NC State Home (https://www.ncsu.edu)										RESOURCES

MENU ☰

(https://wordpress-

CSC 501 (002) Spring 2023 Operating Systems Principles

courses2223.wolfware.ncsu.edu/csc-501-002-sprg-2023/)

PA3

CSC501: Spring 2023

PA3: Demand Paging

## Mid submission Due: Friday, April 7, 2023, 11:59 PM
Final Due: Sunday, April 23, 2023, 11:59 PM

1. Introduction

Demand paging is a method of mapping a large address space into a relatively small amount of physical memory. It allows a program to use an address space that is larger than the physical memory, and access non-contiguous sections of the physical memory in a contiguous way. Demand paging is accomplished by using a "backing store" (usually disk) to hold pages of memory that are not currently in use.

From this point on, only the details of this project are discussed. It is assumed that you have read the Intel documents and are comfortable with paging concepts and the Intel specific details. Here are some suggested reading materials:

› Address Translation example for Intel Processors (https://moodle-courses2223.wolfware.ncsu.edu/pluginfile.php/1237270/mod_assign/introattachment/0/intelvm.html?forcedownload=1) (By: Joe Pfeiffer)
› The image from the above page that is missing:
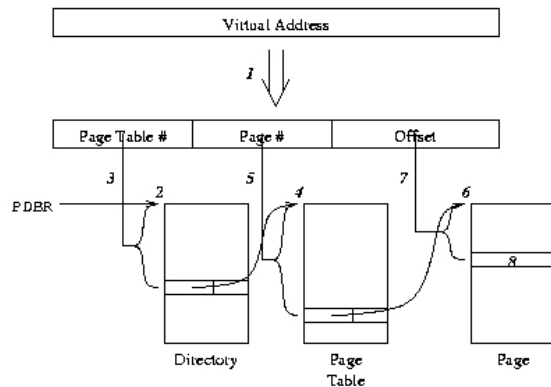
> Intel System Programming Manual (http://www.intel.com/design/pentiumii/manuals/243192.htm)

2. Goal

The goal of this project is to implement the following system calls and their supporting infrastructure.

3. System Calls

SYSCALL xmmap (int virtpage, bsd_t source, int npages)

Much like its Unix counterpart (see man mmap), it maps a source file ("backing store" here) of size npages pages to the virtual page virtpage. A process may call this multiple times to map data structures, code, etc.

SYSCALL xmunmap (int virtpage)

This call, like munmap, should remove a virtual memory mapping. See man munmap for the details of the Unix call.

SYSCALL vcreate (int *procaddr, int ssize, int hsize, int priority, char *name, int nargs, long args)

This call will create a new Xinu process. The difference from create() is that the process' heap will be private and exist in its virtual memory.
The size of the heap (in number of pages) is specified by the user through hsize.

create() should be left (mostly) unmodified. Processes created with create() should not have a private heap, but should still be able to use xmmap().

WORD *vgetmem (int nbytes)

Much like getmem(), vgetmem() will allocate the desired amount of memory if possible. The difference is that vgetmem() will get the memory from a process' private heap located in virtual memory. getmem() still allocates memory from the regular Xinu kernel heap.

SYSCALL srpolicy (int policy)

This function will be used to set the page replacement policy to Second-Chance (SC) or First-In-First-Out (FIFO). You can declare constant SC as 3 and FIFO as 4 for this purpose.

SYSCALL vfreemem (block_ptr, int size_in_bytes)

You will implement a corresponding vfreemem() for vgetmem() call. vfreemem() takes two parameters and returns OK or SYSERR. The two parameters are similar to those of the original freemem() in Xinu. The type of the first parameter block_ptr depends on your own implementation.

