Operating System

Assignment #2

담당 교수 : 김태석 교수님
수업 일시 : 월2, 수1
학과 : 컴퓨터 공학과
학번 : 2013722095
이름 : 최 재 은

1. Introduction

2-1) 생성할 thread의 2배 만큼의 파일에 1~10 사이의 값을 써넣고 Thread가 불러내어 값을 더하는 프로그램을 작성한다. 이 과정에서 thread는 2개의 파일에서 값을 읽어 들여 이를 더하게 된다. 위의 과정을 하나의 값만 남을 때까지 반복한다.

2-2) Cpu scheduling을 사용하여 각 scheduling policy마다의 성능 차이를 비교해 본다. 이를 위해서 2-1 문제와 같이 무작위 값이 저장된 10000개의 파일을 생성하고, 이를 thread가 아닌 process가 읽어 들이도록 한다. 이 과정에서 각 과정을 끝내기 까지의 소요시간을 측정하여 비교하도록 한다.

2. Reference

https://www.joinc.co.kr/w/Site/Thread/Beginning/Mutex
http://www.iamroot.org/xe/index.php?document_srl=13525&mid=Programming

3. Conclusion

- Analysis

2-1)

```
os2013722095@ubuntu:~/os/hw2/2-1$ ls
genfile  gen_file.c  genthread  gen_thread.c  Makefile
os2013722095@ubuntu:~/os/hw2/2-1$ ./genfile
os2013722095@ubuntu:~/os/hw2/2-1$ ls
genfile  gen_file.c  genthread  gen_thread.c  Makefile  tmp1
os2013722095@ubuntu:~/os/hw2/2-1$ cd tmp1
os2013722095@ubuntu:~/os/hw2/2-1/tmp1$ ls
0  1  2  3
```

- max_thread가 2로 하여 genfile을 실행시키면 tmp1경로에 총 4개의 파일이 생성된다.

```
os2013722095@ubuntu:~/os/hw2/2-1$ ./genthread
2 + 4 = 6
7 + 5 = 12
PID : 3025
TID : 3026

6 + 12 = 18
PID: 3025
TID: 3027
os2013722095@ubuntu:~/os/hw2/2-1$
```

- 위의 과정에서 생성된 파일들은 2개의 thread를 통해서 더해지는데 그 결과는 위와 같다. 각각의 thread에서 덧셈이 따로 진행됨을 보이기 위해 PID, TID를 출력해 주었다.

- 과제를 진행하면서 depth1의 thread가 critical section에서 작업 중일 때 depth2의 thread가 critical section에 들어오는 것을 막기 위해 mutex를 사용하였다.

```
#define MAX_THREAD 2

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int value[2];
```

- 공유자원으로 int형 변수를 주었다.

```
pthread_create(&t1, NULL, thread_sum1, "thread1");
usleep(400);
pthread_create(&t2, NULL, thread_sum2, "thread2");
```

- thread가 연속적으로 생성될 때 depth2의 thread가 먼저 생성되면 mutex가 효과적이지 않으므로 400 micro sec 동안 sleep을 주었다.

```
for(i = 0, j = 0; i < MAX_THREAD*2; i+=2, j++)
{

        sprintf(filename, "%d", i);
        f1 = fopen(filename, "r");
        sprintf(filename, "%d", i+1);
        f2 = fopen(filename, "r");

        a = fgetc(f1);
        a -=48;
        b = fgetc(f2);
        b -=48;

        fclose(f1);
        fclose(f2);
        sum = a + b;

        printf("%d + %d = %d\n", a, b, sum);
        value[j] = sum;


}
```

- depth1에서 총 4개의 파일에 접근하여 내용을 읽어 들이고 이를 더하여 공유자원에 저장한다.

```
pthread_mutex_lock(&mutex);

a = value[0];
b = value[1];


pthread_mutex_unlock(&mutex);

printf("%d + %d = %d\n", a, b, a+b);
```

- depth2에서도 공유자원에 접근하여 값을 읽어 들여 이를 더한다.

