Lab 3: Virtual Memory and Demand Paging (100 pts)

CS 503 Spring 2018

Due: Sunday, 29th April 11:59pm

In this last lab, you will be implementing paging mechanism on top of XINU. This lab has more workload than previous ones, so do start early!

Check your assigned port range to use.

The code base for Lab3 could be found at:

```
tar zxvf /u/u3/cs503/xinu-spring2018/xinu-qemu-paging.tar.gz
```

> NOTE: this code base contains the code for the backing store calls, thus it's recommendded to start with provided code base directly (instead of integrating with previous labs if you were doing so).

> NOTE: Similar to Lab2, the provided code base supports both QEMU and Galileo environments. Again, as our grading will be based on Galileo, you should double-check if your submission is working correctly on Galileo.

> NOTE: This is a long handout. Please read this handout very carefully before starting this assignment. If you do not understand anything, please seek the TA in the PSO sessions.

Overview

We have discussed the myriad benefits that virtual memory brings which justifies the investment in hardware support to achieve efficiency and speed. Demand paging allows mapping a large virtual/logical address space onto a potentially much smaller and non-contiguous (i.e. fragmented) physical address space. Main memory (i.e. RAM) is viewed as a cache comprised of fixed size blocks called frames in which pages (of equal size as frames) containing instructions and data referenced by processes are held. Due to the size mismatch between virtual and physical memory, a significant part of a process's virtual address space (i.e. pages that it references) is kept in

secondary persistent storage such as disk or flash memory (i.e., solid state drive).

When a kernel runs out of frames to store pages needed by a process, some pages that are resident in RAM must be evicted to secondary storage to free up frames. In XINU this is accomplished by using a “backing store” that is implemented as a remote disk on a frontend Linux PC in the XINU Lab. The server that handles the remote disk functionality on a frontend is rdserver whose code is in `/rdserver/`. The backing store is accessed by the XINU kernel and its processes running on backend machines through a set of APIs that invoke the services of `rdsproc` which is implemented as the pseudo-device RDISK (check `/config/`). `rdsproc` is a high priority process that shows up under `xsh` when running the `ps` command, you may also have seen it in previous labs. The APIs open RDISK and read/write to it to access the remote disk that serves as secondary storage on a frontend. Find more details on the APIs later.

Note that the XINU backing store may be slower than a local disk since network communication is involved to read/write pages on remote disk. `rdserver` on a frontend communicates with `rdsproc` running on a backend. UDP is used as the transport protocol in order to achieve low overhead, hence, the backing store is unreliable and read/write requests may fail. (Take course CS536 for details of UDP, etc.) This is expected to be rare in the environment of XINU lab, however, you’ll need to consider such error cases during implementation.

Virtual Memory System Support

To help with implementing the VM/demand paging related system/function calls, we provide a description of the overall organization of XINU’s demand paging system, including the key components and how they relate to each other. You may choose to adopt a different implementation strategy if you think it is more effective for realizing the system calls in Section 2. If you are taking your own route, please check with the TAs so that they are kept in the loop.

Important: You will need to consult the Intel manuals, in particular, volume III, chapters 2-5, to acquaint yourself with Intel specific hardware support for virtual memory and demand paging. Intel manuals: Volume 1 I Volume 2 I Volume 3

Demand Paging Services

The coding part of Lab3 centers on implementing a set of system calls that export virtual memory services to user processes:

```
pid32 vcreate(void *funcaddr, uint32 ssize, uint32 hsize, pri16 
```

This system call creates a new XINU process. Its difference from create() is that the process's heap will be private and reside in virtual memory. The size of the heap (in number of pages, not bytes) is specified by `hsize`.

```
char* vgetmem(uint32 nbytes);
```

Much like `getmem()`, `vgetmem()` allocates the requested amount of memory, if available. The difference is that `vgetmem()` gets the memory from a process's private heap in virtual memory. A call to `getmem()` will continue to allocate memory from the regular XINU kernel heap.

```
syscall vfreemem(char *blkaddr, uint32 nbytes);
```

`vfreemem()` is a counterpart of `freemem()` for private heap in virtual memory. It takes two parameters which are similar to those of `freemem()`.

```
syscall srpolicy(int policy);
```

This function will be used to set the page replacement policy. Available policy options are defined in `/include/paging.h`. By default, we are doing FIFO. In the Bonus Problem, you will implement another GCA policy, the `Global Clock Algorithm`.

