

OS Project 1

Scheduling Simulator

모바일시스템공학과

32214362 조윤서

32204288 조민욱

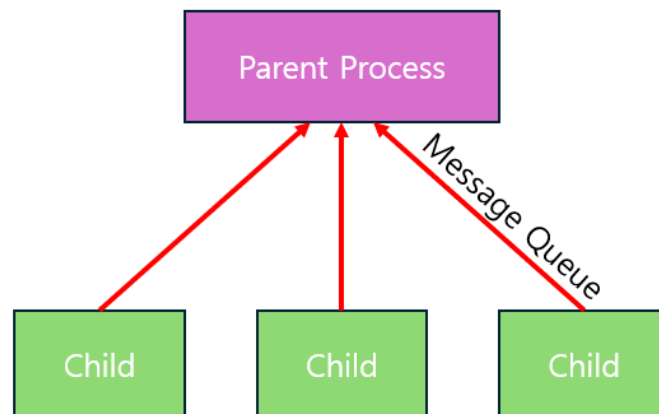
Contents

Introduction.....	3
Analyzing and planning.....	3.
Main explanation.....	5
Important concept.....	6.
Unique features.....	8.
Outro.....	11
Environment.....	11.
Screenshots.....	12.

Introduction

Analyzing and planning

Fork 를 통해 생성된 10 개의 자식 프로세스를 가지고 부모 프로세스의 Round Robin 스케줄링을 구현하기 위해서 프로세스 상태가 담긴 배열 Run-Queue, Wait-Queue 와 메시지 큐를 이용해야 한다. 자식 프로세스는 자신의 CPU 작업이 완전히 종료되었는지, 아니면 추가 실행이 필요한지 알리기 위해서 메시지 큐로 메시지를 보내고 부모는 항상 수신만 하며 시그널로 자식 프로세스를 스케줄링 한다.



Message Queue

1. 자식은 항상 자신의 CPU 사용이 종료되었다는 사실을 **메시지 큐로 송신**
2. 부모는 항상 **메시지 큐에서 수신**만 하며 해당 자식 프로세스를 스케줄링

Figure 1) Message Queue

자식 프로세스는 남은 CPU burst 와 IO burst 를 기준으로 Re-scheduling 되어 큐에 다시 배치된다. FIFO 구조의 큐는 이러한 스케줄링에 효과적으로 활용된다. 즉, 1) CPU burst 가 남아 Run-Queue 에 다시 들어간 프로세스와 2) CPU burst 를 모두 소진하고 IO burst 를 수행하기 위해 Wait-Queue 에 들어간 프로세스는 들어간 순서대로 큐에서 추출되어 부모 프로세스에 의해 처리된다.

