OS Project 1

Scheduling Simulator

모바일시스템공학과

32214362 조윤서

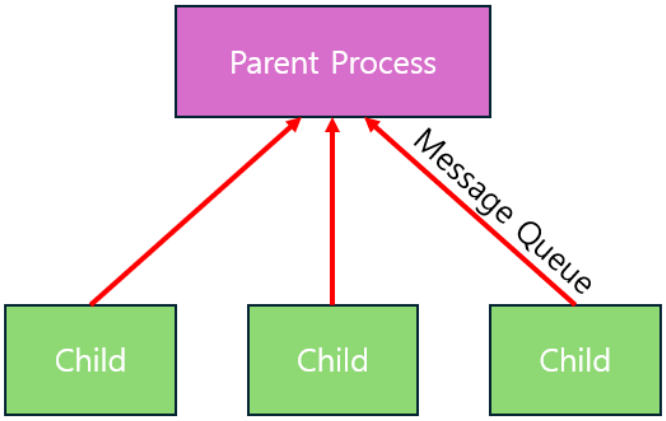
32204288 조민욱

Contents

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 종료를 알리기 위해 메시지 큐로 메시지를 보내고 부모는 항상 수신만 하며 자식 프로세스를 스케줄링 한다.



Message Queue

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

Figure 1) Message Queue

즉 자식 프로세스에 남은 CPU 버스트와 IO 버스트를 기준으로 다시 큐에 집어넣는 Re-scheduling 을 수행한다. 큐의 FIFO 특성은 스케줄링에 적극 활용 가능한 구조이다. **1) CPU 버스트가 남아 Run-Queue 에 들어간 프로세스**와 **2) CPU 를 모두 소진하고 IO 버스트를 수행할 Wait-Queue 에 들어간 프로세스**는 모두 들어간 순서대로 뽑혀 부모 프로세스에 의해 수행된다.