2-2)

```
149   1990  2490  2991  3491  3992  4492  4993  5493  5994  6494  6995  7495  7996  8496  8997  9497  9998
1490  1991  2491  2992  3492  3993  4493  4994  5494  5995  6495  6996  7496  7997  8497  8998  9498  9999
1491  1992  2492  2993  3493  3994  4494  4995  5495  5996  6496  6997  7497  7998  8498  8999  9499
1492  1993  2493  2994  3494  3995  4495  4996  5496  5997  6497  6998  7498  7999  8499  9     95
1493  1994  2494  2995  3495  3996  4496  4997  5497  5998  6498  6999  7499  8     85    90    950
1494  1995  2495  2996  3496  3997  4497  4998  5498  5999  6499  7     75    80    850   900   9500
1495  1996  2496  2997  3497  3998  4498  4999  5499  6     65    70    750   800   8500  9000  9501
1496  1997  2497  2998  3498  3999  4499  5     55    60    650   700   7500  8000  8501  9001  9502
1497  1998  2498  2999  3499  4     45    50    550   600   6500  7000  7501  8001  8502  9002  9503
1498  1999  2499  3     35    40    450   500   5500  6000  6501  7001  7502  8002  8503  9003  9504
os2013722095@ubuntu:~/os/hw2/2-2/tmp2$ cd ..
os2013722095@ubuntu:~/os/hw2/2-2$ ls
genfile  gen_file.c  Makefile  schedtest  schedtest.c  tmp2
os2013722095@ubuntu:~/os/hw2/2-2$
```

- 2-1 문제와 같이 1~10의 값이 저장된 10000개의 파일을 생성하였다.

```
scheduler.sched_priority = sched_get_priority_min(SCHED_OTHER);
sched_setscheduler(0, SCHED_OTHER, &scheduler);
```

- 위와 같은 방법으로 scheduler라는 sched_param 구조체의 priority 값을 현재 얻을 수 있는 최솟값으로 set 해준다.
- 두 번째 줄의 코드는 어떤 정책으로 cpu를 scheduling 할 것인지 정하는 것이다.

```
os2013722095@ubuntu:~/os/hw2/2-2$ ./schedtest
os2013722095@ubuntu:~/os/hw2/2-2$
Consumtion time : 4.747491965
```

- Nomal(sched_other)로 정책을 선택한 경우

```
root@ubuntu:~/os/hw2/2-2# ./schedtest

Consumtion time : 1.64436070
```

- sched_fifo로 정책을 선택한 경우

```
root@ubuntu:~/os/hw2/2-2# ./schedtest

Consumtion time : 1.782824015
root@ubuntu:~/os/hw2/2-2#
```

- sched_rr로 정책을 선택한 경우

```
while(i< MAX_PROCESSES)
{
        pid = fork();
        if(pid< 0 )
                return -1;              // error case

        else if(pid == 0)       // child process
        {
                sprintf(filename, "%d", i);             // read file
                fp = fopen(filename, "r");
                value = fgetc(fp);
                fclose(fp);
                if(i == 9999) break;            // escape loop when last process
                exit(0);                                                // close p

        }
                                                //parent process
        else    // kill parent when the last child is made
        {
                if(i == 9999)
                        return 0;
        }
```

- MAX_PROCESSES = 10000으로 두고 총 10000개의 process들을 생성하는데 fork()를 사용하였다.
- 부모 프로세스는 마지막 자식 프로세스를 생성하는 순간 종료된다.
- 그 과정에서 자식프로세스가 생성되면 그 자식 프로세스는 파일을 읽은 후 종료되는데 마지막 자식 프로세스는 추가적으로 loop를 탈출하여 아래의 코드를 통해 총 소요 시간을 출력한다.

```
clock_gettime(CLOCK_MONOTONIC, &end);
sec = end.tv_sec - start.tv_sec;
nano = end.tv_nsec - start.tv_nsec;
if(nano < 0)
{
        sec -=1;
        nano = 1- nano;
}
printf("\nConsumtion time : %ld.%ld\n", sec, nano);
```

- clock_gettime()을 사용하여 시작할 때의 시간과 종료된 시간을 구하고 이들의 차를 계산하여 총 소요 시간(성능)을 계산해 주었다.

