CS 444 Project 1: Build the Kernel and Concurrency

Morgan Patch, Mark Bereza

Abstract

For the first project of the course, we first built the Linux Kernel and ran it in an emulator, then developed a C program to demonstrate principles of concurrency. The kernel was built using the tools from the Yocto Project, and run inside of Qemu, a hardware hypervisor. The concurrency project involves the production of multiple threads which all fill or empty the same buffer, requiring proper locking and thread-safe protection of the resources.

October 8, 2017

Contents

1 Summary

1.1 Running the Kernel

We first checked out the kernel:

```
$ mkdir /scratch/fall2017/40
$ cd /scratch/fall2017/40/
$ git clone git://git.yoctoproject.org/linux-yocto-3.19
$ cd linux-yocto-3.19
$ git checkout tags/3.19.2
$ /scratch/bin/acl_open /scratch/fall2017/40 patcht
```

Next, we configured the hypervisor with the default kernel:

```
$ source /scratch/files/environment-setup-i586-poky-linux
$ cp /scratch/files/bzImage-qemux86.bin .
$ cp /scratch/files/core-image-lsb-sdk-qemux86.ext4 .
$ qemu-system-i386 -gdb tcp::5540 -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 a separate terminal session, we connected to the hypervisor with GDB:

```
$ cd /scratch/fall2017/40/linux-yocto-3.19
$ $GDB
(gdb) target remote localhost:5540
(gdb) continue
(gdb) disconnect
(gdb) quit
```

In the booted hypervisor, we logged in with the root account and used ‘shutdown‘ to stop it.

Next, we configured the kernel to our options:

```
$ cp /scratch/files/config-3.19.2-yocto-qemu \
  /scratch/fall2017/40/linux-yocto-3.19/.config
$ make menuconfig
```

The changes were to ensure that it would build a 32-bit kernel and change the version to "-group-hw40", then save the configuration and exit.

Finally, we ran the kernel again:

