Operating System (MS)

project #1: Simple scheduling (Round-Robin)

모바일시스템공학과 김태경 (32211203)

목차

Freeday used 3
Freeday left 2

2024.11.13

1. 프로젝트 소개

1.1 프로젝트 개요

본 프로젝트는 운영체제의 스케줄링 정책을 이해하고 구현하기 위해, Round-Robin 스케줄링 알고리즘을 사용하여 프로세스 스케줄링 시뮬레이션을 수행하는 과제입니다. 부모 프로세스는 10개의 자식 프로세스를 생성하고, 각 프로세스는 CPU와 I/O 작업을 반복적으로 수행합니다. 이를 통해 스케줄링 알고리즘의 동작 방식과 프로세스 상태 변화를 확인할 수 있습니다. 자식 프로세스는 CPU-burst와 I/O-burst의 값을 동적으로 관리하며, I/O 대기 중인 프로세스와 실행 가능한 프로세스를 각기 다른 큐로 관리합니다.

1.2 프로젝트 목표

본 프로젝트의 목표는 Round-Robin 스케줄링 알고리즘을 구현하여 각 프로세스의 자원을 균등하게 배분하고, I/O 대기와 CPU 실행이 번갈아 이루어지는 환경을 시뮬레이션하는 것입니다. 이를 통해 스케줄링 방식에 따라 프로세스 상태를 관리하고, 메시지 큐를 통한 IPC 메시지 전송을 이용하여 프로세스 간의 통신을 구현합니다. 궁극적으로 CPU와 I/O의 효율적인 스케줄링을 통해 시스템 자원의 사용과 성능에 대한 이해를 높이는 것을 목표로 합니다.

2. 프로젝트 요구사항

1) **프로세스 생성 및 Round-Robin 스케줄링 설정**
   - 부모 프로세스는 10개의 자식 프로세스를 설정
   - 자식 프로세스에 time slice를 할당하여 CPU 시간을 균등하게 배분
2) **스케줄링 큐 관리**
   - Run-queue & Wait-queue를 사용해 프로세스 상태를 관리
3) **자식 프로세스의 CPU & I/O 작업 수행**
   - CPU-burst와 I/O-burst 값을 기반으로 작업 수행
   - setitimer 시스템 호출으로 ALARM 신호를 전송
4) **IPC Message Queue 구현**
   - msgget, msgsnd, msgrcv 시스템 호출로 프로세스 간의 통신 구현
   - IPC_NOWAIT 옵션 flag를 사용하여 비동기 방식으로 메시지 수신
5) **출력 및 상태 기록**
   - schedule_dump.txt 파일에 Run & Wait-queue의 상태를 기록
   - 각 자식 프로세스는 최소 1분 동안 실행 및 스케줄링은 0 ~ 10000틱 동안 기록
6) **I/O 처리기능 (Optional)**
   - I/O 요청에 따라 Queue의 프로세스 이동
   - ARALM 신호를 기준으로 I/O 작업 시간 관리

3. 프로젝트에 사용된 개념

3.1 제공된 Sample Code 분석

3.1.1 **msg.h 코드 분석**

: 부모 프로세스와 자식 프로세스 간의 메시지 통신을 위해 설계된 구조체입니다. 이 구조체는 메시지 큐를 통해 데이터 전달에 필요한 정보를 담고 있으며, 자식 프로세스가 I/O 작업을 요청하거나 부모 프로세스가 자식 프로세스에 특정 명령이나 데이터를 전달하는 데 사용됩니다.

구조체 분석

- mytpye : 부모 프로세스가 메시지를 수신할 때 메시지 종류를 구분하기 위함
- pid : 자식 프로세스의 pid를 의미, 부모 프로세스가 자식 프로세스의 식별에 사용
- io_time : 자식 프로세스가 I/O 작업을 요청할 때 필요한 대기시간을 나타냄

3.1.2 **msgq.c 코드 분석**

: 메시지 큐를 생성하고, 메시지 큐에 메시지를 보내는 역할을 수행하는 프로그램입니다. 이 과제에서는 자식 프로세스가 CPU 작업(CPU-burst)을 마친 후 I/O 작업을 요청할 때 사용됩니다.

- 메시지 Queue 생성 또는 접근

```
int key = 0x12345;
msgq = msgget( key, IPC_CREAT | 0666);
printf("msgq id: %d\n", msgq);
```

: key는 고유 키 값 0x12345로 고정되며, 다른 프로세스가 접근할 수 있습니다. msgget 함수를 사용하여 메시지 큐를 생성하거나 이미 존재하는 큐를 엽니다. IPC_CREAT 플래그는 큐가 존재하지 않을 때 생성하며, 0666은 큐의 읽기/쓰기 권한을 설정합니다.

- 메시지 구조체 초기화

```
struct my_msgbuf msg;
memset(&msg, 0, sizeof(msg));
```

: msg 구조체를 초기화하여, garbage value를 방지합니다.

- 메시지 필드 설정 (메세지 구조체 설정)

```
msg.mtype = 1;
msg.pid = getpid();
msg.io_time = 10;
```

: 메시지의 타입(mtype)을 1로 설정하고, 현재 프로세스의 PID를 설정하여 부모 프로세스가 특정 자식 프로세스를 식별할 수 있게합니다. I/O 대기시간(io_time)을 10으로 설정합니다.

- 메시지 전송

```
ret = msgsnd(msgq, &msg, sizeof(msg) - sizeof(long), 0);
printf("msgsnd ret: %d\n", ret);
```

: msgsnd 함수를 사용하여 msgq에 메시지를 보냅니다. 시스템에 따라 메시지 헤더 크기가 달라질 수 있기에 sizeof를 사용하여 수정하고, 전송에 성공하면 0을, 실패 시 -1을 반환합니다.

3.1.3 msgrcv.c 코드 분석

: 메시지 큐에서 메시지를 수신하는 역할을 하며, 내용을 출력한 이후 메시지 큐를 삭제하는 역할을 합니다. 부모 프로세스는 메시지를 통해 자식 프로세스의 상태를 파악하고, I/O 대기상태와 CPU 스케줄링을 효율적으로 관리할 수 있습니다.

- **메시지 큐 생성 또는 접근**

```
msgq = msgget( key, IPC_CREAT | 0666);
printf("msgq id: %d\n", msgq);
```

: msgq.c의 메시지 Queue 생성 또는 접근과 같은 역할을 수행함

- **메시지 수신**

```
ret = msgrcv(msgq, &msg, sizeof(msg) - sizeof(long), 0, 0);
printf("msgsnd ret: %d\n", ret);
```

: 메시지 큐의 ID(msgq)로 메시지의 수신할 때 참조할 큐를 지정합니다. &msg는 수신된 메시지를 저장할 버퍼의 주소, msgtype을 0으로 설정하여 가장 먼저 도착한 메시지를 수신하고, msgflg를 0으로 설정하여 수신된 메시지가 없을 경우 차단 모드로 대기합니다.

- **수신된 메시지 내용 출력**

```
printf("msg.mtype: %ld\n", msg.mtype);
printf("msg.pid: %d\n", msg.pid);
printf("msg.io_time: %d\n", msg.io_time);
```

: 메시지 큐를 통해 수신된 메시지의 내용을 확인함(mtype, pid, io_time)

- **메세지 Queue 삭제**

```
if (msgctl(msgq, IPC_RMID, NULL) == -1) {
    perror("msgctl failed");
    return 1;
}
```

: 메세지 큐에 대한 다양한 제어작업을 수행하기 위한 함수 msgct1을 사용하여 IPC_RMID 플래그의 값에 따라 메시지 큐(msgq)를 삭제합니다.

3.1.4 sched.c 코드 분석

: 스케줄링의 기능을 구현하기 위한 코드의 틀을 제공하며, 과제에서는 Round-Robin을 구현할 예정입니다. 위 코드에서 스레드의 생성 및 관리 기능과, 스레드의 우선순위 설정, 스레드의 종료 및 정리 기능을 구현하여야 합니다.

3.1.5 signal.c 코드 분석

: 부모 프로세스가 주기적인 타이머 틱을 발생시키고, 스케줄링 작업을 수행할 수 있게끔 타이머 설정 방법의 예시를 보여주는 샘플 코드입니다. 과제에서 요구하는 Round-Robin 스케줄링의 time quantum을 관리하는데 사용할 수 있습니다.

- **신호 관련 기능 구현**

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>

void signal_handler(int signo);
int count = 0;
```

: <signal.h>으로 sigaction함수와 SIGALRM 상수 등 신호 관련 기능을 사용할 수 있도록 하고, itimerval 구조체와 setitimer 함수를 사용하기 위한 <sys/time.h> 헤더파일을 추가합니다.

- **main 함수**

```
struct sigaction old_sa;
struct sigaction new_sa;
memset(&new_sa, 0, sizeof(new_sa));
new_sa.sa_handler = &signal_handler;
sigaction(SIGALRM, &new_sa, &old_sa);
```

: sigaction 설정 : <signal.h> 헤더파일의 sigaction 구조체를 new_sa로 생성하고, SIGALRM 신호가 발생할 때 signal_handler가 호출되도록 설정하고, sigaction 함수를 사용하여 기존의 핸들러 설정은 old_sa에, 새로운 핸들러는 new_sa로 설정힙니다.

```
struct itimerval new_itimer, old_itimer;
new_itimer.it_interval.tv_sec = 1;
new_itimer.it_interval.tv_usec = 0;
new_itimer.it_value.tv_sec = 1;
new_itimer.it_value.tv_usec = 0;
setitimer(ITIMER_REAL, &new_itimer, &old_itimer);
```

: 타이머 설정 : itimerval 구조체에 주기적인 타이머 설정을 저장합니다. new_itimer.it_interval는 타이머의 반복 간격을, new_itimer.it_value는 타이머의 초기값을 설정하는 변수이며, setitimer 함수를 통해 ITIMER_REAL이 1초마다 SIGALAM을 발생하도록 타이머를 설정합니다.

