Build a synchronized, multi-threaded system

In this assignment, we give you part of a working thread system; your job is to complete it, and then to use it to solve several synchronization problems.

The first step is to read and understand the partial thread system we have written for you. This thread system implements thread fork, thread completion, along with semaphores for synchronization. Run the program `nachos' for a simple test of our code. Trace the execution path (by hand) for the simple test case we provide.

When you trace the execution path, it is helpful to keep track of the state of each thread and which procedures are on each thread's execution stack. You will notice that when one thread calls SWITCH, another thread starts running, and the first thing the new thread does is to return from SWITCH. We realize this comment will seem cryptic to you at this point, but you will understand threads once you understand why the SWITCH that gets called is different from the SWITCH that returns. (Note: because gdb does not understand threads, you will get bizarre results if you try to trace in gdb across a call to SWITCH.)

The files for this assignment are:

- main.cc, threadtest.cc - a simple test of our thread routines.
- thread.h, thread.cc - thread data structures and thread operations such as thread fork, thread sleep and thread finish.
- scheduler.h, scheduler.cc - manages the list of threads that are ready to run.
- synch.h, synch.cc - synchronization routines: semaphores, locks, and condition variables.
- list.h, list.cc - generic list management (LISP in C++).
- synchlist.h, synchlist.cc - synchronized access to lists using locks and condition variables (useful as an example of the use of synchronization primitives).
- system.h, system.cc - Nachos startup/shutdown routines.
- utility.h, utility.cc - some useful definitions and debugging routines.
- switch.h, switch.s - assembly language magic for starting up threads and context switching between them.
- interrupt.h, interrupt.cc - manage enabling and disabling interrupts as part of the machine emulation.
- timer.h, timer.cc - emulate a clock that periodically causes an interrupt to occur.
- stats.h - collect interesting statistics.

You will not need to modify them all. They are located in the code/threads/ directory.

Properly synchronized code should work no matter what order the scheduler chooses to run the threads on the ready list. In other words, we should be able to put a call to Thread::Yield (causing the scheduler to choose another thread to run) anywhere in your code where interrupts are enabled without changing the correctness of your code. You will be asked to write properly synchronized code as part of the later assignments, so understanding how to do this is crucial to being able to do the project.

To aid you in this, code linked in with Nachos can cause Thread::Yield to be called on your behalf in a repeatable but unpredictable way. Nachos code is repeatable in that if you call it repeatedly with the same arguments, it will do exactly the same thing each time. However, if you invoke nachos -rs *integer*, with a different number each time, calls to Thread::Yieldwill be inserted at different places in the code. This simulates a timesharing environment.

Invoking Nachos

Command line	**Result**
nachos	Nachos runs deterministically. Threads are only preempted if they call Thread::Yield or when they finish.
nachos -rs	Nachos dumps core. -rs requires an integer argument.
nachos -rs *integer*	Nachos calls Thread::Yield at random places in the code. Invoking nachos again with the same integer causes the same sequence of preemptions.

Make sure to run various test cases against your solutions to these problems. At the very least make sure that your solutions run for various parameters to -rs.

You *may not* make any modifications to functions in the code/machine directory. It is not necessary to make any changes to the context switching code in switch.h.

There cannot be any busy-waiting in any of your solutions to this assignment.

Warning: in our implementation of threads, each thread is assigned a small, fixed-size execution stack. This may cause bizarre problems (such as segmentation faults at strange lines of code) if you declare large data structures to be automatic variables (e.g., int buf[1000];). You will probably not notice this during the semester, but if you do, you may change the size of the stack by modifying the StackSize define in switch.h.

The assignment is below. Note that you will use locks in later assignments, so making sure that they work correctly now will save you time in later assignments.

1. **(10 points)** Implement locks and condition variables. The comments in the code describe the functions you need to implement. You may either use semaphores as a building block, or you may use more primitive thread routines (such as Thread::Sleep). We have provided the public interface to locks and condition variables in synch.h. You need to define the private data and implement the interface. Note that it should not take you very much code to implement locks and ondition variables.

Your implementation should address issues like a thread trying to release a lock that it does not hold. You must document the error conditions you identify and how your implementation handles them. For example: If a thread tries to release a lock it does not hold, that request is to be ignored.

