Operating Systems : Term Project 1

2019311037 최지원

Round Robin Scheduler 개요

프로그램 구현 내용 및 처리 과정

1. 자식 프로세스 10개 생성, 이때 CPU BurstTime과 IO BurstTime을 랜덤으로 생성함.
2. clock()으로 Scheduler를 부르는 Timer구현
3. 자식 프로세스는 msgrcv로 대기후 msg 오면 요청에 따라 Quantum만큼 CPU Burst 혹은 IO Burst를 수행한다.
4. 스케쥴러는 최초 실행시에 Run Queue와 Wait Queue를 생성함. 각 Queue는 Circular Queue의 Static Queue로 만들어짐 (공룡책에 Circular Queue로 주로 구현한다고 되어있어서 구현해보았습니다.)
5. 두번째 부터는 Run Queue와 Wait Queue를 체크해 한 Queue가 비어있다면 비어있지 않은 다른 Queue에서 dequeue하고, 둘다 비어있지 않다면 random으로 Run Queue혹은 Wait Queue를 선택하여 dequeue 함.
6. Dequeue된 프로세서에 IPC 메세지를 보낸다. 이때, 해당 프로세서가 CPU Burst를 해야하는지 IO Burst 를 해야하는지를 전송하고, 만약 Burst Time이 Quantum 보다 작다면 clock_t의 값을 수정하여 해당 Time Tick의 크기를 남은 Burst Time 만큼으로 조정한다.
7. Burst 가 끝난 프로세서는 랜덤으로 Run Queue와 Wait Queue중 한 Queue에 들어간다. 이때 CPU Burst 와 IO Burst 가 둘다 0이라면 다시 enqueue하지 않는다.
8. Run Queue와 Wait Queue가 모두 비었다면 프로그램을 종료한다.

미구현 내용 및 이상한점

1. Priority Queue를 구현하려고 Priority 정보를 만들었으나 시간이 부족하여 다 구현하지 못함. 시간이 며칠만 더 있었으면 구현할 수 있을 것 같음.
2. 프로그램을 그냥 실행하면 Scheduler - Fetch가 순서대로 잘 작동 되었으나, Shell 명령어를 달아서 출력 내용을 기록하니까 순서대로 표시되지 않았음. 다만 작동은 정상적으로 되는 것으로 보아 File 출력에 시간이 소모되어 Display만 꼬인것으로 보임. (scheduler_dump.txt에 그렇게 나타나 있음.)

Burst Time과 Callback Time 조정

Burst Time과 Callback Time 조정에 따라서 프로그램에 오류가 발생하는지 살펴 보았지만, Time Quantum을 1ms, Callback Time 을 2ms 정도까지 낮추어도 오류가 생기지 않았다. 나노초까지는 실험해보지 못하였다.

Round Robin Scheduler 코드

main.c

```
#include "main.h"
#include "queue.c"
#include "process.c"

pid_t pids[PROCESS_NUM];
process * processData[PROCESS_NUM];
struct sigaction sa;
struct itimerval timerMain, timerLast;
clock_t tickfront, tickrear;
int msgid;


void child(process * pd){
    msgSet tmpMsg;
    pid_t childPID = getpid();
    while(1){
        if(msgrcv(msgid, &tmpMsg, sizeof(msgSet), childPID, 0) != -1){
            printf("\n\n=================================================================");
            printf("\n                    Process %d Fetched!",childPID);
            printf("\n-----------------------------------------------------------------");
            printf("\n                         Remain Burst\n");
            printf("\n             CPU : %d                IO : %d",pd->cpuBurst, pd->ioBurst);
            printf("\n=================================================================");

            if(tmpMsg.mode == WAITING_CPU){
                if(pd->cpuBurst > QUANTUM) pd->cpuBurst -= QUANTUM;
                else pd->cpuBurst = 0;
            }
            else if(tmpMsg.mode  == WATING_IO){
```

main.h

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include <sys/msg.h>
#include <sys/signal.h>
#include <string.h>
#include <sys/ipc.h>

#define QUANTUM 1
#define PROCESS_NUM 10
#define BURSTSIZE 300
#define CALLBACK_TIME 2

#define WAITING_CPU 1
#define WATING_IO 2
#define BURST_CPU 3
#define BURST_IO 4
#define WORK_END 5
typedef struct _process{
    pid_t pid;
    int cpuBurst;
    int ioBurst;
    int status;
    int priority;
```

```
                if(pd->ioBurst > QUANTUM) pd->ioBurst -= QUANTUM;
                else pd->ioBurst = 0;
            }