4. Overall Organization

The following sections discuss at a high level the organization of the system, the various pieces that need to be implemented in Xinu and how they relate to each other. You are welcome to use a different implementation strategy if you think it is easier or better as long as it has the same functionality and challenges.
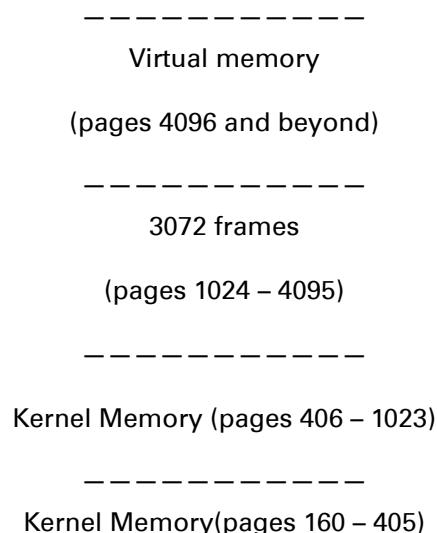
4.1 Memory and Backing Store

4.1.1 Backing Stores

Virtual memory commonly uses disk spaces to extend the physical memory. However, our version of Xinu has no file system support. Instead, we will emulate the backing store (how it is emulated will be detailed in 4.1.3). To access the backing store, you need to implement the following functions in the directory paging:

1. **bsd_t** is the type of backing store descriptors. Each descriptor is used to reference a backing store. Its type declaration is in h. This type is merely **unsigned int**. There are 16 backing stores. You will use IDs 0 through 15 to identify them.

2. **int get_bs (bsd_t store, unsigned int npages)** requests a new backing store with ID store of size npages (in pages, not bytes). If a new backing store can be created, or a backing store with this ID already exists, the size of the new or existing backing store is returned. This size is in pages. If a size of 0 is requested, or the creation encounters an error, SYSERR should be returned. Also for practical reasons, npages should be no more than 128.

3. **int release_bs (bsd_t store)** releases the backing store with the ID store.

4. **SYSCALL read_bs (char *dst, bsd_t store, int page)** copies the page-th page from the backing store referenced by store to dst. It returns OK on success, SYSERR otherwise. The first page of a backing store is page zero.

5. **SYSCALL write_bs (char *src, bsd_t store, int page)** copies a page referenced by src to the page-th page of the backing store referenced by store. It returns OK on success, SYSERR otherwise.

4.1.2 Memory Layout

The basic Xinu memory layout is as follows (page size = 4096 bytes):

— — — — — — — — — —

Virtual memory

(pages 4096 and beyond)

— — — — — — — — — —

3072 frames

(pages 1024 – 4095)

— — — — — — — — — —

Kernel Memory (pages 406 – 1023)

— — — — — — — — — —

Kernel Memory(pages 160 – 405)

————————————
Kernel Memory (pages 25 – 159)

————————————
Xinu text, data, bss (pages 0 – 24)
————————————-

As you can see, our Xinu version compiles to about 100KB, or 25 pages. There is an area of memory from page 160 through the end of page 405 that cannot be used (this is referred to as the "HOLE" in initialize.c). We will place the free frames into pages 1024 through 4095, giving 3072 frames.

The frames will be used to store resident pages, page directories, and page tables. The remaining free memory below page 4096 is used for Xinu's kernel heap (organized as a freelist). getmem()and getstk() will obtain memory from this area (from the bottom and top, respectively).

All memory below page 4096 will be global. That is, it is usable and visible by all processes and accessible by simply using actual physical addresses. As a result, the first four page tables for every process will be the same, and thus should be shared.

Memory at page 4096 and above constitute a process' virtual memory. This address space is private and visible only to the process which owns it. Note that the process' private heap and (optionally) stack are located somewhere in this area.