A comment on locks in synch.h says: In addition, by convention, only the thread that acquired the lock may release it. You may *not* assume this; your implementation must enforce it.

Condition variables are generally used with monitors, but because C++ does not support monitors, we implement the synchronization they provide explicitly using locks.Lock::Acquire() is the equivalent of entering a monitor, and Lock::Release() is leaving it.

You may assume that a Condition variable is always used with the same lock. If you do not assume this, you must document what assumptions you use.

synch.h makes a distinction between Hoare monitors and Mesa monitors. This assignment calls for you to implement Mesa semantics. The distinction is in the semantics ofCondition::Signal(). Under Hoare semantics, when Condition::Signal() is called on a condition variable with a thread waiting, the thread waiting on that condition variable runs immediately. Under Mesa semantics, when Condition::Signal() is called one of the threads waiting on that condition variable is put on the ready queue (that is it is made ready or unblocked), but the next thread to run is determined by the scheduler. The current thread may or may not yield the processor at your discretion. Note also that the thread that is made ready must reacquire the lock before its call to Condition::Wait() returns.

Although this assignment does not require you to implement Hoare semantics, its worth thinking about what issues would arise in implementing those semantics. It's the sort of thing that would make a good exam question.

Test your code by running this test code. Read the code for the expected behavior. Feel free to create your own test cases if you want to test your code more completely.

We will look at your code. Uncommented code will lose points.

2. **Simulating a Doctor's Office (25 points)**
   You are to simulate the operation of a doctor's office. Each of the Doctor Office components below is to be simulated through the use of individual threads. Your Doctor's Office is to have the following types of people and resources:
   - Adult Patient - Someone who has come to see a doctor. Adult Patients come to the doctor's office by themselves.

- Child Patient - Someone who has come to see a doctor. Child Patients cannot go to any room without their parent. They must go together. The Child Patient follows their Parent to the various rooms in the doctor's office.
- Parent - Bring their child to see the doctor. Escorts their child everywhere.
- Waiting Room Nurse - Patients register with the Waiting Room Nurse.
- Nurse - Nurses handle Patients.
- Xray Technician - Takes xrays of Patients
- Cashier - Receives money from Patients/Parents
- Doctors - See Patients in examination rooms

Patient: Patients show up to the doctor's office and register with the Waiting Room Nurse. After registering, they go wait in the waiting room. They wait until a Nurse comes to the waiting room to escort them to an Examinination Room. Once in the examininination room, the Nurse will take their temperature, take their blood pressure, then ask what symptom they have. Patients can have the following symptoms: pain, nausea, hear alien voices. The symptom has nothing to do with the Doctor's examination. However, you must share the data so that the symptom is "written" on the examination sheet. The patient then waits for the Doctor to come to see them. The Patient/Doctor conversation is described under the Doctor section.

If a Patient has to have an xray taken, they must wait for a Nurse to escort them to the Xray Room. After the xray has been taken, a nurse takes them back to their examininination room. They wait for the doctor to come back to read the xray.

After the Doctor says the examination is over, the patient must wait for a Nurse to escort them to the Cashier to pay. Prices are listed in the Cashier section. After paying the Cashier, the Patient leaves the doctor's office.

Child Patient: A Child Patient must have an accompanying Parent. For Child Patients, the Nurse talks to the Parent, who talks to their child. Nurses cannot escort a Parent anywhere until the Parent has told their child to follow them. Only after a Parent has told their child to follow them, will the Parent follow a Nurse.

Waiting Room Nurse: The Waiting Room Nurse stays at her desk in the Waiting Room. Their job is to give a form to Patients to fill out. The form requires the patient name and age. The Waiting Room Nurse does not wait for a Patient to fill out the form. The Patient must bring the form back to the Waiting Room Nurse after it is filled out. Once the Waiting Room Nurse has accepted the completed form, they tell the Patient to go back and wait to be called by a Nurse.

The Waiting Room Nurse create a new Examination Sheet for a registered Patient. The Waiting Room Nurse holds onto all examination sheets for Patients that are in the waiting room. When a Nurse is to escort a Patient to an examination room, they receive the appropriate examination sheet from the Waiting Room Nurse.