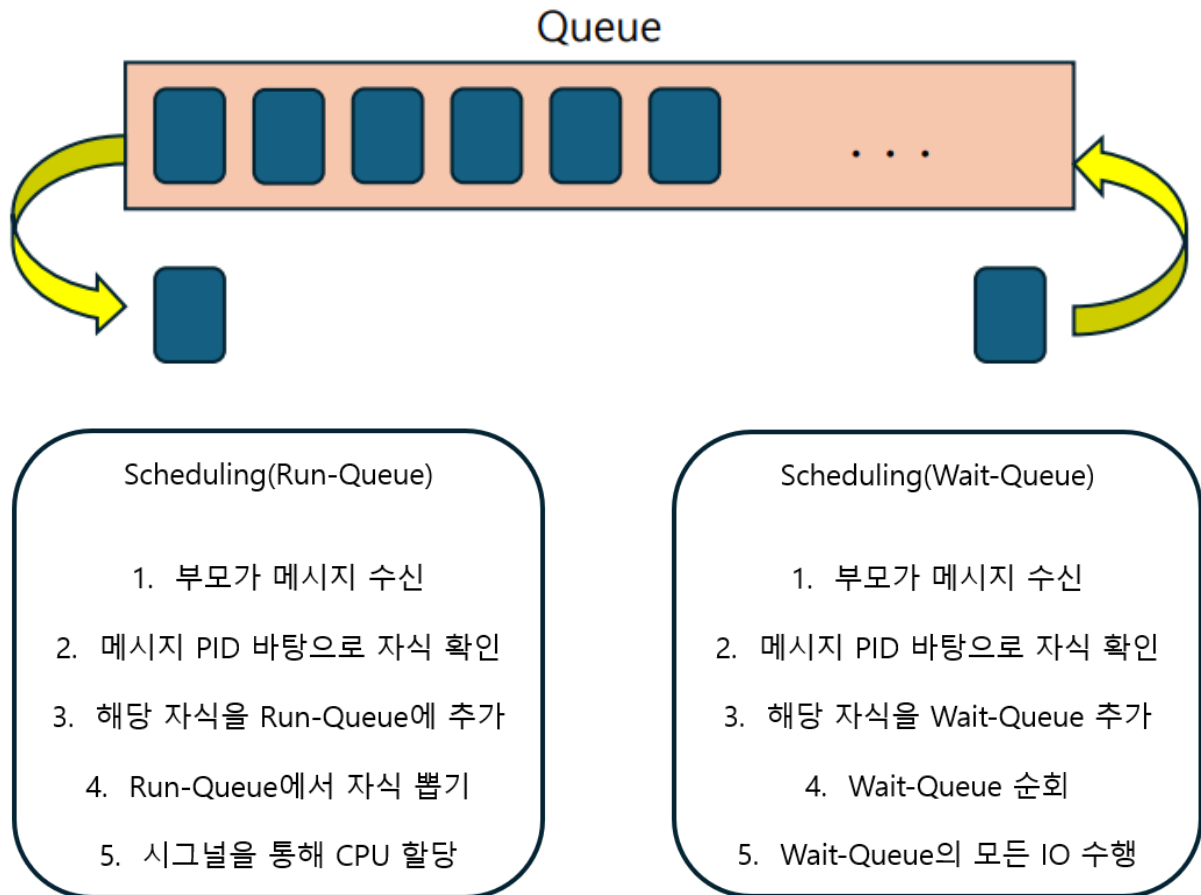


Figure 2) Queue structure

부모 프로세스와 자식 프로세스의 기능을 함수적으로 표현하면 다음과 같다.

	부모	자식
함수적 기능	Wait Queue IO	시그널 받은 후 메시지 전달
	MSG Queue process	
	Run Queue CPU signal	

부모 프로세스

1. Wait-Queue(IO)에서 모든 자식 프로세스의 IO 감소
2. 자식 프로세스가 보낸 메시지 처리
 - 자식 프로세스의 잔여 CPU Burst가 0보다 큰 경우
 - 자식 프로세스의 잔여 CPU Burst가 0보다 작은 경우
3. Run-Queue(CPU)에서 뽑은 자식 프로세스에게 시그널 전달

Figure 3) Parent/Child Process functional work

이미지를 통해 전체 워크 플로우를 표현하면 다음과 같다.

