Computer Science 161: Operating Systems
Assignment 3 Design Document[1]

David Li and Garrett Tanzer

March 10, 2017

1. Introduction

Assignment 3 can be broken up into three major data structures (page tables hierarchies, the core map, and swap) and the operations performed to maintain them (populating/evicting entries from the TLB, swapping pages from disk to RAM as needed, and asynchronously writing dirty pages to disk with a daemon).

We will implement 2-level page tables, with each level indexed by 10 bits of the virtual address; 12 physical bits are then added to the end to get a 32-bit physical address. The core map will be an array of entries for each page of physical RAM with its own bookkeeping information. Swap, twice the size of physical RAM (rounded up to the nearest page) will be tracked using a bitmap. Each address space, the core map, and the swap bitmap will be protected by a coarse-grained spinlock. Each entry of the core map and PTE is also protected by a busy bit, which can be set while coarse synchronization is held for each, in order to prevent accesses during the swap process.

TLB eviction will be largely random. Page swapping will be a variant of the clock algorithm, dependent in part on presence in the TLB. The write-back daemon will loop over the core map sequentially and save everything in RAM to disk when the pages are not the latest version. The main challenge with these operations performing in parallel is synchronization; our algorithms and the primitives we use to ensure they are performed atomically are explained in much greater detail below.

[1] Based on this Piazza post (https://piazza.com/class/iy1oku7camk18k?cid=159), our current design document does not have a section explicitly outlining how performance optimization will happen because it is supposedly only needed in the final submission. In addition to simply running testbin programs to fine-tune parameters of TLB eviction, page swapping, and the write-back Daemon, we can add stat-tracking to see how many page faults, TLB misses, swaps, etc. each strategy has, and we could even time how long it takes to resolve each of them. This collection of metrics would more facilitate much more meaningful analysis than the time to run a certain program.

