CS 503: Fall 2022

Lab 5: Virtual Memory and Demand Paging (460 pts)

Due Dates: Part I - 11/20/2022 | Part II - 12/04/2022

---

1. Objective

We will enable paging on the x86 Galieo backends by setting the PG bit of CR0 to 1 to support virtual memory which, in turn, will be combined with demand paging so that pages may be temporarily stored on swap space to conserve main memory. A region of physical main memory will be used to hold pages belonging to the virtual address space of processes. Swap space, referred to as backing store in XINU, is typically implemented as part of the file system of a local disk. In XINU, backing store support is provided as a remote (i.e., networked) disk on a x86 frontend machine in the XINU Lab. Due to network delay swapping pages between Galileo's main memory and remote disk is very slow. In lab5 we will implement backing store as a region of physical main memory. If the region in physical main memory where pages are held becomes full, the backing store is used to accommodate the overflow.

From the kernel's viewpoint, main memory is organized into blocks of fixed size (4 KB) called frames where code and data, including kernel data structures and per-process heap memory, are kept. It is up to paging enabled XINU to manage what is stored in the frames which are containers where the content of pages are stored. From a process viewpoint, the kernel exports an illusion that a process owns the entire 32-bit address space of a Galileo backend. From a programmer's viewpoint, XINU will export system calls that allow a process to perform dynamic memory allocation using a per-process virtual memory heap. The original shared heap maintained as a free list that getmem() and freemem() operated on will be preserved. New system calls will allow memory to be dynamically allocation from per-process virtual memory heaps.

For lab5 use the paging version of XINU:

```
/homes/cs503/xinu-fall2022-paging.tar.gz
```

To faciliate a modular approach for tackling the problem, the due dates are in two phases with the first milestone (part I) to be submitted by 11/20/2022. Part II is due 12/04/2022. Part I contributes 280 points. Part II contributes 180 points.

Note: Lab5 may be tackled as a group effort involving up to 3 people. If going the group effort route, please specify in lab5.pdf on its cover page who the members are, who did what including programming the various components and testing. Whether you implement lab5 as an individual effort or make it a group effort is up to you. Keep in mind the trade-offs: group effort incurs coordination overhead which can slow down execution. Potential benefits include collaborative problem solving and a modicum of parallel speed-up if efficiently executed. Regarding late days, for a group to use k (= 1, 2, 3) late days, every member of the group must have k late days left.

---

2. Readings

1. XINU set-up
2. Read Chapters 9, 10 and 18 of the XINU textbook.
3. Read Chapter 4.3 from Intel x86 manual, vol 3, whose link is provided in Reference material. Related to Chapter 4.3, selectively confer relevant material in Chapters 2-5 as needed.

---

3. Organization of physical and virtual memory

3.1 Physical memory layout

We will consider main memory in the physical address space as being comprised of fixed-size 4 KB blocks, called frames, which are identified by their frame number 0, 1, ..., 2^{20} - 1. We will partition the physical memory space into 8 regions A, B, C, D, E1, E2, F and G, each serving a specific purpose.

Regions A and G. Region A comprised of frames 0-256 is used by bootloading, and region G comprised of frames 589824-590847 is used for device memory. Our concern is with the regions lying in-between.

Regions B and C. Region B consisting of frames frames 257-326 contains XINU text, data, bss. Region C consisting of frames 327-1023 will be used by legacy dynamic memory allocation (i.e., getmem() and getstk()) to allocate shared heap memory and per-process (but shared/visible) runtime stack for XINU processes. Hence regions B and C correspond to the runtime memory organization of legacy XINU in lab1-lab4.

Region D. Region D comprised of frames 1024-2023 will be used to store page directories and tables. As with GDT and IDT, these kernel data structures act as an interface to x86 whose MMU (memory management unit) references the information therein to perform virtual to physical address translation. Unlike GDT and IDT which are system wide and shared by all processes, page directories and tables stored in region D are per-process kernel data structures.

