CS330: Operating Systems
Assignment#3

1 Preamble

In this assignment, you will implement a shared memory interface and demand paging. You will test your shared memory implementation by writing a few test programs. Download `cs330assignment3.zip` and unzip it in your home directory. This will create a new directory named `cs330assignment3/` and place my version of NachOS under it. You should compile and run NachOS only from the `nachos/code/userprog/` directory. You should run NachOS as follows.

```
> ./nachos -rs s -A n -P p -x ../test/executablename
```

The `-rs` option makes the timer yields happen at random points in time. Here `s` is a randomization seed (an integer). If you use different seeds in different runs, the preemption points will be different. Use zero as the seed value (i.e., `-rs 0`) for reporting all your results. This is important because changing the seed value may change the results slightly leading to potential confusion in grading. However, feel free to experiment with other seed values, but do not report these results. Your implementation should run without any error for arbitrary seed values. Here `n` is an integer from the set $\{1, 2, 3, 4\}$ and `p` is an integer between 0 and 100 (both ends inclusive). The value of `n` determines the chosen scheduling algorithm and the value of `p` determines the base priority of the thread. The four scheduling algorithms are defined in `nachos/code/threads/system.h`. The quantum used by algorithms 3 and 4 are defined as `SCHED_QUANTUM` in `nachos/code/threads/system.h`. You will use algorithm 3 (round robin) for this assignment. Set both `SCHED_QUANTUM` and TimerTicks to 100. The `-P` flag is optional unless `n` is chosen to be 4. You should specify 3 for `n`, unless you are asked to do otherwise.

In the following, I will discuss the steps involved in this assignment.

2 Implementing Shared Memory

To implement shared memory, I have introduced a new system call in `nachos/code/userprog/syscall.h`. This system call is `SysCall_ShmAllocate`. The corresponding function that the user programs can use is also declared: `syscall_wrapper_ShmAllocate()`. This function takes the size of the shared memory to allocate in bytes and returns the starting virtual address where the shared memory region is attached to the calling process's address space. You need to use it in the following way in your test programs. Make sure to allocate all shared memory before invoking any `syscall_wrapper_Fork()`. Suppose you have two integers that you want to share across the parent and the child.

```
int main () {
    ...
    int *array = (int*)syscall_wrapper_ShmAllocate(2*sizeof(int));
    /* Now you can use array[0] and array[1] as the two shared variables. */
}
```

You can make multiple calls to `syscall_wrapper_ShmAllocate` to allocate different types of shared memory regions e.g., int, unsigned, char, etc.. Your task is to implement the `SysCall_ShmAllocate` system call in `nachos/code/userprog/exception.cc`. This system call should do the following. It should allocate a new page table for the caller with number of entries equal to the number of current entries plus the number of pages needed to cover the requested shared memory region size. It will copy the existing page table entries from the old page table into the new one. It will also allocate physical pages for the shared memory region and set up the virtual to physical maps for these pages in the remaining entries of the new page table. You have to introduce a shared field in the TranslationEntry class defined in `nachos/code/machine/translate.h` to remember which pages are shared. You need to modify `SysCall_Fork` as follows. On a `syscall_wrapper_Fork()` call, the virtual to physical maps of the shared pages are just copied from the parent to child's page table and no new physical pages would be allocated for these pages; otherwise `syscall_wrapper_Fork()` works as already implemented. The `SysCall_ShmAllocate` system call returns the starting virtual address of the first shared page within the range of allocated shared pages. Remember to free the old page table of the caller and set the KernelPageTable and KernelPageTableSize fields in the Machine class correctly. For testing your shared memory implementation, I have included two test programs, namely, `shmtest.c` and shmtest1.c in the `nachos/code/test/` directory. In the first program, two threads concurrently increment a shared counter. In the second program, the updates to the shared counter are done by one thread at a time. You should feel free to write new test programs with more than two threads and more complex use of shared memory to stress-test your implementation.