2. Overview

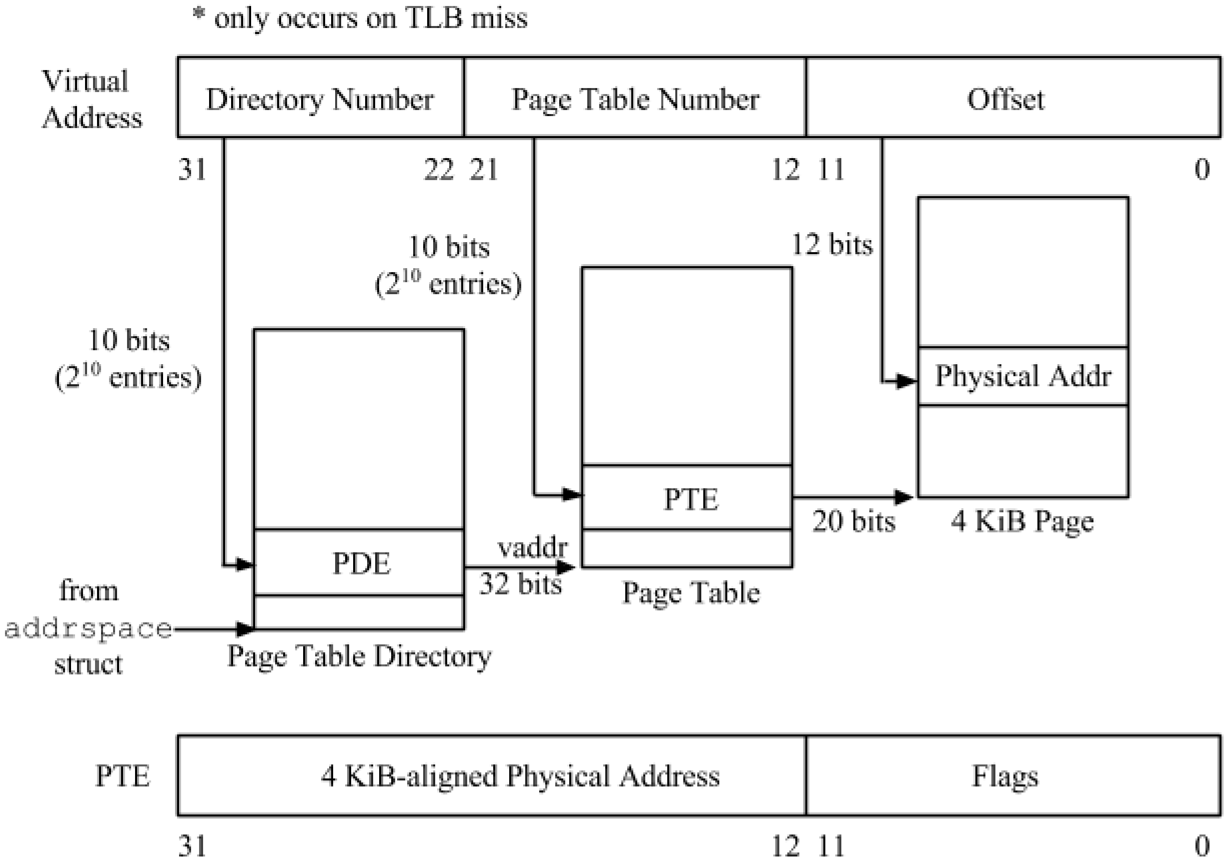
2.1 Page Tables

The data structures for virtual to physical address translation begin in the `addrspace` struct:

```
struct addrspace {
      struct page_table_directory* ptd;
      struct spinlock addr_splk;
      struct wchan *addr_wchan;           // for waiting during swap I/O
      vaddr_t heap_bottom;
      vaddr_t heap_top;
};
```

`ptd` provides an entry point into the hierarchical page tables, while `addr_splk` protects modifications to this address space. `heap_top` is manipulated by `sbrk()` and is used to check for invalid memory accesses.

Here is a diagram showing how two-level page tables are indexed to reach a full 32-bit physical address:



The ptd field of addrspace is a wrapper for an array of page_table pointers (physical addresses of subsequent page tables or NULL if not yet created). Each page_table has an array of 4-byte entries, which store a 20-bit physical frame and 12 bits of flags. Both page table levels are designed to take exactly one page of memory each so that they can fill exactly one use of alloc_kpages().

```
struct page_table_directory{
      struct page_table* pts[1024];
};
```

```
struct page_table {
      uint32_t ptes[1024];
};
```

Each of the PTEs in page_table is organized as follows:

PADDR/SWAP								PTE_X	PTE_R	PTE_W	PTE_B	PTE_P
31-12	11	10	9	8	7	6	5	4	3	2	1	0

PTE_P: 1 if present in memory, 0 if in swap

PADDR: If PTE_P, the physical address is stored in the top 20 bits.

SWAP: If not PTE_P, the index of the page in swap is stored in the top 20 bits.

PTE_B: 1 if busy, 0 if not

PTE_W: 1 if writeable, 0 if not

PTE_R: 1 if readable, 0 if not

PTE_X: 1 if executable, 0 if not

Rather than having to manually traverse the page table with curproc->p_addrspace->ptd->pts[index1]->ptes[index2], we will define a function as_pte(vaddr_t vaddr) that returns the PTE mapped by the virtual address parameter in the current address space (or a negative error value if it has not yet been mapped).

2.2 Core Map

The core map is a global array with bookkeeping data on all page frames in RAM. It will have ram_getsize() / PAGESIZE entries and must be initialized early in the bootstrap process using ram_stealmem(), since the core map is required for kmalloc() to work. We will need to mark the RAM used by the core map in the core map itself. The entire array will be protected by a single spinlock. Each entry will be protected by a busy bit.

```
struct core_map_entry {
      vaddr_t va;
      struct addrspace *as;
      uint32_t metadata;
};
```

`va` and `as` are used to locate the page table entry pointing to the physical address, as well as to acquire the spinlock for the address space.

