운영체제

Assignment3

김태석 교수님

2019202103

이은비

Assignment 3-1

<introduction>

다중 프로세스/쓰레드로 수행하는 프로그램을 작성하는 것으로 numgen.c파일에서 특정 textfile을 파일 입출력을 이용하여 생성 한 뒤, 생성할 프로세스 수(MAX_PROCESSES)의 2배 만큼의 숫자를 기록하게 합니다. 이후에 fork.c와 thread.c 프로그램에서 fork와 exit를 이용하여 부모 프로세스/스레드, 자식 프로세스/스레드에서 textfile안에 있는 숫자를 이용하여 각자 필요한 내용을 수행하도록 합니다.

<conclusion>

실험 전에 pdf에서 주어진 명령어를 수행해서 캐시 및 버퍼를 비워줍니다.

```
os2019202103@ubuntu:~$ rm -rf tmp*
os2019202103@ubuntu:~$ sync
os2019202103@ubuntu:~$ echo 3 | sudo tee /proc/sys/vm/drop_caches
[sudo] password for os2019202103:
3
os2019202103@ubuntu:~$
```

Numgen.c라는 파일 안 MAX_PROCESSES 즉 생성할 프로세스의 개수를 8로 한뒤 생성할 프로세스 개수의 2배의 해당하는 수까지 "./temp.txt"라는 텍스트 파일안에 기록하게 합니다.

```
#include <stdio.h>
#define MAX_PROCESSES 8
int main()
{
FILE *f_write;
FILE *f_write = fopen("./temp.txt","w");
for(int i=0;i<MAX_PROCESSES*2;i++)
{
        fprintf(f_write,"%d",i+1);

}
fclose(f_write);
return 0;
}
~
~
```

fork.c/thread.c 각각 MAXPROCESSES 수만큼 프로세스를 생성합니다.
while문을 사용하여 MAXPROCESSES수가 되기 전까지 while문의 fork();내용을 수행합니다.
최상단 프로세스 2개의 숫자를 읽고 연산을 수행하는 것 은 child()함수 내의 내용 중 fscanf()를 통해 각각의 숫자를 읽고 두개의 숫자를 더한값을 result란 변수에 저장 한 후 그 값을 exit을 통해 부모프로세스로 값을 전달합니다. 부모프로세스에서는 wait(&result);을 통해 값을 받습니다.
수행시간의 측정은 clock_gettime();을 이용하여 측정하기 위해 time.h를 include하고, fork생성 전 &start변수로 저장,wait사용 후 while문이 종료된 후 &end변수로 저장 합니다. 이후 diff에 그 end값에서 start값을 빼고 저장 후 출력합니다.
thread에서는 반복문 사이 pthread_create를 사용하여서 지정된 프로세스 개수만큼 프로세스를 생성합니다. 이후 pthread_create의 인자 중 startfunction인 t_function을 통해 t_function에서 fork.c의 childprocess가 수행한 연산과 같게 합니다. 그리고 종료와 pthread_join을 통해 반환된 값을 받는 과정을 거칩니다.

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>//for exit
#include <time.h>//for clock_gettime();
#define MAX_PROCESSES 8
void child();//function
void parent();
pid_t wait(int *status);
void main(void)
{
        long diff;
        struct timespec begin,end;
        pid_t pid[MAX_PROCESSES];//for make processes
        int runProcess = 0;
        int state;
        int result=0;
        int var=0;
clock_gettime(CLOCK_MONOTONIC,&begin);
while(runProcess<MAX_PROCESSES)
{
         wait(&state);
        pid[runProcess] = fork();//fork();

if(pid[runProcess]<0)//error case
{
 return -1;
}
else if(pid[runProcess] == 0)//child process
{
        child();
}
else//parent process
{
        parent();
}
runProcess++;
}
clock_gettime(CLOCK_MONOTONIC,&end);
diff = (end.tv_sec-begin.tv_sec);
printf("value of fork:%d\n%ld\n",var,diff);