```
while (1);
return 0;
```

: 무한 루프 : 프로그램이 종료되지 않고 계속해서 신호를 받도록 대기합니다.

- **신호 핸들러 함수(signal_handler)**

```
void signal_handler(int signo)
{
    printf("signaled! %d \n", signo);
    count++;

    if (count == 3) exit(0);
}
```

: SIGALRM이 발생할 때 마다 호출되는 함수로, 위의 예제에서는 신호가 3번 발생할 경우 프로그램을 종료하도록 설정하였습니다.

3.2 주요 개념 정리

3.2.1 Round-Robin 스케쥴링 (RR)

운영체제의 CPU 스케쥴링 알고리즘 중 하나로, 각 프로세스에 동일한 시간할당량(time quantum)을 주고 순차적으로 처리하는 방식입니다.

각 프로세스는 주어진 시간 동안만 CPU를 점유할 수 있고, 주어진 시간 할당량이 끝나면 다음 프로세스에 CPU를 할당합니다. 주어진 시간 내에 작업을 하지 못한 경우에는 Context Switching 오버헤드가 발생하여 작업이 미뤄집니다.

이 스케쥴링 기법은 모든 프로세스가 CPU에 공평하게 접근할 수 있어 공정성을 보장하는 스케줄링 기법으로, 시스템의 응답시간을 개선하는데 효과적입니다.

3.2.2 IPC(Inter-Process Communication) & 메시지 Queue

IPC는 프로세스간의 통신으로, 서로 독립된 프로세스들 간의 데이터를 주고받는 방법을 뜻합니다.

메시지 큐는 IPC의 한 방법으로, 운영체제가 제공하는 큐에 메시지를 넣고 빼는 방식으로 통신을 합니다. 메시지의 전송과 수신에서 송신자는 메시지 발송 후 대기하지 않고, 수신자는 필요한 때에 메시지를 받을 수 있는 비동기적 통신을 지원합니다. 메시지들은 선입선출의 방식을 따라 메시지 큐에 저장되며, msgget, msgsnd, msgrcv 등의 시스템 호출로 큐의 생성과 메시지의 송-수신을 실행할 수 있습니다.

3.2.3 CPU-burst & I/O-burst

CPU-burst는 프로세스가 CPU에서 연속적으로 실행되는 구간(시간)입니다. 이 시간 동안 프로세스는 작업을 수행하며, 다른 자원에 접근하지 않습니다.

I/O-burst는 프로세스가 입출력 작업을 수행하는 구간(시간)입니다. 이 시간 동안 프로세스는 CPU를 사용하지 않고, I/O작업이 완료될 때까지 대기상태를 유지합니다.

프로세스는 보통 CPU-burst와 I/O-burst를 번갈아 가면서 수행하며, CPU 작업이 끝나면 입출력 작업을 수행하고, 다시 CPU 작업을 진행하는 과정을 반복합니다.

3.2.4 신호 (Signal)

프로세스에 특정 이벤트가 발생했음을 알리기 위한 비동지적 인터럽트로, 프로세스가 특정 작업을 수행 중일 때 발생하거나 하지 않을 수도 있습니다. signal의 발생에 따라 특정 작업을 수행하도록 사용하는 flag와 비슷한 개념입니다.

* 비동기적인 이벤트인 신호를 프로세스가 즉시 처리할 수 있도록 신호 핸들러(Signal Handler)를 설정할 수 있습니다.

주요 Signal 종류

- SIGINT: 키보드 인터럽트(Ctrl + C)를 의미하며, 기본 동작은 프로그램 종료이다.
- SIGKILL: 프로세스를 강제 종료하는 신호로, 프로세스가 무조건 종료한다.
- SIGTERM: 종료 요청 신호로, 프로그램이 종료되도록 요청한다. SIGKILL과 달리, 프로세스가 종료 동작을 정의할 수 있다.
- SIGALRM: 타이머 만료 시 발생하는 신호이다. setitimer 함수를 통해 설정할 수 있으며, 주기적인 작업에 활용할 수 있다.

3.2.5 타이머 (Timer)

타이머(Timer)는 주기적인 신호의 발생 또는 특정 시간 이후에 신호가 발생하도록 설정할 수 있는 기능이다.

setitimer 함수는 특정 간격으로 SIGALRM 신호를 주기적으로 발생시키는 타이머를 설정하는 시스템 호출입니다. setitimer를 사용하면 타이머 틱마다 프로세스에 SIGALRM 신호가 전달되고, 이를 통해 주기적인 작업을 구현할 수 있습니다.

setitimer 함수 사용 방법 (매개변수)

```
int setitimer(int which, const struct itimerval *new_value, struct itimerval *old_value);
```

- which: 타이머 유형을 지정 (주로 ITIMER_REAL)
  - *ITIMER_REAL: 실제 시간을 기준으로 동작, 만료 시 SIGALRM 신호를 발생
  - *ITIMER_VIRTUAL: 프로세스가 CPU를 사용할 때만 동작, 만료 시 SIGVTALRM 신호를 발생
  - *ITIMER_PROF: 프로세스와 커널이 CPU를 사용할 때 동작, 만료 시 SIGPROF 신호를 발생
- new_value: 타이머 설정을 위한 값. itimerval 구조체로 설정. 타이머 간격(it_interval)과 초기 만료 시간(it_value)을 지정
- old_value: 이전 타이머 설정을 저장할 변수 (필요하지 않다면 NULL로 지정)

itimerval 구조체

: 타이머가 만료되는 시점과 반복 간격을 지정하는데 사용됩니다. 이 구조체를 통해 첫 타이머 만료와 이후 반복 간격을 각각 설정할 수 있습니다. (구조체 구성 ->)

