[i] for 0 <= i < 10. None of the freed blocks add up to 64 bytes, so the blocks allocated for x must be after all of those.
- Question 3
    - This designates that these functions are supposed to only be used internally. In a higher level language you would declare these functions private, but in C you can't do that, so you rely on naming conventions.
- Question 4
    - This is guaranteed by 2 facts:
        - The heap starts off properly aligned
        - Malloc only allocates memory in blocks of 8 bytes (on a 32bit system) or 16 bytes (on a 64bit system).

Sbrk syscall

The sbrk syscall should:

- If the size parameter is greater than 0:
    - Check if the new heap bound would cross **STACK_MIN**. If it does, then error with **ENOMEM**.
    - Allocate the desired number of physical pages for the virtual address growing up from where the current heap is at (this info is stored in the address space)
    - Update the current heap bound in the address space.
    - Return the new heap bound.
- If the size is less than 0:
    - If the new heap bound would make the heap size less than 0, then error.
    - Free any pages that are between the new and old heap bounds. To free a page, delete its PTE, clear its core map entry if it was in physical memory and if it was in physical memory and in the TLB, then call **tlb_shootdown**.
    - Update the heap bound and return the new heap bound.

Plan of Action

We plan to start working on our assignment halfway through spring break

1st week:
F: implement supporting data structures and APIs
Sat: addrspace functions
Sun: addrspace functions

1st week:

M: vm functions
Tu: vm functions (at this point, we should hopefully be able to run our kernel without paging)
W: paging
Th: paging
F: writing daemon
Sat: testing
Sun: testing

3rd week:
M: testing
Tu: testing
W: paging algorithm optimization
Th: paging algorithm optimization
F: paging algorithm optimization
Sat: Discover joy in life once more.
Sun: partayh!

Credits

We would like to thank Prof. James Mickens for teaching us the concepts, and Alex Patel, Victor Domene, and Team Streptococcus pyogenes for helping us refine our design, and all of the TFs for helping with the copious amounts of debugging we had to do.