4.1.3 Backing Store Emulation

Since our version of Xinu does not have file system support, we need to emulate the backing store with physical memory. In particular, consider the following Xinu memory layout:

————————————
Virtual Memory

(pages 4096 and beyond)

————————————
16 Backing stores

(pages 2048 – 4095)

————————————
1024 frames

(pages 1024 – 2047)

————————————
Kernel Memory (pages 406 – 1023)

————————————
Kernel Memory (pages 160 – 405)

————————————
Kernel Memory (pages 25 – 159)

————————————
Xinu text, data, bss (pages 0 – 24)
————————————-

A Xinu instance has 16 MB (4096 pages) of real memory in total. We reserve the top 8MB real memory as backing stores. We have 16 backing stores and each backing store maps up to 128 pages (each page is 4K size). As a result, we have the following map between the backing store and the corresponding physical memory range:

backing store 0: 0x00800000 0x0087ffff	backing store 1: 0x00880000 0x008fffff
backing store 2: 0x00900000 0x0097ffff	backing store 3: 0x00980000 0x009fffff
backing store 4: 0x00a00000 0x00a7ffff	backing store 5: 0x00a80000 0x00afffff
backing store 6: 0x00b00000 0x00b7ffff	backing store 7: 0x00b80000 0x00bfffff
backing store 8: 0x00c00000 0x00c7ffff	backing store 9: 0x00c80000 0x00cfffff
backing store 10: 0x00d00000 0x00d7ffff	backing store 11: 0x00d80000 0x00dfffff
backing store 12: 0x00e00000 0x00e7ffff	backing store 13: 0x00e80000 0x00efffff
backing store 14: 0x00f00000 0x00f7ffff	backing store 15: 0x00f80000 0x00ffffff

In the implementation, you need to "steal" physical memory frames 2048 – 4095 (take a close look at sys/i386.c, and pay attention to the variables npages and maxaddr). As a result, this portion of memory will not be used for other purposes. You can assume that our grading program will not modify this part of memory.

4.1.4 Page Tables and Page Directories

Page tables and page directories (i.e. outer page tables) can be placed in any free frames. For this project you will not be paging either the page tables or page directories. As page tables are always resident in memory, it is not practical to allocate all potential page tables for a process when it is created (you will, however, allocate a page directory). To map all 4 GB of memory would require 4 MB of page tables! To conserve memory, page tables must be created on-demand. That is, the first time a page is legally touched (i.e. it has been mapped by the process) for which no page table is present, a page table should be allocated. Conversely, when a page table is no longer needed it should be removed to conserve space.

4.2 Supporting Data Structures

4.2.1 Finding the backing store for a virtual address

You may realize that there is a problem – if a process can map multiple address ranges to different backing stores, how does one figure out which backing store a page needs to be read from (or written to) when it is being brought into (removed from) a frame?

To solve the problem, you need to keep track of which backing store is allocated when a process is created by vcreate(). Then, a particular page to write/read from can be calculated using its virtual page number within the related store. You may need to declare a new kernel data structure which maps virtual address spaces to backing store descriptors. We will call this the backing store map. It should be a tuple like:

<p align="center">{ pid, vpage, npages, store }</p>

You should write a function that performs the lookup:

<p align="center">f (pid , vaddr)= > {store, pageoffset within store}</p>

The function xmmap() will add a mapping to this table. xmunmap() will remove a mapping from this table.

4.2.2 Inverted Page Table

When writing out a dirty page you may notice the only way to figure out which virtual page and process (and thus which backing store) a dirty frame belongs to would be to traverse the page tables of every process looking for a frame location that corresponds to the frame we wish to write out. This is highly inefficient. To prevent this, we use another kernel data structure, an inverted page table. The inverted page table contains tuples like:

{ frame number, pid, virtual page number }

Of course, if we use an array of size NFRAMES, the frame number is implicit and just the index into the array. With this structure we can easily find the pid and virtual page number of the page held within any frame i. From that we can easily find the backing store (using the backing store map) and compute which page within the backing store corresponds with the page in frame i.

You may also want to use this table to hold other information for page replacement (i.e., any data needed to estimate page replacement policy).

4.3 Process Considerations

With each process having its own page directory and page tables, there are some new considerations in dealing with processes.

4.3.1 Process Creation

When a process is created we must now also create page directory and record its address. Also remember that the first 16 megabytes of each process will be mapped to the 16 megabytes of physical memory, so we must initialize the process' page directory accordingly. This is important as our backing stores also depend on this correct mapping.