```
struct itimerval {
    struct timeval it_interval;
    struct timeval it_value;
};
```

- it_interval : 타이머가 첫 번째 만료 이후 반복될 주기를 정의하고, timeval 구조체로 초(tv_sec)와 마이크로초(tv_usec) 단위의 시간을 설정할 수 있습니다.
- it_value : 타이머가 처음 만료될 때까지 대기할 시간을 정의하며, timeval 구조체로 시간을 설정합니다.

3.2.6 Sigaction 구조체

신호가 발생했을 때 호출될 핸들러 함수와 신호 처리 방식을 정의하는 데 사용됩니다. sigaction 함수를 통해 특정 신호가 발생할 때 수행할 동작을 설정할 수 있으며, 신호 처리 중에 다른 신호를 블록하거나, 신호 처리 방식을 제어하는 다양한 옵션을 제공합니다.

sigaction 구조체 필드 (예시)

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

설명
: 신호가 발생할 때 호출할 핸들러 함수
: 확장된 신호 핸들러
: 신호 처리 중 블록할 다른 신호 집합
: 신호 처리 방식 제어 플래그
: 복원기 (일반적으로 사용되지 않음)

- sa_handler : 핸들러 함수 포인터로, 신호가 발생했을 때 실행될 함수이다. SIG_IGN으로 설정하면 해당 신호를 무시, SIG_DFL로 설정하면 시스템의 기본동작을 수행
- sa_sigaction : sa_flags에 SA_SIGINFO를 설정하면 이 핸들러가 호출되며, sa_sigaction을 설정 시 신호번호와 함께 신호에 대한 추가정보(siginfo_t *)와 호출 컨텍스트를 매개변수로 받을 수 있다.
- sa_mask : 신호가 처리되는 동안 블록할 다른 신호 집합을 정의한다.
- sa_flags : 신호 처리 방식을 제어하는 플래그
  - SA_RESTART: 신호 핸들러 실행 후, 중단된 시스템 호출이 자동으로 재시작되도록 설정
  - SA_SIGINFO: sa_sigaction 핸들러를 활성화하며, 신호에 대한 추가 정보를 전달받음
  - SA_NOCLDWAIT: SIGCHLD 신호에 대해 좀비 프로세스가 생성되지 않도록 설정

3.3 주요 개념의 적용

3.3.1 스케줄링 작업의 실행

사용된 주요 개념

* **Round-Robin 스케줄링** : 자식 프로세스가 공평한 CPU 작업시간을 가진다.
* **타이머**와 **SIGALRM 신호** : 주기적으로 스케줄링 작업을 수행하도록 트리거 하는 역할

작동 과정

1) **타이머 설정**

프로세스는 setitimer를 사용하여 일정 간격마다 SIGALRM 신호가 발생하도록 타이머를 설정합니다. 이 타이머 간격은 Round-Robin 스케줄링에서의 시간 할당량으로 설정됩니다. 타이머 간격은 모든 자식 프로세스가 같은 시간 동안 CPU를 사용할 수 있게 하기 위해 정의됩니다.

2) **SIGALRM 신호 & 스케줄링 핸들러**

GALRM 신호가 발생할 때마다 sigaction으로 설정된 신호 핸들러가 호출됩니다. 이 핸들러는 스케줄러 역할을 수행하며, 현재 run-queue에서 실행 중인 프로세스의 time quantum을 감소시킵니다.

3) **Time Quantum 종료 & 프로세스 전환**

약 현재 프로세스의 time quantum이 0이 되면, 스케줄러는 이 프로세스를 run-queue의 뒤로 이동시키고 다음 프로세스를 실행합니다. 동시에 현재 프로세스가 I/O 작업이 필요하다면, wait-queue로 이동시키고 io_burst가 남아 있는 동안 대기하도록 합니다.

4) **wait-queue 관리**

it-queue에 있는 프로세스들의 io_burst 값을 타이머 틱마다 감소시키고, io_burst가 0이 되면 해당 프로세스를 run-queue로 다시 이동시켜 CPU 사용이 가능하도록 합니다.

3.3.2 부모 -> 자식 프로세스로의 상태 전달

사용된 주요 개념

* **Message Queue(IPC)** : 프로세스 간의 시간슬라이스 정보(Time Qunantum) 전달

작동 과정

1) **부모 -> 자식으로의 시간 할당량 전송**:

부모 프로세스는 자식 프로세스가 CPU를 사용할 수 있는 최대 시간(Time Quantum)을 포함한 메시지를 메시지 큐에 전송합니다. 메시지 큐는 비동기 방식이므로, 자식 프로세스는 필요한 시점에 메시지를 수신할 수 있습니다.

2) **자식의 메시지 수신과 CPU-burst 작업 수행**:

자식 프로세스는 msgrcv를 통해 부모의 명령(시간 슬라이스 정보)을 수신하고, 이 시간을 기반으로 CPU 작업을 수행합니다. 받은 시간 슬라이스 동안 cpu_burst 작업을 수행하고, 시간이 만료되면 CPU 작업을 중단합니다.

3.3.3 **자식 -> 부모 프로세스로의 상태 전달**

사용된 주요 개념

* **Message Queue(IPC)** : 프로세스 간의 시간슬라이스 정보(Time Qunantum) 전달
* **CPU-burst와 I/O-burst** : 작업 상태를 CPU 작업과 I/O 작업으로 구분하여 관리

작동 과정

1) **CPU-burst 종료 시점에 I/O 요청 메시지 전송**

자식 프로세스는 CPU-burst 작업이 끝나면 부모 프로세스에게 I/O 요청 메시지를 전송합니다. 이 메시지에는 자식 프로세스의 pid와 필요한 io_burst 시간이 포함됩니다. 이 메시지를 통해 부모는 자식 프로세스가 CPU 사용을 종료하고, 이제 I/O 작업을 수행해야 한다는 것을 인식하게 됩니다.

2) **부모 프로세스의 run-queue와 wait-queue 관리**

부모 프로세스는 메시지 큐를 통해 자식의 I/O 요청을 수신하면, 해당 자식을 run-queue에서 wait-queue로 이동시킵니다. 이후 부모는 wait-queue에서 해당 자식의 io_burst 값을 타이머 틱마다 감소시키며, io_burst가 0이 되면 자식을 run-queue로 복귀시킵니다.

3.3.4 **Message Queue의 작동 및 프로그램 종료**

사용된 주요 개념

* **Message Queue(IPC)** : 프로세스 간의 시간슬라이스 정보(Time Qunantum) 전달
* **프로그램 종료** : 프로세스가 정상적으로 종료하도록 모든 Resorce 해제

작동 과정

1) **메시지 큐 생성 및 활용**:

부모 프로세스는 프로그램 시작 시 msgget을 사용해 메시지 큐를 생성하며, 이 큐를 통해 자식과 데이터를 주고받습니다. 부모는 CPU 사용 시간(Time Quantum)을 자식에게 전달하고, 자식은 작업 상태를 부모에게 전달하는 데 이 메시지 큐를 사용합니다.

2) 프로그램 종료 시 메시지 큐 삭제:

모든 스케줄링 작업이 완료되고, 모든 자식 프로세스가 종료되면 부모 프로세스는 msgctl을 통해 메시지 큐를 삭제합니다. IPC_RMID 플래그와 함께 msgctl을 호출하여 큐를 삭제하면, 메시지 큐에 할당된 시스템 리소스를 해제하여 시스템 리소스 점유가 남지 않도록 합니다.

3) 프로세스 종료 처리:

모든 자식 프로세스가 종료된 후, 부모 프로세스는 메시지 큐를 삭제하고 프로그램을 종료합니다. 이로써 부모와 자식 프로세스가 정상적으로 종료될 수 있는 환경을 조성하며, 프로그램 종료 시 모든 리소스를 깔끔하게 반환합니다.

4. 프로그램 구현

* 4.1 Project의 Round-Robin과 실제 운영체제의 차이점
* 4.2 Pesudo-Code 프로그램 분석
* 4.3 Main Code 분석
* 4.4 FCFS & SJF 구현

4.1 Project의 Round-Robin과 실제 운영체제의 차이점

4.1.1 CPU와 I/O 관리 방식

1) **Project RR** :

과제에서는 자식 프로세스들이 CPU-burst와 I/O-burst를 번갈아 수행하는 형태로 구현됩니다. I/O 작업을 단순히 wait-queue에서 대기한 상태로 처리합니다.

2) **운영체제** :

실제 운영체제의 RR 스케줄링은 각 프로세스의 CPU 및 I/O요청을 실시간으로 처리합니다. 자원 요구가 복잡하고, I/O 서브 시스템과 디바이스 드라이버를 통해 I/O작업이 비동기적으로 처리됩니다. 실제 운영체제는 비동기적으로 데이터를 디바이스로 전송하고, 작업이 완료되면 인터럽트를 통해 프로세스 상태를 업데이트합니다. CPU를 사용하지 않고 메모리와 디바이스 간 데이터를 전송하는 DMA와 같은 기법을 통해 더 효율적인 자원 관리가 가능합니다.

4.1.2 타이머와 신호 처리 방식

1) **Project RR** :

과제에서는 소프트웨어 타이머로 setitimer를 사용하여, 일정한 간격으로 SIGALRM 신호가 발생하도록 설정합니다. SIGALRM 신호가 발생할 때마다 부모 프로세스의 스케줄러가 Round-Robin방식으로 다음 프로세스에게 CPU를 할당합니다.

2) **운영체제** :

실제 운영체제는 하드웨어 타이머 인터럽트를 사용하여 정확한 주기의 timer tick을 보장합니다. 하드웨어 타이머는 매우 높은 정밀도로 인터럽트를 발생시켜, 운영체제 스케줄러가 정확한 시간간격으로 CPU 할당을 관리할 수 있습니다. 이는 컨텍스트 전환의 정확성과 안정성을 보장하며, timer tick 간격을 조정하여 시스템 부하에 따라 할당 시간을 동적으로 변경할 수도 있습니다.

4.1.3 컨텍스트 전환과 자원 최적화

1) **Project RR** :

과제에서는 기본적인 컨텍스트 전환만을 수행하며, 프로세스의 레지스터와 PC를 저장하고 복원하는 단순한 작업으로 구현합니다.

2) **운영체제** :

실제 운영체제에서는 컨텍스트 전환 시 레지스터와 메모리 주소를 관리할 뿐 아니라, 캐시 locality을 고려하여 성능 최적화를 수행합니다. 자주 전환되는 프로세스의 캐시 데이터를 유지하고, 캐시 미스가 발생하지 않도록 관리하여 성능을 향상시킵니다. 컨텍스트 전환에는 메모리 mapping, page table 관리, 캐시 플러시 등의 작업이 포함되며, 이 과정에서 다양한 자원 사용 최적화 기법이 적용됩니다.

4.1.4 상태 전환 처리와 대기 큐 관리

1) **Project RR** :

과제에서는 프로세스가 run-queue와 wait-queue를 오가며 상태가 전환됩니다. cpu_burst와 io_burst 값에 따라 단순하게 큐에서 이동하며, I/O 작업이 끝나면 flag에 따라 wait-queue에서 run-queue로 복귀하는 구조로 구현합니다.

2) **운영체제** :

실제 운영체제에서는 프로세스가 다양한 상태로 전환됩니다. 준비 상태, 실행 상태, 대기 상태, 종료 상태 외에도 입출력 대기, 페이지 폴트 대기와 같은 다양한 세부 상태들이 있습니다. 프로세스는 입출력 요청 시 비동기적으로 상태를 관리하고, 상태가 변경될 때 커널 내부의 구조체를 통해 정확하게 전환됩니다. I/O 요청은 I/O 서브시스템에서 관리하며, 대기 상태가 끝나면 인터럽트를 통해 프로세스가 준비 상태로 돌아갑니다.

4.1.5 스케줄링 대상과 환경

1) **Project RR** :

과제에서는 메시지 큐를 통해 부모와 자식 프로세스 간의 통신을 구현합니다. 자식 프로세스는 I/O요청이 발생하면 부모에게 메시지를 보내고, 부모는 이를 수신하여 자식의 상태를 전환합니다. 단순한 IPC 메커니즘을 사용하여 프로세스 간의 상태 전달을 수행합니다.

2) **운영체제** :

실제 운영체제는 PCB와 커널의 내부 데이터 구조를 통해 프로세스 상태를 전환하고 관리합니다. IPC 방식은 프로세스 간의 의존성, 데이터 공유 요구, 자원 사용 패턴에 따라 최적화되어 사용됩니다. 커널은 프로세스 간 통신의 복잡성을 관리하고, 프로세스가 서로 독립적이면서도 필요한 데이터를 효율적으로 교환할 수 있도록 지원합니다.

4.2 Pesudo-Code 프로그램 분석

4.2.1 자식 프로세스 생성 함수

Code create_children 함수

```
자식 프로세스 생성 함수 (create_children):
    for i = 0 to NUM_CHILDREN - 1:
        pid = fork()
        if (pid == 0):  // 자식 프로세스인 경우
            랜덤한 cpu_burst와 io_burst 초기화
            메시지 구조체 msg 선언

            while (true):
                msg 큐에서 메시지 수신 (msgrcv 호출)
                if (수신 실패):
                    오류 메시지 출력 후 종료

                cpu_burst -= TIME_QUANTUM

                if (cpu_burst <= 0):
                    msg에 다음 io_burst 값을 저장
                    부모 프로세스에 메시지 전송 (msgsnd 호출)
                    if (전송 실패):
                        오류 메시지 출력 후 종료

            프로세스 종료
        else:  // 부모 프로세스인 경우
            run_queue[i]에 자식 PID 저장
            remaining_burst[i]와 io_burst[i]를 랜덤하게 초기화
```

역할 : NUM_CHILDREN의 수 만큼 자식 프로세스를 생성하여 Run-queue에 등록

작동 방식

1) for 루프를 사용해 NUM_CHILDREN 만큼의 자식 프로세스를 생성
2) 프로세스는 cpu_burst와 io_burst를 무작위로 초기화하여 CPU와 I/O 작업을 설정
3) 생성된 프로세스는 메시지큐에서 time slice 메시지를 수신(msgrcv)하고, cpu_burst 감소
4) cpu_burst가 0 이하가 될 경우 I/O 요청 메시지를 부모에게 전송(msgsnd)
5) 다시 새로운 cpu_burst와 io_burst 값을 설정하여 loop를 반복
6) 부모 프로세스에서는 run_queue 배열에 자식 프로세스의 PID와 초기 remaining_burst, io_burst 값을 저장하야 스케줄링 준비를 완료

4.2.2 타이머 설정 함수

Code setup_timer 함수

```
타이머 설정 함수 (setup_timer):
    타이머 값 설정 (0.05초 주기)