Each `metadata` in the core map is structured as follows:
Our final code will probably use bit fields to represent this instead for simpler flag access.

SWAP	...	...	RECENT	TLB	DIRTY	CONTIG	KERNEL	BUSY	S_PRES
12-31	11-8	7	6	5	4	3	2	1	0

S_PRES: 1 if copy already exists in swap, 0 if not
BUSY: 1 if this page is involved in swapping right now, 0 if not
KERNEL: 1 if this page belongs to the kernel, 0 if to the user level
CONTIG: 1 if this page is the end of a contiguous kernel allocation, 0 if not
DIRTY: 1 if this page has been written to since the last flush to swap, 0 if not
TLB: 1 if this page's translation is currently in a TLB, 0 if not
RECENT: 1 if recently evicted from a TLB, 0 if not
SWAP: Index of the corresponding page on the disk: Assuming a max swap size of 1 GiB (2 * 512 MiB max RAM in kseg0)

2.3 Swap

The swap space is a region of the hard drive that stores pages either already evicted from RAM or preemptively copied from RAM, in anticipation of the page needing to be evicted from RAM. We will keep track of occupied/free sectors on the swap device using a `struct bitmap` of length 2 * `ram_getsize() / PAGESIZE` bits, rounded up to the nearest page. So, with physical RAM limited to 512 MiB, this means our swap device can be between 128 and 1024 MiB inclusive. This space was made larger than RAM to ensure that even if all the pages are preemptively sent to swap, there will still be space for others to actually be swapped in/out.

The bitmap will be protected by a single spinlock. Whenever a new swap index is needed, the first available 0 in the bitmap is switched to 1 under the spinlock's protection.

Read/write to swap using the vnode generated by `vfs_swapon(“lhd0raw”).`

3. Topics

3.1 TLB Misses

The TLB miss handler needs to populate the TLB with the translation for a requested virtual page. This apparently simple task becomes exceedingly complicated when you consider swap and synchronization. Not only must the handler respond to requests for invalid addresses, but for valid requests it must make sure the requested page is in memory (and if not, swap it in, which may require swapping out another page if memory is full), writing that translation to the TLB, and updating bookkeeping data.

Our specific pseudocode implementation is below, but in general terms our strategy for avoiding deadlock and race conditions is to only acquire spinlocks in the order address space > core map > swap and to release those spinlocks as soon as possible (using busy bits in PTEs and the core map for loosely enforced polling synchronization afterwards).

When `VM_FAULT_READ` or `VM_FAULT_WRITE` are triggered:

```
●  If the faulting address is not in user space:
     ○  Kill the thread with a segfault
●  Acquire the address space's spinlock
●  If the faulting address is unmapped but a valid address (i.e. in stack/heap bounds):
     ○  Call alloc_upages to allocate a new page
●  If the faulting address is mapped, but it's read only and you got VM_FAULT_WRITE:
   (You know that once you populate the TLB with the new entry it'll just fail when the
   instruction runs again, so why not catch it now and save the time?)
     ○  Release the address space spinlock
     ○  Kill the thread with a segfault
●  While PTE_B is 1:
     ○  Sleep on the original address space's wchan with its spinlock
●  If the PTE_P is not set
     ○  Set PTE_B to 1
     ○  Release the address space lock
     ○  Acquire the core map spinlock
     ○  Iterate through core map, looking for free entries
     ○  If free entry found:
          ■  Save the index of the core map entry
     ○  Else:
          // swap in/out will probably be helper functions in the final code, but it's
          easier to follow synchronization if it's included here
          // there will probably be an optional parameter to leave pages BUSY
```