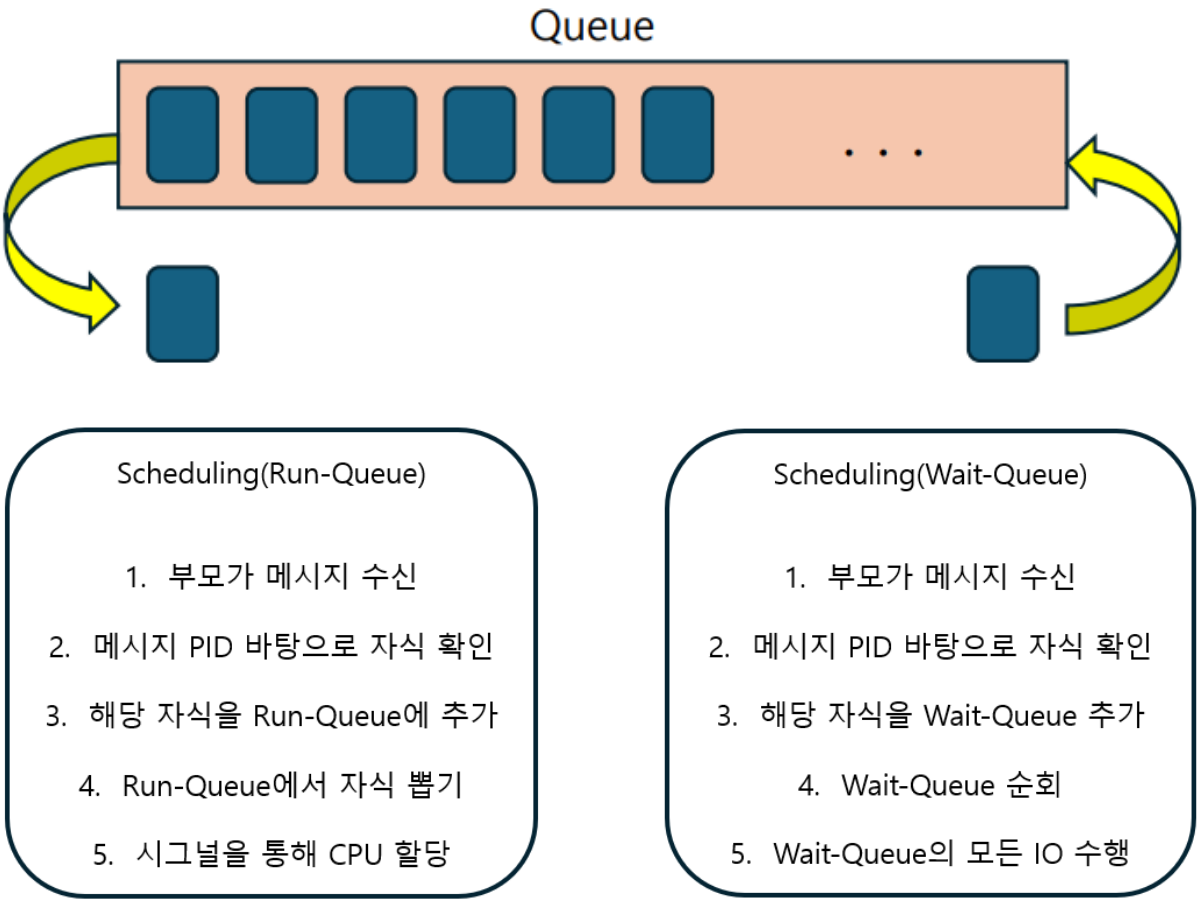


Figure 2) Queue structure

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.

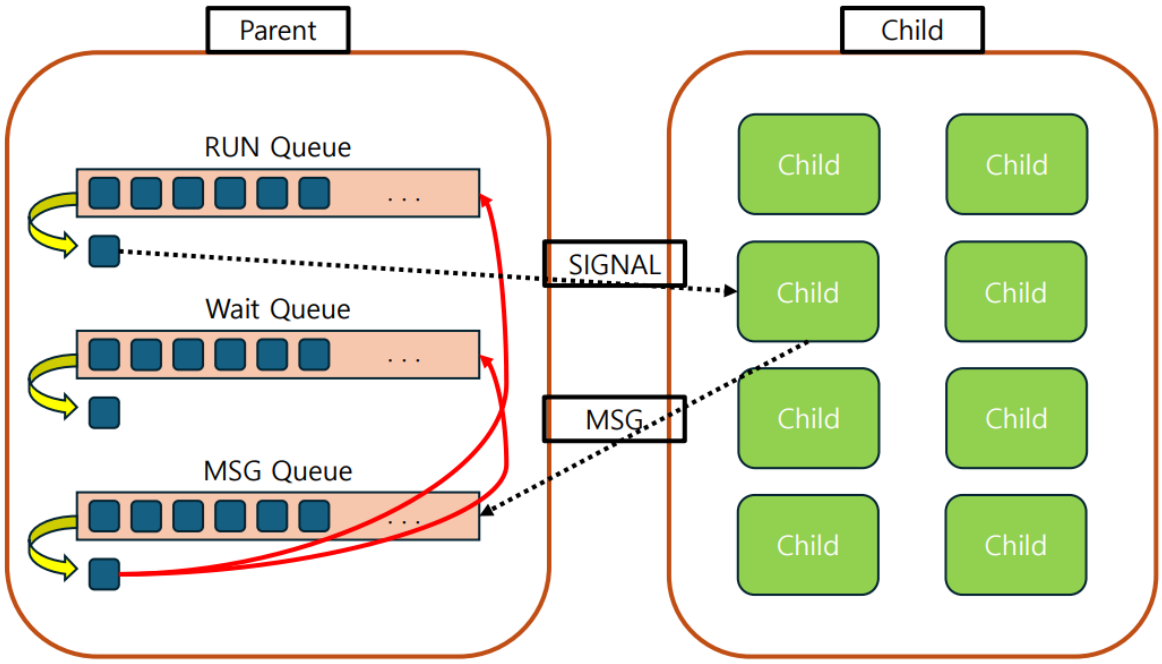


Figure 4) Overall workflow logic

Main explanation

Important concept

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

```c
// 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