}

void child()
{
        FILE *fp1;//flie pointer for read
        int n1=0;
        int n2=0;
        int result=0;
        fp1 = fopen("./temp.txt","r");
        int i;
        for(i=0;i<MAX_PROCESSES;i++)
        {
                fscanf(fp1,"%d\n%d\n",&n1,&n2);
                result= n1+n2;
        sleep(1);
        }
        exit(result);
}
int parent()
{
        wait(&result);
        int i;
        for(i=0;i<MAX_PROCESSES;i++)
        {       var+=result;
        }
        return = var;
}
```

(fork.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>
#include <process.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <linuxunistd.h>
Files
PROCESSES 8

void *t_function(void *n)//thread function
{
        int thread_num = (int)n;
        FILE *fp1;//file pointer for read
        int n1,n2=0;
        int result = 0;
        fp1 = fopen("./temp.txt","r");//file open
        fscanf(fp1,"%d\n",&n1);
        result += n1;
        Sleep(1000*(NUMTHREAD-thread_num));
        exit(result);
}
void main()
        int thread_num = (int)n;
        int* thr_id;
        int status;
        int p;

    for(int i=0;i<MAX_PROCESSES;i++)
{
    thr_id = pthread_create(&p_thread[i], NULL, t_function, (void *)p);
    if (thr_id < 0)
    {
        perror("thread create error : ");
        exit(0);
    }

    // wait
    pthread_join(p_thread[i], (void **)&status);
}
wait(&result)
}
printf("value of thread : %d\n%d\n",status,result);
}
~
~
~
~
```

(thread.c)

<(exit할 때 반환값이 2^8이 아닌 이유)>

linux에서 자식이 exit로 종료될 때의 상태정보는 정상종료일 경우 하위8bits는 0,상위 8 bits는 exit status 즉 자식 프로세스가 exit 명령으로 전달 한 값이 위치하게 됩니다. 또한 signal로 종료될때는 하위 8bits는 signal번호, 상위 8bits는 0으로 위치 합니다. 즉 반환 값인 exitstatus는 8bit 이내, 나머지는 0으로 채워져야하므로 반환값은 8bits이상으로 배치 할 수 없고 따라서 반환 값이 2^8이상일 수 없습니다.

<references>

Clock_gettime()에 대한 예제/https://exeter.tistory.com/86

Multi processor 생성/https://rightg.tistory.com/78

Assignment 3-2

<introduction>

Filegen.c라는 파일을 만들어 특정 디렉토리에 파일 입출력을 통해 textfile에 무작위의 int형 양수를 기록합니다. 이후 assignment3-1에서의 fork.c의 작성 내용 중 각 프로세스에서 sched_setscheduler()를 사용하여 CPU 스케쥴링 정책을 변경 하여 schedtest.c라는 파일의 내용을 작성 한 뒤 2개를 같이 compile합니다.

<conclusion>

새로운 디렉터리를 만든 후 그 안에서 filegen.c,schedtest.c와 Makefile을 생성합니다.

```
os2019202103@ubuntu:~$ ls ./temp
filegen.c  filegen.o  Makefile  schedtest.c  schedtest.o
os2019202103@ubuntu:~$
```

Filegen.c에서는 성능을 비교할 수 있을 정도의 프로세스를 생성하여 실험하기 위해 생성할 프로세스의 개수인 MAX_PROCESSES 를 10000으로 하여 9이하의 무작위의 양수를 "./temp.txt"에 write하게 합니다.

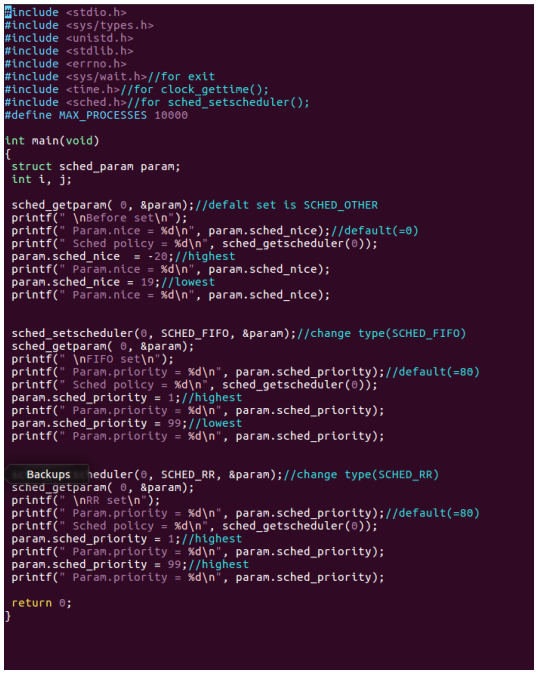
```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

#define MAX_PROCESSES 10000

int main()
{

int i;
FILE *f_write;
for(i=0;i<MAX_PROCESSES;i++)
{
        FILE *f_write = fopen("./temp.txt","w");
        fprintf(f_write," %d", 1+rand()%9);
}
fclose(f_write);
return 0;
}
```

이후 ps를 통해 real time인 SCHED_FIFO(a first-in,first-out policy)와 SCHED_RR(a round-robin policy)의 priority를 확인 한 후, schedtest.c에서 linux가 지원하는 3가지 스케쥴링, 위의 두가지(SCHED_FIFO,SCHED_RR)policy와 SCHED_OTHER(the standard round-robin time-sharing policy)을 각각 highest,default,lowest의 priority/nice값을 설정하고 출력합니다.