- Iterate through the core map starting from a global index variable `clock`, which loops back to the beginning of the array when it reaches the end
  - If `TLB` or `BUSY`, continue (as in skip to the next iteration of the loop)
  - If `RECENT`, set `RECENT` to 0 and continue
  - On the second revolution through the clock, start accepting entries with a set `TLB` flag
  - If you reach this point in the loop without hitting “continue,” choose the current index to swap out to disk
- Set the `BUSY` bit on your chosen core map entry
- Release the core map lock
- Acquire the new address space spinlock (for the core map entry that you want to evict)
- Set `PTE_B` of the PTE referencing the core map entry
- If the `TLB` bit is set:
  - Broadcast a TLB shootdown
  - Go to sleep on the new address space’s wchan and its spinlock (TLB shootdown will wake all on its own addrspace’s wchan when it’s done)
- Release the new address space spinlock
- If `S_PRES` is set:
  - Set `PTE_P` to 0 and set `SWAP` index in the PTE to `SWAP` from the core map entry
  - If the core map entry is `DIRTY`:
    - Write the contents of the physical page to disk
- Else (the cache needs to be written to RAM):
  - Acquire the swap spinlock
  - Find a free spot in the bitmap
  - Mark it taken
  - Release the swap spinlock
  - Set `PTE_P` to 0 and set `SWAP` index in the PTE to the free spot
  - Write the contents of the physical page to disk
- Set `PTE_B` of the PTE to 0
- Acquire the core map lock
- Wake all on the new address space’s wchan

- Acquire original address space spinlock
- Set the core map entry’s `SWAP` to the value from the original (faulting) PTE
- Set the core map entry’s virtual address to the faulting virtual address
- Set the core map entry’s address space to the faulting address space

    - Put the physical address associated with the core map entry into the PTE
    - Release the original address space spinlock
    - Read data from disk at `SWAP` into the physical page of memory
        - We don't need to worry about the PTE being modified in this period because OS/161 is single-threaded, so any other attempted access to this specific PTE will first need to come through the core map entry, which is tagged as `BUSY`
    - Acquire the original address space spinlock
    - Set `PTE_B` on the original address space spinlock to 0
    - Set `BUSY` to 0 on the core map entry (because of the branches above it is possible that this was already 0, but it is impossible that another thread set this `BUSY`)
    - Wake all on the old address space's wchan
    - Save the index of the core map entry
- Holding a spinlock means interrupts are already off, so we don't need to worry about TLB state getting messed up
- Set `TLB` on the core map entry to 1
- Use OS-level random function to determine which TLB entry to evict (because `tlb_random()` only selects 56 of 64 entries)
- Read this randomly chosen entry from the TLB with `tlb_read()`
- While this randomly chosen entry's core map is `BUSY`:
    - Choose a different random one—dealing with someone swapping right now would be annoying
- Set `TLB` to 0 and `RECENT` to 1 for the randomly chosen entry's core map
- Write the replacement entry to the TLB with `tlb_write()`, with global and valid bits set (not writeable)
- Release core map spinlock
- Vector control back to the user level, with the instruction pointer at the faulting instruction

3.2 Page Faults

The page fault handler should respond to accesses with improper permissions. Sometimes these will actually be illegal accesses according to the PTE, in which case we should kill the offending thread, but most of the time these will be intentional faults in order to set the `DIRTY` bit for the swap write-back daemon. In this case, we mark the relevant bookkeeping information and update the TLB to allow writes so that the original instruction can go through.

When `VM_FAULT_READONLY` is triggered: (should only happen for valid TLB entries, but not writeable)

- Acquire the address space spinlock
- If the PTE_W isn't set in the PTE:
    - Release the address space spinlock
    - Kill the thread with a segfault
- Acquire the core map spinlock
- If the physical page is BUSY:
    - Release the core map spinlock
    - While the physical page is BUSY:
        - Go to sleep on the address space wait channel with its spinlock
    - Acquire the core map spinlock
- Mark the associated physical page's DIRTY bit
- Again, the core map spinlock turns interrupts off for us
- Update the TLB entry for the faulting address with a writeable bit using `tlb_write()`
- Release the core map spinlock
- Release the address space spinlock
- Vector control back to user space, with the faulting instruction repeated