    SIGALRM 신호 핸들러 설정 (signal_handler 함수로 설정)
    if (sigaction 실패):
        오류 메시지 출력 후 종료
```

역할 : SIGALRM 신호가 일정 주기마다 발생하도록 타이머를 설정하고,
스케줄링 작업이 주기적으로 트리거 되도록 조정

작동방식

1) itimerval 구조체를 사용하여 타이머를 설정하고, 초기값과 간격을 0,05초로 지정
2) setitimer 함수를 호출하여 TIMER_REAL 타이머를 활성화
3) sigaction을 통해 SIGALRM 신호발생할 때 호출될 signal_handler 함수로 설정

4.2.3 SIGALRM 핸들러

Code signal_handler 함수

```
signal_handler 함수:
    tick_count 증가

    if (tick_count가 100의 배수):
        "Current Timer Tick: tick_count" 출력

    // 모든 프로세스의 대기 시간 및 I/O 대기 시간 업데이트
    for i = 0 to NUM_CHILDREN - 1:
        if (i != current_process && remaining_burst[i] > 0):
            wait_time[i] += 1  // 현재 프로세스가 아닌 경우 대기 시간 증가

        if (io_burst[i] > 0):
            io_wait_time[i] += 1  // I/O 대기 시간 증가

    RR 스케줄러 호출 (RR_scheduler)

    // 현재 Run-queue와 Wait-queue 상태를 로그에 기록
    "Tick tick_count:"를 로그에 기록
    "Run-queue:" 및 각 프로세스의 remaining_burst 값을 로그에 기록
    "Wait-queue:" 및 각 프로세스의 io_burst 값을 로그에 기록
    로그 파일을 플러시 (즉시 기록)
```

역할 : SIGALRM 신호가 발생할 때 스케줄링 작업을 수행하고,
각 프로세스의 대기시간과 I/O 대기시간을 Log에 기록

작동방식

1) tick_count를 증가시키고, 매 100tick 마다 진행상황을 알수 있게 터미널에 출력
2) for 루프를 통해 현재 CPU를 사용중인 프로세스 외의 모든 프로세스의 대기시간을 증가
3) I/O를 대기 중인 프로세스를 I/O 대기시간 증가
4) RR_scheduler를 호출하여 프로세스에 CPU time slice를 할당하고 CPU_burst를 소모
5) 현재 tick의 Run큐, Wait큐의 상태를 schedule_dump.txt 파일에 기록

4.2.4 RR 스케줄러

Code RR_scheduler 함수

```
RR_scheduler 함수:
    if (현재 프로세스의 remaining_burst > 0):
        msg에 현재 프로세스 PID와 타임 슬라이스 저장
        msg 큐에 메시지 전송 (msgsnd 호출)
        if (전송 실패):
            오류 메시지 출력

        // CPU 사용 상황을 로그에 기록
        "at time tick_count, process PID gets cpu time, remaining cpu-burst: remaining_burst" 기록

        remaining_burst 감소 (TIME_QUANTUM만큼)
        현재 프로세스의 CPU 사용 시간 증가 (cpu_usage_time 배열에 기록)

        if (remaining_burst <= 0):
            I/O 처리 함수 호출 (handle_io)

    current_process = (current_process + 1) % NUM_CHILDREN
```

역할 : Round-Robin 스케줄링 기법에 따라 프로세스에 CPU time slice를 할당/
남은 CPU_burst를 감소시키고, CPU_burst가 0이 되면 I/O 처리를 수행.

작동방식

1) remaining_burst가 0보다 클 경우, 프로세스에 time slice를 할당하여 CPU작업을 진행
2) 메시지 구조체에 현재 프로세스의 PID와 time slice를 설정하고,
   부모 프로세스가 메시지를 전송하여 자식 프로세스의 CPU_burst를 감소시킴
3) CPU의 할당 상황을 Log 파일에 기록하고, remaining_burst를 타임 퀀텀 만큼 감소
4) remaining_burst가 0이하가 되면 handle_io() 함수를 호출하고,
   프로세스가 I/O 작업을 수행하고, 다음 Run큐의 프로세스로 이동하며 반복

4.2.5 I/O 처리 함수

Code handle_io 함수

```
handle_io 함수:
    for i = 0 to NUM_CHILDREN - 1:
        if (io_burst[i] > 0):
            io_burst 감소 (1씩 감소)

            if (io_burst[i] == 0):
                remaining_burst[i]와 io_burst[i]에 새로운 랜덤 값 할당
                "Process PID completed I/O, returning to Run-queue" 로그에 기록
```

역할 : Wait큐에 있는 프로세스의 I/O_burst 값을 감소
I/O가 완료되면 Run큐로 돌려보내 CPU를 할당받을 준비

작동방식

1) for 루프를 사용하여 Wait큐에 있는 모든 프로세스의 io_burst 값을 감소시킵니다.
2) io_burst가 0이 된 프로세스의 I/O 작업이 완료된 것으로 간주하고,
   새로운 remaining_burst와 io_burst 값을 무작위로 설정한 뒤 Run큐로 복귀
3) 프로세스의 작업의 완료 메시지를 Log파일에 기록

4.2.6 main 함수

Code main 함수

```
메인 함수:
    현재 시간 기록 (프로그램 시작 시간)

    "Program started" 메시지 출력

    schedule_dump.txt 파일 열기
    if (파일 열기 실패):
        오류 메시지 출력 후 종료

    메시지 큐 생성
    if (메시지 큐 생성 실패):
        오류 메시지 출력 후 종료

    자식 프로세스 생성 함수 호출 (create_children)
    타이머 설정 함수 호출 (setup_timer)

    while (tick_count < MAX_TICKS):
        SIGALRM 신호를 기다림 (pause 함수로 대기)

    모든 자식 프로세스가 종료될 때까지 기다림 (wait 호출 반복)

    메시지 큐 삭제 (msgctl 호출)
    schedule_dump.txt 파일 닫기

    현재 시간 기록 (프로그램 종료 시간)

    프로그램 실행 시간 계산
    각 프로세스별 CPU 사용률 계산

    "Process Statistics" 메시지 출력
    각 프로세스별 CPU 사용 시간, 대기 시간, I/O 대기 시간 출력

    "Total Execution Time", "CPU Utilization" 출력
    "All processes have been terminated" 메시지 출력
```

역할 : 프로그램의 시작&종료를 관리 및 자식 프로세스를 생성하여 스케줄링을 실행.
타이머를 설정하여 SIGALRM 신호가 주기적으로 발생.
모든 프로세스가 종료 된 후 최종 통계 프로그램을 종료.

작동방식

1) Log 파일(schedule_dump.txt)을 생성 및 열고, 메시지 큐를 생성.
2) 프로그램의 시작시간을 기록하고, 자식 함수를 호출하여 자식 프로세스를 생성
3) setup_timer 함수를 호출하고 타이머를 설정
4) while 루프에서 pause()를 사용하여 SIGALRM 신호를 대기하고 신호 발생시 작업을 진행
5) tick_count의 작동 횟수가 MAX_tick에 도달하면 자식 프로세스의 자원을 해제
6) 메시지 큐의 삭제 및 프로그램의 종료 간을 기록한 후, 프로세스별 통계와 실행시간을 출력

4.3 Main Code 분석

4.3.1 헤더파일 호출 및 변수선언

POSIX 기능 활성화 : `#define _POSIX_C_SOURCE 199309L`
sigaction 구조체를 불러내는데 생기는 문제해결 하기 위한 정의

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

헤더파일

- unistd.h : UNIX 표준 심볼과 시스템 호출(fork, pause, close, execl 등) 등 프로세스 제어와 관련된 기능을 지원
- signal.h : 신호처리와 관련된 라이브러리로 sigaction, signal, kill 등을 정의
- sys/wait.h : wait 및 waitpid와 같은 자식 프로세스의 종료 상태를 수집, 프로세스의 종료를 대기하는데 필요한 기능을 제공
- sys/types.h : 시스템 호출이나 구조체에서 사용되는 데이터 타입(pid_t, key_t, ssize_t 등)을 제공
- sys/ipc.h & sys/msg.h : 메시지 큐의 생성과 통신에 필요한 데이터 타입과 함수(msgget, msgsnd, msgrcv, msgctl)를 제공
- sys/time.h : itimerval 구조체와 setitimer 함수로 타이머를 설정하고 제어하는 기능을 제공
- time.h : 시간 관련 함수와 데이터 타입(clock_gettime)을 포함.

매크로 상수 정의

```
#define NUM_CHILDREN 10
#define TIME_QUANTUM 1
#define MAX_TICKS 1500
```

- NUM_CHILDREN: 생성할 자식 프로세스의 수를 정의
- TIME_QUANTUM: 각 프로세스가 할당받는 타임 슬라이스를 정의
- MAX_TICKS: 프로그램 실행의 최대 tick 횟수를 정의

메시지 큐 구조체 정의

```
struct my_msgbuf
{
    long mtype;
    int pid;
    int io_time;
    int cpu_burst;
};
```

- mtype: 메시지의 유형
- pid: 메시지를 받는 자식 프로세스의 PID를 저장
- io_time: 현재 프로세스의 I/O 작업에 소요되는 시간을 기록
- cpu_burst: 프로세스에 할당된 CPU 작업시간을 기록 및 남은 CPU 작업을 추적

프로세스별 통계 추적 변수

```
int cpu_usage_time[NUM_CHILDREN] = {0};
int wait_time[NUM_CHILDREN] = {0};
int io_wait_time[NUM_CHILDREN] = {0};
```

: 각 프로세스의 총 CPU 사용 시간을 기록하는 배열
: 각 프로세스의 총 대기 시간을 기록
: 각 프로세스의 총 I/O 대기 시간을 기록

스케줄링 관련 변수

```
int run_queue[NUM_CHILDREN];
int remaining_burst[NUM_CHILDREN];
int io_burst[NUM_CHILDREN];
```

: 실행 가능한 자식 프로세스를 관리하는 배열, 프로세스의 PID가 저장
: 프로세스의 잔여 CPU burst 시간을 저장하는 배열
: 각 프로세스의 잔여 I/O burst 시간을 저장(I/O 작업 종료 추적)

기타 변수