Nurses will tell the Waiting Room Nurse when there is an open examinination room. The Waiting Room Nurse will tell the Nurse if there is a Patient waiting, or not. It is the Nurses job to tell the next waiting Patient to follow them to the examinination room. It is not the Waiting Room Nurses job to do this.

Cashier: The Cashier is to receive payment from Patients. When a Patient comes up to the Cashier, they hand over their examination sheet. The Cashier will read this and determine the Patient's bill. They then tell the Patient how much they owe. The Patient then "pays" that amount. The Patient then waits for the Cashier to give them a receipt.

Cashiers cannot start with another Patient until they know the current Patient has left their counter.

Xray Technicians: Xray Technicians have nothing to do until a Nurse brings a Patient in for an xray. When first arriving, the Nurse gives the Xray Technician the examination sheet for the Patient, or the Patient must wait in line, if the Xray Technician is busy. The nurse then leaves once they've "dropped off" the Patient.. When a Patient has their turn, the Xray Technician tells the Patient to get onto the table. Once the Patient is on the table, the Xray Technician "takes" an Xray. An xray session can require from 1 to 3 xray images to be taken. After each image is taken, the Xray Technician tells the Patient to "move" so the next image can be taken. Once the Patient has moved, the Xray Technician takes the next image. An xray image can either show 'nothing', a 'break', or a 'compound fracture'.

Once all the xray images are taken, the Xray Technician tells the Patient to wait in the Xray waiting room and puts their examination sheet in a wall pocket for this room. Once the Nurse arrives, the Nurse tells the Patient to follow them. The Xray Technician can work with another Patient once they've put the examination sheet for the just-finished patient in the wall pocket, and that Patient has left the Xray room.

Doctors: Doctors stay in their office until they are informed by a nurse that there is a Patient ready to be examined. The first time a Doctor sees a Patient, they read the examination sheet. The Doctor will randomly determine if the Patient needs an Xray or a shot. There is a 25% chance that a Patient will need an Xray. If a Patient does need xrays, the Doctor must also determine how many images are to be taken. It is to be from 1 to 3 images. There is a separate 25% chance that a Patient will need a shot. If a Patient does not need an xray, or a shot, then the Doctor determines they are fine and leaves the examination room. If a shot/xray is needed, the Doctor tells the Patient what they must have. The Doctor then leaves the examination room. Upon exiting the examination room, the Doctor gives the examination sheet to a Nurse, if one is currently not busy, or if no Nurse is available, the Doctor puts the examination sheet in the wall pocket right by the examination room door. The Doctor then goes back to their office.

The second time a Doctor sees a Patient - after xrays - the Doctor reads the xray images and tells the Patient the results. They also note down on the examination form that the Patient has been informed as to the xray results. The Doctor then leaves the examination room, if a Nurse is available, the Doctor gives the Nurse the examination sheet. If no Nurse is available, they put the form in the wall pocket for that examination room.

Nurses: Nurses escort patients to and from rooms. This includes the Waiting Room, the Xray room, the Cashier, and the Examinination Room. Nurses also take a Patients termperature, blood pressure, and asks for their symptom. Nurses also give shots to Patients.

When a Nurse has nothing to do, they examine the wall pockets outside of each Examination Room. If the wall pocket is not empty, that means a Doctor is done with the Patient in the room. The Nurse is to read the examination sheet to see if the Patient needs an xray, a shot, or is finished with their examination. Whichever task is needed, the Nurse will escort the Patient to the proper location, handing over the examination sheet to the proper person (unless it is just a shot, in that case they give the shot) and then escort the Patient to the Cashier.

For a Nurse to give a shot, they must first go to the supply cabinet to get the needed medicine. Only one Nurse can be using the supply cabinet at a time. To actually give a shot, the Nurse must ensure the Patient is ready for the shot. The Patient must wait for the Nurse to give a shot, once the Nurse enters the examination room with the medicine from the supply cabinet. The Nurse must tell the Patient when the shot is over.

Once a Patient examination is complete - shot given, xrays taken, etc., a Nurse will escort them to the Cashier.