Region E1. Frames 2024-3047 in region E1 will be used as (initially empty) slots to store the content of pages belonging to per-process virtual memory heaps. In the virtual address space of a process, pages 4096-589823 comprise per-process private memory that is not shared and visible to other processes. All other pages are identity mapped to their corresponding physical memory frames which implies they are shared and visible to other processes. Providing support for per-process heap in virtual memory contained in pages 4096-589823 exports the illusion that every process owns this address space.

A page belonging to the per-process heap of a process is resident in physical main memory if one of the frames in region E1 contains the content of the page. This is achieved by setting up the page directory and page tables of a process so that x86 can translate page numbers to frames numbers in E1 that contain the content of the pages. If a page is not resident then either its content is being accessed for the first time and an empty frame in region E1 must be assigned, or the page is stored in the backing store in region E2 which acts as overflow storage. In general, remote disk implemented on a frontend machine is utilized if E1 is filled to capacity. In lab5 region E2 will take on the role of swap space.

Region E2. Region E2 comprised of frames 3038-4095 will act as swap space, called backing store in XINU, for region E1. If E2 reaches full capacity and a system call to allocate memory in a process's private heap requires evicting a page from E1 which has nowhere to go, the system call fails.

Region F. Region F consisting of frames 4096-589823 is not used by XINU. Pages 4096-589823 in per-process virtual address space is used to support per-process private heaps that are not shared and visible to other processes.

Layout of frames in physical memory:

G: device memory

1024 frames starting at 0x90000000 (frames 589824-590847)

F: unused physical address space

(frames 4096-589823)

E2: 1048 frames for use as backing store

(frames 3038-4095)

E1: 1024 frames to store resident pages

(frames 2024-3047)

D: 1000 frames to store page directories and tables

(frames 1024-2023)

C: shared heap memory (frames 327-1023)

B: XINU text, data, bss (frames 257-326)

A: memory reserved for bootloader (frames 0-256)

3.2 Virtual memory layout and per-process private heap

Identity mapped virtual memory. The 2^{20} - 1 pages in the virtual address space of a process are translated to frame numbers in the physical address space with the help of page directory and page tables which are stored in region D of the physical address space. The pages of a process in its virtual address space may also be viewed as partitioned into regions VA, VB, VC, VD, VE1, VE2, VF, and VG corresponding to the same address ranges as A, B, C, D, E1, E2, F, and G, albeit in a process's virtual address space where 4 KB blocks are pages. For every XINU process we will configure its per-process page directory and page tables so that for pages in regions VA, VB, VC, VD, VE1, VE2, and VG, page to frame translation implements the identity map. That is, a page number k in all regions but VF maps to frame number k upon page table lookup

page number k → frame number k

Hence all virtual memory but for region VF is shared by processes. In operating systems such as Linux and Windows that implement isolation/protection user code/data/stack are separate from kernel code/data/stacks and are not shared. Kernel address space in virtual memory is shared but protected from access by processes running in user mode by setting the U/S bit to 0. In legacy XINU without isolation/protection where all processes run in kernel mode it suffices to set the flag 0.

Private virtual memory for per-process heap. Pages 4096-589823 belong to region VF are used to support per-process virtual memory heaps that are not identity mapped to frames 4096-589823 in region F but to frames in region E1. Hence virtual address to physical address translation

VF → E1

per-process virtual memory that is not shared. By default, the content of virtual memory in VF is not

accessible to other processes since it is located in physical memory region E1 which belongs to the kernel and not accessible to user mode processes. If E1 reaches capacity, a page may be evicted from E1 by moving it to backing store E2. To indicate that a page is not resident in E1, the P (present) bit of its page table entry is set to 0. If a memory reference by an instruction maps to a page whose P flag is 0, x86 generates a page fault interrupt. In x86 it is the responsibility of the kernel to handle page faults.

Although the physical frames in region F are not utilized, the corresponding pages in virtual memory VF are at the heart of per-process private virtual memory. For example, one process may store ASCII character 'S' in the bytes of page number 4100 whereas another process may store 'T' in the same page in its virtual address space. Since the kernel will map the content of page 4100 of one process to a different frame (e.g., frame 2040) in region E1 than the content of page 4100 of another process (e.g., frame 2050), each process is provided with the illusion that it owns page 4100.

