Implementation Guide

Threads and Processes:

`Thread::Fork()` spawns a new kernel thread that uses the same `AddrSpace` as the thread that spawned it. If that address space is duplicated and not shared, it is no longer a kernel thread but a Forked Process. If that `AddrSpace` instead contains code and data loaded in from a separate file, it is no longer a forked process, but an executed process. The details of `Fork()` and Exec() are covered in steps 1) and 7) below.

System Calls:

- In `code/userprog/exception.cc` , put function calls for each system call. Just print debug statements in these functions for now, so you can see when system calls get executed. The function may need to return or take an argument. Because the argument lies in user space, you will need to transfer it over using machine register reads and writes. A sample stub illustrating this is shown below.

```
case SC_Join: {
  int result = myJoin(machine->ReadRegister(4));
  machine->WriteRegister(2, result);
  break;
}
```

- If the system calls: `Fork()`, `Yield()`, `Exec()` `Join()`, `Exit()` and `Kill()` are implemented in that order, you will not have to worry about the call you are currently working on depending upon unimplemented calls.
- Remember that the last thing the `ExceptionHandler` function needs to do when executing a system call is to increment the program counter. Write a helper function to do this - it needs to update `PCreg`, `NextPCreg`, and `PrevPCreg` . They should all increment by 32 bits (one word). An integer in `NACHOS` is 8 bits wide.

Guide to a Multi-Programmed Nachos

1. Implement `Fork()` . Fork will create a new kernel thread and set it's AddrSpace to be a duplicate of the CurrentThread's space. It sets then `Yields()` . The new thread runs a dummy function that will will copy back the machine registers, PC and return registers saved from before the yield was performed. You did save the PC, return and other machine registers didn't you?
   Duplicating the AddrSpace requires the implementation of a Memory Manager detailed in steps

2-4. `Fork()` will not work completely until the completion of step 4. Don't get stuck on step 1), steps 2-4 are much more important.

2. Write a Memory Manager that will be used to facilitate contiguous virtual memory. The amount of memory available to the user space is the same as the amount of physical memory, it's not until the virtual memory project that you will have to implement swapping.

   You will need just two methods at first 1) `getPage()` allocates the first clear page and 2) `clearPage(int i)` takes the index of a page and frees it. You can use a bitmap (in `code/userprog/bitmap.*`) with one bit per page to track allocation or your own integer array, which ever you prefer. A possible implementation for this class is shown bellow.

```
#include “synch.h”
#include “bitmap.h”
class MemoryManager {
  public:
     MemoryManager (int numTotalPages);
     ~MemoryManager ();
     int getPage();
     void clearPage (int pageId);
  private:
     BitMap *virtMem;
     Lock *lock;
}
```

   Modify `AddrSpace` (`code/userprog/addrspace.*`) to use the memory manager. first, modify the page table constructors to use pages allocated by your memory manager for the physical page index. The later modification will come in step 4.

3. Write the `AddrSpace::Translate` function, which converts a virtual address to a physical address. It does so by breaking the virtual address into a page table index and an offset. It then looks up the physical page in the page table entry given by the page table index and obtains the final physical address by combining the physical page address with the page offset. It might help to pass a pointer to the space you would like the physical address to be stored in as a parameter. This will allow the function to return a Boolean TRUE or FALSE depending on whether or not the virtual address was valid. If confused, consult AOSC 9.4.1
   or `Machine::Translate()` in `machine/translate.cc`

4. Write the `AddrSpace::ReadFile` function, which loads the code and data segments into the translated memory, instead of at position 0 like the code in the `AddrSpace` constructor already does. This is needed not only for `Exec()` but for the initial startup of the machine when executing any test program with virtual memory.
   You can use the following prototype for the function.

```
int AddrSpace::ReadFile (int virtAddr,
                         OpenFile* file,
                         int size;
                         int fileAddr) {
```

You will also need to use the functions:
`File::ReadAt(buff,size,addr)` and `bcopy(src,dst,num)`
as well as the memory locations at `machine->mainMemory[physAddr].`
At this point, test programs should work the same as before. That is, the halt program and other system calls will still operate the way they did before you modified `AddrSpace`.
Also `Fork()` *should be working. Test your implementation appropriately.*

5. Write the PCB and the process manager classes. The PCB class will store the necessary information about a process. Don't worry about putting everything in it right now, you can always add more as you go along. To start, you can use the following implementation:

```
#include “synch.h”
class Thread;
class pcb {
  public:
     pcb (Thread *input);
     ~pcb ();
     int getID();
  private:
     int MAX_FILES;
     Thread *processThread;
     int processID;
     pcb * parent_process;
}
```