Examination Sheets: Examination sheets are **not** to be stored in a single large array. Each examination room has a wall pocket for storing an examination sheet for the Patient in that room. The Waiting Room Nurse holds onto all examination sheets for Patients in the waiting room. The Xray Technicican holds the examination sheet for the Patient that is having xray images taken. Nurses that are escorting Patients, carry their examination sheet with them.

Number of Entities:

- There can be from 2 or 3 Doctors
- There can be from 2 to 5 Nurses
- There is 1 Waiting Room Nurse
- There is 1 Cashier
- There can be from 1 or 2 XRay Technicians

- There must be at least 30 Patients, but your simulation should run with any number of Patients. This includes Adult, as well as Child, Patients. Every Child Patient must have an associated Parent.

It is required that your project works with variable number of entities and hence, hardcoding will reduce the credit.

Testing:

You are to develop repeatable tests for part 2. Repeatable means that when the test is run, the same behavior is produced - like in the test suite we give you for part 1. You must prove the following:

- Child Patients are never abandoned by their Parent, nor go anywhere without their Parent.
- Waiting Room Nurses only talk to one Patient/Parent at a time.
- Cashiers only talk to one Patient/Parent at a time.
- Patients/Parents never go anywhere without being escorted by a Nurse.
- All Patients leave the Doctor's Office. No one stays in the Waiting Room, Examination Room, or Xray Room, forever.
- Two Doctors never examine the same Patient at the same time.

Specific output guidelines are posted. Please follow the link below.

Output Guidelines

Your submission should meet these guidelines. Any submission that does not meet these guidelines may be graded down.

Assignment 2: System Calls and Multiprogramming

The second phase of Nachos is to support multiprogramming. As in the first assignment, we give you some of the code you need; your job is to complete the system and enhance it.

The first step is to read and understand the part of the system we have written for you. Nachos can run a single user-level 'C' program at a time. As a test case, we've provided you with a trivial user program, `halt'; all halt does is to turn around and ask the operating system to shut the machine down. Run the program `nachos -x ../test/halt' from the userprog directory. As before, trace what happens as the user program gets loaded, runs, and invokes a system call.

The files for this assignment are:

- progtest.cc - test routines for running user programs.

- addrspace.h, addrspace.cc - create an address space in which to run a user program, and load the program from disk.
- syscall.h - the system call interface: kernel procedures that user programs can invoke.
- exception.cc - the handler for system calls and other user-level exceptions, such as page faults. In the code we supply, the `halt' system call is supported, as well as Read, Write, Open and Close - file handling system calls.

  <="" li="">

- filesys.h, openfile.h (found in the filesys directory) - a stub defining the Nachos file system routines. For this assignment, we have implemented the Nachos file system by directly making the corresponding calls to the UNIX file system; this is so that you need to debug only one thing at a time. In assignment four, we'll implement the Nachos file system for real on a simulated disk.
- translate.h, translate.cc - translation table routines. In the code we supply, **we assume that every virtual address is the same as its physical address** - this restricts us to running one user program at a time. You will **generalize this to allow multiple user programs** to be run concurrently. We will not ask you to implement virtual memory support until assignment 3; for now, every page must be in physical memory.
- machine.h, machine.cc - emulates the part of the machine that executes user programs: main memory, processor registers, etc.

  So far, all the code you have written for Nachos has been part of the operating system kernel. In a real operating system, the kernel not only uses its procedures internally, but allows user-level programs to access some of its routines them via *system calls*.

  In this assignment we are giving you a simulated CPU that models a real CPU. In fact, the simulated CPU is the same as a real CPU (a MIPS chip), but we cannot just run user programs as regular UNIX processes, because we want complete control over how many instructions are executed at a time, how the address spaces work, and how interrupts and exceptions (including system calls) are handled.

  Our simulator can run normal programs compiled from C - see the Makefile in the 'test' subdirectory for an example. The compiled programs must be linked with some special flags, then converted into Nachos format, using the program "coff2noff" (which we supply). The only caveat is that floating point operations are not supported - use int's and char's.