        }
    }
}

void burst(process * runningProcess, circularQueue * waitQ, circularQueue * runQ, int count){
    printf("\n\n=================================================================");
    if(runningProcess->status == WATING_IO){
        printf("\n             BURSTING IO || QUAMTUM NUMBER %d",count);
    }
    else if(runningProcess->status == WAITING_CPU){
        printf("\n             BURSTING CPU || QUAMTUM NUMBER %d",count);
    }

    printf("\n-----------------------------------------------------------------");
    printf("\n                          Queue Status\n");
    printf("\n runQ(%d)  :",runQ->num);
    traverseQ(runQ);
    printf(" waitQ(%d) :",waitQ->num);
    traverseQ(waitQ);
    printf("=================================================================");
    msgSet workMsg;
    workMsg.msgType = runningProcess->pid;
    workMsg.process = runningProcess;
    workMsg.mode = runningProcess->status;

    if(runningProcess->status == WATING_IO){
        if(runningProcess->ioBurst < QUANTUM){
            tickrear = tickfront + runningProcess->ioBurst;
            runningProcess->ioBurst = 0;
        }
        else{
            runningProcess->ioBurst -= QUANTUM;
        }
    }
    else if(runningProcess->status == WAITING_CPU){
        if(runningProcess->cpuBurst < QUANTUM){
            tickrear = tickfront + runningProcess->cpuBurst;
            runningProcess->cpuBurst = 0;
        }
        else{
            runningProcess->cpuBurst -= QUANTUM;
        }
    }

    if(msgsnd(msgid, &workMsg, sizeof(msgSet), 0) == -1){
        printf("\nScheduling message send fail\n");
    }
}

int scheduling(){
    static int count = 0;
    static circularQueue * waitQ;
    static circularQueue * runQ;
    process * runningProcess;
    srand(time(NULL));
    if(count == 0){
        printf("\n=== Scheduling Start ===\n\n");
        waitQ = makeQ();
        runQ = makeQ();
        for(int i=0; i<PROCESS_NUM; i++){
            addQ(makeP(pids[i]),runQ);
        }

    }
    else{
        if(isEmpty(waitQ)){
            runningProcess = popQ(runQ);
            burst(runningProcess, waitQ, runQ, count);
            int tmpRand = rand();
            if(runningProcess->cpuBurst != 0 && runningProcess->ioBurst != 0){
                if(tmpRand%2 == 0){
                    runningProcess->status = WAITING_CPU;
                    addQ(runningProcess,runQ);
                }
                else{
                    runningProcess->status = WATING_IO;
                    addQ(runningProcess, waitQ);
                }
            }
            else if(runningProcess->cpuBurst == 0 && runningProcess->ioBurst != 0){
                runningProcess->status = WATING_IO;
                addQ(runningProcess, waitQ);
            }
            else if(runningProcess->cpuBurst != 0 && runningProcess->ioBurst == 0){
                runningProcess->status = WAITING_CPU;
                addQ(runningProcess, runQ);
            }
            else if(runningProcess->cpuBurst == 0 && runningProcess->ioBurst == 0){

            }
        }
        else if(isEmpty(runQ)){
            runningProcess = popQ(waitQ);
            burst(runningProcess, waitQ, runQ, count);
            int tmpRand = rand();
```

```
} process;

typedef struct _circularQueue{
    int front;
    int rear;
    int max;
    int num;
    process * queue[PROCESS_NUM];
} circularQueue;
typedef struct _msgSet{
    long msgType;
    int mode;
    process * process;
} msgSet;


void child(process * pd);
```

queue.c

```
circularQueue * makeQ(){
    circularQueue * tmpQ = (circularQueue *)malloc(sizeof(circu
    tmpQ->front = -1;
    tmpQ ->rear = -1;
    tmpQ -> max = PROCESS_NUM+1;
    tmpQ->num = 0;
    return tmpQ;
}

void addQ(process * p, circularQueue * q){
        q->rear = (q->rear + 1)%(q->max);
        q->num++;
        q->queue[q->rear] = p;

}

process * popQ(circularQueue * q){
    q->front = (q->front +1)%q->max;
    q->num--;
    process * tmp = q->queue[q->front];
    q->queue[q->front] = NULL;
    return tmp;
}

void traverseQ(circularQueue * q){
    for (int i=0; i < q->num; i++){
        printf(" %d",q->queue[(q->front +1 +i)%q->max]->pid);
    }
    printf("\n");
}

int isEmpty(circularQueue * q){
    if(q->front == q->rear){
        return 1;
    }
    else{
        return 0;
    }
}
```

process.c