```c
// 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.

```c
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

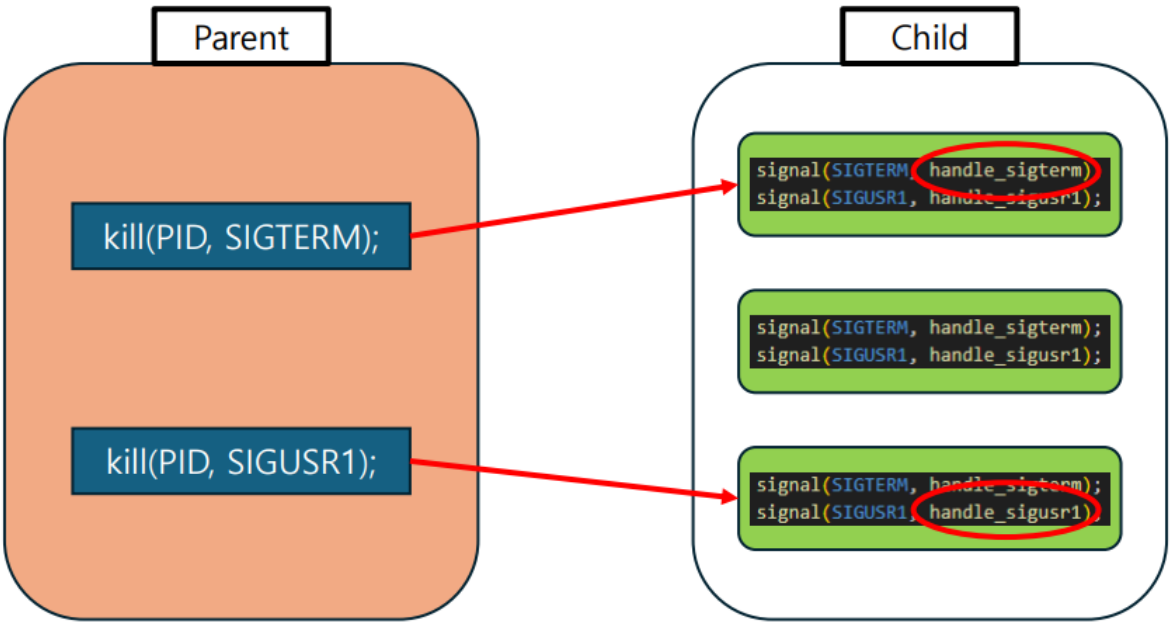


Figure 8) Parent-Child Signal send/receive

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 필요 작업 잔여 유무를 판단하기 때문에, 상태 확인이 먼저 이루어진 후 Run-Queue 에서 다음 실행할 프로세스를 적절히 선택하여야 한다. 실제로는 이런 순차적인 처리 과정이 상황에 맞게 바뀌어야 유연하며 효율적인 CPU Scheduling 이 가능할 것이다.

각 자식 프로세스가 일정 시간 이상을 돌아가게 하기 위해, 다음 기준을 세우고 각 변수를 이용하였다. TIME QUANTUM 을 CPU/IO burst 타임을 감소시키는 어떤 단위로 설정하고, TIME TICK 은 실제 시간으로 간주한다. 따라서 실제 시간에서의 TIME QUANTUM 값을 위 두 값의 곱으로 설정하고 current time 을 1 씩 증가하도록 하였다.

- TIME QUANTUM = 3 단위
- TIME TICK = 100ms
- Current time = 1 씩 증가 = 300ms 씩 증가 (TIME QUANTUM * TIME TICK)

즉 300ms 단위로 실행되기에 1 분을 실행하고 싶다면 minute 값을 1m/300ms=200, 10 분을 실행하고 싶다면 10m/300ms=2000 로 두면 된다. Current time 값과 minute 값을 비교하여 프로세스를 다시 Run-Queue 에 집어넣기에, 간단한 변수 값 변경으로 시뮬레이션 시간을 인위적으로 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

추가적인 Running time, Waiting time, CPU use time 계산은 이 시간 간격에 기반하여 구현되었다. Running Time 은 각 프로세스가 시스템에 도착한 시점부터 종료된 시점까지의 전체 시간을 의미한다. CPU 와 IO 에서 대기 시간을 포함한 전체 실행 시간을 나타내므로, 마지막 IO 까지 종료될 프로세스에 대해 직전 현재 시간을 넣도록 하였다. 추후 해당 시간에 300ms 단위를 곱해주어야 실제 시간이 나오게 된다.