- 고찰

2-1의 경우 thread를 통해 임계구역에 접근할 때 mutex를 사용하여 한 번에 한 thread만 임계구역에 접근할 수 있도록 제한하였다. 이를 통하여depth1의 thread가 임계구역에서 작업 중일 때 depth2의 thread가 임계구역에 접근할 수 없도록 하며, 이러한 방법을 '공유 자원에 대한 동기화'라고 한다. 이 과정을 수행할 때에 순차적으로 thread1, 2가 생성되도록 하였으나 thread2가 1보다 먼저 생성되어 critical section에 접근하는 경우가 자주 발생하였다. 이를 막기 위해 약간의 sleep을 주었다. 또한 공유자원을 전역변수로 선언하여 접근, 공유할 수 있도록 하였다. 또한 과제를 수행하면서 서로 다른 thread에서 공유자원에 접근, 작업하고 있음을 보여주기 위해 pid, tid를 출력하여 구분하였다.

2-2의 경우 cpu의 process를 처리하는 정책을 re-scheduling하는 것을 목적으로 한다. 이 과정에서 성능 차이를 보이기 위해 충분히 많은 프로세스를 생성하도록 한다. cpu는 프로세스의 작업을 수행할 때에 time sharing 기법을 사용하는데 이는 단위 시간(time quantum) 동안만 cpu를 할당해 주는 방법으로 cpu utilization을 극대화하는 방법이다. 위에서 언급한 3개의 scheduling 방법 중 FIFO를 제외한 두 스케줄링은 time sharing 방법을 사용한다. sched_other는 모든 프로세스들이 0의 priority(우선순위가 없음)를 가지며, time share하면서 동등하게 스케쥴 된다. sched_fifo의 경우, 1~99까지의 priority를 가질 수 있으며 자신보다 높은 priority를 갖는 프로세스에 의해 선점될 수 있으며(preemptive) 선점한 프로세스가 cpu를 놓아주게 되면 계속 동작하는 방식이다. sched_rr의 경우 기본적으로 sched_ff와 동일하게 동작하되 , time quantum을 모두 소진하게 되면 priority의 대기 리스트 뒤로 이동되는 차이점이 있다.

FIFO는 기본적으로 다른 프로세스에 의해 선점되기 전까지는 작업 중인 프로세스가 끝날 때까지 기다려야 한다. 그에 비해 Round Robin방식은 time quantum이 소진되면 자동으로 대기열의 최후위로 이동되고 다음 프로세스가 실행되는데, 이 방법은 user입장에서는 모든 프로세스가 동시에 조금씩 진행되는 것처럼 보이는 효과를 보이지만, 실제로는 레지스터 등의 값을 변경하기 위한 overhead(time)가 증가하게 되어 시간적인 면에서는 더 비효율적이다.

Round Robin은 time share할 대상이 없으면 fifo와 같은 동작을 하므로 other 보다는 더 효율적으로 프로세스들을 처리할 수 있다. 즉 프로세스 변경에 따른 overhead가 적다.

FIFO는 현재 프로세스가 끝나기 전 또는 다른 프로세스에 의해 선점되기 전까지는 cpu를 소유하고 있으므로 프로세스 변경에 따른 overhead가 상대적으로 매우 적다.

즉, 세 스케쥴링 방법을 비교하면 이론적으로 FIFO > RR > NORMAL(OTHER)이 성립된다.