  You are to compile Nachos in the userprog directory. Nachos user programs are to be compiled from the test directory. Once this is done, you can run a simple test of the file handling system calls using this command:

  ```
  nachos -x ../test/testfiles (run Nachos from the userprog directory)
  ```

You should see the following output (or something very close).

```
testing a write
Machine halting!

Ticks: total 1854, idle 1600, system 170, user 84
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 16
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

Assignment

1. (10 points) Implement system calls and exception handling. This consists of two parts:
   - You must support the following system calls already defined in syscall.h: Fork, Exec, Exit, and Yield. You will complete the implementation of Exec and Fork in part two, because you need multiple address spaces to be able to test their proper functionality. We have provided you an assembly-language routine, "syscall", to provide a way of invoking a system call from a C routine (UNIX has something similar - try 'man syscall'). **You are NOT to implement Join.**
   - You are to implement system calls for the Lock class Acquire() and Release() methods and the Condition class Wait(), Signal(), and Broadcast() functions. These are five NEW system calls that do not exist in Nachos currently.

     You must also implement constructor and destructor system calls to create a Lock and a Condition object. These are to be called: CreateLock, DestroyLock, CreateCondition, DestroyCondition. The CreateLock and CreateCondition system calls can either take no argument, or you can provide a character array parameter to give them a name. It is your choice. When calling CreateLock or CreateCondition, you are to return an integer value. This value is an index position into a kernel structure array of actual Locck and Condition objects. You **are not** to return a kernel space pointer, as that would allow the user program to manipulate it directly. The user program sees it as an identifier that is somehow associated by you (in the kernel of Nachos) with the kernel-level Lock, or Condition, object that you created. This way, on an Acquire system call, you pass an int argument -; the value returned by the CreateLock system call - so the OS can use that argument (cast appropriately to a Lock pointer in kernel mode) to get to the Lock object created in CreateLock.

For the DestroyLock and DestroyCondition system calls, they take a single integer parameter - the identifier for the Lock or Condition kernel object that is to be deleted.

2. We have provided code for the five system calls that are concerned with files: Create, Open, Close, Read and Write. NOTE: To output messages from within a Nachos user program, you must use the Write system call. printf will not work, as it is a Unix system call and is not implemented in Nachos user programs (you can use it in kernel mode).
3. The Exit system call must ensure that Thread::Finish is called, except for the very last thread running in Nachos. For the very last thread (not the last thread in a process - unless it's the last process in Nachos), you must call interrupt->Halt() to actually stop Nachos. If you don't do this, Nachos will not terminate. This assignment requires that Nachos terminates.
4. Yield is very simple to implement.
5. Exec and Fork require actual implementation (which you will do in part 2) - not just organizing calls to existing capabilities in the Nachos kernel. You will have to deal with creating address spaces, initializing them with the code of the program to be run when necessary, creating a new thread, and binding the thread to its address space and creating a new stack.
6. You will have to maintain a process table, in order to maintain the mapping from the SpaceIds that you pass up to the user level, and the actual address space data structures. When Exiting a thread, if it is the last thread in the process, you will need to ensure that you delete the address space (and all other resources). You may think that you don't need a process table for this project, but trust me, you will defintely need it for project 3 - besides, every OS has a process table.
7. Note that you will need to 'bullet-proof'' the Nachos kernel from user program errors - there should be nothing a user program can do to crash the operating system (with the exception of explicitly asking the system to halt). Bullet-proofing includes, but is not limited to making sure that any pointers passed to the nachos kernel point to valid memory in the user's address space, and that any parameters are reasonable. You may set limits on parameters (i.e. maximum buffer sizes or other values) but they must be documented in your writeup.
8. Your answer to this part must include a test suite of user programs that demonstrate the correctness of your system calls (other than Exec and Fork which are covered in part 2). You must demonstrate not only that your system calls work when used correctly, but that they do not crash Nachos when used incorrectly. You should exercise your creativity in finding many different possible types of abuse and showing that your system is safe from them.

9. Document your test suite. Tell what error each program or subroutine is checking for and make the output as clear as possible. Comment your test code. The test suite should be user programs like the halt example. Note that your test case should also demonstrate the line buffering behavior of your synchronized console.
10. (10 points) Implement multiprogramming. The code we have given you is restricted to running one user program at a time. You will need to:

    Come up with a way of allocating physical memory frames so that multiple programs can be loaded into memory at once (cf. bitmap.h). Some of the work has already been done - see the AddrSpace constructor.

    Provide a way of copying data to/from the kernel from/to the user's virtual address space (now that the addresses the user program sees are not the same as the ones the kernel sees). For big hints on how this is done, see the file handling system calls that we've provided.

    Complete the implementation of the Exec and Fork system calls.

    Note that scheduler.cc now saves and restores user machine state on context switches - when you compile from the userprog directory.

    Now that you have independent address spaces, and a way of switching between processes in a controlled manner, you must complete the implementation of Exec. The routine 'StartProcess' in progtest.cc may be of use to you in implementing the 'Exec' system call.

    Nachos Exec creates a new address space and starts a single thread running that code. You can think of it as a combination of Unix fork and exec.

    Design and demonstrate a test suite that shows that Exec and Fork perform correctly, when used in any combination and cannot be used to break the OS. Your tests must show not only that proper input to Fork and Exec work properly, but that improper input is captured and dealt with appropriately. In no case should Nachos terminate abnormally.

11. (20 points) Implement part 2 of project 1 as a set of multithreaded Nachos user programs. You must use the new Lock and Condition system calls that you implemented in part 1 for synchronization.

    You will have to use the Fork system call to create all of your threads.

If your Exec system call is working properly, then it should be possible to have multiple Doctor Office simulations, each running independently, executed through different processes simultaneously.

You are to provide a test suite that proves your implementation is correct. You must fully explain what Locks and Condition Variables you use in your solution and how it guarantees proper synchronization between the threads in your tests. These tests should be equivalent to the set of tests you had for your simulation in project 1 part 2.

Adding New System Calls

You will need to make the following changes to add the new system calls - so the trap interrupt will work.

Change start.s in the test subdirectory. You will need to copy and paste the lines from one of the other system calls for your new system calls. I've included the lines below from Halt. Each system call will need the same set of lines, changing 'halt' to your new system calls: Acquire, Release, Wait, Signal, and Broadcast, etc.