3.3 Backing store and demand paging

Backing store mapping. In addition to region E1 in physical main memory where pages belonging to per-process virtual memory in region VF are held, region E2 will be used as backing store (or swap space) to hold pages when E1 overflows. Typically a local disk is used as swap space. In XINU we may use a remote disk on a frontend machine to store pages if frames in region E1 are filled up. In lab5 we will use E2 as backing store which simplifies coding and testing. If space in E1 needs to be freed, page replacement policy determines which frame(s) to evict from E1 to E2. Therefore a kernel data structure that keeps track of where pages evicted from E1 are stored in E2 must be implemented

VF → E2

Describe in lab5.pdf the data structure you use to perform this backing store mapping.

A page fault interrupt, unless it is caused by access violation, signals that an address falls within a page that is not resident in one of the frames of region E1. In demand paging the missing page is brought in from backing store E2 to region E1 of physical main memory. If all frames in region E1 are occupied, page replacement policy determines which page is evicted from region E1 to backing store to free up a frame. Kernel bookkeeping keeps track of where pages are stored: paging directory and tables to translate page numbers to frame numbers if resident in E1, and additional kernel data structures to specify where in backing store E2 a page is stored if not resident in E1.

Demand paging. If backing store E2 has free space when a page in E1 needs to be evicted, page replacement policy determines which page to evict and backing store mapping is updated. E2 may be full when the kernel considers freeing up space in E1. For this to happen a process must have called vmhgetmem() (see Section 4) to request new memory. System call vmhgetmem() allocates memory in virtual address space VF and updates the process's page directory and tables to indicate new pages in VF have been allocated. However, vmhgetmem() does not allocate physical memory in E1 for the newly allocated pages in VF, and the P flag of page table entries are set to 0. It is when the process accesses the newly allocated memory in VF that x86 generates a page fault and XINU's page fault handler is tasked with allocating free frames in E1.

In the case where both E1 and E2 are full we will make the process that page faulted block by entering new process state PR_FRAME (define in include/process.h to an unused value) and wait its turn in a new FIFO queue, qid16 framewait, that utilizes XINU's legacy enqueue() and dequeue() functions. When a frame in E1 becomes free because a process calls vmhfreemem() or terminates, the kernel checks if a process is queued in state PF_FRAME. If so, the process is dequeued, allocated a free frame in E1 to the page on which it faulted, and readied. When the process becomes current, resched() returns to the page fault handler which returns to the page fault dispatcher. The dispatcher executes iret which results in the instruction that caused the page fault to be executed again with the previously missing page now resident. If a frame in E2 but not E1 becomes free, a page from E1 is evicted before allocating the free frame in E1 to the process waiting in FIFO queue framewait.

3.4 Multi-level caching of content and address translation

Collectively, pages in virtual memory, frames in E1, and backing store E2 make up the three layers of memory management that determines where content in virtual memory is stored in physical memory. These three layers are under the control of XINU. x86 utilizes multi-level caching to speed up content access and address translation by exploiting locality of reference of real-world processes. In x86 managing of multi-level caches, including TLB, is the purview of hardware. The kernel is responsible for configuring page directories and page tables, and conveying the location of a page directory through the PDBR register embedded in CR3 (its most significant 20 bits). The kernel is also responsible for invalidating TLB entries if changes to paging structures are made that may render page table entries cached in TLB stale.

---

4. Virtual memory system calls

Modifying XINU to support virtual memory entails changing legacy XINU code such as create() and ctxsw() to implement per-process page table allocation, configuration, and dynamic resource management. VM services are exported to processes using the following system calls to be coded in system/:

char *vmhgetmem(uint16 msize)