```
int tick_count = 0;
int current_process = 0;
int msgq_id;
FILE *log_file;
```

: 진행된 타이머 tick 수를 저장하는 변수 (최대 MAX_TICK까지 작동)
: 현재 CPU 할당 중인 프로세스의 인덱스를 기록
: 메시지 큐의 ID를 저장하는 변수
: schedule_dump.txt 파일에 대한 파일 포인터, 스케줄링 로그를 기록

4.3.2 create_children 함수

자식 프로세스 생성:

```
void create_children()
{
    for (int i = 0; i < NUM_CHILDREN; i++)
    {
        pid_t pid = fork();
        if (pid == 0) {
            srand(getpid());
            int cpu_burst = rand() % 10 + 1;
            int io_burst = rand() % 5 + 1;
            struct my_msgbuf msg;
```

```
        else
        {
            run_queue[i] = pid;
            remaining_burst[i] = rand() % 10 + 1;
            io_burst[i] = rand() % 5 + 1;
        }
    }
}
```

- fork()를 이용하여 NUM_CHILDREN 개수만큼 자식 프로세스를 생성
- 자식 프로세스는 cpu_burst와 io_burst 시간을 랜덤하게 초기화

프로세스에서의 메시지 수신 및 전송:

```
// 메시지 수신
if (msgrcv(msgq_id, &msg, sizeof(msg) - sizeof(long), getpid(), 0) == -1)
{   // 오류처리
    perror("msgrcv");
    exit(1);
}
```

…

```
    // CPU-burst 감소
    cpu_burst -= TIME_QUANTUM;
    if (cpu_burst <= 0)
    {
        msg.mtype = 1;
        msg.pid = getpid();
        msg.cpu_burst = io_burst;
        if (msgsnd(msgq_id, &msg, sizeof(msg) - sizeof(long), IPC_NOWAIT) == -1)
        {   // 오류처리
            perror("msgsnd");
            exit(1);
        }
    }
}
exit(0);
```

자식 프로세스에서의 메시지 수신 및 전송:

- 각 자식 프로세스는 메시지 큐에서 부모 프로세스로부터 메시지를 수신(msgrcv)하여 할당된 CPU burst를 감소, cpu_burst가 0 이하가 되면 I/O 요청 메시지를 부모에게 전송(msgsnd)
- msgrcv나 msgsnd에서 오류가 발생 시 perror로 오류를 출력하고 자식 프로세스는 exit(1)로 종료

부모 프로세스에서의 실행 대기 큐 및 burst 초기화:

- 자식 프로세스의 PID와 초기 cpu_burst와 io_burst를 부모 프로세스의 run_queue, remaining_burst, io_burst 배열에 저장하여 스케줄링 준비

4.3.3 setup_timer 함수

SIGALRM 타이머 설정:

```
struct itimerval timer;
timer.it_value.tv_sec = 0;
timer.it_value.tv_usec = 50000;
timer.it_interval = timer.it_value;

if (setitimer(ITIMER_REAL, &timer, NULL) == -1)
{   // 오류처리
    perror("setitimer");
    exit(1);
}
```

- itimerval 구조체를 통해 타이머를 0.05초 간격으로 설정하여 주기적으로 SIGALRM 신호를 발생
- setitimer 호출이 실패할 경우 perror를 사용해 오류 메시지를 출력 및 프로그램 종료

SIGALRM 핸들러 등록:

```
struct sigaction sa;
sa.sa_handler = signal_handler;
sa.sa_flags = 0;
sigemptyset(&sa.sa_mask);

if (sigaction(SIGALRM, &sa, NULL) == -1)
{   // 오류처리
    perror("sigaction");
    exit(1);
}
```

- sigaction을 통해 SIGALRM 신호 발생 시 signal_handler 함수를 호출
- sigaction 호출이 실패할 경우 perror로 오류를 출력 및 종료

4.3.4 signal_handler 함수

Tick 증가와 진행 상황 출력:

```
tick_count++;
if (tick_count % 100 == 0)
{
    printf("Current Timer Tick: %d\n", tick_count);
}
```

- tick_count를 증가시키고, 매 100 tick마다 현재 tick 상태를 터미널에 출력

대기 시간 추적:

```
for (int i = 0; i < NUM_CHILDREN; i++)
{
    if (i != current_process && remaining_burst[i] > 0)
    {
        wait_time[i] += 1;
    }
    if (io_burst[i] > 0)
    {
        io_wait_time[i] += 1;
    }
}
```

- 현재 실행 중인 프로세스를 제외한 나머지 프로세스의 wait_time을 증가, I/O 대기 중인 프로세스의 io_wait_time도 증가

RR_scheduler 실행:

```
RR_scheduler();
```

로그 기록:

```
fprintf(log_file, "Tick %d:\n", tick_count);
fprintf(log_file, "Run-queue: ");
for (int i = 0; i < NUM_CHILDREN; i++)
{
    fprintf(log_file, "[%d:%d] ", run_queue[i], remaining_burst[i]);
}
fprintf(log_file, "\nWait-queue: ");
for (int i = 0; i < NUM_CHILDREN; i++)
{
    if (io_burst[i] > 0)
    {
        fprintf(log_file, "[%d:%d] ", run_queue[i], io_burst[i]);
    }
}
fprintf(log_file, "\n");
fflush(log_file);
```

- RR_scheduler 함수 호출 후 현재 run_queue와 wait_queue의 상태를 schedule_dump.txt 파일에 기록
- fflush를 사용해 버퍼의 내용을 즉시 파일에 기록하여, 로그 파일에 최신 상태를 유지

4.3.5 RR_scheduler 함수

타임 슬라이스 할당 및 CPU burst 감소:

```
void RR_scheduler()
{
    if (remaining_burst[current_process] > 0)
    {
        struct my_msgbuf msg;
        msg.mtype = run_queue[current_process];
        msg.cpu_burst = TIME_QUANTUM;

        if (msgsnd(msgq_id, &msg, sizeof(msg) - sizeof(long), IPC_NOWAIT) == -1)
        {   // 오류처리
            perror("msgsnd");
        }
```

…

```
        remaining_burst[current_process] -= TIME_QUANTUM;
        cpu_usage_time[current_process] += TIME_QUANTUM;
```

…

```
    }
    current_process = (current_process + 1) % NUM_CHILDREN;
}
```

- msgsnd를 통해 현재 프로세스에 타임 슬라이스를 할당하여 CPU 작업을 수행, 할당된 TIME_QUANTUM만큼 remaining_burst를 감소
- 할당 과정에서 msgsnd가 실패하면 perror로 오류를 출력

CPU 사용 시간 기록:

```
// CPU 할당 상황을 파일에 기록
fprintf(log_file, "at time %d, process %d gets cpu time, remaining cpu-burst: %d\n",
        tick_count, run_queue[current_process], remaining_burst[current_process]);
```

- 각 프로세스의 cpu_usage_time 배열에 사용한 CPU 시간을 기록

I/O 요청 처리:

```
if (remaining_burst[current_process] <= 0)
{
    handle_io();
}
```

- remaining_burst가 0 이하가 되면 handle_io 함수를 호출하여 현재 프로세스가 I/O 요청을 할 수 있도록 관리
- 스케줄링 작업 완료 후 다음 프로세스로 current_process를 이동하여 Round-Robin 방식으로 모든 프로세스가 공평하게 스케줄링

4.3.6 handle_io 함수

I/O burst 감소:

```
void handle_io()
{
    for (int i = 0; i < NUM_CHILDREN; i++)
    {
        if (io_burst[i] > 0)
        {
            io_burst[i]--;
            if (io_burst[i] == 0)
            {
                remaining_burst[i] = rand() % 10 + 1;
                io_burst[i] = rand() % 5 + 1;
                fprintf(log_file, "Process %d completed I/O, returning to Run-queue\n", run_queue[i]);
            }
        }
    }
}
```

- for 루프를 통해 Wait-queue에 있는 모든 프로세스의 io_burst 값을 감소시켜 I/O 작업이 점진적으로 완료될 수 있도록 유도

I/O 완료 후 Run-queue 복귀:

- io_burst가 0이 된 프로세스는 I/O 작업이 완료된 것으로 간주, 새로운 remaining_burst와 io_burst 값을 랜덤하게 설정 및 Run-queue로 되돌려 CPU 할당을 받을 수 있게 준비
- I/O 작업이 완료된 상태는 schedule_dump.txt 파일에 기록

4.3.7 **Main 함수**

프로그램 시작과 종료 관리:

```c
int main()
{
    printf("Program started. Creating
    child processes and initializing scheduling.\n");

    log_file = fopen("schedule_dump.txt", "w");
    if (log_file == NULL)   // 오류처리
    {
        perror("fopen");
        exit(1);
    }

    // 프로그램 시작 시간 기록
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);
```

- clock_gettime으로 프로그램 시작 시간 및 종료 시간을 기록하여 전체 실행시간을 계산
- "Program started" 메시지를 출력, 프로그램의 시작을 알림

로그 파일 생성:

- fopen으로 로그 파일 schedule_dump.txt를 열어 스케줄링 작업 로그를 저장

...

```c
    // 프로그램 종료 시간 기록
    clock_gettime(CLOCK_MONOTONIC, &end);
    double elapsed = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
```

메시지 큐 생성 및 제거:

```c
    // 메시지 큐 생성
    msgq_id = msgget(IPC_PRIVATE, IPC_CREAT | 0666);
    if (msgq_id == -1)      // 오류처리
    {
        perror("msgget");
        exit(1);
    }
```

- msgget을 통해 부모와 자식 프로세스 간의 통신을 위한 메시지 큐를 생성
- msgctl을 통해 모든 자식 프로세스가 종료된 후 msgctl을 사용하여 메시지 큐를 제거 (자원 누수방지)

...