This system call will NOT be invoked at arbitrary places inside your code to force changing from one replacement policy to another. So don't worry about switching from one replacement policy to another midway through execution. You may assume that the default policy is FIFO and if `srpolicy(...)` is called, it will be the first statement in the program.

Backing Stores and Memory Management

Backing Stores

backing store

Virtual memory commonly uses disk space or flash RAM (i.e. SSD) to extend the memory of the machine. This version of XINU does not provide local disk file system support. In place of secondary storage implemented on local disk, we will use a networked remote disk on a frontend Linux PC. A page server, `rdserver`, running on the frontend will act as a backing store. To access the services of `rdserver`, the XINU kernel and its processes must go through `rdsproc`, the high priority process that runs on a backend. `rdsproc` conveys backing store requests by XINU and its processes to `rdserver`, and relays the response from `rdserver` to XINU and its processes. `rdsproc` (source code in `/device/rds/`) implements the

remote disk pseudo-device RDISK (see `/config/`) which is accessed via `open()` / `read()` / `write()`. An API for backing store related operations on RDISK is provided through following definitions/functions (see `/paging/`):

```
bsd_t
```

The backing store descriptor type is used to reference a backing store. The type declaration is specified in `/include/kernel.h`. For practical reasons, each student is limited to 8 mappings at a time (ranging from `MIN` ID `to` MAX `ID`). See `paging.h` and also `page` server.h `in` /include/`.

```
bsd_t allocate_bs(uint32 npages);
```

Allocates a free backing store; returns a bs number if successfull or SYSERR if they are all occupied.

```
bsd_t deallocate_bs(bsd_t store);
```

Deallocate the backing store if it is not being used and return the bs number; return SYSERR otherwise.

```
bsd_t open_bs(bsd_t store);
```

Return the store number if the backing is allocated and increase the use count; return SYSERR otherwise.

```
bsd_t close_bs(bsd_t store);
```

Return the store number if successful and decrease use count; return SYSERR otherwise.

```
syscall read_bs(char *dst, bsd_t store, uint32 pagenum);
```

This copies the `pagenum`'th page from the backing store referenced by ID `store` to memory pointed by `dst`. It returns OK on success, SYSERR otherwise. The first page of a backing store is page zero.

```
syscall write_bs(char *src, bsd_t store, uint32 pagenum);
```

This copies a page pointed to by *src* to the *pagenum*'th page of the backing store referenced by store. It returns OK on success, SYSERR otherwise.

Note that `read_bs()` and `write_bs()` are both blocking calls. But they can still be used inside our page fault interrupt handling routine. In fact, we

are unable to avoid that under the environment of “remote network disk”.

Generally, there is no need to modify them, unless you decide to implement additional optimizations.

As noted above, please take into consideration that demand paging may incur long delays due to its networked nature in XINU. There is also the possibility that calls may fail and return SYSERR due to lost packets. This is expected to be rare in the environment of XINU lab. However, you must check for error conditions in your code and take appropriate actions to prevent erroneous program execution. Note that the provided API are blocking calls that result in rescheduling. This is a consequence of disk I/O (even more so, remote disk I/O) being significantly slower than RAM.

The paging API uses a global table `bstab[]` with 8 entries to store information about each mapping. You may choose to make a similar one or extend it when maintaining your data structures later on.

Side note: there was also `get_bs()` and also `release_bs()` API calls which may lead to the scenario where different processes have conflicts using backing stores. The API we provide here is building on top of these, but tries to avoid that potential problem with the help of additional book-keeping. So basically you don’t need to invoke those two old functions directly. Text

Using the Paging Server API

> NOTE: This step is not required when testing on QEMU, as RDISK device is emulated in QEMU environments (i.e., no need to run `rdserver` on a frontend machine). For more details, you may have a look at implementations in `device/qrds`.