```
$ qemu-system-i386 -gdb tcp::5540 -S -nographic \
  -kernel ./arch/i386/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"
```

Then connected with GDB, continued the kernel, disconnected, and logged in, as above.

1.2 Building the Concurrency

The concurrency problem required us to build a "producer" – a thread which randomly generates a value and a wait time, places them into a buffer, and waits a brief time in a loop – and a consumer – a thread which pulls items from a buffer, waits the required wait time, then prints the value. The producer blocks if the buffer is full, and the consumer blocks if the buffer is empty.

The buffer is a custom queue struct, which is implemented as a dynamically-allocated array, as well as an index for the frontmost and backmost elements. If either index reaches the end of the array, it wraps around to the beginning, forming a circular queue. The buffer holds 'event's, which are structs that store an integer value and wait time.

The producer runs in an infinite loop. First it creates an event with a random payload and a wait time between 2 and 9, then it waits for the buffer to have space, by locking the buffer mutex, checking the size, and if it is full, unlocking the mutex and waiting. If the buffer is not full, it inserts the event into the buffer, then unlocks the mutex and waits for 3 to 7 seconds.

The consumer follows a similar pattern. In an infinite loop, it locks the mutex, checks if the buffer is empty – if it is, it unlocks the mutex and waits briefly – otherwise it grabs an item out of the buffer, unlocks the mutex, waits the time the item requires, then prints the payload.

2 Qemu flags and options

The command was 'qemu-system-i386 -gdb tcp::5540 -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"'. The flags are, as follows:

- '-gdb tcp::5540'

  Sets the port that the gdbserver will listen on to 5540.

- '-S'

  Starts qemu suspended in debug mode, and opens a gdbserver.

- '-nographic'

  Boots without a graphical interface, saving resources while still allowing the Linux kernel to be booted in a serial display.

- '-kernel'

  Uses the bzImage file listed next as the Linux kernel to boot inside qemu.

- '-drive file=core-image-lsb-sdk-qemux86.ext4,if=virtio'

  Initializes the main hard drive, which loads from the provided ext4 file, and is a "virtual system", i.e. it is not a real, physical drive.

- '-enable-kvm'

  Enables booting qemu as a Kernel-based Virtual Machine, which gives it access to low-level hardware virtualization.

- '-net none'

  Do not initialize any network devices

- ‘-usb‘

  Enable a USB driver.

- ‘-localtime‘

  Set the time inside the emulator as equal to the host system’s local time.

- ‘–no-reboot‘

  If a reboot command is set, instead simply shut down.

- ‘–append ”root=/dev/vda rw console=ttyS0 debug”‘

  Sends the command line parameters ‘root=/dev/vda rw console=ttyS0 debug‘ to the Linux kernel. They are as follows:

  - ‘root=/dev/vda‘
    Sets the device holding the root filesystem to /dev/vda, the virtual filesystem.
  - ‘rw‘
    Mounts the root device as read-write.
  - ‘console=ttyS0‘
    Sets the virtual console to the serial device, because of the ‘-noconsole‘ qemu option.
  - ‘debug‘
    Enables kernel debugging by logging debug events.

3 Concurrency Questions

3.1 What do you think the main point of this assignment is?

The main points of the first half of the assignment are to become familiar with the build processes and structure of large scale applications like the Linux kernel, to learn how to problem solve issues in the kernel, to learn about the concepts involved in a hypervisor, and to get used to debugging applications remotely.

The second half of the assignment is designed to remind us of all of the concepts in concurrency, including multithreading, the pthreads library, mutexes, etc.

3.2 How did you personally approach the problem?

We began by building the producer. It would run in a loop: create an event, add it to the buffer, and wait. The numbers were provided by a pseudocode ‘genrand()‘ function. We decided to make that a real function and split it into a separate file. In order to create an event and add it to the buffer, we needed to create both. Initially we just put it in a static array with an index, as a stack.

Then we needed to figure out how to apply mutexes to the function to be safe. We simply wrote pseudocode ‘lock()‘ and ‘unlock()‘ functions temporarily. Finishing this led to the current implementation structure.

The next step was to write the consumer, which started as a copy of the producer and was altered from there. It was rather simple to build.

Next, we filled out the 'genrand()' functions, using the example file provided on the class website, and then filling in the correct calls to RDRAND or to the Merseinne Twister file.

For the last major piece, we realized that we were instead supposed to use a queue rather than a stack. So we implemented a round queue in the same way as we learned in CS261.

Finally, we just had to wrap the whole thing in a main() function that would initialize the queue, create the threads, and register a signal handler for a clean exit. We also at this point replaced 'lock()' and 'unlock()' with the actual pthreads functions.

3.3 How did you ensure your solution was correct?

We largely tested our solution with by running it on our local machines and the class server, and seeing that it continually worked without any apparent issues. We also used Valgrind to ensure there were no memory leaks.

3.4 What did you learn?

Among the lessons learned were how the kernel works in brief and how to run it in a hypervisor, how to integrate ASM into C code, how to source environment variables from Bash files, how ACLs work, how to set up Visual Studio for Linux development and debugging, many lessons about LaTeXand building it, and how Makefile macros work.

4 Version Control Log

acronym	**meaning**
V	`version`
MF	Number of `modified files`.
AL	Number of `added lines`.
DL	Number of `deleted lines`.

V	**date**	**commit message**	**MF**	**AL**	**DL**
1	2017-10-06	Added source files and .gitignore file	6	366	0
2	2017-10-06	Added makefile	1	19	0
3	2017-10-06	Minor fixes to makefile	1	1	0
4	2017-10-07	fixes to random number generation	5	22	33
5	2017-10-07	Changed default thread counts to #defines, added comments	1	8	7
6	2017-10-07	Now implemented with a queue	5	104	28
7	2017-10-07	Added README	1	11	0
8	2017-10-07	Moved files into assignment1 subdirectory	18	462	462
9	2017-10-07	slight change to folder structure	18	462	462
10	2017-10-07	change folder layout again	18	462	462
11	2017-10-07	Fixed errors in makefile, updated README, added VS files	5	534	7
12	2017-10-07	changed line encodings to LF	7	271	271
13	2017-10-08	Consume message prints AFTER waiting, changes to .gitignore	4	23	17
14	2017-10-08	"make" now performs "make release", updated README	3	7	7

V	date	commit message	MF	AL	DL
15	2017-10-08	mutexs now behave properly	1	8	12
16	2017-10-08	now unlocks the buffer mutex before "working"	2	18	14

5 Work Log

- Monday, 2017-10-02
  - Looked over assignment description.
- Tuesday, 2017-10-03
  - Cloned git repo for linux-yocto
  - Switched to the 3.19.2 tag
  - Sourced using environment configuration script
  - Attempted to build the kernel, but could not due to error in qemu command
  - Began attempting to set up LaTeXdevelopment environment locally
- Thursday, 2017-10-05
  - Set permissions using ACL script
  - Built kernel
  - Configured kernel version to reflect our group name
  - Built "our" kernel
  - Wrote up list of commands needed to accomplish assignment
  - Planned structure of concurrency assignment
  - Wrote up psuedocode for producer and consumer threads
  - Got LaTeXenvironment set up and makefile working
- Friday, 2017-10-06
  - Set up remote development/build environment for Linux using Visual Studio
  - Created Git Repo for source code
  - Created corresponding remote repo on GitHub
  - Wrote code for base code for producer/consumer and random number generation
  - Created makefile for assignment
  - Created preliminary work log and wrote abstract
- Saturday, 2017-10-07
  - Changed code so that events are processed via FIFO, not FILO
  - Fixed a bunch of errors in code and makefile
  - Changed line encodings to Linux-style
  - Wrote section one of writeup, debugged LaTeXmakefile

- Sunday, 2017-10-08
  - Changed code so that consumed message prints after "work" is done, not before
  - Split single mutex into buffer_mutex and print_mutex
  - Tested code on server and with Valgrind
  - Wrote remainder of writeup

Appendix 1: Source Code

5.1 eventqueue.h