```c
    // 모든 자식이 종료된 후에 메시지 큐 제거
    msgctl(msgq_id, IPC_RMID, NULL);
    fclose(log_file);
```

주기적 스케줄링:

```c
    create_children();
    setup_timer();
    // 메인 스케줄링 루프
    while (tick_count < MAX_TICKS)
    {       pause();    }
    // 모든 자식 프로세스가 종료될 때까지 대기
    for (int i = 0; i < NUM_CHILDREN; i++)
    {       wait(NULL);    }
```

- MAX_TICKS에 도달할 때까지 pause()로 SIGALRM 신호를 대기하여 주기적 스케줄링을 수행
- 모든 자식 프로세스가 종료될 때까지 wait(NULL)을 통해 기다림

프로그램 종료 후 통계 출력: (편의를 위해 모든 통계를 터미널에 출력하도록 수정함)

```c
    // CPU 사용률 계산
    double cpu_utilization = 0;
    for (int i = 0; i < NUM_CHILDREN; i++)
    {       cpu_utilization += cpu_usage_time[i];    }
    cpu_utilization = (cpu_utilization / (MAX_TICKS * NUM_CHILDREN)) * 100;
    // 터미널에 프로세스별 통계 출력
    printf("\nProcess Statistics:\n");
    for (int i = 0; i < NUM_CHILDREN; i++)
    {          printf("Process %d: CPU Usage Time = %d, Wait Time = %d, I/O Wait Time = %d\n"
               run_queue[i], cpu_usage_time[i], wait_time[i], io_wait_time[i]);    }
        // 총 실행시간 & CPU 사용률 & 자원 해제
    printf("\nTotal Execution Time: %f seconds\n", elapsed);
    printf("CPU Utilization: %f%%\n", cpu_utilization);
    printf("All processes have been terminated. Resources cleaned up.\n");
```

- 각 프로세스의 CPU 사용률과 실행 시간 등의 통계를 터미널에 출력

4.4 FCFS & SJF 스케줄링 구현 (project1_n.c) // 실행안됨

4.4.1 FCFS_scheduler() 함수

: FCFS 스케줄링 방식을 통해 run_queue의 첫 번째 프로세스를 끝날 때까지 실행한다.

```
void FCFS_scheduler()
{
    for (int i = 0; i < NUM_CHILDREN; i++) {
        if (remaining_burst[i] > 0) {
            struct my_msgbuf msg;
            msg.mtype = run_queue[i];
            msg.cpu_burst = remaining_burst[i];

            if (msgsnd(msgq_id, &msg, sizeof(msg) - sizeof(long), IPC_NOWAIT) == -1) {
                perror("msgsnd");
            }
            fprintf(log_file, "at time %d, process %d gets cpu time, remaining cpu-burst: %d\n",
                    tick_count, run_queue[i], remaining_burst[i]);
            cpu_usage_time[i] += remaining_burst[i];
            remaining_burst[i] = 0;
            handle_io();
            break;
        }
    }
}
```

- for 반복문을 통해 run_queue의 첫 번째 프로세스부터 순차적으로 검사한다. remaining_burst[i]가 0보다 크면 해당 프로세스를 현재 실행 대상으로 선택한다.
- 프로세스를 선택하면 IPC 메시지 송신(msgsnd)으로 프로세스에 remaining_burst 만큼의 CPU 시간을 할당한다. 여기서 msg.mtype은 해당 프로세스의 run_queue[i]로 지정해 특정 프로세스가 메시지를 받도록 설정되고 남은시간을 0으로 설정하여 프로세스가 작업을 모두 완료한 것으로 표시한다.
- I/O 작업 처리: handle_io() 함수를 호출하여 I/O burst를 처리하다. 이는 FCFS에서 각 프로세스가 CPU burst를 끝낸 후 I/O 작업이 발생하도록 구현된 방식이다..
- 이 함수는 처음 발견한 작업 완료 프로세스에 대해서만 실행을 완료하고 break로 반복을 종료한다. 다른 프로세스들은 다음 tick에서 순차적으로 처리된다.

4.4.2 SJF_scheduler() 함수

: SJF 스케줄링 방식을 통해 현재 run_queue에 있는 프로세스 중 남은 burst 시간이 가장 짧은 프로세스를 선택하여 실행한다.

```
void SJF_scheduler() {
    int shortest_job = -1;
    int min_burst = __INT_MAX__; // 초기값 설정
    for (int i = 0; i < NUM_CHILDREN; i++) {
        if (remaining_burst[i] > 0 && remaining_burst[i] < min_burst) {
            min_burst = remaining_burst[i];
            shortest_job = i;
        }   }
    if (shortest_job != -1) {
        struct my_msgbuf msg;
        msg.mtype = run_queue[shortest_job];
        msg.cpu_burst = remaining_burst[shortest_job];

        if (msgsnd(msgq_id, &msg, sizeof(msg) - sizeof(long), IPC_NOWAIT) == -1) {
            perror("msgsnd");       }
        fprintf(log_file, "at time %d, process %d gets cpu time, remaining cpu-burst: %d\n",
                tick_count, run_queue[shortest_job], remaining_burst[shortest_job]);

        cpu_usage_time[shortest_job] += remaining_burst[shortest_job];
        remaining_burst[shortest_job] = 0;
        handle_io();
    }
}
```

- shortest_job 변수를 통해 가장 작은 burst 시간을 가진 프로세스의 인덱스를 추적한다.
- 반복문을 통해 remaining_burst[i]가 가장 작은 프로세스를 찾고, shortest_job에 해당 인덱스를 저장한다.
- shortest_job이 설정되면, msgsnd()를 통해 해당 프로세스에 CPU 시간을 할당하고 스케줄링 정보를 로그에 기록한다.
- 남은 시간을 0으로 설정하여 프로세스가 완료되었음을 표시한다.
- 완료 후 handle_io() 함수를 호출하여 I/O burst를 처리한다.

4.4.3 **handle_io() 함수**

: I/O 작업을 완료하고 다시 run_queue로 돌아오는 프로세스를 관리한다.

```
void handle_io()
{
    for (int i = 0; i < NUM_CHILDREN; i++)
    {
        if (io_burst[i] > 0)
        {
            io_burst[i]--;
            if (io_burst[i] == 0)
            {
                fprintf(log_file, "Process %d completed I/O, returning to Run-queue\n", i);
                remaining_burst[i] = rand() % 10 + 1; // 새로운 CPU burst 시간
                io_burst[i] = rand() % 5 + 1;         // 새로운 I/O burst 시간
                run_queue[run_queue_size++] = i;
            }
        }
    }
}
```

- 각 프로세스의 io_burst 값을 검사하고, io_burst[i]가 0보다 큰 경우에는 1씩 감소시켜 I/O 작업이 진행 중임을 나타낸다.
- io_burst[i]가 0이 되면 I/O 작업이 완료된 것으로 간주하고, 해당 프로세스를 run_queue로 다시 복귀시킨다.
- 남은 CPU burst와 I/O burst를 새로 설정하여 프로세스가 다시 실행될 준비를 하도록 설정한다.
- run_queue 복귀 시 로그에 이를 기록하여 스케줄링 정보 추적이 가능하게 한다.

5. 프로그램 빌드 환경

컴파일 환경: Ubuntu 22.04 -> VS Code 1.93
프로그래밍 언어: C언어
실행 파일: project1_.c (RR구현) / project1_n.c (RR, FCFS, SJF구현) // 작동 안함
실행 방법: F5로 실행
실행결과 확인 ; 터미널 창 및 scheduler_dump.txt 파일에서 log확인

6. 문제점과 해결방법

6.1 타이머 설정과 SIGALRM 핸들러 처리 방식

SIGALRM 신호와 관련된 sigaction 및 itimerval 구조체의 세부 구성이 직관적으로 와닿지 않아 초기구현에 어려움이 있었고, 세부 구성과 SIGALRM 의 함수들의 사용 방식에 대해 따로 찾아보고서 구현할 수 있었다. 특히, SIGALRM 신호가 발생했을 때 바로 스케줄링 작업을 수행해야 할지, 아니면 다른 작업 이후에 수행해야 할지에 대해 기준을 잡지못해 신호 발생 시의 처리 흐름을 설정하는데 시간을 많이 잡아먹은 것 같다.

해결한 방법으로는 setitimer 함수를 사용해 일정 주기로 SIGALRM 신호를 발생하도록 설정하고, sigaction을 사용하여 SIGALRM 신호가 발생할 때마다 signal_handler 함수가 호출되도록 설정하였다. 이는 SIGALRM 신호가 발생할 때마다 signal_handler가 즉시 실행되도록 구성할 수 있다.

6.2 msgrcv: Identifier removed 오류 문제

MAX_TICK의 수를 일정 수 이상으로 지정하면, 꾸준히 뜨던 오류로 결국 해결하지 못해 MAX_TICK를 줄이는 쪽으로 수정하고 타이머 주기(tv_usec)를 늘려 실행시간을 1분 이상이 되도록 조정하였다. 대략적으로 1200tick을 넘어가면서 생기는 꾸준히 생성되는 오류였다.

문제점은 과제에서 부모 프로세스가 자식 프로세스와 메시지 큐를 통해 통신하고, 일정 작업 후 메시지 큐를 삭제하는 작업이 있는데 메시지 큐를 삭제한 이후에도 자식 프로세스가 msgrcv를 통해 메시지를 수신하려고 시도하는 경우에 발행하는 오류였다.

해결하는 방법으로는 메시지 큐를 삭제하기 전에 모든 자식 프로세스가 종료되도록 설정하는 것인데. 제대로 구현되지는 못한 것 같다.

6.3 GPT 사용 효율

1) setitimer를 이용한 타이머 설정과 주기적인 SIGALRM 신호 처리

: Round-Robin 스케줄링 정책을 구현하여 모든 프로세스가 순환하며 일정한 TIME_QUANTUM 동안 CPU를 할당받도록 설정하는 것이 어려웠고, GPT의 도움으로 RR_scheduler 함수를 구현하여 현재 프로세스에 TIME_QUANTUM을 할당하고, TIME_QUANTUM이 만료되면 다음 프로세스로 전환되도록 설정했다.

2) Run-queue & Wait-queue 발동 조건과 작동 방식

: 프로세스가 I/O 작업을 요청할 때와 CPU 작업을 완료한 후의 처리 방식에서 Run-queue와 Wait-queue가 어떻게 연계되는지 이해하지 못 했었다. 각각의 프로세스가 CPU 작업(CPU burst) 동안 실행되다가 그 시간이 다했을 때 Wait-queue로 이동하도록 설계하고, Wait-queue에 있는 프로세스가 다시 Run_queue로 돌아오는 기준을 잡는 것이 어려웠다. GPT의 도움을 통해 Wait-queue와 Run-queue 순차적인 작업과정을 정리하고, 발동 조건을 명확히 정리하여 구현을 마무리하였다.

7. 실행결과 및 분석

7.1 project1_.c 실행 결과창

1) **디버그 콘솔**

```
Loaded '/lib/x86_64-linux-gnu/libc.so.6'. Symbols loaded.
[Detaching after fork from child process 21819]
[Detaching after fork from child process 21820]
[Detaching after fork from child process 21821]
[Detaching after fork from child process 21822]
[Detaching after fork from child process 21823]
[Detaching after fork from child process 21824]
[Detaching after fork from child process 21825]
[Detaching after fork from child process 21826]
[Detaching after fork from child process 21827]
[Detaching after fork from child process 21828]
[Inferior 1 (process 21815) exited normally]
```

: 프로그램은 정상적으로 자식 프로세스들을 생성하고 분리되는 과정을 보여주며, 각 자식 프로세스의 PID를 보여준다.

: exited normally 메시지는 메인 프로세스가 정상적으로 종료되었음을 의미한다.

2) **터미널**

```
Program started. Creating child processes and initializing scheduling.
Current Timer Tick: 100
```

... : 프로그램이 정상적으로 시작됨을 알린다.