```
  .globl Halt
  .ent  Halt
Halt:
  addiu $2,$0,SC_Halt
  syscall
  j    $31
  .end Halt
```

Make two changes to syscall.h.

1. Add the new syscall codes. The existing codes in syscall.h are below. You must start with a number greater than these numbers.

```
/* system call codes -- used by the stubs to tell the kernel which system call
 * is being asked for
 */
#define SC_Halt   0
#define SC_Exit   1
#define SC_Exec   2
#define SC_Join   3
#define SC_Create 4
#define SC_Open   5
#define SC_Read   6
```

```
#define SC_Write  7
#define SC_Close  8
#define SC_Fork   9
#define SC_Yield  10
```

2. Add the new function prototypes for the new system calls. The function prototype for Halt() in syscall.h is shown below.

```
/* Stop Nachos, and print out performance stats */
void Halt();
```

Assignment 3: Virtual Memory

The third phase of Nachos is to investigate the use of caching. In this assignment we use caching for two purposes. First, we use a software-managed translation lookaside buffer (TLB) as a cache for page tables to provide the illusion of fast access to virtual page translation over a large address address space. Second, we use memory as a cache for disk, to provide the abstraction of an (almost) unlimited virtual memory size, with performance close to that provided by physical memory. We provide no new code for this assignment (the only change is that you need to compile with the '-DVM -DUSE_TLB' flags - which is already set if you compile from the vm directory); your job is to write the code to manage the TLB and to implement virtual memory. You can edit files in the userprog directory, but you must compile and run Nachos from the vm directory.

Page tables were used in assignment 2 to simplify memory allocation and to isolate failures from one address space from affecting other programs. For this assignment, the hardware knows nothing about page tables. Instead it only deals with a software-loaded cache of page table entries, called the TLB. On almost all modern processor architectures, a TLB is used to speed address translation. Given a memory address (an instruction to fetch, or data to load or store), the processor first looks in the TLB to determine if the mapping of virtual page to physical page is already known. If so (a TLB 'hit'), the translation can be done quickly. But if the mapping is not in the TLB (a TLB 'miss'), page tables and/or segment tables are used to determine the correct translation. On several architectures, including Nachos, the DEC MIPS and the HP Snakes, a 'TLB miss' simply causes a trap to the OS kernel, which does the translation, loads the mapping into the TLB and re-starts the program. This allows the OS kernel to choose whatever combination of page table, segment table, inverted page table, etc., it needs to do the translation (which is what you get to do). On systems without software-managed TLB's, the hardware does the same thing as the software, but in this case, the hardware must specify the exact format for page and segment tables. Thus, software managed TLB's are more flexible, at a cost of being somewhat slower for handling TLB misses. If TLB misses are very infrequent, the performance impact of software managed TLB's can be minimal.

The illusion of unlimited memory is provided by the operating system by using main memory as a cache for the disk. For this assignment, page translation allows us the flexibility to get pages from disk as they are needed. Each entry in the TLB has a valid bit: if the valid bit is set, the virtual page is in memory. If the valid bit is clear or if the virtual page is not found in the TLB, a software page table is needed to tell whether the page is in memory (with the TLB to be loaded with the translation), or the page must be brought in from disk. In addition, the hardware sets the use bit in the TLB entry whenever a page is referenced and the dirty bit whenever the page is modified.

When a program references a page that is not in the TLB, the hardware generates a *TLB* exception, trapping to the kernel (eventually reaching the exceptionHandler() function). The operating system kernel then checks its own page table. If the page is not in memory, it reads the page in from disk, sets the page table entry to point to the new page, sets the TLB correctly, and then resumes the execution of the user program. Of course, the kernel must first find space in memory for the incoming page, potentially writing some other page back to disk, if it has been modified.

As with any caching system, performance depends on the policy used to decide which things are kept in memory and which are only stored on disk. On a page fault, the kernel must decide which page to replace; ideally, it will throw out a page that will not be referenced for a long time, keeping pages in memory those that are soon to be referenced