Similar to getmem(), vmhgetmem() allocates the requested amount of memory albeit in page granularity (not bytes). If msize contiguous pages are available, a virtual memory pointer to the start of the first allocated page is returned. Otherwise, vmhgetmem() returns SYSERR. Indirection through virtual to physical address translation allows external fragmentation in physical address space (region E1) to be hidden and exported as contiguous memory in virtual address space. If requested per-process heap allocation exceeds 4096 + MAXHSIZE - 1 (see 5.3 for definition of MAXHSIZE), vmhgetmem() returns SYSERR. Whereas legacy getmem() continues to allocate heap memory from shared memory region C, vmhgetmem() allocates heap memory in private virtual memory spanning pages 4096-589823 (in our implementation limited to MAXHSIZE pages) that are mapped to frames in region E1. vmhgetmem() returns a pointer to a virtual address in pages in region VF.

syscall vmhfreemem(char *blockaddr, uint16 msize)

vmhfreemem() is a counterpart of freemem() with corresponding arguments for private heap in virtual memory, albeit with the second argument in unit of pages (not bytes). Recall that in the case of getmem() memory leakage ensues when a process terminates if it does not call freemem() to free its memory. For XINU with VM support, even if vmhfreemem() is not called the kernel will frees up all frames and backing stores that a process was allocated when it terminates.

As usual, code the system calls in their own files under system/.

---

5. Enabling, initializing, and managing virtual memory

Virtual memory is enabled on x86 by setting the PG bit of CR0 to 1. This is done after relevant paging related kernel structures have been configured and initialized. Subsequently they are dynamically managed. The following summarize key elements involved in supporting demand paging.

5.1 Null process and identity map page tables

When XINU's null/idle process is configured in initialize.c its page table must be set up to perform identity map for all pages that correspond to frames in regions A, B, C, D, E1, E2, and G. Subsequently, all processes spawned using system call create() can share the process tables set up for the null process for regions A, B, C, D, E1, E2, and G. With respect to virtual memory, the distinguishing feature of a process created using create() from the null process is that the former may utilize a private heap heap by calling vmhgetmem() which necessitates additional page tables to translate page numbers 4096-589823 to frame numbers in E1.

All page table entries that undergo identity mapping to frames in regions A, B, C, D, E1, E2, and G are marked resident by setting the P (present) bit to 1. Confer Intel manual vol. 3 to set the values of other fields

of page table entries. The shared page tables are statically assigned when a process is spawned using create(). After updating CR3 to set up PDBR (page directory base register) as part the null process's context, it is ready to execute instructions in its infinite while-loop. Since all pages are identity mapped to frames resident in physical main memory, the null process will not page fault. This provides a sanity checkpoint that shared page table entries that are identity mapped have been correctly configured.

Of course, the null process calls create() to spawn a process before entering the while-loop. Hence, kernel data structures to manage per-process heaps in virtual memory must be configured, a page fault handler for exception (interrupt 14 in IDT) must be set up, and context-switching needs to be updated to account for a richer process context. Paging initialization is carried out by calling, void init_paging(void), in system/init_paging.c, right after sysinit() in initialize.c.

5.2 x86 2-level page table

x86 Galileo uses a 2-level page table structure. Note that in Intel nomenclature the second-level (or outer) page table indexed by the most significant 10 bits of a 32-bit virtual address is called page directory. The first-level (or inner) page table indexed by the next 10 bits is called page table. In our memory layout described in Section 3, the first 4096 frames (regions A-E) to which pages are identity mapped comprise 16 MB of the both virtual and physical memory starting at address 0. Since each page directory entry of which there 1024 spans a 4 MB address range, we need to configure the first 4 entries of the null process's page directory. Each page directory entry points to a page table containing 1024 page table entries. Hence 4 page tables containing a total of 4096 entries for frames 0-4095 to which pages are identity mapped must be created. These page tables can be shared by all processes. Paging structure initialization must also be carried out for 1024 pages starting at 0x90000000 which are identity mapped to frames 589824-590847 in region G. This means updating entry 576 (0x90000000 >> 22) of the page directory and all entries in the associated page table.

Page directories and page tables are stored in region D of the physical memory layout. To conserve memory, first-level page tables are allocated on-demand. That is, the first time a page is referenced, a page table is allocated. Conversely, when a page table is no longer needed, it is removed to conserve space.