The paging server API (`get_bs()`, `read_bs()`, `write_bs()`, `release_bs()`) through RDISK (implemented using `rdsproc` and `rdserver`) creates a file on a frontend Linux PC which serves as the backing store of XINU’s demand paging subsystem. The mapping between the pages in a backing store and the blocks on the remote disk file are calculated in the `read_bs()`/`write_bs()` calls. The block size for the remote disk is 512 bytes (`RD_BLKSIZ`, as defined in */include/rdisksys.h*), and each mapping is translated to 205 pages (`RD_PAGES_PER_BS`, as defined in */include/page_server.h*), which is equivalent to 1640 blocks on the remote disk (recall a page is of size 4096 bytes on our x86 backends). In the current version of XINU, only 200 pages (`MAX_PAGES_PER_BS`, as defined in */include/page_server.h*) are allowed per mapping.

The remote disk server needs to be started before you load and run the XINU kernel. The code for rdserver is in */rdserver/*. Run

```
make clean
make
```

in this directory and then start the server by:

```
./rdserver [port number]
```

The port number used in the code given to you is “33124” as specified in */include/rdisksys.h*, `RD_SERVER_PORT`. The server runs in broadcast mode, hence anyone who uses the same port number will access the same server.

Important: To avoid conflict, each student will be assigned a range of port numbers to use. Check here for which port numbers you could use. Then change the port number in */include/rdisksys.h*, and also specify the right port when starting `rdserver` on a frontend PC.

If you look into the code of `psinit()`, you will find that it opens a connection to the remote disk and creates a file named “backing_store”. This file will contain all the data you read/write using the paging API. The rdserver can be terminated by issuing “`Ctrl-C`” but note that the file is persistent. You will need to manually delete the file *backing_store* in the */rdserver/* directory to reset and start afresh. Note that command

```
make clean
```

will also delete the *backing_store* file. Remember to do `make clean` before submitting. Please note that it is the programmer’s responsibility to make sure a backing store is allocated before using it, and to open/close in right sequence.

Memory Layout

The basic XINU memory layout for our x86 backend is as follows:

Desc.	Frame number	Virtual Address	Global?	Identity map?	
Memory reserved for Boot loader	0 - 256	0x00000000 - 0x000FFFFF	Y	Y	
XINU text, data, bss	257 - 326	0x00100000 - 0x00145FFF	Y	Y	
free memory	327 - 1023	0x00146000 - 0x003FFFFF	Y	Y	
metadata space	1024 - 4095	0x00400000 - 0x00FFFFFF	Y	Y	
virtual heap	-	0x01000000 - 0x8FFFFFFF	N	N	
device memory	-	0x90000000 - 0x903FFFFF	Y	Y	

As gleaned from the memory layout, the paging version of XINU we will be using compiles to about 70 pages.

We will place the ***metadata space*** into pages 1024 through 4095, giving a total of 3072 frames. These (initially) meatadata space will be used to store resident pages, page directories, and page tables. Hence they are part of what would be considered “kernel memory”.

The ***free memory*** (in the range from page 327 to page 1023) is used for XINU’s kernel heap which is organized as a free list. `getmem()` and `getstk()` obtain memory from this area, allocated from the bottom and top, respectively. This conforms with the XINU memory layout.

All memory below page 4096 will be “global” (or shared). That is, it is usable and visible by all processes. Moreover, these global pages are accessible by simply using physical addresses. In other words, all page tables for page entries 0-4095 implement an identity map (i.e., the physical address is the virtual address).

Device memory is 1024 pages starting at memory address 0x90000000 and must be mapped at the same physical address. Hence, the first four page tables and the page table for device memory for every process will be the same, i.e. they are shared/global with an identity map.

Private heap is private memory space for each process, and it is located at page 4096 and above. This private address space is visible only to the process that owns it. A process's private heap is mapped somewhere in this area by `vcreate()`.

Page Tables and Page Directories

Page tables and page directories can be placed in any free frame of the designated area within the metadata space. 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 page, 4K.

For Lab3, you will not be paging either the page directories or page tables. That is, page directories and page tables should not be relying on backing stores. For page directory, they must always be allocated. However, it is not practical to allocate all potential page tables for a newly created process. Therefore, to conserve memory, page tables are created on-demand. The first time a page is touched (i.e. it has been mapped by a process), a page table needs to be allocated. Conversely, when a page table is no longer needed, it should be removed in order to conserve space.

In `/include/paging.h`, there are two struct `pd_t` and `pt_t` which represent Page Directory and Page Table, repectively. They both contain a lot of bit fields. When initializing these fields:

- For page directories, set the following bits: set `pd_write` always and set `pd_pres` whenever the corresponding page tables are present in the main memory). You can make all the other bits zero.
- For the four global page tables, set the following bits, `pt_pres` and `pt_write`. You can make others zero.