3.3 MAT Daemon

The MAT (Memory Access Traversal (and Write-Back)) Daemon is a background process spawned during boot to proactively write dirty pages from RAM to the swap space. It will also update pages on the swap if they have already been placed onto swap. This kernel-mode process should be given a low priority in the scheduler since it is not crucial to the functioning of the operating system. MAT Daemon will `thread_yield()` after a certain amount of time, and because it uses spinlocks, it has to give up control voluntarily; the amount of time it takes to do so will be tuned based on performance once implemented. MAT Daemon will not run if no swap space is left. Below is the algorithm for the function:

- Acquire the core map spinlock
- Iterate through core map continually and backwards, from the greatest entry of the core map to the least entry of the core map, restarting after the lowest entry is hit. This backwards accessing is used to ease thrashing between the MAT Daemon and our clock swapping algorithm.
    - If the entry is KERNEL or TLB, continue
    - If the core map entry is DIRTY:
        - Set BUSY to 1
        - Release the core map spinlock
        - If not S_PRES: (need to find a swap index)
            - Acquire the swap spinlock

            - Find first free spot in the bitmap
            - Mark it taken
            - Release the swap spinlock
            - Set `SWAP` index in the core map entry to the free spot
        - Set `DIRTY` to 0
        - Write the contents of the physical page to disk at SWAP index
        - Set `BUSY` to 0
        - Acquire the core map spinlock
    - At some point decide to yield to the scheduler
        - Release the core map spinlock
        - `thread_yield()`

3.4 TLB Shootdown

The TLB shootdown handler will use `tlb_probe()` to find the requested entry then invalidate it. Once it finishes, it must wake all on its address space's wchan (to signal to the thread that issued the shootdown that it completed). Use the constants/masks defined in `tlb.h` to construct and make sense of TLB entries.

The struct `tlbshootdown` in `vm.h` will be defined as follows:

```
struct tlbshootdown {
      uint32_t oldentryhi;    // tlb_probe() doesn't need entrylo
};
```

3.5 Miscellaneous Functions

```
void* sbrk(intptr_t amount);
```

Extends the top of the heap for the address space (i.e. `heap_top`) by the `amount`, updating the page tables accordingly. We should create a constant to indicate the greatest extent of heap expansion.

- Get address space and take the spinlock
- Check page-alignment of amount

If amount is positive:

- Test for `ENOMEM` condition
- Increase `heap_top`

- Release address space spinlock

If amount is negative:

- Test for `EINVAL` condition using `heap_bottom`
- Get core map's spinlock
- Decrease `heap_top`
- Zero out pages in core map corresponding to virtual addresses between old and new heap_top, marking associated swap pages as free in the process
- Mark associated pages only in swap as available with the swap spinlock
- Remove entries marked `TLB` from the TLB (this is in the same address space, so since we're multithreaded, that means it's our TLB, which makes it much easier)
- Release core map spinlock

Return pointer to new `heap_top`

Errors:

- `EINVAL` for an `amount` that is either not page-aligned or would make `heap_top` below `heap_bottom`
- `ENOMEM` for insufficient virtual memory

```
struct addrspace* as_create(void);
```

Create a new empty address space (create minimum page table entries, initialize spinlock and wchan, NULL everything else). Return NULL if out of memory.

```
int as_copy(struct addrspace *src, struct addrspace **ret);
```

Do a deep copy of the `src` address space into a new address space saved to `ret`. During the process, the `src` spinlock must be held.

