CSC501 Lab 3 Spring 2009
Demand Paging
Part 1 Due: Mar 26th 11:45 PM
Part 2 Due: Apr 9th 11:45 PM

1 Introduction

Demand paging is a method of mapping a large address space into a relatively small amount of physical memory. This is accomplished by using a "backing store" (usually disk) to hold pages of memory not currently needed. Paging is what allows us to use more address space than is physically available, and to run programs in non-contiguous sections of memory.

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.

2 Goals

The ultimate goal of this project is to provide the facilities to implement the system calls listed below.

3 Implementation

3.1 System Calls

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

Much like its Unix counterpart (see man mmap) it will map 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's heap will be private and exist in its virtual memory.
The size of the heap (in number of pages) is specified by the user.

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. A call to getmem() will continue to allocate memory from the regular Xinu kernel heap.

SYSCALL srpolicy (int policy)

This function will be used to set the page replacement policy to FIFO or Global clock management (GCM). You can declare constant FIFO as 3 and GCM as 4 for this purpose.

SYSCALL vfreemem (block_ptr, int size_in_bytes)

You will implement a corresponding vfreemem() call for the vgetmem(bytes) call. It will take two parameters and will return OK or SYSERR. The two parameters are similar to those of the freemem() call in the original Xinu. The type of the first argument *block_ptr* will depend on your own implementation.

4 Overall Organization

The following sections discuss at a high level the organization of the system, the various pieces we need to implement demand paging in Xinu and how they relate to each other. This handout describes my ideas for implementation in Xinu. 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. If you are going to deviate greatly please check your design with me first.

4.1 Memory and Backing Store

4.1.1 Backing Stores

Virtual memory commonly uses disk space to extend the memory of the machine. However, we must remember that our version of Xinu has no file system support. Also, we cannot use the network to communicate with a page server which will manage the backing stores. Instead, we will emulate the backing store (how it is emulated will be detailed in 4.1.2). To access the backing store you are given the following (These calls are in the directory *paging*.):

1. **bsd_t** The backing store descriptor type is used to reference a backing store. The type declaration for it is in paging.h. This type is merely unsigned int.

   For practical reasons each student is limited to 8 mappings at a time. You will use the IDs zero through seven to identify them.

2. **int get_bs (bsd_t store, unsigned int npages)** This call requests from the page server a new backing store with id store of size npages (in pages, not bytes). If the page server is able to create the new backingstore, or a backingstore with this ID already exists, the size of the new or existing backingstore is returned. This size is in pages. If a size of 0 is requested, or the pageserver encounters an error, SYSERR is returned.

   Also for practical reasons, the npages will be no more than 256.

3. **int release_bs (bsd_t store)** This call requests that the page server release the backing store with ID store.

4. **SYSCALL read_bs (char *dst, bsd_t store, int page)** This copies *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)** This copies a page pointed to 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 will be as follows:

```
        ---------------------------------
                  Virtual heap

             (pages 4096 and beyond)

        ---------------------------------
                  3072 frames

               (pages 1024 - 4096)


        ---------------------------------

          Free Memory (pages 406 - 1024)

        ---------------------------------
              HOLE (pages 160 - 406)

        ---------------------------------
           Free Memory (pages 25 - 160)


        ---------------------------------
  Free Memory Xinu text, data, bss (pages 0 - 25)
        ---------------------------------
```

As you can see the version of Xinu we will be using will compile to about 100k, or 25 pages. The PC has 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 its 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 and network support, we need to emulate the backing store with physical memory. In particular, consider the following Xinu memory layout:

```
        ---------------------------------
                  Virtual heap

             (pages 4096 and beyond)
```

```
---------------------------------
        Backing store

     (pages 2048 - 4096)

---------------------------------
         1024 frames

     (pages 1024 - 2048)

---------------------------------

 Free Memory (pages 406 - 1024)

---------------------------------
      HOLE (pages 160 - 406)

---------------------------------
   Free Memory (pages 25 - 160)


---------------------------------
Free Memory Xinu text, data, bss (pages 0 - 25)
----------------------------------
```

Each backend machine has 16 MB (4096 pages) of real memory. And we reserve the top 8M real memory as backing stores. For practical reasons, we only allow the support of 8 backing stores and each backing store will only map maximum 256 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 - 0x008fffff

backing store 1: 0x00900000 - 0x009fffff

backing store 2: 0x00a00000 - 0x00afffff

backing store 3: 0x00b00000 - 0x00bfffff

backing store 4: 0x00c00000 - 0x00cfffff

backing store 5: 0x00d00000 - 0x00dfffff

backing store 6: 0x00e00000 - 0x00efffff

backing store 7: 0x00f00000 - 0x00ffffff

In the implementation, you need to "steal" physical memory frames 2048 - 4096 (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 port of memory.