Ps -l을 이용하여 PRI(priority)의 default 값은 80, NI의 default값은 0임을 알았습니다.

```
os2019202103@ubuntu:~/temp$ ps -l
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   2206   2177  0  80   0 -  7377 wait   pts/2    00:00:00 bash
0 R  1000   6786   2206  0  80   0 -  8996 -      pts/2    00:00:00 ps
```

이후 자료조사 하여 realtime policy에서 highest 값과 lowest값은 각각1과 99이며
Nonrealtime policy에서 우선순위를 비교하는 nice 의 highest 값과 lowest값은 각각-20,+19임을 알아냈습니다. 따라서 이후 schedtest.c에서 sched_setscheduler()를 통해 policy를 변경하고, param.sched_nice,param.sched_priority를 default,highest,lowest value로 초기화하였습니다.



```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>//for exit
#include <time.h>//for clock_gettime();
#include <sched.h>//for sched_setscheduler();
#define MAX_PROCESSES 10000

int main(void)
{
 struct sched_param param;
 int i, j;

 sched_getparam( 0, &param);//defalt set is SCHED_OTHER
 printf(" \nBefore set\n");
 printf(" Param.nice = %d\n", param.sched_nice);//default(=0)
 printf(" Sched policy = %d\n", sched_getscheduler(0));
 param.sched_nice  = -20;//highest
 printf(" Param.nice = %d\n", param.sched_nice);
 param.sched_nice = 19;//lowest
 printf(" Param.nice = %d\n", param.sched_nice);


 sched_setscheduler(0, SCHED_FIFO, &param);//change type(SCHED_FIFO)
 sched_getparam( 0, &param);
 printf(" \nFIFO set\n");
 printf(" Param.priority = %d\n", param.sched_priority);//default(=80)
 printf(" Sched policy = %d\n", sched_getscheduler(0));
 param.sched_priority = 1;//highest
 printf(" Param.priority = %d\n", param.sched_priority);
 param.sched_priority = 99;//lowest
 printf(" Param.priority = %d\n", param.sched_priority);


  Backups eduler(0, SCHED_RR, &param);//change type(SCHED_RR)
 sched_getparam( 0, &param);
 printf(" \nRR set\n");
 printf(" Param.priority = %d\n", param.sched_priority);//default(=80)
 printf(" Sched policy = %d\n", sched_getscheduler(0));
 param.sched_priority = 1;//highest
 printf(" Param.priority = %d\n", param.sched_priority);
 param.sched_priority = 99;//highest
 printf(" Param.priority = %d\n", param.sched_priority);

 return 0;
}
```

<각 cpu스케쥴링 별 lowest,default,highest 숫자 지정 이유>

SCHED_FIFO는 우선순위 real-time 스케줄링 방법으로, 1-99 까지의 priority 를 가집니다. 낮은 숫자일 수록 높은 priority를 가지므로 1이가장 높은 priority임을 알 수 있습니다. SCHED_OTHER과 같이 time-sharing 방식이 아니 여서 priority 가 높은 process 가 항상 선점하는 방식이며 SCHED_FIFO1은 항상 SCHED_OTHER의 어떤 값보다 먼저 scheduled됩니다.

SCHED_RR는 SCHED_FIFO와 같이 우선순위 real-time 스케줄링 방법으로, 1-99 까지의 priority를 가지며 낮은 숫자일수록 높은 priority를 가집니다.

SCHED_OTHER는 위의 두 가지 policy와 다르게 non-realtime스케쥴링 방법이며 linux의 default time-sharing 스케줄링 방법입니다. 즉 FIFO나 RR 을 수동으로 지정해 주지 않는다면, User가 생성하는 모든Process 는 모두 이 방법으로 스케줄링 됩니다.
해당 프로세스들은 nice or setpriority() 시스템콜을 통해, 동적으로 우선순위 조절이 가능하며. 기본적인 스케줄링 알고리즘은 time-shared Round Robin policy입니다.

<references>

리눅스 프로세스 스케쥴링 API/ https://blog.daum.net/tlos6733/115

Assignment 3-3

<introduction>

pid를 바탕으로 프로세스 이름,현재프로세스의 상태,프로세스 그룹 정보등의 프로세스 정보를 출력하는 module을 작성합니다.

<conclusion>

3-1,3-2compile완료하지 못해 이후 단계 진행이 어려웠습니다.