Prof. Heonyoung Yeom

System Programming (001)

HW - Semaphores

Report

Dept. of Computer Science and Engineering
2021-13194
Jaejun Ko

2023.5.25

HW - Semaphores Report

2021-13194 Jaejun Ko

Introduction

In this lab session, I will program two Linux modules, `ptree`, and `paddr`. By implementing these, I can understand *Linux Kernel Module Programming* based on *Debug File System* interface. `ptree` is tracing the process tree from a specific process id, and `paddr` is finding a physical address using a virtual address.

Description

The following is my development environment. I'm using Ubuntu 20.04.6 LTS, which has Linux kernel 5.15.0-72-generic. It works on x86_64 architecture and CPU op-modes are 32-bit and 64-bit. It has 8 core CPU, and each core has 2 threads, which means the hyper-threading is enabled. The clock speed is s1800.000 MHz, where the maximum is 3900.000 MHz and the minimum is 400.0000MHz.

```
~ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.6 LTS
Release:        20.04
Codename:       focal
~ uname -ar
Linux kjj-Ubuntu 5.15.0-72-generic #79~20.04.1-Ubuntu SMP Thu Apr 20 22:12:07 UTC 2023 x86_64 x86_64 x86_64 GNU/Linux
~ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           142
Model name:                      Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz
Stepping:                        11
CPU MHz:                         1800.000
CPU max MHz:                     3900.0000
CPU min MHz:                     400.0000
```
s

In this environment, I built two c files, `badcnt.c`, and `goodcnt.c`. The contents of the two files are written below. The decisive difference between the two is whether *semaphore* is applied. *Semaphore* is a variable or abstract data type used to control access to a common resource by multiple threads and avoid critical section problems in a concurrent system such as a multitasking operating system. Thus, `badcnt.c` is so buggy, and `goodcnt.c` is safe to execute. In this example, *mutex* is used, which is binary semaphore used for mutual exclusion.

There are two operations in mutex, P operation and V operation. P operation locks the mutex, and V operation unlocks it. In `goodcnt.c`, mutex is locked before incrementing `cnt` using P operation, and unlock it after incrementation, using V operation. However, since there are more instructions in `goodcnt.c` than `badcnt.c`, it takes longer than `badcnt.c`.

```
/*
 * badcnt.c - An improperly synchronized counter program
 */
/* WARNING: This code is buggy! */
```

```c
#include "csapp.h"

void *thread(void *vargp);  /* Thread routine prototype */

/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    /* Check input argument */
    if (argc != 2) {
       printf("usage: %s <niters>\n", argv[0]);
       exit(0);
    }
    niters = atoi(argv[1]);

    /* Create threads and wait for them to finish */
    Pthread_create(&tid1, NULL, thread, &niters);
    Pthread_create(&tid2, NULL, thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
       printf("BOOM! cnt=%ld\n", cnt);
    else
       printf("OK cnt=%ld\n", cnt);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    long i, niters = *((long *)vargp);

    for (i = 0; i < niters; i++) //line:conc:badcnt:beginloop
       cnt++;                   //line:conc:badcnt:endloop

    return NULL;
}
```

```c
/*
 * goodcnt.c - A correctly synchronized counter program
 */
/* $begin goodcnt */
#include "csapp.h"

void *thread(void *vargp); /* Thread routine prototype */

/* Global shared variables */
    volatile long cnt = 0; /* Counter */
    sem_t mutex;           /* Semaphore that protects counter */

int main(int argc, char **argv)
{
    int niters;
    pthread_t tid1, tid2;

    /* Check input argument */
    if (argc != 2) {
       printf("usage: %s <niters>\n", argv[0]);
       exit(0);
    }
    niters = atoi(argv[1]);

    /* Create threads and wait for them to finish */
    Sem_init(&mutex, 0, 1);  /* mutex = 1 */
    Pthread_create(&tid1, NULL, thread, &niters);
    Pthread_create(&tid2, NULL, thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
```

```
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    int i, niters = *((int *)vargp);
    for (i = 0; i < niters; i++) {
        P(&mutex);
        cnt++;
        V(&mutex);
    }
    return NULL;
}
```

Result

The following is the result of the execution of `badcnt`. The correct output is cnt=20000, but in outputs cnt=16021, which means it accessed some unsafe regions when working on it. As unsafe regions are not blocked, the result of `badcnt` is unstable and works in the wrong way. By using `clock()` in <`time.h`>, I could get execution time, the average time of execution was 0.000267 s, and the standard deviation was 7.17E-5 s.

```
~/../../../conc  main ?3  ./badcnt 10000
OK cnt=20000
~/../../../conc  main ?3  ./badcnt 10000
BOOM! cnt=16021
```

The following is the result of the execution of `goodcnt`. Unlike `badcnt`, it always outputs the correct result of execution. it means the blocking is performed well, and any thread did not access any unsafe regions of the progress graph. By using `clock()` in <`time.h`>, I could get the execution time, the average time of execution was 0.006243 s, and the standard deviation was 0.00292 s. Compared to `badcnt`, it was much slower(almost 23 times slower).

```
~/../../../conc  main ?3  ./goodcnt 10000
OK cnt=20000
~/../../../conc  main ?3  ./goodcnt 10000
OK cnt=20000
~/../../../conc  main ?3  ./goodcnt 10000
OK cnt=20000
~/../../../conc  main ?3  ./goodcnt 10000
OK cnt=20000
~/../../../conc  main ?3  ./goodcnt 10000
OK cnt=20000
~/../../../conc  main ?3  ./goodcnt 10000
OK cnt=20000
```

Conclusion

In this HW, I can use mutex, a one of the semaphores. I can understand that performance of `goodcnt` is very low comparing to `badcnt`, instead of safety. I can deduce that the overhead of locking is very large.