This should be fairly straightforward after reading the Intel manuals carefully.

Used paging server API to take a remote disk as backing store of pages

Installed page fault interrupt service routine (ISR) and implemented page faults handling scheme

Applied the FIFO and the GCA(Global Clock Algorithm) policies for page replacement when handling page faults

Page Faults & Page Replacement

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 handle a page fault, 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 valid address, if not, print an error message and kill the process.
5. Let `p` be the upper ten bits of `a`.
6. Let `q` be the bits [21:12] of `a`.
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 bring 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`.

Page Fault Interrupt Service Routine (ISR)

A page fault triggers interrupt number 14 (see Intel manual). When an interrupt occurs, CS:IP are pushed onto the stack, followed by an error code (see Intel Manual, volume III, chapter 5).

```
-----------
error code  <--- stack top
-----------
IP
-----------
CS
-----------
...
```

Execution then jumps to the ISR. We do not use a common dispatcher for this ISR. To specify the page fault ISR in the interrupt vector, please use

```
set_evec(uint32 interrupt_number, (void (*isr)(void)));
```

Your ISR should be a routine written in assembly (you cannot write the entire code in C). You may invoke a function written in C from the assembly to simplify assembly code writing. The first and last things the ISR does are to

save and restore all general purpose registers. It must also remove the error code from the top of the stack at some point. Like other ISRs, it should use `iret` (see Intel Manual, volume II), not `ret`, to return when finished.

This is supposed to involve some assembly code. After Lab1, you should already have some experience writing in assembly. But since we all know that writing full handler routine in assembly may be error-prone and hard to debug, it's recommended to implement only those necessary parts in assembly, and call a C function which implements full functionality. Here we provide a pseudo code snippet for guidance.

1. clear all interrupts
2. Store error code in a global variable. (If you use any temp register to do this, then make sure that you save/restore that value as well).
3. save all registers
4. call a C function to handle the interrupt and do all the required processing
5. restore all registers
6. remove error code from stack
7. restore interrupts
8. iret

Obtaining a Free Frame

When a free frame is needed, it may be necessary to remove a resident page from a frame. How you pick the page to remove depends on your page replacement policy. The function to find a free page may have the steps below:

1. Search the inverted page table for an empty frame. If one exists, stop.
2. Else, pick a page to replace (using the current replacement policy).
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 process id 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 Manual, volumes II and III for more details on this instruction).
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 "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 the store, given `pid` and `a`. If the lookup fails, something is wrong. Print an error message and kill the process with id `pid`. 2. Write the page back to the backing store.

Page Replacement Policies

Implement FIFO as the default page replacement policy. In the Bonus Problem, you will be asked to implement another GCA policy.

To test your replacement policies, you may need to modify `NFRAMES` macro defined in `/include/paging.h`.

The specifications say that the free memory in the main memory from the 1024th page to the 4095th page accounts for 3072 free frames. It's generally recommendded not to have more that 200 pages in a single backing store (the hard line is 205, as defined in `RD_PAGES_PER_BS` in */include/page_server.h*. There are 8 backing stores available for you. Hence, you can have at most 1600 pages of virtual memory in total for different processes. This entire 1600 pages can be easily accommodated in our 3072 free frames. Then there will be no need for any page replacement at all!

There is a constant called `NFRAMES` in *`/include/paging.h`* which has a value of 3072. Make sure that your entire code depends on this constant (not its value) 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 value of the constant to (say) 400, then the number of free frames initially available is only 400, i.e. your main memory looks as if it only has `1024 + NFRAMES = 1024 + 400 = 1424` frames of memory. Thus, you have an ample scope to test your replacement policy by changing the NFRAMES constant.

Initially, `NFRAMES` is set to 3072, so that you don't need to consider about page replacement policy until you've finished everything else. In grading, we may use a small number to test page replacement policy.