A mapping must be created for the new process' private heap and stack , if created with vcreate(). As you are limited to 16 backing stores, you may want to use just one mapping for both the heap and the stack (as with the kernel heap), vgetmem() taking from one end and the stack growing from the other. (Keeping a private stack and paging it is optional, but a private heap must be maintained).

4.3.2 Process Destruction

When a process dies, the following should happen.

1. All frames which currently hold any of its pages should be written to the backing store and be freed.
2. All of its mappings should be removed from the backing store map.
3. The backing stores for its heap (and stack if have chosen to implement a private stack) should be released (remember backing stores allocated to a process should persist unless the process explicitly releases them).
4. The frame used for the page directory should be released.

4.3.3 Context Switching

It should also be clear that now as we switch between processes we must also switch between memory spaces. This is accomplished by adjusting the PDBR register with every context switch. We must be careful, however, as this register must always point to a valid page directory which is in RAM at a page boundary.

Think carefully about where you place this switch if you put it in resched() – before or after the actual context switch.

4.3.4 System Initialization

The NULL process is somewhat of a special case, as it builds itself in the function sysinit(). The NULL process should not have a private heap (like any processes created with create()).

The following should occur at system initialization:

1. Set the DS and SS segments' limits to their highest values. This will allow processes to use memory up to the 4 GB limit without generating general protection faults. Make sure the initial stack pointer (initsp) is set to a real physical address (the highest physical address) as it is in normal Xinu. See sys/i386.c. And don't forget to "steal" physical memory frames 2048 – 4096 for backing store purposes.
2. Initialize all necessary data structures.
3. Create the page tables which will map pages 0 through 4095 to the physical 16 MB. These will be called the global page tables.
4. Allocate and initialize a page directory for the NULL process.
5. Set the PDBR register to the page directory for the NULL process.
6. Install the page fault interrupt service routine.
7. Enable paging.

4.4 The Interrupt Service Routine (ISR)

As you know, a page fault triggers an interrupt 14. When an interrupt occurs the machine pushes CS:IP and then an error code (see Intel Volume III chapter 5)

———-
error code
———-
IP
———-
CS
———-
...
...

It then jumps to a predetermined point, the ISR. To specify the ISR we use the routine:

set_evec(int interrupt, (void (*isr)(void))) (see evec.c)

4.5 Faults and Replacement Policies

4.5.1 Page Faults

A page fault indicates one of two things: the virtual page on which the faulted address exists is not present or the page table which contains the entry for the page on which the faulted address exists is not present. To deal with a page fault you must do the following:

1. Get the faulted address **a**.
2. Let **vp** be the virtual page number of the page containing the faulted address.
3. Let **pd** point to the current page directory.
4. Check that **a** is a legal address (i.e. that it has been mapped in pd). If it is not, print an error message and kill the process.
5. Let **p** be the upper ten bits of **a**. [What does p represent?]
6. Let **q** be the bits [21:12] of **a**. [What does q represent?]
7. Let **pt** point to the **p**-th page table. If the **p**-th page table does not exist, obtain a frame for it and initialize it.
8. To page in the faulted page do the following:
    1. Using the backing store map, find the store **s** and page offset **o** which correspond to **vp**.
    2. In the inverted page table, increase the reference count of the frame that holds **pt**. This indicates that one more of **pt's** entries is marked as "present."

3. Obtain a free frame, **f**.

4. Copy the page **o** store **s** to **f**.

5. Update **pt** to mark the appropriate entry as present and set any other fields. Also set the address portion within the entry to point to frame **f**.