```
Current Timer Tick: 1200
msgrcv: Identifier removed
msgrcv: Identifier removed
```

: 100틱마다 타이머를 기록하여 스케줄링 이벤트의 진행상황 확인

... : 각 프로세스별 PID, CPU 사용 시간, 대기 시간, I/O 대기 시간이 기록되어 있다.

```
Process Statistics:
Process 21819: CPU Usage Time = 108, Wait Time = 946, I/O Wait Time = 1210
Process 21820: CPU Usage Time = 97, Wait Time = 820, I/O Wait Time = 1210
Process 21821: CPU Usage Time = 114, Wait Time = 1011, I/O Wait Time = 1210
Process 21822: CPU Usage Time = 105, Wait Time = 919, I/O Wait Time = 1210
Process 21823: CPU Usage Time = 116, Wait Time = 1016, I/O Wait Time = 1210
Process 21824: CPU Usage Time = 89, Wait Time = 739, I/O Wait Time = 1210
Process 21825: CPU Usage Time = 92, Wait Time = 780, I/O Wait Time = 1210
Process 21826: CPU Usage Time = 108, Wait Time = 949, I/O Wait Time = 1210
Process 21827: CPU Usage Time = 103, Wait Time = 891, I/O Wait Time = 1210
Process 21828: CPU Usage Time = 114, Wait Time = 1001, I/O Wait Time = 1210
```

... : 스케줄링의 총 실행시간, CPU의 사용률이 기록되어 있고, 프로그램의 종료 메시지를 출력한다.

```
Total Execution Time: 60.504623 seconds
CPU Utilization: 8.716667%
All processes have been terminated. Resources cleaned up.
```

3) **schedule_dump.txt 파일 log**

```
at time 1, process 21819 gets cpu time, remaining cpu-burst: 4
Tick 1:
Run-queue: [21819:3] [21820:8] [21821:4] [21822:7] [21823:10] [21824:3] [21825:1] [21826:4]
Wait-queue: [21819:2] [21820:1] [21821:1] [21822:3] [21823:2] [21824:3] [21825:5] [21826:2]
at time 2, process 21820 gets cpu time, remaining cpu-burst: 8
Tick 2:
Run-queue: [21819:3] [21820:7] [21821:4] [21822:7] [21823:10] [21824:3] [21825:1] [21826:4]
```

...

```
at time 22, process 21820 gets cpu time, remaining cpu-burst: 1
Process 21819 completed I/O, returning to Run-queue
Process 21822 completed I/O, returning to Run-queue
Process 21824 completed I/O, returning to Run-queue
Process 21826 completed I/O, returning to Run-queue
Tick 22:
Run-queue: [21819:10] [21820:0] [21821:7] [21822:4] [21823:2] [21824:2] [21825:0] [21826:10
```

... = 일정하지 않은 패턴 == Round-Robin의 특징

```
at time 1210, process 21828 gets cpu time, remaining cpu-burst: 8
Tick 1210:
Run-queue: [21819:9] [21820:6] [21821:9] [21822:0] [21823:8] [21824:9] [21825:0] [21826:7]
Wait-queue: [21819:2] [21820:3] [21821:3] [21822:2] [21823:2] [21824:2] [21825:3] [21826:3]
```

: at time X, process Y gets cpu time, remaining cpu-burst: Z 형식으로 각 시간 틱마다

프로세스가 CPU를 할당받는 상황이 기록되어 있다. Tick X 이후에 Run-queue와 Wait-queue 상태가 출력되며, 각 프로세스의 남은 CPU burst와 I/O burst 시간이 표시됩니다.
: Tick 번호가 표시되면서 각 틱마다 Run-queue와 Wait-queue의 상태가 갱신된다.
: 스케줄링 중인 프로세스와 I/O 작업 중인 프로세스가 번갈아 가며 대기열에 들어가거나 돌아오면서 라운드 로빈 방식으로 CPU 시간을 할당받는 과정을 보여준다.

4) **Round-Robin 스케줄링의 일정하지 않은 패턴**
: 라운드 로빈 스케줄링은 각 프로세스에 일정한 타임 퀀텀(TIME_QUANTUM)을 할당하여 실행합니다. 이로 인해 모든 프로세스가 차례대로 CPU 시간을 받게 됩니다. 프로세스의 CPU burst와 I/O burst가 랜덤으로 설정되어 있기 때문에, 각 프로세스가 실행되면서 남은 CPU burst나 I/O 요청이 다르게 발생합니다. 그로인해 특정 시간 이후에도 각 프로세스의 남은 burst 값이 서로 달라지며, 이로 인해 일정하지 않은 패턴이 나타납니다.

7.2 인덱스 값 조정 실험

7.2.1 자식 프로세스 수 변경

1) NUM_CHILDREN == 1

```
Process Statistics:
Process 37017: CPU Usage Time = 4, Wait Time = 0, I/O Wait Time = 1201
Total Execution Time: 60.051625 seconds
CPU Utilization: 0.333333%
```

- 자식 프로세스가 1개 뿐이라 연속적으로 CPU할당과 I/O대기 후 CPU 작업을 반복한다.
- 총 실행 시간은 약 60초로, I/O 대기시간이 프로그램 전체 실행시간의 대부분을 차지한다.
- 대기시간이 0인 것은 프로세스가 다른 프로세스와 CPU를 공유할 필요가 없기 때문이다.

2) NUM_CHILDREN == 50

```
Process 30327: CPU Usage Time = 25, Wait Time = 1224, I/O Wait Time = 1250
Process 30328: CPU Usage Time = 25, Wait Time = 1189, I/O Wait Time = 1250
Total Execution Time: 62.527768 seconds
CPU Utilization: 2.081667%
```

- 각 프로세스의 대기시간이 상당히 높아졌다. 이는 자식 프로세스가 많아지면서 CPU 자원을 얻기 위한 경쟁이 발생했음을 의미하며, 총 실행시간이 증가하는 결과를 보여준다.

3) NUM_CHILDREN == 100

```
Process 32735: CPU Usage Time = 13, Wait Time = 1283, I/O Wait Time = 1300
Process 32736: CPU Usage Time = 13, Wait Time = 1287, I/O Wait Time = 1300
Total Execution Time: 65.087837 seconds
CPU Utilization: 1.083333%
```

- I/O Wait Time이 증가하여 약 1300이 되었는데, 이는 CPU 할당을 받아도 대부분의 프로세스가 I/O 대기 상태로 전환된다는 것을 의미한다.

4) 자식 프로세스 수의 변화에 따른 분석
- 프로세스 수가 증가할수록 총 실행 시간은 늘어나며, 각 프로세스의 대기 시간과 I/O 대기 시간도 증가했습니다.
- CPU 사용률은 근소한 차이지만 1일 때 가장 낮고, 50일 때는 늘어났다가 100에서는 다시

감소했다.
 - 프로그램이 CPU보다는 I/O 작업이 많은 프로세스를 다루고 있어, 프로세스의 수를 늘려도 CPU 사용률이 크게 향상되지 않는다는 것을 알 수 있었습니다.

7.2.2 **Time quantum 변경**

1) TIME_QUANTUM == 1

```
Process 106901: CPU Usage Time = 103, Wait Time = 891, I/O Wait Time = 1210
Process 106902: CPU Usage Time = 114, Wait Time = 1001, I/O Wait Time = 1210
Total Execution Time: 60.504945 seconds
CPU Utilization: 8.716667%
```

 - CPU 사용 시간이 제한되어 I/O 대기 시간이 비교적 길고, 대기 시간이 높게 나타난다.

2) TIME_QUANTUM == 5

```
msgsnd: Resource temporarily unavailable
msgsnd: Resource temporarily unavailable
msgsnd: Resource temporarily unavailable
Process 106157: CPU Usage Time = 600, Wait Time = 947, I/O Wait Time = 1210
Process 106158: CPU Usage Time = 590, Wait Time = 958, I/O Wait Time = 1210
Total Execution Time: 60.504583 seconds
CPU Utilization: 49.583333%
```

 - 타임 슬라이스가 5로 늘어나면서, 프로세스가 CPU에 머무르는 시간이 길어짐에 따라 CPU 사용률이 증가했다.
 - msgsnd: Resource temporarily unavailable 오류가 발생했는데, 이는 자식 프로세스가 타임 슬라이스 내에서 메시지 전송 시 자원이 일시적으로 부족하여 발생하는 오류이다.

3) TIME_QUANTUM == 10