```
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

위 계산을 통해 나온 결과를 통해 대략적인 프로그램 흐름을 판단할 수 있다. 실제 실행 결과를 보게되면 Waiting time + CPU use time 이 Running time 보다 작다. 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

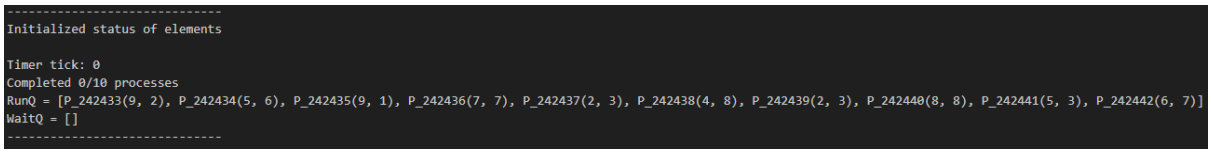


Figure 13) Initialized status of elements

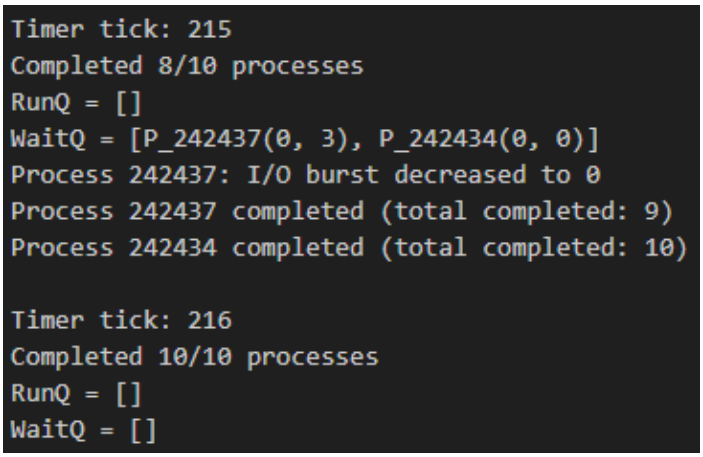


Figure 14) End of CPU/IO Burst loop

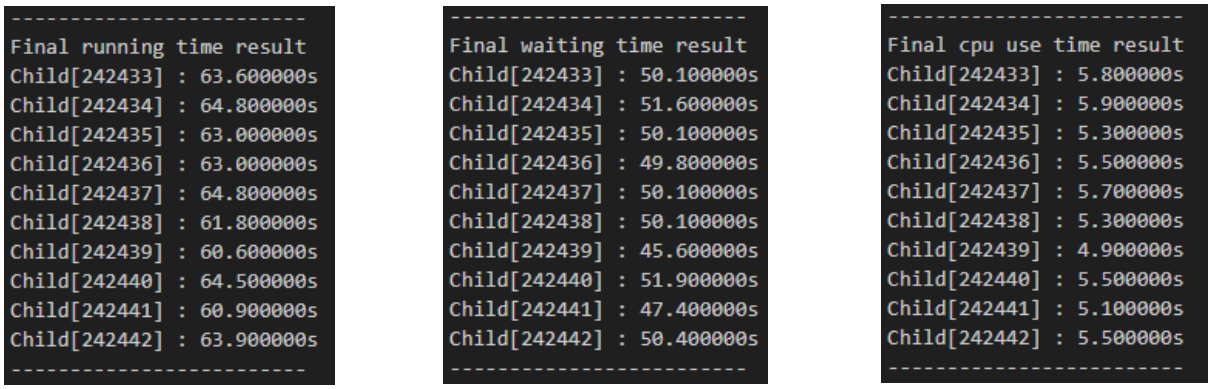
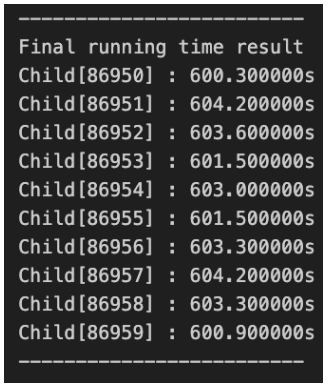
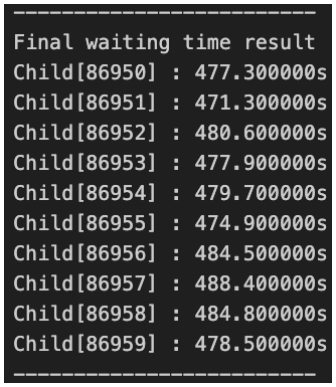


Figure 15) minute=200 result(1m)





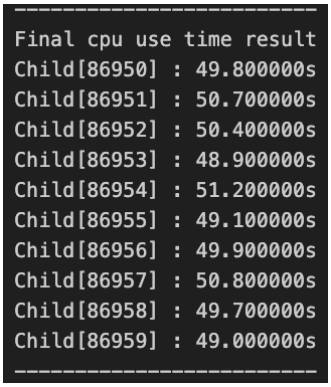


Figure 16) minute=2000 result(10m)