If you increased the number of physical pages for Project 2, you must return it to its original value of 32 for this assignment.

1. (10 points) Implement software-management of the TLB. For this, you will need to implement some kind of software page translation, for handling TLB misses. Note that with the compile time flag -DUSE_TLB, the hardware no longer deals with page tables; thus, you need to do something about making sure the TLB state is set up properly on a context switch. Most systems simply invalidate all the TLB entries on a context switch; the entries get re-loaded as the pages are referenced. Your page translation scheme should keep track of the dirty and use flags for each page set by hardware in the TLB entry.

   For this part, you must run test cases that show that a single program that fits in main memory runs to completion. The most convenient test case to run is 'matmult'. matmult, when correctly executed passes the value 7220 to the Exit system call. If you print out this value, you can easily find out if your virtual memory is working. You also need to show that two programs that fit in main memory together run to completion in different address spaces (to show that the TLBs are managed properly.) You should convince yourself, and us, that threads running in different address spaces cannot corrupt each other's memory. This must also work with multiple threads in a single address space. To show multiple programs working, you must have a functioning Exec system call.

2. (10 points) Implement virtual memory. For this, you will need routines to move a page from disk to memory and from memory to disk. Use the Nachos file system to contain your swap file. You are required to have a single swap file for all Nachos processes and threads. You are to organize the swap file by pages and nothing is to be preloaded from the executable when a process starts up. Thus, you will have to keep track of each virtual page and whether it is in memory, in the swap file, or in the original executable. You do not have to worry about the executable being changed during a Nachos execution.

   When memory is filled, you must select a page to remove from memory to make room for it. In order to find unreferenced pages to throw out on page faults, you will need to keep track of all of the pages in the system which are currently in use. A simple way to do this is to keep a 'core map', which is an inverted page table - instead of translating virtual page numbers to physical pages, an inverted page table translates physical page numbers to the virtual pages that are stored there. **Implement Random page replacement policy and FIFO page replacement ONLY**. See the submission guidelines for howto select the policy from the Nachos flags.

   You should demonstrate that two or more programs that are each larger than your 32 pages of main memory are able to run to completion.

   We have test cases like matmult and sort, but be advised that we may run other user-level programs on your system to test your implementation. NOTE: You are not required to run any test case that utilizes the Join system call.

3. (15 points) For part 3 you are to implement remote procedure calls for the Lock and Condition Variable system calls that you implemented in project 2. You are to also implement new system calls for handling monitor variables. You need to be able to create them, retrieve their value, and set their value. You may assume that monitor variables are all integers. Any of these system calls are to be sent from a 'client' , instance of Nachos to a "server" instance of Nachos. Your server is to be able to handle multiple clients - just like the operating system can handle multiple address spaces. Any of the system calls mentioned above that is made by a client is to be sent to the server Nachos. You can hardcode the machine ID of your server, if you like, in the Nachos clients. The server performs the system call for the client and returns any result back to the client. **NOTE: This means that all user programs share all Locks and CVs. They are no longer process specific. If you didn't get these system calls to work for project 2, you will have to fix them for this project.**

