Assignment2

운영체제

2-1: Process & Thread

2-3: CPU scheduling policy

제출일: 11월 03일 일요일

담당 교수: 김태석

학 번: 2015722025

학 과: 컴퓨터정보공학부

이 름: 정용훈

1. Introduction

2-1과제는 Linux에서의 명령 fork() → process생성과 thread를 생성하는 명령으로 기존 text파일에서 값을 읽어와 각 process와 thread가 계산하여, 순차적으로 값을 도출하여, 최종적으로 메모장에 있는 모든 값들이 더하는 과정을 process와 thread의 동작 시간비교를 하며, 2-2과제는 여려 종류의 CPU schedule에 대하여 존재하는 text파일을 읽는데 얼마나 걸리는지 schedule에 대한 비교와 결과를 이해하는데 목적을 두고 있습니다.

2. Requirements for each task(Conclusion)

A. Assignment 2-1

(1) Process

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <time.h>
#include <stdlib.h>

#define MAX_PROCESSES 64

int P_NUM = 0;
int data=0; //For saving data
pid_t P_id[MAX_PROCESSES]; //For saving process id
int result[MAX_PROCESSES*2]; //For saving result of addition

int main()
{
    FILE *f_read = fopen("temp.txt","r"); //open data file
    struct timespec start={0,0}, end={0,0}; //For check excution time
    char number[10]; //For saving string from text file

    for(int i=0; i<MAX_PROCESSES*2;i++) //Save data from text to array
    {
        fgets(number,sizeof(char)*10,f_read); //load data
        result[i]=atoi(number); //make array
    }
```

위 코드는 Process생성전 계산을 하기 위해 준비되는 code입니다. Process의 개수를 정의 해줌과 동시에 계산을 위한 array와 각 상태를 저장해줄 변수들을 선언합니다. 다음으로 main함수에 진입하면 계산을 하기 위해 존재하는 text파일에서 값을 읽어 array에 저장하는 과정을 확인할 수 있습니다. 계속해서 아래는 실제 process생성 과정과 process들의 동작과정을 나타내는 code입니다.

```
28  ////////////////////////////////////////////////////////////////
29      clock_gettime(CLOCK_MONOTONIC, &start); //get time
30      for(int j=MAX_PROCESSES; j>=1; j/=2)
31      {
32          for(int i=0; i<j; i++, P_NUM++)
33          {
34              P_id[P_NUM]=fork(); //create porcess
35
36              if(0 < P_id[P_NUM])
37                 continue;
38
39              else
40                  exit(result[2*P_NUM]+result[(2*P_NUM)+1]); //cacluation
41
42
43          }
44
45          for(int i=0; i<P_NUM; i++)
46          {
47              waitpid(P_id[i],&result[i],0); //wait child prcess
48              result[i]=result[i]>>8; //save result from child processes
49          }
50          P_NUM=0;
51      }
52      clock_gettime(CLOCK_MONOTONIC, &end); //get time
53  ////////////////////////////////////////////////////////////////
54
55      printf("value of fork: %d\n",result[0]); //result
56      printf("%.6f\n", ((double)end.tv_sec + 1.0e-9*end.tv_nsec)
57      -((double)start.tv_sec + 1.0e-9*start.tv_nsec) ); //excution time
58      return 0;
59  }
```

이전 code에서 준비 과정을 모두 끝낸 후 처음 반복 조건에서는 정의된 MAX_PROCESSES만큼의 process를 fork명령을 통하여 생성한 후 각 데이터들을 2개씩 읽어 각각의 결과를 exit함수를 이용하여 main process에 전달하며, main process는 wait함수를 통하여, 해당 결과 값을 array에 저장합니다. 이 과정을 도출되는 값이 1개가 될 때까지 반복하게 되면, 최종적으로 process가 text파일의 값을 모두 계산하는 execution time을 도출할 수 있습니다.

(2) Thread

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <time.h>
#include <stdlib.h>
#include <pthread.h>

#define MAX_PROCESSES 64

int P_NUM = 0;
pthread_t P_id[MAX_PROCESSES]; //For saveing thread id
int result[MAX_PROCESSES*2]; //For saving result
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER; //For prevent race condition

int sum(void *arg) //Working of thread
{
    pthread_mutex_lock(&counter_mutex); //lock
    int sum=result[2*(P_NUM)]+result[(2*P_NUM++)+1]; //Shared resource
    pthread_mutex_unlock(&counter_mutex); //unlock

   return sum; //return value of reslut
}

int main()
{
    FILE *f_read = fopen("temp.txt","r"); //open text file
    struct timespec start={0,0}, end={0,0}; //For checking excution time
    char number[10]; //For get string from text file

    for(int i=0; i<MAX_PROCESSES*2;i++) //For saving data to array from text
    {
        fgets(number,sizeof(char)*10,f_read);
        result[i]=atoi(number);
    }
```