```
#ifndef EVENTQUEUE_H_INCLUDED
#define EVENTQUEUE_H_INCLUDED

#include <stdbool.h>

struct event {
        unsigned long val;
        unsigned long wait;
};

struct event_queue {
        struct event *events;
        int capacity;
        int size;
        int front;
        int back;
};

struct event_queue *create_queue(int capacity);

void delete_queue(struct event_queue *queue);

bool is_empty(struct event_queue *queue);

bool is_full(struct event_queue *queue);

void enqueue(struct event_queue *queue, struct event item);

struct event dequeue(struct event_queue *queue);

#endif /* EVENTQUEUE_H_INCLUDED */
```

5.2 eventqueue.c

```
#include <stdlib.h>
#include "eventqueue.h"
```

```
struct event_queue *create_queue(int capacity)
{
        struct event_queue *queue = malloc(sizeof(*queue));
        queue->events   = calloc(capacity, sizeof(struct event));
        queue->capacity = capacity;
        queue->size     = 0;
        queue->front    = 0;
        queue->back     = 0;

        return queue;
}

void delete_queue(struct event_queue *queue)
{
        free(queue->events);
        free(queue);
}

bool is_empty(struct event_queue *queue)
{
        return (queue->size == 0);
}

bool is_full(struct event_queue *queue)
{
        return (queue->size >= (queue->capacity - 1));
}

void enqueue(struct event_queue *queue, struct event item)
{
        queue->events[queue->back] = item;

        if (queue->back < (queue->capacity - 1))
                (queue->back)++;
        else
                queue->back = 0;

        (queue->size)++;
}

struct event dequeue(struct event_queue *queue) {
        struct event e = queue->events[queue->front];

        if (queue->front < (queue->capacity - 1))
                (queue->front)++;
        else
                queue->front = 0;

        (queue->size)--;

        return e;
```

```
}
```

5.3 mt.h

```
#ifndef MT_H_INCLUDED
#define MT_H_INCLUDED

void init_genrand(unsigned long s);

unsigned long genrand_int32(void);

long genrand_int31(void);

double genrand_real1(void);

double genrand_real2(void);

double genrand_real3(void);

double genrand_res53(void);

#endif /* MT_H_INCLUDED */
```

5.4 rand.h

```
#ifndef RAND_H_INCLUDED
#define RAND_H_INCLUDED

void rand_init();

unsigned long rand_uint();

unsigned long rand_uint_inclusive(int, int);

#endif // RAND_H_INCLUDED
```

5.5 rand.c

```
#include <stdio.h>
#include <time.h>
#include "mt.h"
#include "rand.h"

void rand_init()
{
        init_genrand(time(NULL));
}

unsigned long rand_uint()
{
```

```
        unsigned long eax;
        unsigned long ebx;
        unsigned long ecx;
        unsigned long edx;
        eax = 0x01;