3 Implementing Demand-paged Virtual Memory

The current implementation of NachOS allocates the complete address space of a thread when the thread is created. This is not how realistic systems work. The entire executable may not be needed by the thread at the same time. You will change the implementation such that as and when the first access to a virtual page comes (causing what is known as a page fault), we allocate the next available physical page, and register this mapping in the page table entry corresponding to the virtual page. When a new test program is started, the number of virtual pages is calculated from the executable file, as usual. Accordingly, the page table is allocated and all entries are initialized as invalid (i.e., valid is FALSE). At this point no physical page is allocated. When a virtual page is accessed for the first time, the next available physical page is assigned to the virtual page, the page table entry is changed to valid state, and the virtual to physical page mapping is registered in the entry. At this time, the page is zeroed out and the contents of the page from the executable file are copied into the correct locations of the memory array. When a thread is forked, you should, as usual, create a number of page table entries equal to the number of page table entries of the parent and copy the contents of the valid physical pages of the parent into child's physical pages. Beyond this point the parent and child will allocate their pages on demand independently as and when accesses come. Also, change the implementation of `SysCall_Exec` appropriately. In other words, `SysCall_Exec` will not allocate any physical page frame, which will be allocated on demand as the new executable makes memory accesses. It should free the page table of the caller and allocate a new one depending on the size of the new executable. It should free all the physical pages belonging to the caller, except the shared ones. These pages are now available for allocation to other threads. All memory accesses go through the Translate method of the Machine class. This is where the page table is looked up to find the virtual to physical page mapping. When a virtual page is found in an invalid state in the page table, it should be counted as a page fault, and the page fault is handled by allocating the next free physical page and registering this mapping in the page table entry. Your task is to detect page faults, count them, and handle them on demand. Your simulation should report the number of page faults at the end. Assume a page fault latency of 1000 ticks. During this time the thread undergoing the page fault must remain in BLOCKED state. Make use of the sleep queue developed in the first assignment for this purpose. When the process is scheduled again, it should start execution from the instruction that suffered from a page fault. In summary, the page fault needs to be modeled as a restartable exception. Remember to model the page faults correctly in a `syscall_wrapper_Fork()` call i.e., the caller may transition in and out of the BLOCKED state

multiple times depending on the number of pages to be copied. We are not modeling copy-on-write.

One way to incorporate demand paging in the implementation of `SysCall_Fork` is to run a loop over the parent's page table. Each iteration `i` of the loop examines the entry `i` of parent's page table and if it is valid, it allocates a physical page frame for the child. Suppose the virtual page `i` of parent is mapped to physical page frame `p`. Suppose the page frame allocated to the child is `c`. You will record the mapping of virtual page `i` to physical page frame `c` in the child's page table and copy the contents of physical page frame `p` into physical page frame `c`.

To simplify matters, we will assume that shared pages are not allocated on demand. Instead, the `SysCall_ShmAllocate` call allocates all the necessary shared physical pages. These should be counted as page faults, but no latency should be charged i.e., the caller will just continue executing. This is because there is usually nothing to be copied from the disk at the time of allocating a shared physical page.

Handling page faults inside system calls involves one additional step. These page faults arise when a system call handler invokes ReadMem or WriteMem on a virtual address which is not yet present in the physical memory. In the case of a page fault, the ReadMem and WriteMem methods return FALSE. Depending on the return value of these methods, you may have to keep retrying the ReadMem or WriteMem until you get a return value of TRUE.

In this part of the assignment, we will make sure that the running threads never run out of physical pages. In other words, set NumPhysPages to a high enough value. Notice that our demand-paged virtual memory model is a highly simplified version of the reality. We do not model a disk or a swap space.

4 Implementing Page Replacement Algorithms

As NumPhysPages is gradually reduced, eventually a situation will arise when all the physical pages are currently occupied and a thread undergoes a page fault. This can happen if NumPhysPages is less than the requirement of all the threads. In this case, you have to select one physical page for replacement, whose contents will now be overwritten by the new page. You will implement four replacement policies, namely, Random, FIFO, LRU, and LRU-CLOCK, and evaluate the number of page faults returned by each. Also, at the time of replacing a page, you need to do the following things. To simplify matters, we will ensure that shared pages are never considered for replacement.