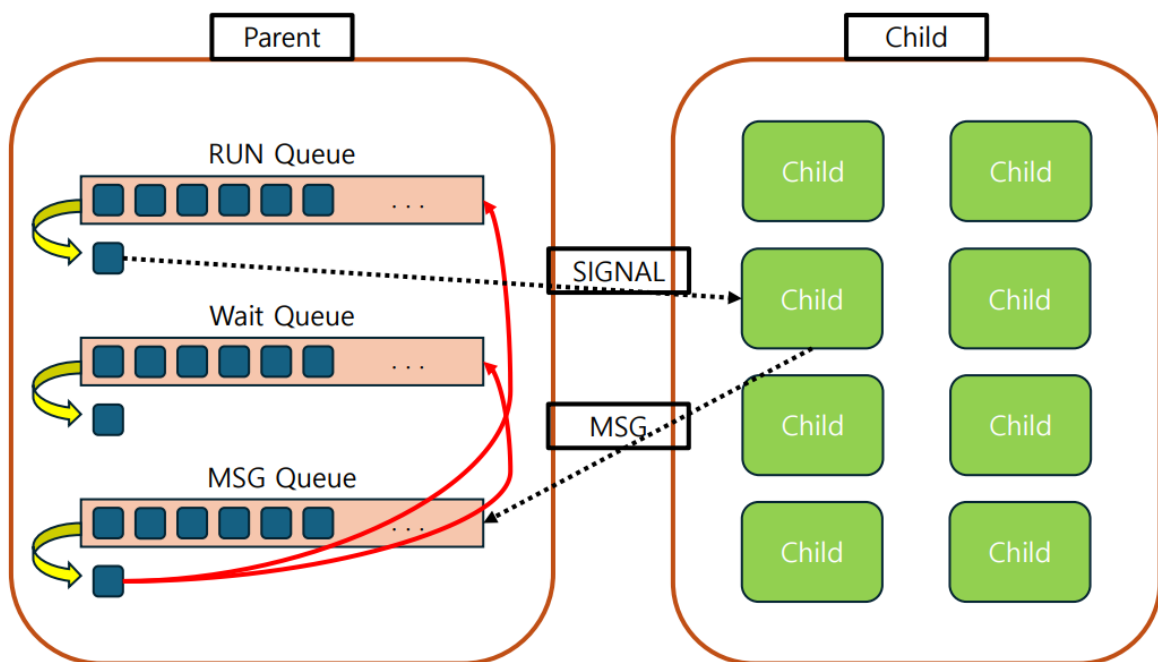


Figure 4) Overall workflow logic

Main explanation

Important concept

앞서 언급한 부모 프로세스는 다음과 같은 여러 동작으로 분리할 수 있다.
실제로 코드에 적용하기 전, 각 기능의 명확한 의미를 명시한다.

1. Wait Queue 처리

- 큐를 순회하며 들어있는 모든 프로세스의 IO 감소(Time Quantum)
- 큐의 앞을 가리키는 Front 기준으로, IO 가 완료되었다면 de-queue
- 현재 실행시간 기준
 - i. (1 분 이하인 경우) → CPU/IO Burst 값 재할당 후 en-queue
 - ii. (1 분 이상인 경우) → 프로세스가 종료된 것으로 간주하고 시그널 전송

2. MSG Queue 처리

- 큐가 비지 않을 때까지 NO WAIT 을 이용하여 연속적으로 수신
- PID 를 이용하여 프로세스 정보 확인(프로세스 구조체 불러오기)
- 잔여 CPU Burst 시간 기준
 - i. (0 이하인 경우) → Wait Queue 에 en-queue
 - ii. (0 이상인 경우) → Run Queue 에 en-queue

3. Run Queue 처리

- 큐에서 de-queue 를 한 뒤, 해당 프로세스 CPU 감소(Time Quantum)
- 해당 자식 프로세스에게 실행 허가 시그널 전송

```
// 2. 자식으로부터 메시지를 받아 처리
while (msgrcv(msgq_id, &msg, sizeof(MsgBuf) - sizeof(long), 1, IPC_NOWAIT) != -1) {
    printf("[Parent] Received message from child PID: %d\n", msg.pid);
    Process* current_process = NULL;
    for (int i=0; i<NUM_PROCESSES; i++) {
        if (processes[i]->pid == msg.pid) {
            current_process = processes[i];
            break;
        }
    }
}
```

Figure 5) Parent Process MSG Queue recieving

```
// 3. run queue 처리
if (!isEmpty(run_queue)) {
    Process* current_process = dequeue(run_queue);
    if (current_process != NULL) {
        current_process->remaining_cpu -= TIME_QUANTUM;
        current_process->remaining_cpu = (current_process->remaining_cpu <= 0) ?
        kill(current_process->pid, SIGUSR1);
    }
}
```

Figure 6) Parent Process sending SIGUSR1 to Child Process

신호를 보내려는 자식 프로세스의 PID 와 신호 종류를 이용해서 부모 프로세스는 두개의 시그널을 전송한다. 프로세스 종료 시그널 또는 CPU 사용 시그널을 받은 자식 프로세스는 1) **EXIT** 혹은 2) **프로세스 실행**을 하게 된다. 부모의 CPU 사용 허가 시그널에 맞춰 실행된 자식 프로세스는, 실행 완료를 메시지 큐를 이용해서 알리게 된다.

```
void execute_child_task(int pid) {
    // 부모에게 보낼 메시지 구성
    MsgBuf msg;
    printf("Child process %d CPU ing...\n", pid);
    msg.mtype = 1;
    msg.pid = pid;

    // 부모 프로세스로 상태 메시지 전송
    if (msgsnd(msgq_id, &msg, sizeof(MsgBuf) - sizeof(long), 0) == -1) {
        perror("msgsnd failed sending to parent process");
        exit(1);
    }
    // 실제로 해당 시간 동안 처리하는 것으로 가정
}
```