4.5.2 Obtaining a Free Frame

When a free frame is needed, it may be necessary to remove a resident page from frame. How you pick the page to remove depends on your page replacement policy,

Your function to find a free page should do the following:

1. Search inverted page table for an empty frame. If one exists stop.
2. Else, Pick a page to replace.
3. Using the inverted page table, get vp, the virtual page number of the page to be replaced.
4. Let **a** be **vp*4096** (the first virtual address on page vp).
5. Let **p** be the high 10 bits of **a**. Let **q** be bits [21:12] of **a**.
6. Let **pid** be the pid of the process owning **vp**.
7. Let **pd** point to the page directory of process **pid**.
8. Let **pt** point to the **pid's p**-th page table.
9. Mark the appropriate entry of **pt** as not present.
10. If the page being removed belongs to the current process, invalidate the TLB entry for the page **vp** using the **invlpg** instruction (see Intel Volume III/II).
11. In the inverted page table decrement the reference count of the frame occupied by **pt**. If the reference count has reached zero, you should mark the appropriate entry in pd as being not present. This conserves frames by keeping only page tables which are necessary.
12. If the dirty bit for page **vp** was set in its page table you must do the following:
    1. Use the backing store map to find the store and page offset within store given **pid** and **a**. If the lookup fails, something is wrong. Print an error message and kill the process **pid**.
    2. Write the page back to the backing store.

4.5.3 Page Replacement Policies

You must implement two page replacement algorithms: **SC** (default) and **FIFO**.

For SC, when a frame is allocated for a page, you insert the frame into a circular queue. When a page replacement occurs, SC first looks at the current position in the queue (current position starts from the head of the queue), checks to see whether its reference bit is set (i.e., pt_acc = 1). If it is not set, the page is swapped out. Otherwise, the reference bit is cleared, the current position moves to the next page and this process is repeated. If all the pages have their reference bits set, on the second encounter, the page will be swapped out, as it now has its reference bit cleared.

For FIFO algorithm, you need to ensure that the page that was allocated first in the main memory is the one that gets replaced first when a victim page is needed. You need to keep track of when each page has been brought into the memory which would give you the needed information while evicting a page.

Default page replacement policy should be SC; srpolicy(FIFO) will be called in main() to switch the replacement policy from SC to FIFO. When srpolicy(SC) or srpolicy(FIFO) is called, your program should turn on a debugging option, so that when replacements occur, it will print ONLY the replaced frame numbers (not addresses) for grading.

Note that you are free to add whatever structures you'd like in your inverted page table.

5. Required API Calls

You must implement the system calls listed in the beginning of this handout exactly as specified. Be sure to check the parameters for validity. For example, no process should be allowed to remap the lowest 16 MB of the system (global memory).

None of Xinu's other system call interfaces should be modified.

6. Details on Intel Architecture and Xinu

After having read chapters two and three in volume 3 (http://www.intel.com/design/pentiumii/manuals/243192.htm) you should have a basic understanding of the details of memory management in the Intel architecture.

The following might be useful for you to know:

1. We are using the Intel Pentium chip, not the Pentium Pro or Pentium II. Some details of those chips do not apply.
2. Xinu uses the flat memory model, i.e. physical address = linear addresses.
3. The segments are set in i386.c in the function setsegs().
4. Pages are 4k (4096 bytes) in size. Do not use 2M or 4M page size
5. The backend machines have 16 MB (4096 pages) of real memory.

Some example code is given in the project directory for getting and setting the control registers. A useful function, dump32 (unsigned long), for dumping a binary number with labelled bits is also given.

7. Code Framework

In this lab, we will use another version of Xinu that can be downloaded in the following link: csc501-lab3.tgz (https://moodle-courses2223.wolfware.ncsu.edu/pluginfile.php/1237270/mod_assign/introattachment/0/csc501-lab3.tgz?forcedownload=1). This version of Xinu contains the skeleton code for the backing store calls. It also has .h files needed for this project. Here are a test file testmain.c (https://moodle-courses2223.wolfware.ncsu.edu/pluginfile.php/1237270/mod_assign/introattachment/0/testmain.c?forcedownload=1) and its sample output (https://moodle-courses2223.wolfware.ncsu.edu/pluginfile.php/1237270/mod_assign/introattachment/0/sample_output.txt?forcedownload=1).