5.3 Process creation

When a process is created by calling create() it is provided a per-process page directory. All page tables identity mapping to regions A, B, C, D, E and G can be shared. Page tables for a process's private heap that map to frames 2024-4095 are part of per-process context and unique to each process. When vmhgetmem() is called it searches through its free list in virtual memory spanning pages 4096 to 4096 + MAXHSIZE - 1 where MAXHSIZE (to be defined in include/memory.h as 1024) in unit of pages is a bound on the maximum number of pages a process is allowed to be allocated from region VF. vmhgetmem() updates the process's page directory and allocates page tables and initializes their page table entries. All entries are marked as non-resident, i.e., the P bit is set to 0. vmhgetmem() does not perform frame allocation in region E1. Accessing virtual memory in region VF will trigger a page fault which is handled by the kernel's page fault dispatcher pgfdisp which calls the page fault handler pgfhandler() to update page table entries to allocate frames in E1 and set up page-to-frame translation. Describe in lab5.pdf your per-process kernel structures for paging related processing.

5.4 Paging structure updates and TLB flush

x86 Galileo utilizes TLB to cache recently accessed page-to-frame mappings to speed up virtual-to-physical address translation. When paging structures are updated such as during a context-switch, TLB needs to be flushed due to aliasing where the same page number belonging to the two processes being context-switched in and out map to different frames in E1. This is accomplished by overwriting CR3. If mappings involving a select few pages are affected, the invlpg instruction may be used to selectively flush a page table entry that contains the address in the operand of invlpg. You may opt to use invlpg when incremental changes to page table entries are made to improve efficiency. However, for lab5 where focus is on correctness this is not necessary, i.e., you may flush the entire TLB. Although x86 manages multi-level content and TLB caches,

the kernel manages paging structures in main memory and instructs x86 if TLB needs to be flushed when updates to page table entries are made.

5.5 Process termination

When a process terminates all frames which currently hold its pages are freed. All pages stored on backing store are removed, and frames in region D used for page directory and page tables are released.

---

6. Managing physical memory regions

6.1 Region D: page directories and tables

Kernel data structures to track page directories and tables. Frames 1024-2023 are used to store per-process page directories and page tables. Page tables used by the null process that perform identity mapping are shared. When a process is spawned using create() it is allocated a page directory that is stored in a free frame in region D. If all frames in region D are occupied, create() returns SYSERR. Shared page tables set up for the null process spanning regions A, B, C, D, E1, E2, and G are stored in region D. Both page directory and page table are of size 4 KB (i.e., 1024 entries each of size 4 B). Page tables for per-process virtual memory heap are also allocated from region D. Devise a kernel data structure to track where in region D page directories and page tables belonging to which processes reside. Describe the data structure in lab5.pdf. Your data structure should facilitate ease of frame allocation in region D when a process is created, and deallocation when a process terminates. Make sure that page directories and page tables are aligned at 4 KB frame boundaries.

Page table allocation: vmhgetmem(). System call vmhalloc(hsize) specifies that a process wishes to use hsize pages as private virtual memory heap. Actual heap memory from virtual memory region VF is allocated by calling vmhgetmem() which treats bytes in pages 4096 to (4096 + hsize - 1) as a per-process free list from which requested pages are allocated per first-fit policy. If memory requested by vmhgetmem() cannot be met, vmhgetmem() returns SYSERR. If contiguous virtual memory allocation succeeds, a pointer in the virtual address range of page numbers 4096 to (4096 + hsize - 1) is returned. Page tables are allocated in region D for all pages in the range 4096 to (4096 + hsize - 1) that were allocated by vmhgetmem(). The page directory of the process is updated to point to the allocated page tables. The P bit of the allocated page table entries are set to 0. This means that until a virtual memory address allocated by vmhgetmem() is accessed by an instruction (e.g., movl) memory allocation exists virtually in the per-process heap region VF. vmhgetmem() does not allocate physical frames in E1 (or E2) to pages that it allocated in region VF. Physical frame allocation is performed by the page fault handler.