Figure 7) Child Process is recived signal and sending MSG

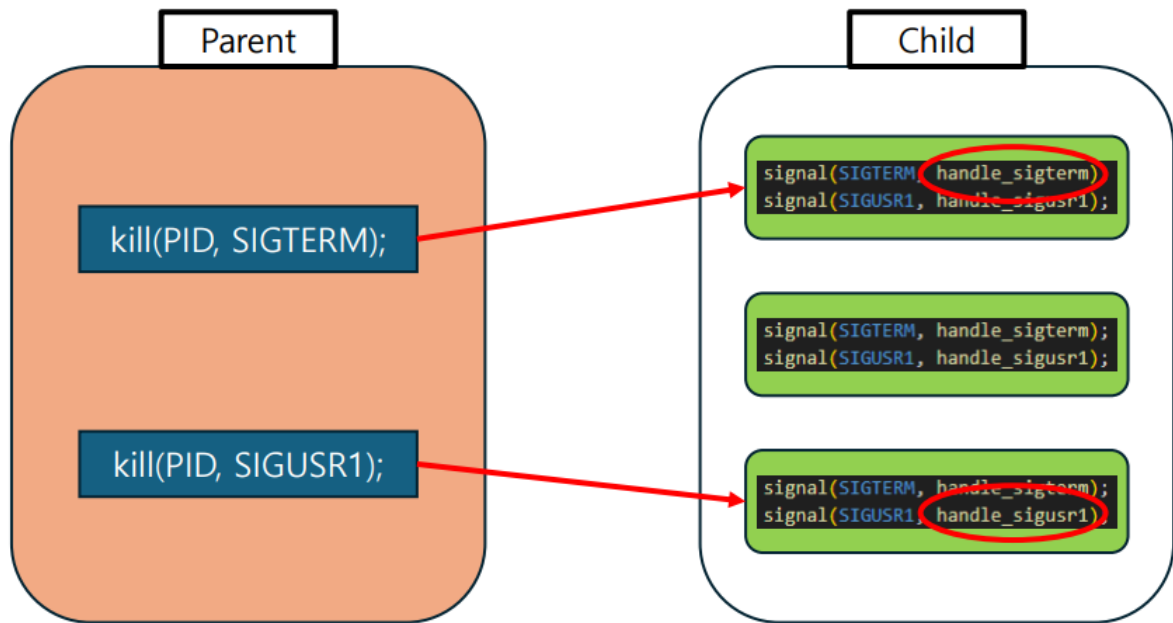


Figure 8) Parent-Child Signal send/receive

Unique features

기본적으로 Round Robin Scheduling 을 이용하지만, 미묘한 작업 수행 순서 차이로 제어 로직이 달라질 수 있다. 가령 Run-Queue 와 Wait-Queue 가 모두 비었을 경우 메시지 큐를 우선적으로 이용해 빈 큐들을 채워줘야 한다. 즉 Run-Queue 를 채울 수 있을 때 채우고, 이후 해당 큐에서 뽑은 자식 프로세스에게 CPU 타임 슬라이스를 할당한다. 부모 프로세스는 먼저 실행 중이던 자식 프로세스로부터 메시지를 전달받아 CPU 작업이 완전히 종료되었는지, 아니면 추가 실행이 필요한지를 판단하기 때문에, 상태 확인이 먼저 이루어진 후 Run-Queue 에서 다음 실행할 프로세스를 선택하여야 한다. 이와 같은 순차적인 처리 과정을 준수해야 정확하고 효율적인 CPU Scheduling 이 가능해진다.

각 자식 프로세스가 일정 시간 이상 실행되도록 다음과 같은 기준과 변수를 설정하였다. TIME QUANTUM 은 CPU/IO burst 시간을 감소시키는 단위로, TIME TICK 은 실제 시간을 나타낸다. 따라서 실제 시간에서의 TIME QUANTUM 은 이 두 값을 곱한 값으로 정의되며, current time 은 1 씩 증가하여 매 300ms(= TIME QUANTUM * TIME TICK)마다 갱신된다.