Support Data Structures

Backing Store Map: Finding the Backing Store for a Virtual Address

If a process can map multiple address ranges to different backing stores, how does one determine which backing store a page needs to be read from (or written to) when it is being brought into (removed from) a frame? To handle this problem, please keep track of which backing store a process

was allocated when it was created using `vcreate()` (vcreate needs to call **backing store APIs** to allocate a backing store for the virtual heap). Finding the offset (i.e. the particular page within the store to write/read from) can be computed 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 is a tuple such as:

```
<pid, vpage, npages, store>
```

This mapping is maintained inside the kernel. Since the *store* can take only 8 values at 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 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 8 entries of the type of the mapping and that's all. It is pretty similar to `semtab[]` and `proctab[]`.

Also write a function that performs the following lookup:

```
f(pid, vaddr) => (store, page_offset within store)
```

Inverted Page Table: Mapping from physical memory space to virtual memory space

Page tables basically map from virtual address space (i.e., virtual page number) to physical address space (i.e., frame number). Thus, when writing back a dirty page, the only way to determine which virtual page and process (and thus which backing store) a dirty frame belongs to, is to traverse the page tables of every process looking for a frame location that corresponds to the frame we wish to write back. This is extremely inefficient. To circumvent this, we use another kernel data structure: *inverted page table*, which maps from frame number to pid and virtual page number.

The inverted page table is an array, where each element is a tuple of the following form:

```
(frame_number, pid, virtual_page_number)
```

If we use an array of size `NFRAMES`, the frame number is implicit and we can just index into the array. With this additional 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 to the page in

frame *i*. You may also extend to use this table to hold other information for page replacement.

Considerations

Please take into consideration the following when completing this assignment.

System Initialization

The NULL process is special in that it is custom created (not through `create()`) in `sysinit()`. The NULL process does not have a private heap, and neither do processes spawned by `create()`.

To set up demand paging, write a function `initialize_paging()` which does the following:

1. Initialize all necessary data structures.
2. Allocate and initialize a page directory for the NULL process.
3. Create the page tables which map pages 0 through 4095 to the 16 MB physical address range (Note: 4096 * 2^12 = 16 * 2^20). We will call these *global page tables*.
4. Create a page table for mapping the device memory starting at `0x90000000`. You need to map 1024 pages starting at this address. This means you will need to fill page directory entry at index #576 (0x90000000 >> 22) and all the entries in the corresponding page table.
5. Set the `PDBR` register (i.e., `CR3` register) to the page directory of the NULL process.
6. Install the page fault interrupt service routine.
7. Enable paging.

`initialize_paging()` should be called in initialize.c immediately after the call to `sysinit()`.

> Note: `PDBR` stands for the page directory base register, also known as `CR3`.

Process Creation

When a process is created, we must also create a page directory and record its address. As noted above, the first 16MB of each process's virtual address space will be mapped to the 16MB of physical memory, and the 1024 pages starting at `0x90000000` will be mapped to the same physical memory range for memory-mapped devices. Hence this initialization is common to all processes.

A separate mapping must be created for a new process's private heap when created with `vcreate()`.

Process Destruction

When a process terminates, 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 should be released (recall that backing stores allocated by a process should persist unless the process explicitly releases them).
4. Frames used for the process's page directory and page tables should be released.

Context Switching

As we switch context between processes, we must also switch between memory spaces. This is accomplished by updating the `PDBR` register with every context switch. This register must always point to a valid page directory that is in RAM at a page boundary.

Hints/Notes

After reading chapters two and three in Volume 3, you should have got a basic understanding of memory management in the Intel x86 architecture. Some more remarks/reminders are as below:

1. XINU uses the flat memory model, i.e. physical address = linear addresses. This is achieved by "neutralizing" the addressing contribution of segmentation.
2. The segments are set in *meminit.c* in the function `setsegs()`.
3. Pages are 4K (4096 bytes) in size.
4. To compute the virtual page number from a virtual address, simply grab the most significant 20 bits of a virtual address form the virtual page number. After reading the materials, the answer shall be straightforward.
5. Some demo code is given by asm_ex.c for getting and setting the control registers, and dump32.c for dumping a binary number with labeled bits.

Debugging

Using `objdump -D` on the file `xinu.elf` can help you locate where your program crashed. 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. It is better to test your code module-by-module, and maintain working copies (to which you can revert back) using some version control tool.

Instrumentation Hooks

You'll need to implement this function for the instrumentation:

```
uint32 get_faults();
```

Your implementation should maintain a count of the number of times the page fault handler has been called. `get_faults()` will return this number.

