Part A. [70] Implement the matrix multiplication project (Project 2) of Chapter 4, once using POSIX threads, and once using conventional processes (fork(2)) with shared memory (shmget(2)). Conduct a series of experiments to record the turnaround times and memory usage, and compare the efficiency of processes vs threads. Use the following test cases:

- 64 × 64 matrix size, with 4 and 16 threads/processes
- 128 × 128 matrix size, with 4, 16, and 64 threads/processes
- 256 × 256 matrix size, with 4, 16, and 64 threads/processes
- 512 × 512 matrix size, with 4, 16, 64 and 256 threads/processes
- 1024 × 1024 matrix size, with 4, 16, 64 and 256 threads/processes

Once you complete the experiments, answer the following questions:

1. [50] Design and provide the proper graphs/charts that demonstrate the memory usage and turnaround time, and how they increase with respect to matrix size and with number of threads. It is part of the assignment to figure out and decide how to record the memory usage and turnaround time (i.e. the methodology of the experiment), and, also, what graphs are needed and how to be designed to demonstrate these facts. In your report, talk about your chosen methodology.

2. [10] From the experiments conducted, is there an ideal number of threads with respect to matrix size? Does this ideal mapping follow a function, and, if so, what is this function? What can you tell about the number of processes with respect to the matrix size?

3. [10] What can you tell about the scalability of each of the two approaches (threads vs processes) with respect to matrix size and with respect to number of threads/processes?

Notes: You may implement this on any system of your choice. Do not forget to give the specifications of the test system, and to consider the number of existing processors or cores. Use the graphs of question 1 to answer questions 2 and 3. In your report, include the complete source code of both programs. Source codes should be well commented and properly formatted in fixed-width (a.k.a. monospaced) font; pay particular attention to taking care of lines that exceed 80 characters in length.

Part B. [40] Do ASST0 of the OS/161 (see next pages)

Notes: For ASST0, you may help each other up to but not including Section “Using GDB for ASST0”. From that point on, you should do ASST0 absolutely individually. Reports are student-individual.
CSE 430 OS/161 Programming Assignment 0 (ASST0)
Introduction to OS/161

This assignment is primarily designed to serve as a quick introduction to OS/161 and SYS/161 and the process of programming and handing in project assignments. It is also intended to demonstrate how to connect to SYS/161 with GDB. This will be valuable in debugging future assignments. It shows how to install and use OS/161 and System/161. It includes adding a simple routine to OS/161 and running and debugging the new kernel.

Due date and total points for ASST0 are indicated on the class outline.

**OS/161 and System/161**

OS/161 is a self-contained operating system that is designed to run on a MIPS based processor. You will be making changes to the OS/161 kernel, and running the modified code. However, since we do not have MIPS stations available to run OS/161 on, we will need an emulator. This is the purpose that System/161 will serve. System/161 provides a virtual set of hardware that OS/161 can run on. You will be providing a kernel image to run on System/161. That is the way you will “boot” your OS/161 operating system. No changes will be made to System/161. We will only be using it as it was designed.

**GDB (The GNU Project Debugger)**

GDB is a debugger that we recommend when testing your code. Perhaps in previous projects you have only used print statements to have your program print out what is occurring during runtime, but with OS/161 you will quickly find that a debugger has great benefits. Additionally, in the future you may not always have the luxury of providing print statements, and may have to debug code that you didn’t actually write yourself. Using GDB is fairly straightforward. A small tutorial is included for you to practice using it on ASST0. This tutorial also shows how to connect the debugger to the kernel while it is running on System/161.

**Toolchain Note**

Often the set of the compiler, linker, debugger, etc. are referred to as a toolchain. The toolchain that is being used for OS/161 is for a MIPS based processor. Because most likely you will be running System/161 and OS/161 on an x86 based processor, you will need a compiler and linker that will produce and link the MIPS code that System/161 will emulate the hardware for. Additionally, this means that GDB is built for the MIPS system as well. This is because it now has to recognize the MIPS binary that it will be analyzing. There are two ways to run cse430 OS/161 projects.