- TIME QUANTUM = 3 단위
- TIME TICK = 100ms
- Current time = 1 씩 증가하여 매 300ms 간격으로 실행

이렇게 300ms 단위로 실행되므로, 예를 들어 1 분을 실행하려면 minute 값을 $1\text{min}/300\text{ms} = 200$ 으로 설정하고, 10 분을 실행하려면 $10\text{min}/300\text{ms} = 2000$ 으로 설정하면 된다. current time 과 minute 값을 비교하여 프로세스를 다시 Run Queue 에 추가함으로써, 간단한 변수 값 변경을 통해 시뮬레이션 시간을 쉽게 조정할 수 있다.

```
front_process = dequeue(wait_queue);  
// 실행 시간을 길게 시뮬레이팅 하기 위해서 다시 큐 혹은 종료  
if (current_time < minute) {  
    front_process->cpu_burst = front_process->remaining_cpu = rand() % 10 + 1;  
    front_process->io_burst = front_process->remaining_io = rand() % 10 + 1;  
    front_process->cpu_use_time += front_process->cpu_burst;  
    enqueue(run_queue, front_process);  
}
```

Figure 9) Current time and Minute unit

추가적으로 Running time, Waiting time, CPU use time 은 이 300ms 간격에 기반하여 계산된다. Running Time 은 각 프로세스가 시스템에 도착한 시점부터 종료된 시점까지의 전체 시간을 의미하며, CPU와 IO 대기 시간을 포함한다. 마지막 IO가 완료된 프로세스에

대해 직전 current time 값을 기록하고, 이를 300ms 단위로 환산하여 실제 시간으로 변환한다.

```
front_process->running_time = current_time;
kill(front_process->pid, SIGTERM);
completed_processes++;
```

Figure 10) Running time calculation

Waiting Time 은 각 프로세스가 실행되기 전까지 CPU 를 할당받기 위해 기다린 총 시간이다. 즉, 프로세스가 CPU 에 접근할 수 없을 때의 누적된 시간을 의미한다. Waiting time 은 매번 부모 프로세스가 마지막에 연산을 하도록 하였다. 큐를 순회하며 프로세스 구조체의 waiting time 필드의 값을 1 단위씩 더해준다.

```
// 런 큐 웨이팅 타임 계산
int i = run_queue->front;
while (i != run_queue->rear) {
    Process* process = run_queue->processes[i];
    if (process != NULL) {
        process->waiting_time++;
    }
    i = (i + 1) % MAX_QUEUE_SIZE;
}
```

Figure 11) Waiting time calculation

CPU use Time 은 각 프로세스가 실제로 CPU 를 사용한 누적 시간을 의미한다. 이 값은 프로세스가 CPU 에서 작업하는 동안에만 증가하며, I/O 작업 및 대기 중에는 증가하지 않는다. CPU use time 은 CPU Burst 타임이 랜덤으로 지정될 때마다 더해주도록 하였다.

```
front_process->cpu_use_time += front_process->cpu_burst;
enqueue(run_queue, front_process);
```

Figure 12) CPU use time calculation

이 계산을 통해 프로그램의 대략적인 흐름을 파악할 수 있다. 실제 실행 결과를 보면 $\text{Waiting Time} + \text{CPU Use Time}$ 이 Running Time 보다 작게 나온다. 즉 프로세스가 IO 작업을 위해 Wait-Queue 에 머무르는 시간과 완전히 맞아 떨어지지 않는 약간의 오차들이 포함되었다고 볼 수 있다. 또한 비정상적으로 Waiting time 이 큰 값이 나오는데, 프로세스가 종료되자마자 메시지 큐를 통해 종료를 알리고, 이후 메시지 큐에서 일괄적으로 Run-Queue 로 들어가기 때문에 필연적으로 큰 값이 나올 수밖에 없다. 실제로 Schedule dump 파일을 살펴보면 매 시간마다 각 큐의 상태들이 출력되는데, Run-Queue 는 거의 항상 찬 상태를 유지하게 되고, Wait-Queue 는 상대적으로 적은 프로세스(3~5 개)들이 쌓이다가 일괄적으로 빠져나간다. 순회하며 동시에 IO 를 감소시키기에, 앞의 프로세스 IO 버스트가 0 이 되는 순간 연속적으로 de-queue 된다.