```
process * makeP(pid_t tmppid){
    process * tmpP = (process *)malloc(sizeof(process));
    tmpP -> pid = tmppid;
    tmpP->cpuBurst = (rand() % BURSTSIZE) + BURSTSIZE;
    tmpP->ioBurst = (rand() % BURSTSIZE) + BURSTSIZE;
    tmpP->status = WAITING_CPU;
    tmpP->priority = rand()%3;
    return tmpP;
}
```

```
            if(runningProcess->cpuBurst != 0 && runningProcess->ioBurst != 0){
                if(tmpRand%2 == 0){
                    runningProcess->status = WAITING_CPU;
                    addQ(runningProcess,runQ);
                }
                else{
                    runningProcess->status = WATING_IO;
                    addQ(runningProcess, waitQ);
                }
            }
            else if(runningProcess->cpuBurst == 0 && runningProcess->ioBurst != 0){
                runningProcess->status = WATING_IO;
                addQ(runningProcess, waitQ);
            }
            else if(runningProcess->cpuBurst != 0 && runningProcess->ioBurst == 0){
                runningProcess->status = WAITING_CPU;
                addQ(runningProcess, runQ);
            }
            else if(runningProcess->cpuBurst == 0 && runningProcess->ioBurst == 0){

            }
        }
        else{
            int tmpRand1 = rand();
            if(tmpRand1%2 == 0){
                runningProcess = popQ(runQ);
                runningProcess->status = WAITING_CPU;
            }
            else{
                runningProcess = popQ(waitQ);
                runningProcess->status = WATING_IO;
            }
            burst(runningProcess, waitQ, runQ, count);

            if(runningProcess->cpuBurst == 0 && runningProcess->ioBurst != 0){
                runningProcess->status = WATING_IO;
                addQ(runningProcess, waitQ);
            }
            else if(runningProcess->cpuBurst != 0 && runningProcess->ioBurst == 0){
                runningProcess->status = WAITING_CPU;
                addQ(runningProcess, runQ);
            }
            else if(runningProcess->cpuBurst != 0 && runningProcess->ioBurst != 0){
                int tmpRand2 = rand();
                if(tmpRand2%2 == 0){
                    runningProcess->status = WAITING_CPU;
                    addQ(runningProcess, runQ);
                }
                else{
                    runningProcess->status = WATING_IO;
                    addQ(runningProcess, waitQ);
                }
            }
            else if(runningProcess->cpuBurst == 0 && runningProcess->ioBurst == 0){

            }
        }

    }
    count++;
    if(isEmpty(runQ) && isEmpty(waitQ)){
        return 1;
    }
    else{
        return 0;
    }
}


int main(){
    key_t key = ftok("~/os", 65);
    msgid = msgget(key, IPC_CREAT | 0666);
    if(msgid >0){
        printf("Message Queue Created! %d\n",msgid);
    }
    else{
        printf("Message Queue Create Failed :(\n");
    }
    int i=0;
    srand(time(NULL));
    for (i=0; i<PROCESS_NUM; i++){
        pids[i] = fork();
        if(pids[i] <0){
            printf("fork ERROR!\n");
        }
        processData[i] = makeP(pids[i]);
        if(pids[i] == 0){
            child(processData[i]);
            exit(0);
        }
    }


    tickfront = clock();
    int end = 0;
    while(!end){
        tickrear = clock();
        if((tickrear-tickfront)>=QUANTUM*1000){
            tickfront = tickrear;
```

```
            end = scheduling();
        }
        usleep(CALLBACK_TIME);
    }
    printf("\n\nALL PROCESSES ENDED!\n\n");
    return 0;

}
```

Round Robin Scheduler Dump

```
================================================================
             BURSTING CPU || QUAMTUM NUMBER 187
----------------------------------------------------------------
                         Queue Status

 runQ(8)  : 5712 5713 5714 5705 5706 5707 5708 5710
 waitQ(1) : 5709
================================================================

================================================================
                    Process 5711 Fetched!
----------------------------------------------------------------
                         Remain Burst

             CPU : 509                IO : 372
================================================================

================================================================
             BURSTING CPU || QUAMTUM NUMBER 188
----------------------------------------------------------------
                         Queue Status

 runQ(8)  : 5713 5714 5705 5706 5707 5708 5710 5711
 waitQ(1) : 5709
================================================================
```