When an allocated virtual address is accessed by a process a page fault interrupt (exception 14) is generated since page table entries were marked as non-resident with the P flag set to 0. Your page fault handler, pgfhandler(), will perform the task of finding free frames in region E1, updating the relevant paging structures in region D and setting the P flag to 1. Upon returning from exception 14, the instruction that cause the page fault is executed again by x86 -- upon interrupt 14 x86 pushes EFLAGS, CS, EIP onto the runtime stack where EIP points to the instruction that caused the page fault -- which will find the missing page installed in a frame in region E1. If both E1 and E2 are full, the page faulting process will be context-switched out and wait its turn for frames to become available.

6.2 Managing regions E1 and E2: page fault handler

When a page fault is generated, identify the faulting virtual address. Determine the page number of the page that contains the address. Using the page directory of the current process consider what actions to take.

Page fault pertaining to E1 but not E2. A page fault is generated when a page belonging to per-process virtual memory heap is marked not resident. This may mean that vmhgetmem() has updated per-process page tables that are marked not resident because either a page is being referenced for the first time, or it has been evicted to backing store in region E2. In both cases, it is the responsibility of the page fault handler, void

pgfhander(void), in system/pgfhandler.c that is called by the page fault dispatcher, pgfdisp in system/intr.S, to find a free frame in E1 for the deferenced page. If a page is being accessed for the first time, pgfhandler() allocates free frames in E1, updates relevant page table entries, and sets the P flag to 1. When pgfdisp returns by executing iret, the instruction that caused the page fault is executed again. To install the page fault dispatcher, pgfdisp, replace _Xint14 and its code with pgfdisp assembly code in system/intr.S.

Page fault pertaining to E1 and E2. If a missing page resides in backing store E2 because it was evicted from E1, a free frame is found in E1 and the missing page moved to E1. If E1 is full, a page resident in E1 is evicted to E2. The default policy for selecting a page to evict is FIFO. Note that even if both E1 and E2 are full, an exchange of the page in E1 to be evicted and the missing page in E2 to be brought into E1 can be effected "in place," i.e., without needing to free frames in E1 and E2. The two pages just swap places.

Page fault pertaining unallocated memory. A page fault can also mean that a page is being accessed that has not been allocated by vmhgetmem(). If so, we will terminate the offending process which correponds to the default disposition of segmentation fault in Linux.

System page fault. The last condition for a page fault can be access violation such as trying to write to a page that is readable but not writable. x86 pushes an error code at the top of the runtime stack above the return address upon generating exception 14 which should be inspected to help determine what caused the interrupt.

Implementing XINU's backing store in physical main memory removes dependence on very slow and unreliable remote disk access via UDP over Ethernet on a frontend x86 machine. This simplifies programming and testing. If backing store E2 corresponded to a memory-mapped block device, swap space may physically be hard disk or SSD, even though E2 is used to interface with it.

6.3 Location of kernel data structures for virtual memory management

You will devise kernel data structures to perform bookkeeping chores such as pages allocated in VF, frames assigned to pages in E1, frames storing evicted pages in E2, and queue to implement FIFO page replacement. By default, you may allocate memory to hold kernel structures in region C using getmem(). If global variables are used to implement new kernel data structures, note that they are part of region B and may overflow into frame 327 and above which results in physical memory layout that deviates slightly from Section 3.1. In practice, this will not matter since the start of the shared heap space, minheap, where getmem() allocates memory from, is set to the end of the XINU image in meminit.c, and virtual memory management affects regions VF, E1, and E2. When implementing kernel modifications that impact memory layout check that they do not introduce unintended side effects.

---

7. Submission in two phases

To facilitate a modular approach for tackling the problem, the due dates are in two phases with part I due on 11/20/2022 and part II due on 12/04/2022. The scope of part I is virtual memory paging without backing store support. That is, during testing per-process heap allocation across all processes will not exceed the 1024 frames available in region E1. Hence page replacement and managing of backing store E2 which can entail context-switching out a page faulting process if both E1 and E2 are full does not come into play.