The first way to run OS/161 projects is to run on one of the five remotely accessible BY214
machines that have the toolchain already set up. These can be remotely accessed from home via SSH (or other remote access and file transfer program) or from the machines in the open lab in BY214. If you run remotely from home or from the BY214 open lab machines the only ‘tool’ you will need to install is the MIPS emulator, System/161.

The second way to run the OS/161 projects is on your own machine. In this case you will have to install the provided CS161 toolchain, or build your own toolchain. Detailed instructions are provided for installing the provided CS161 toolchain in the “Installing the cs161 Toolchain Information” write-up available via a link on the OS/161 Information web page. There are two limitations. First the Toolchain available on the web site was designed to work with linux 2.4. If you are using the supplied Toolchain, it may not work if you run it on a linux 2.6 version. It depends on the compiler version. Some of the early linux 2.6 versions came with a compiler that used an accompanying version of the libc library that is compatible with our Toolchain. Later versions come with an incompatible libc. If you are expert at dealing with these things – you can change the compiler and install OS/161. Second, if you are building your own toolchain this can be a quite difficult task. But if you are still interested, you will need a toolchain that generates code for a “mips-elf” target. In both cases we do not have the time to help you. The BY214 machines are set up to do the job and can handle the entire class.

System/161
If you are remotely accessing one of the BY214 lab machines (e.g., by214ba.eas.asu.edu via SSH), download tarballs for sys161 and os161, sys161-1.12.tar.gz and os161-1.10.tar.gz, respectively, to that machine. These are available at:

http://www.public.asu.edu/~dmiller/os161

Then upload these tarballs to your home directory, /export/your_asurite_id, on that BY214 machine.

If you are running on your own Linux box, download the tarballs from the abovementioned site into your home directory, e.g., /home/your_user_name.

A skeleton of the resulting directory structure, starting at the home directory, is shown in cse430 OS/161 Handout #1, OS/161 Directory Skeleton for ASST0. Decompress and extract sys161-1.12 with the following command (‘v’ can be omitted).

    tar vzxf sys161-1.12.tar.gz

Then change to the sys161-1.12 directory that was created (see OS/161 HO#1 top right hand side).

    cd sys161-1.12

You will have to run the configure script to prepare for the compiling and installation of System/161. If you used an alternate location for installation in the previous steps, you will want to run the script with the --help argument to get more information. It isn’t necessary to install
System/161 to a location in the shell’s path.

```
./configure mipseb
```

This will configure System/161 to be installed to ~/cs161/root (see OS/161 HO#1). After this command finishes type the following to both build and install System/161.

```
make && make install
```

This will take a relatively short amount of time when compared to the toolchain steps needed if you build your own tool chain. Once it has completed System/161 has been installed.

**Building and Installing OS/161 Source Programs**

We will now build the user programs that come with OS/161. These are utilities that you may use every day on a UNIX type system. Some programs this includes are: `ls`, `pwd`, and `cat`. Get into your home directory again and decompress and extract `os161-1.10.tar.gz` into it with the following commands (you may omit the ‘v’ in the tar command):

```
   cd ~
   tar vzxvf os161-1.10.tar.gz
```

(Note: the shell expands ‘~’ to your home directory).