Find out the page table entry corresponding to the replaced page and change its state to invalid. If the page table entry is in dirty state (happens if the page is modified by the running program), you have to back up the contents of the replaced page so that when the page is needed later, it can be loaded back correctly. For this purpose, you will allocate a character array (just like the main memory array) per thread when a thread is created. The length of the array should be equal to the number of virtual pages in the executable multiplied by the page size. When a page is replaced, its contents are stored in the corresponding locations of this array. Now, at the time of a page fault, you need to know if the contents of the page should be loaded from the executable file or the backup array. Introduce a new field in the page table entry for this purpose. Initialize this at the time of thread creation to mention that the page should be loaded from the executable file. When a page is replaced, change the field to mention that it should be loaded from the backup array on the next fault for this page. Finding the page table entry corresponding to a replaced page may be tedious. Think of data structures that can accelerate this.

We have already mentioned that one way to incorporate demand paging in the implementation of `SysCall_Fork` is to run a loop over the parent's page table. Each iteration `i` of the loop examines the entry `i` of parent's page table and if it is valid, it allocates a physical page frame for the child. Suppose the virtual page `i` of parent is mapped to physical page frame `p`. While allocating the child's page frame, it may be necessary to replace a page if all page frames are occupied. You should invoke your page replacement algorithm and can replace any page selected by the algorithm except the page `p`. The reason why you cannot replace the page `p` of the parent should be obvious (you need to copy the contents of page `p` into the newly allocated child's page frame, so you want this page to be resident in memory). Suppose the page frame allocated to the child is `c`. Once the contents of page frame `p` are copied into page frame `c`, you should correctly update the replacement states of these two pages. In other words, if the replacement policy is LRU, the page frame `c` should be the MRU frame and page frame `p` should be the next

MRU frame (copying involves reading from p first and then writing to c making c the MRU frame and p the next MRU frame). If the replacement policy is LRU-CLOCK, the reference bits of both p and c should be set to 1 (or TRUE).

For the test programs that do not use `syscall_wrapper_Fork()` or shared memory, your implementation should run correctly with just two pages (one page to keep the current instruction and another page for data), although there may be a large number of page faults. However, your patience will be tested here because these simulations may take a long time. For example, the vectorsum program on my implementation takes about three minutes with two frames and random replacement. The programs that use `syscall_wrapper_Fork()` and create multiple threads will require at least two pages per thread. The programs that use shared memory will require additional number of pages to accommodate the shared pages, which can never be replaced as per our design.

Introduce a new command line flag `-R` followed by an integer from $\{1, 2, 3, 4\}$ for specifying the page replacement algorithm according to the following list.

1. Random
2. FIFO
3. LRU
4. LRU-CLOCK

If this flag is not specified, the simulator should fall back to the demand-paged virtual memory system with large enough physical memory so that page replacement need not be invoked. This flag should be used as follows, where r belongs to the set $\{1, 2, 3, 4\}$.

```
> ./nachos -rs s -A n -P p -R r -x ../test/executablename
```

5 What to Submit

Report the number of page faults and the total number of ticks needed to complete the execution for each of the following two programs provided in the `nachos/code/test/` directory: `vmtest1`, `vmtest2` using each of the four page replacement policies. Report the statistics for the following six values of NumPhysPages: 16, 32, 64, 128, 256, 512. Remember that these results should be collected with randomization seed zero (i.e., `-rs 0`) and round-robin scheduling (i.e., `-A 3`) with both `SCHED_QUANTUM` and TimerTicks set to 100. Explain these results. Put these tables and explanations along with any other comments related to the other parts of the assignment in a pdf document. Name the pdf file groupX.pdf (replace X by your assigned group id).

Note: One simulation of vmtest1 with NumPhysPages set to sixteen and random replacement policy will take around fifteen minutes. With more frames and a better policy, simulation time will go down. Please be prepared for this and start your test runs early.

Execute make clean in the threads and userprog directories. Prepare the submission zip ball of your machine/, threads/, and userprog/ directories. Your submission at the end of simulation must print the number of page faults. You are welcome to provide more statistics.

```
> cd nachos/code/
> zip -r groupX.zip machine/ threads/ userprog/
```

Replace X by your assigned group id. Send an email to cs330autumn2017@gmail.com with subject "[CS330] Assignment3" and attach groupX.zip and groupX.pdf to the mail. The body of the mail should contain the following two sentences.

1. I have not copied any part of this submission from any other group.

2. I have not helped any other group copy any part of this submission.

All the members of the group should put their names and roll numbers below the statement.

6 Punishment for Cheating

Please refer to the section on academic integrity on course web page.