위 코드는 thread를 사용하기 위한 준비 code입니다. Fork code와 마찬가지로 정의된 MAX_PROCESSES를 기반으로 프로그램이 동작하며, 각 thread의 id를 저장하기 위한 array와 값을 저장하기 위한 array가 선언되어 있으며, execution time을 측정하기 위한 변수가 선언되어 있습니다. 가장 다른 특징으로는 각 thread가 동작해야 하는 함수를 따로 선언해주어야 하며, 해당 함수는 sum으로 덧셈 값을 return하는 함수가 선언되어 있는 것을 확인할 수 있습니다. 또한 fork와는 다르게 race condition을 막아 주는 code를 작성하였는데 이는 fork와 thread의 실행 속도 차이나 접근 방법에 대한 고찰이 필요하다고 생각됩니다. (fork또한 안전하게 쓰기 위해서는 semaphore를 사용 추천)

```c
////////////////////////////////////////////////////////////////////
    clock_gettime(CLOCK_MONOTONIC, &start);
    for(int j=MAX_PROCESSES; j>=1; j/=2)
    {
        for(int i=0; i<j; i++) //create thread
            pthread_create(&P_id[i], NULL, (void *)sum, NULL);


        for(int i=0; i<j; i++) //equal to wait
            pthread_join(P_id[i], (void **)&result[i]);
        P_NUM=0;
    }
    clock_gettime(CLOCK_MONOTONIC, &end);
////////////////////////////////////////////////////////////////////

    printf("value of thread: %d\n",result[0]); //print value
    printf("%.6f\n", ((double)end.tv_sec + 1.0e-9*end.tv_nsec) //time
    -((double)start.tv_sec + 1.0e-9*start.tv_nsec) );
    return 0;
}
```

반복적인 동작을 하는 알고리즘은 fork code와 크게 다르지 않습니다.

(3) Result 2-1 & compare

```
os2015722025@ubuntu:/mnt/hgfs/shared/2-1$ ls
fork.c  Makefile  numgen.c  thread.c
```

현재 directory의 파일들을 나타낸 것이다. 아래는 해당 파일들을 컴파일 하기 위한 makefile이다.

```
all:
    gcc -o numgen numgen.c
    gcc -o fork fork.c
    gcc -o thread thread.c -pthread

check:
    $ ./numgen
    $ ./fork
    $ ./thread

clear:
    $ rm -rf tmp*
    $ sync
    $ echo 3 | sudo tee /proc/sys/vm/drop_caches
```

Make 명령을 통해 세가지 파일을 모두 컴파일 할 수 있으며, 해당 값들은 check를 통해 결과 값을 확인할 수 있으며, clear를 통하여 과제에서 요구하는 실험 마다 버퍼나, 캐시를 비워줄 수 있다. 아래는 실험 과정이다.

```
os2015722025@ubuntu:/mnt/hgfs/shared/2-1$ make
gcc -o numgen numgen.c
gcc -o fork fork.c
gcc -o thread thread.c -pthread
os2015722025@ubuntu:/mnt/hgfs/shared/2-1$ ls
fork  fork.c  Makefile  numgen  numgen.c  thread  thread.c
```

```
os2015722025@ubuntu:/mnt/hgfs/shared/2-1$ make clear
rm -rf tmp*
sync
echo 3 | sudo tee /proc/sys/vm/drop caches
[sudo] password for os2015722025:
3
```

```
os2015722025@ubuntu:/mnt/hgfs/shared/2-1$ make check
./numgen
./fork
value of fork: 64
0.022292
./thread
value of thread: 8256
0.020426
```

해당 결과 같은 MAX_PROCESSES를 64로 놓은 결과 값이다. 아래는 좀 더 많은 표본을 모아 놓은 것이다.

