Assignment 1 - Write-Up

October 9, 2017

Tyler Jones (jonesty) & Tair Maimon (maimonc)

1 Kernel

1.1 Setting up the kernel

- Part 1:
  - Open two putty sessions, connect to os2 with ssh username@os2.engr.oregonstate.edu
  - In screen 1:
    - Create a folder with group number as the name, in /scratch/fall2017/???
    - Run the script /scratch/bin/"acl_open" to change the permissions of the repo
    - Export the kernel from the git repository: git://git.yoctoproject.org/linux-yocto-3.19 (add actual command)
    - In the linux-yocto-3.19 folder in your repo run the command: git checkout 'v3.19.2' to get the right version
    - Source the environment configuration script with the command: source /scratch/opt/environment-setup-i586-poky-linux (for the bash shell)
    - Run the commend to run qemu: qemu-system-i386 -gdb tcp::5531 -S -nographic -kernel bzImage-qemux86.bin -drive file=core-image-lsb-sdk-qemux86.ext4,if=virtio -enable-kvm -net none -usb -localtime –no-reboot –append "root=/dev/vda rw console=ttyS0 debug".
  - In screen 2:
    - Source the environment configuration script with the command: source /scratch/opt/environment-setup-i586-poky-linux (for the bash shell)
    - Run the command: $GDB.
    - In the line for gdb() run: target remote : 5531
    - In the line for gdb() run: continue
    - Go back to screen 1
    - In screen 1:
    - Enter: root for login
    - you're in the kernel
    - Can run: uname -a to see what's the name of the kernel.
    - Run: reboot, to exit kernel
- Part 2:
  - In screen 1:
    - copy /scratch/fall2017/files/config-3.19.2-yocto-qemu to $SRC_ROOT/.config
    - In the linux-yocto-3.19 folder in your repo, run: make menuconfig. Go to General setup, local version and change the name variable

    * In the linux-yocto-3.19 folder in your repo there is a make file
    * Run: make -j4 all. This creates a kernel image. It will take a few minutes
    * After that, check you have a file bzimage in arch/x86/boot
    * Go back to the linux-yocto-3.19 folder in your repo and run: source /scratch/opt/environment-setup-i586-poky-linux (for the bash shell)
    * Run the commend to run qemu: qemu-system-i386 -gdb tcp::5531 -S -nographic -kernel arch/x86/boot/bzImage -drive file=core-image-lsb-sdk-qemux86.ext4,if=virtio -enable-kvm -net none -usb -localtime –no-reboot –append "root=/dev/vda rw console=ttyS0 debug". Make sure to change the -kernel option to the new image
  - In screen 2
    * Source the environment configuration script with the command: source /scratch/opt/environment-setup-i586-poky-linux (for the bash shell) (if you didn't close the window from part 1, skip a and b)
    * Run the command: $GDB.
    * In the line for gdb() run: target remote : 5531
    * In the next line for gdb() run: continue
    * Go back to screen 1
  - In screen 1
    * Enter root
    * Run: uname -a
    * Make sure your name is the name -31-hw shows after the location
    * Run reboot to exit.

1.2 qemu-system-i386 Options Explanations

- **gdb tcp::5531** wait for gdb connection on the port 5531
- **S** freeze CPU at startup
- **nographic** disable graphical output and redirect serial I/Os to console
- **kernel bzImage-qemux86.bin** specify kernel image
- **drive file=core-image-lsb-sdk-qemux86.ext4,if=virtio** defines a new drive. There are a few options that are valid
- **enable-kvm** enable KVM full virtualization support
- **net none** use it alone to have zero network devices
- **usb** enable the USB driver

- **localtime** set the clock to local time. The default is UTC time
- **–no-reboot** exit instead of rebooting
- **–append "root=/dev/vda rw console=ttyS0 debug"**. use 'cmdline' as kernel command line

2 Concurrency: Producer-Consumer Problem

2.1 What's the point?

The main point of this assignment is to demonstrate the effective use of parallel processing on a shared resource. Parallel programming is an extremely effective tool, and the producer- consumer problem is a very common problem that requires it's use.