```
msgsnd: Resource temporarily unavailable
msgsnd: Resource temporarily unavailable
msgsnd: Resource temporarily unavailable
Process 114378: CPU Usage Time = 1210, Wait Time = 929, I/O Wait Time = 1210
Process 114379: CPU Usage Time = 1210, Wait Time = 942, I/O Wait Time = 1210
Total Execution Time: 60.505225 seconds
CPU Utilization: 100.833333%
```

 - 타임 슬라이스가 10으로 매우 길어지면서, 각 프로세스가 CPU에 오래 머물며 충분한 작업을 수행한다. CPU 사용률이 극대화되었고, 총 실행 시간은 여전히 약 60초로 일정하다.

4) Time Quantum의 값의 변화 따른 분석
 - 타임 슬라이스가 짧을수록 스케줄링 오버헤드가 증가하여 CPU 사용률이 낮아지고, 타임 슬라이스가 적당히 길어지면 CPU 사용률이 상승합니다.
 - 타임 슬라이스가 너무 길면 I/O 중심의 작업에서 효율이 떨어질 수 있다는 것을 알 수 있습니다.
 - msgsnd: Resource temporarily unavailable 1000tick 이후로 계속 발생했는데 CPU 할당 시간이 길어지면서 자원 요구가 증가하였고, 자원 경쟁이 발생하여 생긴 문제로 타임 슬라이스가 클수록 빈번하게 발생합니다.

7.2.3 **타이머 주기 값 변경** // 큰 의미는 없는 듯

1) tv_usec == 10000

```
Process 116100: CPU Usage Time = 103, Wait Time = 891, I/O Wait Time = 1210
Process 116101: CPU Usage Time = 114, Wait Time = 1001, I/O Wait Time = 1210
Total Execution Time: 12.103761 seconds
CPU Utilization: 8.716667%
```

- 실행 시간이 12초로 줄어들었지만, CPU 사용률은 8.716667%로 유지되었다.

2) tv_usec == 50000

```
Process 21819: CPU Usage Time = 108, Wait Time = 946, I/O Wait Time = 1210
Process 21820: CPU Usage Time = 97, Wait Time = 820, I/O Wait Time = 1210
Total Execution Time: 60.504623 seconds
CPU Utilization: 8.716667%
```

3) tv_usec == 100000

```
Process 115315: CPU Usage Time = 103, Wait Time = 891, I/O Wait Time = 1210
Process 115316: CPU Usage Time = 114, Wait Time = 1001, I/O Wait Time = 1210
Total Execution Time: 121.005537 seconds
CPU Utilization: 8.716667%
```

- 스케줄링이 자주 발생하지 않아 프로세스가 대기 시간이 증가하고, 실행 시간이 길어졌다.

4) 타이머 주기 값에 따른 분석

- 타이머 주기가 짧을수록 프로그램의 총 실행 시간이 단축되며, 주기가 길수록 실행 시간이 증가하는 경향이 있습니다.
- CPU 사용률은 타이머 주기에 관계없이 일정하게 유지되어 CPU 사용에는 영향을 미치지 않는 것을 알 수 있습니다.
- 타이머 주기를 50밀리초로 설정하였을 때 오버헤드 없이 CPU와 I/O 대기시간이 가장 적절한 것으로 볼 수 있습니다.

7.3 스케줄링 기법별 효율 분석(예상)

7,3,1 Round-Robin 스케줄링

: RR 스케줄링은 모든 프로세스에 동일한 Time Quantum을 할당하고, 순서대로 CPU 시간을 분배하는 방식이다. 과제에서는 TIME_QUANTUM 값을 지정하여 각 프로세스가 주기적으로 CPU를 할당받도록 구현할 수 있다.

장점:

- 모든 프로세스가 공정하게 CPU를 할당받아 대기 시간이 지나치게 길어지지 않음.
- 주기적인 프로세스 전환으로 응답 시간이 짧아짐.
- 프로세스가 일정 시간 이상 CPU를 사용하지 못하도록 제한하여 CPU 오버헤드가 줄어듦.

단점:

- CPU 전환이 많아져 컨텍스트 스위칭 오버헤드가 증가함.
- 짧은 Time Quantum 설정 시 작업 처리 효율이 떨어질 수 있음.

과제에서의 효율성: CPU와 I/O를 번갈아가며 사용하는 프로세스들이 많이 생성되므로, RR 스케줄링이 적합이라 예상된다.. Time Quantum이 너무 작으면 오버헤드가 커지므로, 적절한 Time Quantum 값을 설정하는 것이 중요하다.

7.3.2 First-Come-First-Serve 스케줄링

: FCFS 스케줄링은 프로세스가 도착한 순서대로 CPU를 할당하는 방식이다. 프로세스의 요청 시간이 빠를수록 먼저 CPU를 할당받게 되며, 작업의 종료 순서도 동일하게 유지된다.

장점:

- 구현이 간단하고 컨텍스트 스위칭 오버헤드가 적음.
- 선착순 방식이므로 모든 프로세스에 예측 가능한 종료 순서를 보장함.

단점:

- Convoy 효과 발생 가능성: 긴작업이 먼저 실행되면 뒤의 짧은작업들이 오래 기다리게 됨.
- 응답 시간이 불규칙적으로, I/O 중심의 프로세스에서 비효율적임.

과제에서의 효율성: 여러 프로세스가 각기 다른 CPU burst와 I/O burst를 가지므로 비효율적일 가능성이 크다. 특히 CPU 중심의 작업이 많으면 I/O 중심의 작업이 오랫동안 기다려야 하므로, 대기 시간이 길어질 수 있다.

7,3,3 Shortest Job First 스케줄링

: SJF 스케줄링은 CPU burst가 짧은 작업부터 우선적으로 CPU를 할당하는 방식이다. SJF에는 비선점형 방식과 선점형 방식이 있으며, 선점형 SJF는 남은 CPU burst가 더 짧은 프로세스가 나타나면 기존 프로세스를 중단하고 새로운 프로세스를 할당하는 방식이다.

장점:

- 평균 대기 시간이 가장 짧아지는 방식으로, 특히 짧은 작업들이 빨리 끝남.
- 처리 효율성이 높아 CPU 사용률이 극대화됨.

단점:

- 각 프로세스의 정확한 CPU burst 시간을 예측해야 함.
- Starvation 발생 가능성: 긴 작업은 계속 대기하게 될 수 있음.
- I/O 중심의 프로세스와 CPU 중심의 프로세스 간의 균형을 맞추기 어려움.

과제에서의 효율성: 과제에서는 CPU와 I/O burst가 다르므로, 정확한 burst 예측이 어려워 보인다. 짧은 작업에 우선순위를 두는 방식이지만, 긴 작업들이 기다리게 되어 Starvation 문제가 발생할 수 있다.

8. 느낀점

초기에 과제를 이해할 때, 자식 프로세스가 CPU burst와 I/O burst를 반복하는 무한 루프 구조로 설계되어 CPU 작업을 수행한다고 명시되어 있었지만, 이 "CPU 작업"이 구체적으로 무엇을 의미하는지 바로 이해하기 어려웠다. CPU 작업은 단순히 할당된 타임 슬라이스 만큼의 CPU burst를 소모하는 것인지, 실제로 CPU 연산을 수행하는 것인지 모호한 부분이 있었고, 자식 프로세스가 실제로 수행해야 할 작업이 무엇인지에 대한 정의가 명확하게 와닿지 않았다. CPU 작업을 단순히 CPU burst 시간 감소로 간주하여 시뮬레이션하는 방식으로 구현하게 되었다.

또한, setitimer를 이용한 타이머 설정과 주기적인 SIGALRM 신호처리, 그리고 msgget, msgsnd, msgrcv 시스템 호출을 사용한 메시지 큐 설정과 IPC를 구현하는 부분에 대해서 순서상으로의 복잡함 때문에 어려울 거라고 생각했는데, 수도코드로 기능을 차례차례 미리 정리해 보니 구현 과정이 생각보다 수월하게 진행되었다. 특히, 주기적인 신호 처리를 통해 스케줄링을 구현하는 방식이 처음에는 복잡하게 느껴졌으나, 구체적으로 각 신호와 타이머 설정을 코드로 옮기기 전에 필요한 기능들을 구성한 후 신호 발생과 처리, 메시지 큐를 통한 프로세스 간 통신을 보다 명확히 이해할 수 있었다.

부모 프로세스의 run-queue와 wait-queue 관리 부분도 초기 설계에서 어려울거라 생각했는데, GPT의 도움을 받아 run_queue와 wait_queue를 배열로 구현하고, 배열의 인덱스를 통해 각 프로세스의 상태를 관리하도록 설계하니 효과적으로 해결할 수 있었습니다. 배열 인덱스를 활용하여 각 프로세스의 남은 burst와 대기 상태를 쉽게 확인하고 업데이트할 수 있었다. 아직 실행의 과정이 완전히 머리에 정리된 것은 아니지만, 이 과정을 통해 라운드 로빈 스케줄링의 공정성과 효율성을 유지할 수 있다는 것을 확실하게 알수 있었다.

이번 과제에서 RR 스케줄링과 다른 스케줄링 기법이 이번 과제의 작업에서 얼마나 효율적인지를 분석하는게 가장 해보고 싶었는데 Main Code를 구현하는데도 오랜시간이 걸리고, 과제 마감기한도 짧아 코드를 완성시키지 못한 것이 아쉬웠다. project1_n.c에서 프로그램을 실행하면 작업을 진행할 스케줄링 기법을 선택해서 작업을 진행하도록 설계한다고 해봤는데, 급하게 해보느라 FCFS, SJF 스케줄링은 GPT 한테 만들어 달라해버리고, RR 스케줄링에서 추가적으로 구현하려 했기에 실행에 필요한 세부구성이 다른지 제대로 실행되지는 않았아서 스케줄링 기법들의 비교분석을 마무리하지 못한 것이 아쉬웠다.

9. 참고자료

* CPU scheduling (CPU_burst, I/O_burst)
https://ws-pace.tistory.com/23

* SIGALRM 타임아웃 구현
https://simsimjae.tistory.com/161

* Signal handler 개념 및 구현
https://medium.com/@ykm7003/signal-handler-%EC%97%90-%EB%8C%80%ED%95%B4%EC%84%9C-3b4f09c64aa6

* Run_Queue & Wait_Queue 개념
https://jj-0.tistory.com/6

* Message Queue 설정 & IPC & 시스템 호출
https://reakwon.tistory.com/97

* Message Queue 구현 참조
https://velog.io/@jhlee508/%EC%9A%B4%EC%98%81%EC%B2%B4%EC%A0%9C-Message-Queue-IPC