8. Debugging

Please try to debug by yourself first. Also realize that you know your program best.

Furthermore, if it helps you, you can uncomment the #define's in evec.c to get a stack trace and register dump. Using this and nm on the file xinu.elf can help you locate where your program crashed. Or you may recompile everything using the compiler's-g flag, disassemble xinu.elf using objdump-d xinu.elf > xinu.dis, load xinu.dis into your text editor and search for the return address in the stack. In the disassembly the addresses are the numbers on the left (e.g. ab3e:). This will show you the function name (may be some lines above) of the function the crash occurred in and (if you compiled that particular file with-g) the C line number in the []'s.

The most difficult problem to diagnose is when the machine simply reboots itself. This is usually the result of having a bad stack pointer. In such a case the machine cannot give a trap.

9. What to Turn In

The goal of the this assignment is to provide support for :

› Memory mapping: mapping of the first 16 Mb of physical memory, and the xmmap() and xmunmap() system calls
   › Different running processes created with *vcreate* can have its own private heap
   › vgetmem, vfreemem: implemented and fully functional
   › All running processes can simply share the same page table
› Demand paging: data is retrieved from the backing stores only when needed
› Backing store management:
   › get_bs, release_bs, read_bs, write_bs: implemented and fully functional
› Page replacement policies: SC, FIFO

Remember that, per the specification, page tables are created and destructed on demand. In other words, you system must not pre-allocate page tables. Also, page tables that do not contain at least one valid entry pointing to a data page should be destroyed (the frame should be freed) as soon as their last entry is invalidated. Page tables and page directories are not paged out.

Turn-in Instructions

Electronic turn-in instructions

1. go to the csc501-lab3/compile directory and do "make clean".
2. go to the directory of which your csc501-lab3 directory is a subdirectory (NOTE: please do not rename csc501-lab3, or any of its subdirectories.) e.g., if **/home/csc501/csc501-lab3** is your directory structure, goto **/home/csc501**
3. create a subdirectory TMP (under the directory csc501-lab3) and copy all the files you have modified/written, both .c files and .h files into the directory.
4. compress the csc501-lab3 directory into a tgz file and submit your file on Moodle. Please only upload one tgz file.

```
tar czf csc501-lab3.tgz csc501-lab3
```

Grading Policy

› (28%) Source code can be compiled and the generated image is bootable
› (72%) All test cases worth 72 points

10. One Last Note

Even with the design given to you, this is not necessarily an easy project. Dealing with low-level aspects of the machine is always difficult. Please do not procrastinate. It is very easy (especially with Xinu and even more so when working at a low level) to run into problems.

FAQs:

1) How do I get the virtual page number from a virtual address?

The most significant 20 bits of a virtual address form the virtual page number.

2) About the mapping < pid, vpage, npages, store >

This mapping is maintained inside the kernel. Since the "store" can take only 16 values at the most (because there are only 16 backing stores possible for any user), and no store can be mapped to more than one range of virtual memory at any time, the table that contains these mappings will contain only 16 entries. This table is placed in the kernel data segment in the first 25 pages of the physical memory. You need not take any extra care about placing this table. Just create an array of 16 entries of the type of the mapping and that's all. It is pretty similar to semaph[] and proctab[].

3) srpolicy()

This system call will not be called at arbitrary places inside your code to force changing from one replacement policy to another. You can assume that the default policy is SC. srpolicy(FIFO), if called, will be the first statement in the program. So, need not worry about switching from one replacement policy to another midway through the execution.

4) Paging.h contains two structures pd_t and pt_t which contains a lot of bit fields. Initially which fields should be set in a page directory and a page table?

For page directories, set the following bits and make the other bits zero : pd_write always and pd_pres whenever the corresponding page tables are present in the main memory.

For the four global page tables, set the following bits: pt_pres and pt_write. You can make others zero.

(This answer should be fairly obvious if you have read the Intel manuals carefully.)