        __asm__ __volatile__(
                "cpuid;"
                : "=a"(eax), "=b"(ebx), "=c"(ecx), "=d"(edx)
                : "a"(eax)
        );

        unsigned long rand;
        if (ecx & 0x40000000) {
                __asm__ __volatile__(
                        "rdrand %%eax"
                        : "=a"(rand)
                );
        } else {
                rand = genrand_int32();
        }

        return rand;
}

unsigned long rand_uint_inclusive(int lower, int upper)
{
        unsigned long output = (rand_uint() % (upper - lower + 1)) + lower;
        return output;
}
```

5.6 main.c

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <signal.h>
#include "rand.h"
#include "eventqueue.h"

#define BUFFER_SIZE             32
#define DEFAULT_NUM_PRODUCERS 5
#define DEFAULT_NUM_CONSUMERS 3

pthread_mutex_t buffer_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t print_mutex  = PTHREAD_MUTEX_INITIALIZER;
struct event_queue *buffer;

void sig_handler(int signal)
{
```

```
        perror("\nTerminating...\n");
        pthread_mutex_lock(&buffer_mutex);
        delete_queue(buffer);
        exit(0);
}

void *producer(void *dummy)
{
        while (1) {
                struct event e;
                e.val = rand_uint();
                e.wait = rand_uint_inclusive(2, 9);
                pthread_mutex_lock(&buffer_mutex);
                if (!is_full(buffer)) {
                        enqueue(buffer, e);
                        pthread_mutex_unlock(&buffer_mutex);
                        pthread_mutex_lock(&print_mutex);
                        printf("Added event %lu to buffer.\n", e.val);
                        pthread_mutex_unlock(&print_mutex);
                        sleep(rand_uint_inclusive(3, 7));
                } else {
                        pthread_mutex_unlock(&buffer_mutex);
                        sleep(1);
                }
        }
}

void *consumer(void *dummy)
{
        while (1) {
                pthread_mutex_lock(&buffer_mutex);
                if (!is_empty(buffer)) {
                        struct event e = dequeue(buffer);
                        pthread_mutex_unlock(&buffer_mutex);
                        sleep(e.wait);
                        pthread_mutex_lock(&print_mutex);
                        printf("Consumed event %lu.\n", e.val);
                        pthread_mutex_unlock(&print_mutex);
                } else {
                        pthread_mutex_unlock(&buffer_mutex);
                        sleep(1);
                }
        }
}

int main(int argc, char **argv)
{
        /* Ignore Ctrl+C until buffer is allocated */
        signal(SIGINT, SIG_IGN);

        /* Initialize random number generator */
```

```
        rand_init();

        /* Create the event buffer */
        buffer = create_queue(BUFFER_SIZE);

        /* Now that the buffer has been allocated, Ctrl+C will free and exit */
        signal(SIGINT, sig_handler);

        /* Use defaults unless their number is specified via command line */
        int num_producers = DEFAULT_NUM_PRODUCERS;
        int num_consumers = DEFAULT_NUM_CONSUMERS;
        if (argc > 2) {
                num_producers = atoi(argv[1]);
                num_consumers = atoi(argv[2]);
        }

        /* Create threads for producers */
        pthread_t pro_threads[num_producers];
        for (int i = 0; i < num_producers; ++i) {
                pthread_create(&pro_threads[i],
                               NULL,
                               producer,
                               NULL);
        }

        /* Create threads for consumers */
        pthread_t con_threads[num_consumers];
        for (int i = 0; i < num_consumers; ++i) {
                pthread_create(&con_threads[i],
                               NULL,
                               consumer,
                               NULL);
        }

        /* Wait for interrupt signal */
        pause();

        return 0;
}
```

5.7 Makefile

```
CC      = gcc
CFLAGS  = -std=c99 -pthread
SOURCES = main.c rand.c mt19937ar.c eventqueue.c
HEADERS = rand.h mt.h eventqueue.h
FILES   = $(SOURCES) $(HEADERS)
OUTDIR  = ./build
OUTPUT  = $(OUTDIR)/assignment1

release: create_outdir $(FILES)
```

```
	$(CC) $(CFLAGS) -O2 -o $(OUTPUT) $(SOURCES)

debug: create_outdir $(FILES)
	$(CC) $(CFLAGS) -g -Wall -o $(OUTPUT) $(SOURCES)

clean: create_outdir
	rm -rf $(OUTPUT)

rebuild_release: clean release

rebuild_debug: clean debug

create_outdir:
	mkdir -p $(OUTDIR)
```