2.2 How did we approach the problem?

We approached the problem as viewing the producer and consumer as the two separate processes. Essentially, our main function would create one type of thread that calls a producer function, while the other creates another type of thread that calls a consumer function. Within both of these threads there needed to be some way to control exclusive access to the shared resource - the buffer. Essentially, we needed to implement a way that blocks the consumer/producer while the other is performing an operation on it. If we hadn't done this, then if the consumer and producer are reading at the same time, or if the producer creates an item when the buffer is full, or the consumer tries to consume an event when the buffer is empty, there would have been an abundance of errors.

2.3 Algorithm

Our algorithm for this problem was to make one mutex which would block access to the buffer, and two semaphores which would be used to make sure the producer and consumer do not make or consume an event when the buffer is either full or empty. The algorithm (written in pseudocode) was as follows:

Consumer:

- Decrement # of items sem. *//checking for empty buffer*
- Lock mutex
- Consume buffer event
- Unlock mutex
- Increment # of spaces sem.

Producer:

- Decrement # of spaces sem. *//checking for full buffer*
- Lock mutex
- Produce event
- Add event to Buffer
- Unlock mutex
- Increment # of items sem.

2.4 How did we ensure our solution was valid?

We ensured that our solution was correct by first starting with one producer and one consumer. We also reduced the size of the buffer from size(32) to size(5) for testing simplicity. Essentially there were 3 main tests that we needed to perform:

- When the buffer is full, producer doesn't create a new item until the consumer creates an empty space
- When the buffer is empty, the consumer doesn't try to consume an item until the producer creates an item.
- The producer and consumer never act on the buffer at the same time

We were able to manipulate the threads to create each of the situations above. The first and second cases were tested by forcing the wait time of the producer or consumer to be shorter than the other depending on the case. This either caused the consumer to "try" to consume an event on an empty buffer because it always acted faster than the producer, or caused the producer to try to produce an event on a full buffer because it always acted faster than the consumer. The third test case was implicitly tested and passed by observing the output of all of our other tests and clearly they never acted simultaneously due to the mutex locking and unlocking.

After we ensured the above passed with one thread of each type, we created multiple threads of both types and performed the same tests once more, each one passing.

2.5 What did we learn?

We learned a lot about p_threads and semaphores through this project. We learned firstly that they are not difficult to implement, even in a relatively difficult to debug language like C. Secondly, we learned that without the function p_thread_join function, the main program will actually terminate even though it creates the subprocesses/threads properly. In order to get the program to run infinitely, we found that making each spawned thread exist in an infinite while loop, and then forcing our main function to wait for them to join was the best/intended solution. Without join, main would just terminate practically instantly. Thirdly, we learned that semaphores are a tool to make threads work with one another. Before this project, we had the misconception that semaphores were what allowed parallel programming to work, but this was proved incorrect. Semaphores were what allowed us to track the properties of buffer_size easily, while our mutex allowed the desired exclusive access. P_threads were the actual construct that made our implemetation of the parallel programming possible.

3 Version Control & Work Logs

3.1 Version Control Log

Version Control Log	
10/2 @ 11 am	Pseudocode for producer/consumer. Implemented buffer and buffer_item struct.
10/2 @ 2 pm	Implemented mutex/semaphores in p/c. Set-up p_threads in main().
10/4 @ 6 pm	Implemented Mersenne Twister + wait times within threads.
10/6 @ 7pm	Implemented rdrand conditional w/ Mersenne Twister.
10/8 @ 2 pm	Implemented multiple consumers + producers. Added p_thread_join()
10/8 @ 1 am	Implemented command line args to handle any number of p/c.

3.2 Work Log

Similar to the concurrency assignment requirement, we did the two portions of Assignment 1 in parallel. We met up a handful of times over the course of the last 2 weeks to work on the assignment together. Ultimately, Tair ended up focusing more on the Kernel portion when we were not together working, and Tyler focused more on the concurrency portion.

The concurrency portion was built in stages. The most important part of the assignment was understanding the problem statement. The pseudocode and algorithm were discussed and implemented first. From there, we implemented the buffer and the struct that the buffer would hold. These both happened very early on and our future build just added the necessary features based on the assignment specifications. The randomness implementation was actually one of the more difficult portions of this assignment. Implementing rdrand and Mersenne twister were both their own challenges. Mersenne twister was the easier of the two and was thus implemented first. Rdrand was one of the final features of the program that we implemented, along with the arguments to handle multiple threads of consumers and producers at the same time.