5) Where do we place the page tables and page directories?

If your memory is divided into 4096 pages, page tables and page directories should be placed in the range 1024-2047. They should be placed on page boundaries only, i.e., the starting address of any page table or page directory should be divisible by the size of the pages NBPG.

6) What is the use of xmmap()?

There was a big misconception about the usage of xmmap() among many previous students. When does a user process call xmmap()? Why is it used for?

Even though xmmap() is given in the first page of your handout, it is not the most important system call that you should implement. Also, it is not main part of the project. Also, it is not the only way by which you can access virtual memory and test your implementation.

Then, how else can a process try to use virtual memory? Here is one example. This example shows how to access virtual memory, and when a page fault could happen.

```
main()
{
vcreate(process A, , , hsize = 100, ,,,);      /* process A is created with a virtual heap of 100 pages */

}

process A()
{
int *x;
int temp;
x = vgetmem(1000);  /* allocates some memory in the virtual heap which is in virtual memory */
/* the following  statement will cause a page fault. The page fault handling routing will read in the required page from
```

backing store into the main memory, set the proper page tables and the page directory entries and reexecute the statement. */

*x = 100;

x++;

*x = 200;

temp = *x; /* You are reading back from the virtual heap to check if the previous write was successful */

vfreemem(–x, 1000); /* frees the allocation in the virtual heap */

}

The virtual heap in the example is present in a backing store that is exclusive for the process alone (no backing store should be shared across processes; neither should the same backing store be mapped to multiple memory ranges). The heap is mapped in the address range of 4096th page to 4196th page of the process. So, the backing store mapping table should contain an entry < process A's pid, 4096, 100, backing store number >.

Then, why do we need xmmap() and what does it do? xmmap() is very similar to mmap() of Unix. It treats the backing stores as "files." One potential usage of xmmap() is as follows:

Process A:

char *x;

char temp;

get_bs(4, 100);

xmmap(7000, 4, 100);   /* This call simply creates an entry in the backing store mapping */

x = 7000*4096;

x = 'Y';   / write into virtual memory, will create a fault and system should proceed as in the prev example */

temp = *x;   /* read back and check */

xmunmap(…);

It can be thought of as you create a file, whose name is "4". It is a big empty file of size 100 pages. You store a character 'A' as the first character in the file. However, instead of using file I/O operations, you modify the file by means of a memory modification, as you have mapped the file to a memory location.

Let us say there is another process B which executes the following code after process A writes to its virtual memory mapped by xmmap.

  Process B:

char *x;

char temp_b;

xmmap(6000, 4, 100);

x = 6000 * 4096;

temp_b = *x: /* Surprise: Now, temp_b will get the value 'Y' written by the process A to this backing store '4' */

These examples should make the usage of xmmap() more clear. Think about it.

7) Page fault handling routine (page fault ISR) – What should be done here?

Psuedo code for the implementation (which would be easier if you do it in assembly)

1) store error code in a global variable

2) store flag register

3) clear all interrupts

4) save all general registers

5) page fault handling

6) restore general registers

7) restore flag register

8) iret

If you have not written in assembly language before, look at some code written in assembly in xinu itself. Or else, disassemble some simple C code and check the assembly code. Note that not everything has to be implemented in assembly as it would be very difficult. Thus, you could include a call to a C function which handles 5).

8) Are read_bs and write_bs blocking calls and can they be used inside our interrupt handling routine?

They are, and can be used inside the page fault handling routine.

9) How do I test replacement policies?

There is a constant NFRAMES in paging.h. Its default value is 1024. Make sure that your entire code depends on this constant as a measure of the available free frames. If we change the constant value to say 400, then the number of free frames initially available is only 400, i.e., your main memory looks as if it has only 1024 + NFRAMES = 1024 + 400 = 1424 frames of memory. So, you have an ample scope to test your replacement policy by changing NFRAMES.

Search …   Search

Recent Comments

Archives

Categories

> No categories