This will create an os161-1.10 directory in your home directory (see OS/161 HO#1). We want to run the configure script for OS/161 now. If you want it to install the kernel you build to the default location (~/cs161/root) (see OS/161 HO#1), you only need to run the following configuration command at this point from the os161-1.10 directory that was created.

```
   cd os161-1.10
   ./configure
```

However, if you want to use an alternate location, use the following command to get more information on doing this. We recommend simply using the default location though.

```
   ./configure -help
```

After configuring OS/161, type the following to both build and install the OS/161 source.

```
make
```

**Building an OS/161 Kernel**

We are now ready (finally) to build an actual object OS/161 kernel, called `kernel-ASST0` that will run on the System/161 emulator. From the `os161-1.10` directory perform the following steps. They will be explained after doing this.

```
   cd kern/conf
```
./config ASST0
cd ../compile/ASST0
make clean
make depend
make && make install

This will configure a kernel, called kernel-ASST0 and compile, link and install it to the
~/cs161/root directory if you used the default installation locations. Now it is time to run
the kernel. Switch to the ~/cs161/root directory. Perform the following to set the default
configuration for System/161:

cp sys161.conf.sample sys161.conf

Note: the above is only needed the first time you build the OS/161 kernel

Now, use the following code to boot the new kernel on the sys161 MIPS emulator:

./sys161 kernel-ASST0

You should see the following message displayed, and the cursor will be waiting at a command
prompt.

```
sys161: System/161 release 1.12, compiled Feb 7 2005 19:59:39
OS/161 base system version 1.10
Copyright (c) 2000, 2001, 2002, 2003
   President and Fellows of Harvard College. All rights reserved.
Put-your-group-name-here's system version 0 (ASST0 #1)
   Cpu is MIPS r2000/r3000
   348k physical memory available
   Device probe...
lamebus0 (system main bus)
   emu0 at lamebus0
   ltrace0 at lamebus0
   ltimer0 at lamebus0
   htclock on ltimer0 (100 hz)
   beep0 at ltimer0
   rtclock0 at ltimer0
   lrandom0 at lamebus0
   random0 at lrandom0
   lr0 at lamebus0
   lr1 at lamebus0
   lser0 at lamebus0
  ristol0 at lser0
   pseudorand0 (virtual)
   OS/161 kernel [? for menu]:
```

Just to check out how to run a program and that everything’s ok, you can go through the
following sequence:

Enter ? to get the kernel menu
Enter ?t to get the tests menu
Enter sy1 to get the Semaphore test

You should get a printout of 32 thread numbers (0-31) in no particular order each followed by 63
copies of the ASCII character corresponding to the thread number + 64 in the ASCII character
code set.
Modifying the OS/161 Kernel

You are now going to make a modification to the OS/161 kernel. You are essentially going to be adding a new function/system call. This will be nothing more than a simple ‘hello world’ function, but it will give an insight on how additions are made to OS/161. So you will be adding a routine to the kernel, i.e., hello.c. Note: hello.c is not a user level program.

First, download the hello.c file. It is available via a link on the OS/161 Information page on the class web site. This file contains the needed code for implementing the hello.c addition to OS/161. However, you need to tell the build system that it is there, and then have it linked into the kernel. You will want to put hello.c into the kern/main/ directory of OS/161. So, for example if I have my OS/161 source in the root of my home directory, I would want to put hello.c into (see os/161 Handout #1 os/161 Directory Skeleton for ASST0):

```
~/os161-1.10/kern/main/
```

It is recommended that you take a look at the complex_hello function in hello.c. This is the function that we want to invoke with our new command. If you list the abovementioned directory, you will see a file called menu.c. That sounds like a good place to start. You should look over menu.c and see how they have implemented other commands for OS/161. A good one to look at would be the command to print out the heap stack information. This is pretty close to what we want to do isn’t it? Simply print some information to the console? As a hint, you will want to find the array of structs called cmdtable in the file menu.c.

From there, you can see that you will be using a two-step approach to invoke complex_hello().

1. Add a command such as

   ```
   “ch”, cmd_complex_hello
   ```

   to the command table.