Besides, you'll need to add instrumentation hooks to your page fault handler. These hooks are functions that will be called by your code to report what it is doing:

```
void hook_ptable_create(uint32 pagenum);
```

For *pagenum*, you may pass in either the frame number, or any page in this page table. This hook should be called when your implementation is creating a page table.

```
void hook_ptable_delete(uint32 pagenum);
```

For *pagenum*, you may pass in either the frame number, or any page in this page table. This hook should be called when your implementation is deleting a page table.

```
void hook_pfault(int16 procid, void *addr, uint32 pagenum, uint3
```

where *addr* is the address that for which the page fault was generated, *pagenum* is the pagenumber this corresponds to, and *framenum* is the frame number which has been selected for this page. This hook should be called right before the page fault handler is about to return.

```
void hook_pswap_out(int16 procid, uint32 pagenum, uint32 framenu
```

where *pagenum* is the number of the page being swapped out; *framenum* is the number of the frame that contains the page being swapped out, and *procid* is the process whose page is being swapped out. This hook should be called when your implementation is replacing (swapping out) a frame.

The code for hooks is already provided in */system/hooks.c*, remember to invoke them at the right moment. You might want to comment out the `HOOK_LOG_ON` macro in */system/hooks.c* when you do not want to be disturbed by this output. Note that we will replace this file in grading, so do not let your implementation depend on this file.

Testcases

For the purpose of this lab, you will be provided two simple testcases. These testcases are provided in */system/page_policy_test.c*. This file checks both the page fault handling and page replacement policy within one process created using *vcreate*. The test virtual process creates 99 pages and attempts to read and write to all of them which induces page faults.

For page fault handling, please set PAGE_REPLACEMENT to 0 and NFRAMES (include/paging.h) to a value greater than or equal to 200. Since there are enough free frames available, the page fault handler (if implemented correctly) will simply allocate these frames to the process. If the process ends gracefully i.e., it doesn't hit an assert, the test has passed successfully.

For page replacement testing, please set PAGE_REPLACEMENT to 1 and NFRAMES (include/paging.h) to a value less then or equal to 50. Also, if your instrumentation hooks are not correctly placed, there is no guarantee on the accuracy of this test. For this part, since NFRAMES < ALLOCATED_PAGES, there is bound to be page replacement. The hooks will panic in case the page replacement policy (FIFO) is not being followed. If the test runs till the end, with the hooks in place, your test passes.

Bonus Problem (25 pts)

The default page replacement policy is FIFO, in the bonus problem, you are asked to implement another GCA (*Global Clock Algorithm*) policy. The description on GCA can be found at Chapter 10.14 in textbook. Use the GCA macro as defined in */include/paging.h*. Select your policy by calling `srpolicy(GCA)`. Note that your `srpolicy(GCA)` call should return OK if you have added the support for GCA policy. To traverse through the list of frames for testing/setting bits in GCA policy, you may just start from the frame next to where you last stopped, thus in a round-robin fashion.

Also create a workload of memory intensive processes, and demonstrate that your policy improves performance comparing to FIFO. (Hint: using the hooks and monitor statistics in different cases). Discuss your results in **Lab3Answers.pdf** (put under */system/*). Since we didn't require writings for

non-bonus part, if you do not plan to implement the bonus problem, you don't need to submit **Lab3Answers.pdf**.

Turn-in Instructions**

Electronic turn-in instructions:

1. Go to `xinu-spring2018/compile/` directory and do `make clean`.
2. Go to `xinu-spring2018/rdserver/` directory and do `make clean`.
3. Go to the directory of which your `xinu-spring2018` directory is a subdirectory. (NOTE: please do NOT rename xinu-spring2018, or any of its subdirectories!)

   e.g., if ~/cs503lab3/xinu-spring2018 is your directory structure, go to ~/cs503lab3/ and turnin there.
4. Type the following command:

```
turnin -c cs503 -p lab3 xinu-spring2018
```

5. After submitting your files, you may want to check that it's indeed submitted using command:

```
turnin -v -c cs503 -p lab3
```

Important: Please comment your code changes in XINU such that (a) where changes are made is highlighted, and (b) what changes are made is conveyed.

Important: You can write code in main.c to test your procedures but please note that when we test your programs we will replace the main.c file. Therefore, do not put any functionality in the main.c file. All debugging output should be turned off before you submit your code.