4.1.4 Page Tables and Page Directories

As stated before page tables and page directories can be placed in any free frame. For this project you will not be paging either the page tables or page directories. Because page tables are always resident 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 Support Data Structures

4.2.1 Finding the Backingstore 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 accomplish this you need to keep track of which backing store the process was allocated when it was created using vcreate. Finding the offset (i.e the particular page within the store to write/read from) can be calculated using the virtual page number. 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:

{ pid, vpage, npages, store }

You should write a function that performs the lookup:

f (pid , vaddr)= > {store, pageoffset within store}

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 have noticed 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:

{ framenumber, pid, virtual pagenumber }

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 (the age of the page in the frame, any data needed to estimate page replacement policy, etc.).

4.3 Process Considerations

With each process having its own page directory 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 the 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()**. Because you are limited to 8 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 it's 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 by a process should persist unless the process explicitly releases them).
4. The frame used for the process' 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 process 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 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

set evec(int interrupt, (void (*isr)(void)))

routine (see evec.c).

Your ISR should be a routine written in assembly (it will not work correctly if you write it in C). The first and last things it should do are save and then restore all general purpose registers respectively. It must, at some point, remove the error code from the top of the stack. It should use iret (see Intel Volume II), not ret, to return once finished.

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 of 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). 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 pth page table. If the pth 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 increment the reference count of the frame which holds pt. This indicates that one more of pt's entries is marked "present."
    3. Obtain a free frame, f .
    4. Copy the page o of 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 pth 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. Using the backing store map 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, viz. FIFO and global clock page replacement. For this you will want to make use of page's accessed bits. There are also three bits in the page table entries which you may use for any purpose. You are free to add whatever structures you'd like in your inverted page table, or extra processes if needed. You may wish to consult Silberschatz chapters 9 and 10 and the Silberschatz slides on the web.

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 calls interfaces should be modified.

6 Details on the Intel Architecture and Xinu

After having read chapters two and three in volume 3 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.
6. 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 labeled bits is also given.

7 Given Code

In this lab, we will use another version of xinu that can be downloaded in the following link: http://fairfax.csc.ncsu.edu/csc501/lab3/lab3-fall09.tgz This version of Xinu contains the skeleton code for the backing store calls. It also has .h files needed for this project.

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

9.1 Part 1 (Due: Mar 26th 11:45 PM)

The goal of the first part of this lab is to provide an intermediary checkpoint for lab3, and pave the way to a more functionnal virtual memory system that you will complete in part2.

With the first part of the lab done, your system supports:

- Memory mapping: mapping of the first 16 Mb of physical memory, and the xmmap() and xmunmap() system calls
    - 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: 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 on 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.

Electronic turn-in instructions (**Make sure that your code compiles and that you have turned off all your debugging output!):**

i) copy the entire csc501-lab3 directory to csc501-lab3p1, go to the csc501-lab3p1/compile directory and do "make clean".

ii) go to the directory of which your csc501-lab3p1 directory is a subdirectory (NOTE: please do not rename csc501-lab3p1, or any of its subdirectories.)

e.g., if /home/csc501/csc501-lab3p1 is your directory structure, goto /homes/csc501

iii) create a subdirectory TMP (under the directory csc501-lab3p1) and copy all the files you have modified/written, both .c files and .h files into the directory.

iv) compress the csc501-lab3p1 directory into a tgz file and use Wolfware's Submit Assignment facility. Please only upload one tgz file.

```
tar czf csc501-lab3p1.tgz csc501-lab3p1
```

9.2 Part 2 (Due: Apr 9th 11:45 PM)

Your system should have a full support of all above system calls and features. In particular, in addition to those features implemented in part1, your system is also expected to support

- Memory mapping
  - different running processes created with *vcreate* can have its own private heap
  - vgetmem, vfreemem: implemented and fully functional
- Page replacement: FIFO and GCM

Electronic turn-in instructions (**Make sure that your code compiles and that you have turned off all your debugging output!):**

i) copy the entire csc501-lab3 directory to csc501-lab3p2, go to the csc501-lab3p2/compile directory and do "make clean".