- First, `as_create()` (this might fail, in which case return the error)
- Acquire the `src` spinlock
- Iterate through the entire address space and for each mapped entry:
    - Get one new user page with `alloc_upages()` (this acquires the new address space's lock, but because it's not yet running on a thread, there is no worry about deadlock from holding the spinlocks for two address spaces)
    - If the PTE in `src` is in memory
        - `memcpy` from it into the new page
    - If the PTE is in swap

        - Read from swap directly into the new memory
- Release the address space spinlock
- Return success.

```
void as_activate();
```

Flush the TLB. Update the TLB flags in core map entries while doing so (with spinlock protection).

```
void as_deactivate();
```

Do nothing.

```
void as_destroy(struct addrspace *as);
```

Iterate through the address space and free all remaining referenced pages (using helper functions). Clean up the address space's spinlock, wchan, etc. Remember to `kfree` the addresses of the pagetables themselves; not just clear what they refer to.

```
int as_define_region(struct addrspace *as, vaddr_t vaddr, size_t
sz, int readable, int writeable, int executable);
```

Map the designated virtual memory region to physical memory with the specified permissions in user space.

- Align `vaddr` and `sz` based on `PAGE_FRAME` and `PAGE_SIZE`
- Use `alloc_upages()` (described below) to allocate an aligned `sz / PAGE_SIZE` pages of user memory in `as` starting at `vaddr`
- Iterate through the allocated pages in virtual memory and set their permissions to the specified ones

---

```
int as_prepare_load(struct addrspace *as);
```

Do nothing.

```
int as_complete_load(struct addrspace *as);
```

Allocate the bottom of the heap, and set up the top in `addrspace` so that `sbrk()` will work later.

```
int as_define_stack(struct addrspace *as, vaddr_t *stackptr);
```

Allocate space and populate page tables from the top of user address space down to the given stack pointer.

```
vaddr_t alloc_kpages(unsigned pages);
```

Helper function that allocates `pages` contiguous pages of kernel memory in the core map and returns a pointer to the first one (use `PADDR_TO_KVADDR`). Make sure to use the `CONTIG` bit to mark where the allocation starts and stops and mark pages as `KERNEL`.

- Get core map lock
- Keep track of:
    - Max number of contiguous free pages so far
    - Index of that page chain
    - Max number of contiguous non-KERNEL non-TLB non-BUSY pages so far
    - Index of that page chain
    - Max number of contiguous non-KERNEL non-BUSY pages so far
    - Index of that page chain
- Iterate over core map, counting the above stats
    - If at any point you reach the required number of contiguous free pages, break
- If the number of contiguous free pages is sufficient, use that index
- Else if the number of contiguous non-KERNEL, non-TLB, non-BUSY pages is sufficient:
    - Swap the relevant pages out of memory
    - Remember the starting index
- Else if the number of contiguous non-KERNEL non-BUSY pages is sufficient:
    - Swap the relevant pages out of memory
    - In the process, TLB shootdown the relevant pages and sleep on wchan until done
    - Remember the starting index

- Else return `ENOMEM`
- Iterate over the chain of free pages and mark them as belonging to the kernel
- Convert `PADDR` to `VADDR` with the macros and return the `VADDR` value

---

```
void free_kpages(vaddr_t addr);
```

Helper function that frees the kernel allocation starting at `addr` (in kseg0). Remember to use the `CONTIG` bits to free the entire allocation.

```
int alloc_upages(struct addrspace *as, vaddr_t vaddr, unsigned
pages);
```

Helper function that allocates `pages` pages of user memory in the core map and populates `as`'s page tables with them starting at `vaddr`. Does not mark pages `KERNEL`.

You must not hold any locks for potentially active address spaces or the core map lock when calling `alloc_upages().`

- Acquire address space lock
- Acquire core map lock
- Set `counter = ceil(sz/PAGESIZE)`
- Loop through every entry in core map
  - If entry is empty (has no vaddress)
    - Map it to the appropriate virtual page, adding more page table entries/page tables if necessary; set the permissions appropriately
    - Set the `vaddr` and `as` fields in coremap
    - Decrease `counter` for every successful addition; break when `counter == 0`
- Release core map lock
- Release address space lock
- While `counter > 0`:
  - Swap out a page (Refer to the `PTE_P` not set code in the paging section)
  - Acquire original address space lock
  - Acquire core map lock
  - Attempt to mark that page as ours and put it into our page table, as in the body of the for loop
  - Decrement `counter`

