

# 1 Log of Commands

First we cloned the linux-yocto-3.19 git hub repo into our group 50 directory. We then setup the environment using

```
source ../../files/environment-setup- i586-poky-linux
```

then we moved the config file into the yocto directory using the command

```
cp ../../files/config-3.19.2-standard .config
```

Next to change the local version name we ran the command

```
make menuconfig
```

After we had that setup we then tested that the kernel worked by copying the bzImage-qemu86.bin file and the core-image-lsb-sdk-qemu86.ext4 file into our yocto directory. we then ran the command

```
qemu-system-i386 -gdb tcp::5550 -S -nographic -kernel bzImage-qemu86.bin -drive file=core-image-lsb-sdk-qemu86.ext4,if=virtio -enable-kvm -net none -usb -localtime -no-reboot -append "root=/dev/vda rw console=ttyS0 debug"
```

to start the vm in a halted state. Next we used gdb to connect to the vm remotely by using the commands

```
$GDB
```

```
target remote os2:5550
```

Finally we built our kernel by running

```
make -j4 all
```

and then test that it worked by running the vm again and using the new bzImage by

```
qemu-system-i386 -gdb tcp::5550 -S -nographic -kernel arch/x86/boot/bzImage -drive file=core-image-lsb-sdk-qemu86.ext4,if=virtio -enable-kvm -net none -usb -localtime -no-reboot -append "root=/dev/vda rw console=ttyS0 debug"
```

and then re-ran gdb to connect remotely.

# 2 Concurrency Solution

We create ten threads, five producer threads, and five consumer threads. The producer thread will call the function `producer_start` that will run forever. In the function we get a random value and sleep for that amount of time. We then get two random values one of which we use to determine the amount of time that the consumer will sleep for and the other is the value that the consumer will print out after the consumption time. We then use a semaphore to keep track of the space available in the buffer. If the space becomes full this will block the producer until another space becomes available. Next we use a mutex to lock the other threads out while we put the item into the buffer. Finally we will increment the index of the buffer, unlock the buffer, and increment an items

semaphore that will keep track of whether or not there are any items in the buffer and if none are it will block. When we create the consumer threads we call the `consumer_start`. This function will first decrement the items semaphore, lock the mutex, get the item out of the buffer, decrement the index of the buffer, unlock the mutex and increment the space semaphore, and finally sleep for the consumption time and prints the value and frees the memory. We also have an infinite loop at the end of the main function so the program will not end.

### 3 List of Commands

`-nographic` disables the graphical output and redirects serial I/Os to the console

`-kernel` uses a 'bzImage' as the kernel image.

`-drive` uses a 'file' as a drive image.

`-enable` enable KVM full virtualization support.

`-net none` use it alone to have zero network devices.

`-usb` enable the USB driver.

`-localtime`

`-no-reboot` exit instead of rebooting.

`-append` cmdline use 'cmdline' as kernel command line

### 4 Point of the assignment

To understand how multiple threads working at the same time need to use synchronization when they are sharing data that can be manipulated by each of the threads individually.

### 5 how was the problem approached

We first read how to solve the problem in the Little Book Of Semaphores. we then looked up the semaphores syntax in C. We also looked pthread syntax. We first got the random numbers to generate using one of the options available. We then created the struct and the pthreads. We got the producer function working and then we got the consumer function working.

### 6 How did we test the solution

We added in print statements and manipulated the number of producers and consumers to make sure that the semaphores were properly blocking. We made an abundant amount of producers and a small amount of consumers. Initially the program had a lot of statements being printed by all of the producer threads and eventually the number of print statements decreased and got to a point where they only printed after a consumer message printed proving that the producer was being blocked until a consumer removed something from the buffer. To test that the semaphore was blocking when there was nothing in the buffer we performed the same test except we had an abundant amount of consumers and a small amount of producers.

## 7 What did we learn

That the semaphores are very useful tools for helping with synchronization across multiple threads. Type declarations can be very important. We accidentally had the get random number fuction returning the wrong type and it caused our whole program to not run the way we expected.

## 8 Version control log

commit 1	commit 2	commit 3	commit 4
Author: Shawn Cross	Author: Ryan Crane	Author: Shawn Cross	Author: Shawn Cross
Date: 2017-10-08	Date: 2017-10-08	Date: 2017-10-08	Date: 2017-10-08
initial commit	concurrency	updating the tex file And adding the pdf	changes to the tex file.

## 9 Work Log

Tuesday 10/3 10:00 am to 11:30 am

Met with eachother and began to set up our group directory and tried to get the kernel set up.

Thursday 10/5 10:00 am to 11:00 am and 3:30pm to 6:00pm

Finished getting the kernel working and started on the concurrency assignment.

Sunday 10/8 12:00 pm to 5:00 pm

Finished the concurrency assignment and started the  $\text{\LaTeX}$ document.

Monday 10/9 12:00 pm to 130 pm

Finished the  $\text{\LaTeX}$ document.