2. Define the cmd_complex_hello function, which interfaces to hello.c in menu.c as follows:

   ```
   static int cmd_complex_hello(int nargs, char **args)
   {
     complex_hello();
     return 0;
   }
   ```

These steps enable you to invoke complex_hello() by typing ch when you are in the tests menu in the same way as you invoked the semaphore test by typing sy1 in the unmodified system. Now, after you have added the necessary code to menu.c, you need to tell the build system to build hello.c as well. This is done with the configuration system that you have already used. Change to the ~/os161-1.10/kern/conf directory in OS/161’s source tree. In this directory you need to edit the file conf.kern to have the build system compile and link hello.c into the kernel. To do this, add the following line in conf.kern.

```
file main/hello.c
```
Note: If you are doing your modifications on a local PC for either speed or convenience with a tool such as Microsoft’s Visual Studio and then uploading the modified code to one of the BY214 computers, to avoid problems caused by the difference between line terminating characters under Windows and Linux, after uploading it’s a good idea to use the \texttt{dos2unix} utility after uploading the modified text as follows:

\begin{verbatim}
dos2unix conf.kern
dos2unix hello.c
dos2unix menu.c
\end{verbatim}

After adding a reference for \texttt{hello.c} into \texttt{conf.kern} you need to reconfigure as you did previously. Here is the command again for your reference (invoked from the \texttt{~/os161-1.10/kern/conf} directory):

\begin{verbatim}
./config ASST0
\end{verbatim}

Then build and install your kernel into the \texttt{~/cs161/root} directory as you did before. Here is the whole set of commands for your reference as well:

\begin{verbatim}
cd ~/os161-1.10/kern/conf
./config ASST0
cd ../compile/ASST0
make clean
make depend
make
make install
\end{verbatim}

Now change directory to \texttt{~/cs161/root} and issue the command

\begin{verbatim}
./sys161 kernel-ASST0
\end{verbatim}

to run the new kernel. This is just as you have done before. When the main menu comes up, type the command that you specified for the provided \texttt{complex_hello} function, e.g., \texttt{ch}.

Uh oh, looks like the kernel is having a problem with running the code. The kernel is receiving a panic and shutting down.

\textbf{Using GDB for ASST0}

Luckily we have a debugger at our disposal. We can use GDB to help with the tracking down of bugs in the code. Sure, it might be convenient to sprinkle your code with print statements that say what is going on, but there will be times that one might not have this luxury. Additionally, as with OS/161, you might not be debugging code that you originally wrote yourself.

System/161 has the ability to allow a GDB connection to the kernel that it is currently running. This is done with the ‘-w’ argument. Essentially, it tells System/161 to wait for a debugger connection. To do this, as before, change directory to \texttt{~/cs161/root} and issue the command
The version of GDB we are using has been patched to allow connections through UNIX sockets. You can use any debugger you want, but the cs161-gdb version that is supplied will definitely connect to System/161. The reason that we can’t directly connect to the kernel is because it is running in the System/161 emulator. You will need to open another terminal in the current directory (~/cs161/root). Now type the following command to get the debugger going in a second window:

```
cs161-gdb kernel-ASST0
```

Note: You have to have PATH set properly so the shell can find cs161-gdb for this to work. PATH is already set up for you if you run remotely on the BY214 machines.

This tells gdb to load the symbols found in the binary that you specified. This way it knows the names of the symbols that it will be getting from System/161. Now, type the following command to connect to System/161 and start debugging the kernel:

```
target remote unix:.sockets/gdb
```

You should see something like the following after doing this:

```
Remote debugging using unix:.sockets/gdb
__start () at .././arch/mips/mips/start.S:24
24     .././arch/mips/mips/start.S: No such file or directory.
in .././arch/mips/mips/start.S
Current language:  auto; currently asm
```

This means that the debugger is now talking to System/161 on which kernel-ASST0 is running. Below are some commands that you can use to find the bug in the complex_hello() routine in the kernel. Also, note that many of the commands can be abbreviated to save typing. The abbreviations are also included.