    - Release core map lock
    - Release address space lock
    - Because of the way the locking works, it might be possible for another thread to swoop in and claim the page we just swapped out from, but that's okay because `counter` only decreases upon success
- Return success, unless we ran out of both memory and swap space at some point during the function

---

```
void free_upage(struct addrspace *as, vaddr_t vaddr, unsigned
pages);
```

Helper function that frees (from the core map) `pages` contiguous pages of user memory in the virtual address space `as`, starting at address `vaddr`. This function will also need to shoot down the associated TLB entries and deal with removing pages that were swapped out.

4. Plan of Action

Sunday:	core map
Monday:	core map bootstrap
Tuesday:	page tables
Wednesday:	`as_create()`, `as_destroy()`, `as_copy()`, `as_prepare_load()`
Thursday:	Paging
Friday:	`as_activate()`, `as_deactivate()`, `tlbshootdown()`
Saturday:	TLB Misses (No swap yet)
Sunday:	MAT Daemon
Monday:	catch-up
Tuesday:	catch-up
Wednesday:	Swapping
Thursday:	Swapping
Friday:	catch-up
Saturday:	catch-up
Sunday:	`sbrk()`
Monday:	catch-up
Tuesday:	`as_define_region()`, `as_complete_load()`, `as_define_stack()`
Wednesday:	test
Thursday:	test
Friday:	test

5. Code-Reading Questions

5.1 Design Questions

1. TLB miss, page fault
    - page not present in TLB upon attempted read or write (can be in RAM, in swap, or nonexistent)

   TLB miss, no page fault
    - not possible, unless you consider use of kseg0 without the TLB a "miss"

   TLB hit, page fault
    - page present in TLB, wrong permissions (e.g. write to read-only address)

   TLB hit, no page fault
    - normal kuseg memory access (or kseg2 I suppose, but we're not really dealing with that)

2. It reads the faulting instruction, puts the address in the TLB, reruns the instruction, puts the instruction back in the TLB, etc. Infinite loop.

3. Requested address is invalid (Negative or not aligned to word), TLB entry does not exist, Process has no permission for the memory region it's trying to access, It's write only

5.2 Malloc Questions

1. a. malloc only calls sbrk once throughout the loop
   b. 0x21000

2. The last allocation does not need sbrk. The allocation pointed to by x is on the same page or same page range as the first allocation.

3. A user program might want to define a function of the same name; identifiers with underscore prefixes are reserved for the implementation of the C dev environment to prevent collisions with names used by applications.

4. Allocations are rounded up to an integral number of blocks. Non-fundamental data types are aligned to their greatest member. The sizes of the greatest members is a multiple of where each block starts.

5.3 Paging Algorithms

We can/will separate each eviction strategy into a separate function, then define a series of macros with the same text that each replace to a different eviction strategy. We would then comment out all but one and compile. Alternatively, instead of macros we could make an array of function pointers that we index into using a #defined index that can be easily changed. If we wanted to use different structs/stats for each algorithm, we could define them all with `#if`, `#endif`, and such but that would get very convoluted.

The above strategy is in our opinion the reasonable one, but if we just *need* instantaneous switching (these answers are more speculative):

- Maybe there's a way to set a flag in sys161.conf and have the OS choose its swapping algorithm based on it at startup (so we wouldn't need to recompile to switch page swapping algorithms), but that would probably require deeper knowledge of the inner workings of sys161.

- Theoretically we could also make a kernel program that can change a synchronized variable that serves the same purpose as a `#define`d index (and that way we could switch the page swapping algorithm once the OS is running), but then they would have to all use the same struct/stats. Many algorithms might also depend on data structures that are cultivated from boot, so they wouldn't be compatible.