   We will provide you with some low-level network communications facilities; you will build a nicer abstraction on top of those, and then use that abstraction in building a distributed application. Each separate Nachos (which actually exists as a UNIX process) is a node in the network; a network (emulated via UNIX sockets) provides the communication medium between these nodes. You can test out the basic network functionality by running `nachos -m 0 -o 1' and `nachos -m 1 -o 0' (from the network directory) simultaneously (preferably, on different terminals or in different windows). [Note: this test case requires a working implementation of locks and condition variables from assignment 1, but nothing else.]

There are only a few new files for part 3:

nettest.cc -- network test routines.

post.h, post.cc -- a post office abstraction, built in software on top of the network. This provides synchronized delivery and receipt of messages to/from specific mailboxes; there may be multiple mailboxes per machine.

network.h, network.cc -- emulation of the physical network hardware. The network interface is similar to that of the console, except that the transmission unit is a packet rather than a character. The network provides ordered, unreliable transmission of limited size packets between nodes. All routing issues (how the message gets from node to node) are taken care of by the network. The post office provides a more convenient abstraction than the raw network; you are to continue this layering process -- at each level, the software removes one physical constraint and replaces it with an abstraction. Thus, reliable messages can be built on top of an unreliable service, large messages can be built on top of fixedlength messages, etc.

Assignment 4: Distributed Doctor's Office

This final assignment is to create a distributed system for allowing user programs across multiple Nachos clients to participate in a single Doctor's Office Simulation, without having to know where any other user program is executing.

For this assignment you do not have to worry about running out of memory (you can set memory to any number of pages, but you must document what your solution requires), nor are you required to use the TLB, so project 3 code (parts 1 and 2) are not required.

The files for this assignment are the same as for part 3 of project 3.

1. (15 points) You are to implement all the entities from the project 1 Doctor's Office Simulation. You are to do this in such a way that they all work together across Nachos clients. It is the server's responsibility to connect the various entities together. User programs are not to know anything about the fact that remote procedure calls are being used. In addition, Nachos clients do not directly talk to each other. **All client messages are to go to a Nachos server. The server connect the clients - and the user programs. You are required to use the Exec system call to create each entity. The Fork system call is not to be used.**

You are to use the system calls that you created in project 3 part 3 in your distributed simulation.

For part 1, you only have 1 server, like project 3 part 3.

Each instance of Nachos that is running user programs is to now be able to run up to 10 user programs simultaneously. You must assign mailbox numbers to threads as they are created. Mailbox numbers can no longer be hardcoded.

2. (10 points) Fully Distributed Servers
   Now you are to change your server code so that you can have up to 5 servers working together in a group. Each server can receive a request from any client. In fact, the client Nachos kernels are to randomly generate a server machine ID for each request message they send.

   Your servers are to implement the total ordering algorithm to determine when a message is to be processed. When a server receives a client request message, it is immediately to be forwarded to all other servers. Servers must process server forwarded-messages in timestamp order.

   Your simulation is to work with any number of servers from 1 to 5.

3. **(10 points) Extra Credit**
   This does not require the Doctor's Office. You are to implement an Election Algorithm within your servers. You can implement the Bully, or the Ring, election algorithm. The purpose of the election algorithm is to determine which user program should receive ownership of a lock when the user program that had the lock has failed (terminated with a Control-C from the command line).

   You are to have from 2 to 5 user programs. They can be single-threaded, one per client Nachos instance. Your servers are to detect when a user program, that owns a lock, has died. This is easy to do with Nachos. A postOffice->Send() to a dead Nachos returns a -1 value. Once the determination has been made, the servers work together to "elect" another user program to be given the lock. All servers must agree on the choice, so that only one user program is given the lock. One of the servers sends a reply to that winning user program - as if a Release request was received.