MAX_PROCESSES = 4

```
os2015722025@ubuntu:/mnt/hgfs/shared/2-1$ make check
./numgen
./fork
value of fork: 36
0.004867
./thread
value of thread: 36
0.000640
```

MAX_PROCESSES = 8

```
os2015722025@ubuntu:/mnt/hgfs/shared/2-1$ make check
./numgen
./fork
value of fork: 136
0.010016
./thread
value of thread: 136
0.002322
```

MAX_PROCESSES = 16

```
os2015722025@ubuntu:/mnt/hgfs/shared/2-1$ make check
./numgen
./fork
value of fork: 16
0.010717
./thread
value of thread: 528
0.002519
```

MAX_PROCESSES = 32

```
os2015722025@ubuntu:/mnt/hgfs/shared/2-1$ make check
./numgen
./fork
value of fork: 32
0.016439
./thread
value of thread: 2080
0.004727
```

실험결과 **공유하는 자원이 thread가 많고 fork가 적기** 때문에 대체로 fork를 통한 측정시간이 thread에 비해 더 많이 나오는 것을 확인할 수 있었다. 하지만 측정시간은 컴퓨터 현재 환경에 많은 영향을 받으므로 음악을 틀어놓고 있다거나 측정 중 다른 작업을 한다면, 그 순간 값이 다르게 측정되는 경우도 있다 그런 예외 상황을 배제한다면 거의 thread의 측정시간이 짧게 나온다. 또한 fork의 결과 값이 MAX_PROCESSES 16이후부터 정의된 값으로 나오는데 이러한 이유는 8 bit를 shift하는 이유에서 나타납니다. Child 종료 상태 값은 16비트로 값을 구성하며, 하위 8비트가 0이면 상위 8비트의 값이 exit함수에 전달한 인자 값 이므로, wait를 통해 값을 전달 받기 위해서는 8만큼 shift해주어야 합니다. 그러므로 반환 받을 수 있는 값은 최대 255이므로 초과되는 값들은 결과 값이 thread와 다른 것을 확인할 수 있습니다.

B. Assignment 2-2

(1) Scheduling policy of CPU

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <stdlib.h>
#include <sched.h>
#include <sys/resource.h>

#define MAX_PROCESSES 10000 //Number of process
FILE* f_read; //File
pid_t IDs[MAX_PROCESSES]; //array
char name[10];
char data[10];
int status;

int main()
{
    int sel = 0; //For select menu
    double time = 0;
    struct sched_param s;
    struct timespec start = { 0,0 }, end = { 0,0 };

    while(1)
    {
        printf("--------Menu--------\n"); //View menu
        printf("1.RR       - highest\n");
        printf("2.RR       - default\n");
        printf("3.RR       - lowest \n");
        printf("4.FIFO     - highest\n");
        printf("5.FIFO     - default\n");
        printf("6.FIFO     - lowest \n");
        printf("7.Standard - highest\n");
        printf("8.Standard - default\n");
        printf("9.Standard - lowest \n");
        printf("10.END PROGRAM\n");
        printf("--------------------\n");
        printf("Select: ");
        scanf("%d",&sel);
        printf("\n");
```

위 code는 해당 과제를 Instruction에서 소개했던 것과 마찬가지로 Linux에서 지원하는 scheduling policy의 차이를 알아보기 위해 작성된 code이다. 우선 각각의 scheduling policy를 선택하여 시간 측정을 할 수 있도록 메뉴를 만들었으며, 종료가 필요하면 10을 입력하면 된다. 나머지 선언된 변수들은 time측정과 이에 출력에 필요한 변수들이다.

```
68  ////////////////////////////////////////////////////////////////////
69          clock_gettime(CLOCK_MONOTONIC, &start); //check excution time
70          for(int i=0; i<MAX_PROCESSES;i++)
71          {
72              IDs[i] = fork(); //make processes
73
74              if(IDs[i]>0)
75                  continue;
76
77              else
78              {
79                  if(sel==1)
80                  { //Each scheduling policy of CPU
81                      s.sched_priority = sched_get_priority_max(SCHED_RR);
82                      if(sched_setscheduler(getpid(), SCHED_RR, &s)==-1)
83                          printf("Fail!\n");
84                  }
85                  else if(sel==2)
86                  {
87                      s.sched_priority = 50;
88                      if(sched_setscheduler(getpid(), SCHED_RR, &s)==-1)
89                          printf("Fail!\n");
90                  }
91                  else if(sel==3)
92                  {
93                      s.sched_priority = sched_get_priority_min(SCHED_RR);
94                      if(sched_setscheduler(getpid(), SCHED_RR, &s)==-1)
95                          printf("Fail!\n");
96                  }
```