ii) go to the directory of which your csc501-lab3p2 directory is a subdirectory (NOTE: please do not rename csc501-lab3p2, or any of its subdirectories.)

e.g., if /home/csc501/csc501-lab3p2 is your directory structure, goto /homes/csc501

iii) create a subdirectory TMP (under the directory csc501-lab3p2) and copy all the files you have modified/written, both .c files and .h files into the directory.

iv) compress the csc501-lab3p2 directory into a tgz file and use Wolfware's Submit Assignment facility. Please only upload one tgz file.

```
tar czf csc501-lab3p2.tgz csc501-lab3p2
```

10 A 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) In the specification of the vcreate() system call, the parameter *hsize* refers to the heap size as number of pages (not number of bytes).

2) Where can I get notes on GCM?

You can read the class slides for a fairly high level description of GCM.

3) Should I integrate this lab with the previous labs?

No, you need not.

4) 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.

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

This mapping is maintained inside the kernel. Since the "store" can take only 8 values at the most (because there are only 8 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 8 entries. This table is placed in 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 8 entries of the type of the mapping and that's all. It is pretty similar to semaph[] and proctab[].

6) 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 *fifo* and srpolicy(GCM), 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.

7) 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. But, I am mentioning them just in case and don't read too much into this answer and confuse yourself).

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

The page tables and page directories are to be placed in the following memory range: If your memory is divided into 4096 pages, then they should be placed in the range 1024-2048. 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.

*** 9) What is the use of xmmap()?

There was a big misconception about the usage of xmmap() among many students. When does a user process call xmmap()?. Why is it used for?. Before answering these questions, let me make one point very clear.

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? I will give you one **example**

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

}
```

/* This virtual heap will be present in a backing store that is exclusive for this process alone. (No backing store will be shared across processes. Neither will the same backing store be mapped to multiple memory ranges.) */

/* This virtual heap present in a backing store will be mapped from in the address ranges 4096th page to 4196th page of this process. So, the backing store mapping table you maintain will contain an entry < process B's pid, 4096, 100, backing store number > */

```
   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 virtual heap to check if the previous write was successful */

            vfreemem(--x, 1000); /* frees the allocation in the virtual heap */
}
```

In the previous example, you accessed virtual memory, found that the access creates a page fault and will handle the page fault. This example some fair idea about how to use your virtual memory and how to test your implementation of virtual memory.

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(...);
```

/* This can be thought of as you creating 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. But, instead of using file I/O operations, since you have mapped the file to a memory location, you modify the file by means of a memory modification !! */

Let us say there is another process B which executes the following code after a while the prev code was executed (assume that 'A' has not executed release_bs(4))

```
   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' */

I think, these examples should make the usage of xmmap() more clear. Think about it.

10) Page fault handling routine (page fault ISR) - What should be done here?

Most of the students have had trouble writing this correctly. So I am giving a psuedo code for the implementation (which is easier if you do it in assembly)

```
clear all interrupts
Store error code in a global variable  /* if you use any temp register to do this, then make sure that you restore that value too */
save all registers
call C function to handle the interrupt and do all the required processing
restore all registers
remove error code from stack
restore interrupts
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. Not everything has to be implemented in assembly. It is very difficult. So, you include a call to a C function which will handle everything.

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

They are, but can be used inside the page fault handling routine. In fact, you cant escape from that.

12) A request/directive

While using get_bs(store, pages), please don't make the *pages* parameter take values greater than 200. This might cause unnecessary congestion in the page server resulting in its crash (which happens often nonetheless). So, limit to get_bs (store, 200) at the maximum.

13) How do I test my replacement policies?

The spec says that free memory in main memory from 1024th page to 4096th page accounts for 3072 free frames. The above request asks you not to have more that 200 pages in a single backing store. There are 8 backing stores available for you. So, totally, you can have 1600 pages of virtual memory of different processes. This entire 1600 pages can be accommodated in our 3072 free frames easily. So, there will be no need for any page replacement at all???? Then, how do we check the page replacement policy.

There is a constant called NFRAMES in paging.h which has a value of 3072. Make sure that your entire code depends on this constant as a measure of the available free frames. If this constant has a value of 3072, then we will have the problem stated above. But, if we change the constant's 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 ample scope to test your replacement policy by changing the NFRAMES constant.