   The process manager should do the same thing as the memory manager - it has getPID and clearPID methods, which return an unused process id and clear a process id respectively. Again, use a bitmap or similar integer array. You'll also need an array of PCB* to store the PCBs. Modify the AddrSpace constructors to include a PCB as an attribute.
6. Implement Yield() . Given that forked processes are almost the same as kernel threads, this one should be trivial.
7. Implement `Exec()` . Exec is creating a new process (kernel thread) with new code and data segments loaded from the `OpenFile` object constructed from the filename passed in by the user.
   In order to get that file name, you will have to write a function that copies over the string from user space. This function will start copying memory from the physical address pointed to by the virtual address in `machine->ReadRegister(4)` . It should go until it hits a `NULL` byte.
   Fork the new thread to run a dummy function that sets the machine registers straight and runs the machine. The calling thread should yield to give control to the newly spawned thread. *At this point you should be able to call test files from another test using* `Exec(someTestFile)` *from* `someOtherTestFile.` *(in the* `test/` *dir).*
8. Implement `Join()` . Join should force the current running thread to wait for some process to finish. The PCB manager can keep track of who is waiting for who using a condition variable for each PCB.

9. Implement `Exit(int status)`. This function should set the status in the PCB being exited. It should also force any threads waiting on the exiting process to wake up.
10. Test your system calls using your own test programs. If you have time, test using some programs to do complex and even malicious things.

Steps to implement the System Calls

Fork()

1. Save old process registers.

2. Create a new AddrSpace and copy old AddrSpace to new AddrSpace. (Improve the current AddrSpace).

3. Create a new Thread, associate new AddrSpace with new thread.

4. Create a PCB (Process Control Block) and associate the new Address and new Thread with PCB.

5. Complete PCB with information such as pid, ppid, etc.

6. Copy old register values to new register. Set pc reg value to value in r4. save new register values to new AddrSpace.

7. Use Thread::Fork(func, arg) to set new thread behavior: restore registers, restore memory and put machine to run.

8. Restore old process register

9. Write new process pid to r2

10. Update counter of old process and return.

Yield()

1. CurrentThread yield

Exec()

1. Read register r4 to get the executable path.

2. Replace the process memory with the content of the executable.

3. Init registers.

4. Write 1 to register r2 indicating exec() invoked successful;

5. Return -1 if any step failed. e.g., the executable is unrecognizable.

Join()

1. Read process id from register r4.

2. Make sure the requested process id is the child process of the current process.

3. Keep on checking if the requested process is finished. if not, yield the current process.
4. If the requested process finished, write the requested process exit id to register r2 to return it.

Exit()
1. Get exit code from r4;
   if current process has children, set their parent pointers to null;
   if current process has a parent, remove itself from the children list of its parent process and set child exit value to parent.
2. Remove current process from the pcb manager and pid manager.
3. Deallocate the process memory and remove from the page table;
   current thread finish.

Kill()
1. Read killed ID from r4.
2. Make sure if the killed process exists.
3. If killed process is the current process, simply call exit
   if the killed process has children, set their parent pointers to null
   if the killed process has a parent, remove itself from the child list.
4. Remove itself from the pcb list and pid list.
5. Free up memory and remove page table item.
6. Remove killed thread from scheduler.
7. Return 0 in r2 to show succeed.

Additional suggestions:

- *Error handling*: System calls invoked by a user process in user mode should *never* ``crash'' Nachos. This means you should never trust that the arguments supplied for a system call are correct or reasonable. Instead, these arguments should be checked, and if they are incorrect, a failure status should be passed back to the caller. You will want to think of some scheme for accomplishing this.
- *Build and test incrementally*: We suggest not trying to implement everything at once, but rather do a little at a time, testing that each bit you do works before going on to the next modification. This incremental approach seems to work best for operating systems, since it enables you never to be too far from having a version that ``runs''. If you make too many changes all at once, the debugging task to get back to a running system becomes enormous.
- *Adding source files*: Don't be afraid to add new source files to Nachos, especially if it makes your program more modular. For example, it might be reasonable to add a new source file for the physical memory manager.

- *Debugging flags and switches*: Using the ``-s'' flag to Nachos along with the ``-x'' flag causes Nachos to single-step while in user mode. This might be helpful for debugging and understanding. Also, have a look at the file `threads/utility.h` to see all the code letters that can be supplied along with the ``-d'' flag to enable various kinds of debugging printout from Nachos. The ``-d m'' option prints out each MIPS instruction as it is executed, which is very helpful for tracing problems with `Fork()` and `Exec()`.
- *Incrementing Program Counter after a System Call*: Before returning to user mode after a system call, it is necessary to increment the MIPS program counter to point to the next instruction. A system call instruction takes 4 bytes, so you must add 4 to the program counter. If you don't do this, the application making the system call will go into a loop in which it repeatedly makes the same call over and over. This is hard to figure out otherwise, so I'm telling you now so you don't have to.