```
97                  else if(sel==4)
98                  {
99                      s.sched_priority = sched_get_priority_max(SCHED_FIFO);
100                     if(sched_setscheduler(getpid(), SCHED_FIFO, &s)==-1)
101                         printf("Fail!\n");
102                 }
103                 else if(sel==5)
104                 {
105                     s.sched_priority = 50;
106                     if(sched_setscheduler(getpid(), SCHED_FIFO, &s)==-1)
107                         printf("Fail!\n");
108                 }
109                 else if(sel==6)
110                 {
111                     s.sched_priority = sched_get_priority_min(SCHED_FIFO);
112                     if(sched_setscheduler(getpid(), SCHED_FIFO, &s)==-1)
113                         printf("Fail!\n");
114                 }
115                 else if(sel==7)
116                 {
117                     s.sched_priority = 0;
118                     nice(-20);
119                     if(sched_setscheduler(getpid(), SCHED_OTHER, &s)==-1)
120                         printf("Fail!\n");
121                 }
122                 else if(sel==8)
123                 {
124                     s.sched_priority = 0;
125                     nice(0);
126                     if(sched_setscheduler(getpid(), SCHED_OTHER, &s)==-1)
127                         printf("Fail!\n");
128                 }
```

```
129                 else if(sel==9)
130                 {
131                     s.sched_priority = 0;
132                     nice(19);
133                     if(sched_setscheduler(getpid(), SCHED_OTHER, &s)==-1)
134                         printf("Fail!\n");
135                 }
136 
137                 chdir("temp");
138                 sprintf(name,"%d.txt",i);
139                 f_read = fopen(name, "r");
140                 fscanf(f_read,"%s\n",data); //read content file
141                 fclose(f_read);
142 
143                 exit(0); //end of child porcess
144             }
145         }
146 
147         for(int i=0;i<MAX_PROCESSES;i++)
148             waitpid(IDs[i],&status,0); //wait
149 
150         clock_gettime(CLOCK_MONOTONIC, &end); //check excution time
151 ////////////////////////////////////////////////////////////////
152 
153         printf("Excution time : %.6f\n\n",((double)end.tv_sec + 1.0e-9 * end.tv_nsec)
154          - ((double)start.tv_sec + 1.0e-9 * start.tv_nsec));
155     }
156 }
```

위 코드들은 menu에 맞게 Scheduling이 동작할 수 있도록 작성한 것이며, 처음부터 RR-highest, RR-default, RR-lowest, FIFO-highest, FIFO-default, FIFO-lowest, OTHER-highest, OTHER-default, OTHER-lowest순서데로 CPU scheduling policy가 선택 되어있는 것을 확인할 수 있다. OTHER을 제외한 RR과 FIFO는 우선순위 값을 설정 할 수 있으며 이는 1~99까지 설정이 가능하다. OTHER같은 경우는 nice값을 받게 되는 nice 값이란 **동적 우선순위**를 계산 할 때 사용되는 **것으로 time slice + (20-nice 값)→** 계산이 된다 그러므로 값이 음수인 경우 우선순위가 올라가는 것으로 해석 할 수 있다. 그렇다면 다음 문항에서 각각의 CPU scheduling policy에 대한 결과 값을 확인해보자

(2) Result 2-2

```
os2015722025@ubuntu:~/Desktop/2-2$ make
gcc -o filegen filegen.c
gcc -o schedtest schedtest.c
```

위 명령은 두 파일을 모두 컴파일 하기 위한 명령입니다.

```
os2015722025@ubuntu:~/Desktop/2-2$ make clear
rm -rf tmp*
sync
echo 3 | sudo tee /proc/sys/vm/drop_caches
[sudo] password for os2015722025:
3
```

실험 전 clear를 통해 과제에서 요구하는 사항을 수행합니다.

```
--------Menu--------
1.RR       - highest
2.RR       - default
3.RR       - lowest
4.FIFO     - highest
5.FIFO     - default
6.FIFO     - lowest
7.Standard - highest
8.Standard - default
9.Standard - lowest
10.END PROGRAM
--------------------
Select: 1

RR - highest (1~99), Select 99
Excution time : 0.544615
```

```
Select: 2

RR - default (1~99), Select 50
Excution time : 0.560984
```

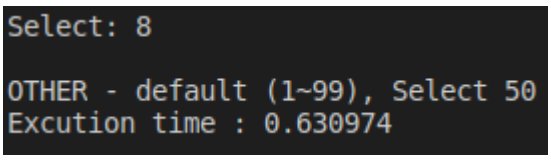
```
Select: 3