Outro

Environment

- Linux environment. Basically, Local Ubuntu with window and MAC.
- To Run code, Place the scheduling.c and scheduling.h files inside the same folder.
- Compiler it and execute out file.
- Finally, you will get a schedule dump which includes element status at each time.
 - Time tick, completed processes, Run-Queue status, Wait-Queue status.
 - Process movement (Run/Wait)

- Process CPU/IO burst shift
- Running time, waiting time, CPU use time

Screenshots

```
-----
Initialized status of elements
Timer tick: 0
Completed 0/10 processes
RunQ = [P_242433(9, 2), P_242434(5, 6), P_242435(9, 1), P_242436(7, 7), P_242437(2, 3), P_242438(4, 8), P_242439(2, 3), P_242440(8, 8), P_242441(5, 3), P_242442(6, 7)]
WaitQ = []
-----
```

Figure 13) Initialized status of elements

```
Timer tick: 215
Completed 8/10 processes
RunQ = []
WaitQ = [P_242437(0, 3), P_242434(0, 0)]
Process 242437: I/O burst decreased to 0
Process 242437 completed (total completed: 9)
Process 242434 completed (total completed: 10)

Timer tick: 216
Completed 10/10 processes
RunQ = []
WaitQ = []
```

Figure 14) End of CPU/IO Burst loop

-----	-----	-----
Final running time result	Final waiting time result	Final cpu use time result
Child[242433] : 63.600000s	Child[242433] : 50.100000s	Child[242433] : 5.800000s
Child[242434] : 64.800000s	Child[242434] : 51.600000s	Child[242434] : 5.900000s
Child[242435] : 63.000000s	Child[242435] : 50.100000s	Child[242435] : 5.300000s
Child[242436] : 63.000000s	Child[242436] : 49.800000s	Child[242436] : 5.500000s
Child[242437] : 64.800000s	Child[242437] : 50.100000s	Child[242437] : 5.700000s
Child[242438] : 61.800000s	Child[242438] : 50.100000s	Child[242438] : 5.300000s
Child[242439] : 60.600000s	Child[242439] : 45.600000s	Child[242439] : 4.900000s
Child[242440] : 64.500000s	Child[242440] : 51.900000s	Child[242440] : 5.500000s
Child[242441] : 60.900000s	Child[242441] : 47.400000s	Child[242441] : 5.100000s
Child[242442] : 63.900000s	Child[242442] : 50.400000s	Child[242442] : 5.500000s
-----	-----	-----

Figure 15) minute=200 result(1min)

```
-----  
Final running time result  
Child[86950] : 600.300000s  
Child[86951] : 604.200000s  
Child[86952] : 603.600000s  
Child[86953] : 601.500000s  
Child[86954] : 603.000000s  
Child[86955] : 601.500000s  
Child[86956] : 603.300000s  
Child[86957] : 604.200000s  
Child[86958] : 603.300000s  
Child[86959] : 600.900000s  
-----
```

```
-----  
Final waiting time result  
Child[86950] : 477.300000s  
Child[86951] : 471.300000s  
Child[86952] : 480.600000s  
Child[86953] : 477.900000s  
Child[86954] : 479.700000s  
Child[86955] : 474.900000s  
Child[86956] : 484.500000s  
Child[86957] : 488.400000s  
Child[86958] : 484.800000s  
Child[86959] : 478.500000s  
-----
```

```
-----  
Final cpu use time result  
Child[86950] : 49.800000s  
Child[86951] : 50.700000s  
Child[86952] : 50.400000s  
Child[86953] : 48.900000s  
Child[86954] : 51.200000s  
Child[86955] : 49.100000s  
Child[86956] : 49.900000s  
Child[86957] : 50.800000s  
Child[86958] : 49.700000s  
Child[86959] : 49.000000s  
-----
```

Figure 16) minute=2000 result(10min)