When implementing and testing part I, first consider processes that do not make system calls of Section 4 to allocate per-process virtual memory heap. With paging enabled and virtual addresses translated to physical address via page tables implementing identity mapping, processes should run behave in the same way as before in XINU without virtual memory support. In the second step of part I, add support for per-process private heap. Keep the total virtual memory heap allocation across all process below 1024 so that page replacement and managing of backing store E2 does not come into play. Describe in lab5.pdf the test cases you considered to gauge correctness of your VM XINU implementation for part I. In README.1 list all the functions that you introduced for part I that are not specified in the problem description, along with a brief description of their roles.

In part II, processes fill up all 1024 E1 frames necessitating that pages are evicted to backing store E2 and restored from backing store as needed. This entails supporting FIFO page replacement policy and managing frames across E1 and E2 to implement demand paging. When both E1 and E2 are full page fault leads to the current process being blocked until free frames become available. Describe in lab6.pdf the test cases you considered to gauge correctness of your VM XINU implementation for part II. In README.2 list all the functions that you introduced for part II that are not specified in the problem description, along with a brief description of their roles.

Bonus problem [40 pts]

Up to 40 points (20 points for each part) will be given in consideration of (a) modularity, clarity, style, and (b) efficiency of your implementation. With respect to (a), a modular design that is not monolithic, adequately annotated with comments, not cluttered will be recognized with up to 10 points. With respect to efficiency, although the focus of lab5 (part I and II) is on correctness all else being equal implementing efficient solutions is desirable and will be noted. For example, in part I pages in region VF and frames in region E1 will need to be managed and allocated/deallocated. Doing so reasonably efficiently (but not necessarily optimally) indicates careful consideration and execution that will be recognized with up to 10 points. The same goes for part II.

Note: The bonus problem provides an opportunity to earn extra credits that count toward the lab component of the course. It is purely optional.

Turn-in instructions

Due dates:

The lab is divided into two parts with part I due on 11/18/2022 and part II due on 12/04/2022. Using turnin submit part I as lab5 and part II as lab6.

General instructions:

When implementing code in the labs, please maintain separate versions/copies of code so that mistakes such as unintentional overwriting or deletion of code is prevented. This is in addition to the efficiency that such organization provides. You may use any number of version control systems such as GIT and RCS. Please make sure that your code is protected from public access. For example, when using GIT, use git that manages code locally instead of its on-line counterpart github. If you prefer not to use version control tools, you may just use manual copy to keep track of different versions required for development and testing. More vigilance and discipline may be required when doing so.

The TAs, when evaluating your code, will use their own test code (principally main()) to drive your XINU code. The code you put inside main() is for your own testing and will, in general, not be considered during evaluation.

If you are unsure what you need to submit in what format, consult the TA notes link. If it doesn't answer your question, ask during PSOs and office hours which are scheduled M-F.

Specific instructions:

1. Format for submitting written lab answers and kprintf() added for testing and debugging purposes in kernel code:

- Provide your answers to the questions below in lab5.pdf and place the file in lab5/. You may use any document editing software but your final output must be exported and submitted as a pdf file.
- For problems where you are asked to print values using kprintf(), use conditional compilation (C preprocessor directives #define combined with #if and #endif) with macro XINUTEST (in

include/process.h) to effect print/no print depending on if XINUTEST is defined or not. For your debug statements, do the same with macro XINUDEBUG.

2. Before submitting your work, make sure to double-check the TA Notes to ensure that any additional requirements and instructions have been followed.

3. Electronic turn-in instructions:

i) Go to the xinu-fall2022/compile directory and run "make clean".

ii) Go to the directory where lab5 (containing xinu-fall2022-paging/ and lab5.pdf) is a subdirectory.

For example, if /homes/alice/cs503/lab5/xinu-fall2022-paging is your directory structure, go to /homes/alice/cs503

iii) For part I of lab5, type the following command

turnin -c cs503 -p lab5 lab5

You can check/list the submitted files using

turnin -c cs503 -p lab5 -v

If you tackled lab5 as a group effort only one member of the group need run turnin. The cover page of lab5.pdf should note who the group members are and what each person contributed. The same goes when part II is submitted.

For part II of lab5,

turnin -c cs503 -p lab6 lab6

Please make sure to disable all debugging output before submitting your code.

Back to the CS 503 web page