- **next (n):** Step to the next line of code. This will “Step Over” any subroutine calls.
- **step (s):** Step to the next line of code. This will “Step Into” any subroutine calls.
- **list (l):** List lines of code. Type “help list” for more information.
- **break (b):** Set a breakpoint. You can provide line numbers, function names, etc. For example, to break at our new function, type: “b complex_hello”. Or if you wanted to do it by line number you would do: “b hello.c:33”.
- **info (i):** Get information. For example, do “info b” for info on breakpoints.
- **continue (c):** Continue running the code until a signal or breakpoint causes it to stop.
- **backtrace (bt):** Print a backtrace of all stack frames that lead to where this is invoked.
- **quit (q):** Quit the debugger.

So, you will want to switch back and forth between the OS/161 kernel and gdb. For example, you might want to start testing the code by setting a breakpoint at complex_hello and
see if you can figure out where the kernel is crashing. After setting the breakpoint, you would want to **continue**, at which time you will see that OS/161 is responding on its terminal again. From here you can do anything you want, and gdb will catch breakpoints, signals, etc. Basically, you will have to switch between two terminals to control OS/161 and to debug it with GDB.

**Additional debugger usage info**

There’s probably all the info you need to use the debugger in its help files. When you get the debugger running, for example by doing `cs161-gdb kernel-ASST0`, when in `~/cs161/root`, do a `help info`. There’s a list of info commands. Two of the most helpful in addition to the ones mentioned above are `info set` and `info locals`. `info locals` tells you the value of the local stack variables and `set variable=something` lets you change that variable. Also be careful to use `n` and `s` appropriately, i.e., don’t step into something you don’t want to go through all the steps of.

**Restarting Notes:**

You may need to restart the kernel and/or reconnect to System/161 to run another debugging session. To get OS/161 rebooted again in the sys161 window:

```
cd ~/cs161/root and
./sys161 -w kernel-ASST0
```

To get the debugger started and reconnected again in the debugger window:

```
cs161-gdb kernel-ASST0
target remote unix:.sockets/gdb
```

This now leaves you with the first assignment. It is your task to find out why the kernel is crashing. Once you find the problem, you are to correct it, recompile the kernel, and run it. If you are successful you will see a statement similar to the following displayed by OS/161.

```
Hello World!!!
Operation took 0.001896160 seconds
OS/161 kernel [? for menu]:
```
Questions about the OS/161 Kernel

Listed below are questions that you will need to answer and submit as part of this project. They will give you a much better insight to OS/161 while answering them. The sections of the source code where you can find relevant information for the question are listed before each set.

kern/arch/mips/conf
1. The kernel for this assignment is configured to use a particular vm system. What is this vm system called?

kern/arch/mips/include
2. Which register number is used for the stack pointer (sp) in OS/161?
3. What bus/busses does OS/161 support?
4. What is the difference between splhigh and spl0?
5. Why do we use typedefs like u_int32_t instead of simply saying int?
6. What must be the first thing in the process-control block?

kern/arch/mips/mips
7. What does splx return?
8. What is the highest interrupt level?
9. What function is called when user-level code generates a fatal fault?

kern/include
10. How frequently are “hardclock” interrupts generated?
11. What functions comprise the standard interface to a VFS device?
12. How many characters are allowed in an SFS volume name?
13. What is the standard interface to a file system (i.e., what functions must you implement to implement a new file system)?
14. What function puts a thread to sleep?
15. How large are OS/161 pids?
16. What operations can you perform on a vnode?
17. What is the maximum path length in OS/161?
18. What is the system call number for a reboot?
19. Where is STDIN_FILENO defined?
20. What does kmain() do?

kern/main
21. Is it okay to initialize the thread system before the scheduler? Why (not)?

ASST0 Deliverables.
1. (20 points) Answers to the questions about the OS/161 kernel.
2. (10 points) Give the screenshot of the result when you execute the modified kernel.
3. (5 points) Printout of the parts of kern/main/menu.c that you have modified.
4. (5 points) Printout of updated (with the fix) hello.c