RR - lowest (1~99), Select 1
Excution time : 0.560313
```

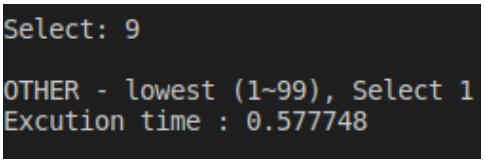
```
Select: 4

FIFO - highest (1~99), Select 99
Excution time : 0.561712
```

```
Select: 5

FIFO - default (1~99), Select 50
Excution time : 0.556140
```

```
Select: 6

FIFO - lowest (1~99), Select 1
Excution time : 0.561035
```

```
Select: 7

OTHER - highest (1~99), Select 99
Excution time : 0.754948
```





해당 결과 값들을 확인하였을 때 각 CPU scheduling policy에 대한 시간차이는 거의 없다고 생각됩니다. 조금 더 명확한 차이를 얻고자 MAX_PROCESSES의 값을 임의로 요구된 10000보다 많은 값을 설정하여 하는 경우 virtual machine자체가 멈추는 현상이 일어나 측정이 불가한 상황이었습니다. 우선 각각의 policy에 대한 설명은 아래와 같습니다.

SCHED_OTHERS: 일반적인 사용자 프로세스에 적용되는 정책으로 time slice와 kernel에 의해 지속적으로 변경되는 동적 우선순위를 사용

SCHED_FIFO: 긴급한 경우 사용되는 정책으로 모든 SCHED_OHTERS 그룹보다 높은 우선순위를 가지고 time slice에 대한 개념이 없습니다. 스스로 CPU를 반납하기 전에는 CPU를 계속 사용합니다.

SCHED_RR: 마찬가지로 SCHED_OTHERS 그룹보다 높은 우선순위를 가지며, FIFO와 다르게 time slice를 갖게 됩니다. Time slice가 다하면 큐의 마지막으로 삽입되며 같은 순위의 프로세스 동작이 끝나면 CPU를 할당 받게 됩니다.

위와 같은 설명을 보았을 때 FIFO와 RR의 경우 OHTERS보다 우선순위를 높게 차지하므로 시간적인 부분에서 적게 소요되는 것으로 예상되지만 실험 결과 자체는 그렇지 않았습니다. 파일을 읽는데 많은 시간이 필요한 것이 아니라 각각의 정책에 대한 시간이 비슷하게 나오게 되었으며, 시간의 차이는 컴퓨터 환경(ex 음악을 틀어 놓는 등)에 따라 다르고 각각 개인의 컴퓨터 사양에 따라 다르기 때문에 많은 영향을 미칠 수 있다고 생각됩니다. 개인적인 소견으로는 CPU정책에 대한 차이를 실험하기 위해서는 각각의 process가 CPU를 좀더 오래 사용하는 프로그램을 동작하는 것이 좋다고 생각됩니다.

3. Reference

[1] 제공된 강의 자료 (2-1, 2-2)

[2] Scheduling에 대한 전반적 개념 & 동적 우선순위 (2-2)

https://palpit.tistory.com/622

[3] Shift 8에 대한 이유 (2-1)

http://ehpub.co.kr/%EB%A6%AC%EB%88%85%EC%8A%A4-%EC%8B%9C%EC%8A%A4%ED%85%9C-%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D-7-8-%ED%94%84%EB%A1%9C%EC%84%B8%EC%8A%A4-%EC%A2%85%EB%A3%8C-%EB%8C%80%EA%B8%B0-%EB%B0%8F/

4. Consideration

해당 과제에서는 thread와 fork의 실행 시간 차이와 Linux에서 지원하는 CPU scheduling 정책에 대한 execution time을 비교하게 되었습니다 2-1과제에 경우 시간차이가 미세하지만 thread에서 효율적으로 나타난 것에 비해 CPU scheduling에 경우 시간차이가 많이 나지 않고 어떠한 정책 자체가 우수하다는 평가를 받지 못하는 결과가 나왔습니다. 이에 위에서도 설명했지만 각각의 정책에 대해 생성되는 process가 CPU를 좀 더 점유 할 수 있는 동작을 구현하는 것이 시간차이를 알아 볼 수 있도록 하는데 더 효율적이라고 생각됩니다. 2-1 실험은 대체로 만족스러웠지만 2-2 실험은 아쉬운 과제였습니다.