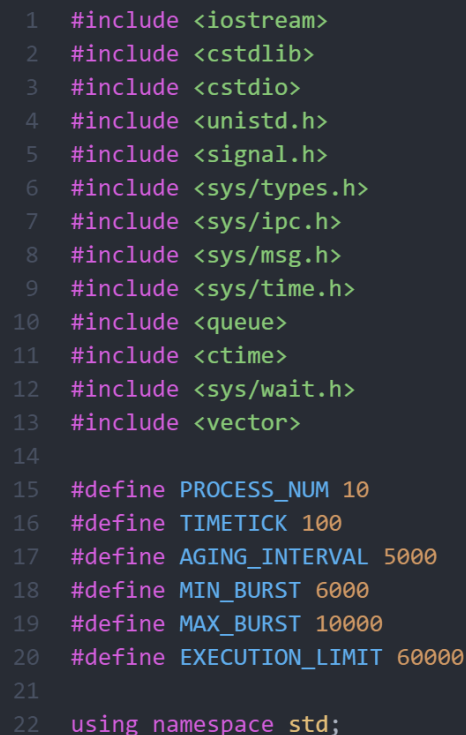


OS Term Project #1

2021310410 엄지우

1. 소스 코드

a. 헤더 파일



```
1  #include <iostream>
2  #include <cstdlib>
3  #include <cstdio>
4  #include <unistd.h>
5  #include <signal.h>
6  #include <sys/types.h>
7  #include <sys/ipc.h>
8  #include <sys/msg.h>
9  #include <sys/time.h>
10 #include <queue>
11 #include <ctime>
12 #include <sys/wait.h>
13 #include <vector>
14
15 #define PROCESS_NUM 10
16 #define TIMETICK 100
17 #define AGING_INTERVAL 5000
18 #define MIN_BURST 6000
19 #define MAX_BURST 10000
20 #define EXECUTION_LIMIT 60000
21
22 using namespace std;
```

- `#include <iostream>`: C++ 입출력 스트림 기능
- `#include <cstdlib>`: 일반적인 유틸리티 함수 (`malloc`, `free`, `rand` 등)
- `#include <cstdio>`: C 스타일 입출력 함수 (`printf`, `fprintf` 등)
- `#include <unistd.h>`: 유닉스 표준 함수 (`fork`, `sleep`, `getpid` 등)
- `#include <signal.h>`: 시그널 처리 기능 (`signal`, `SIGALRM` 등)
- `#include <sys/types.h>`: 데이터 유형 정의 (`pid_t`, `size_t` 등)
- `#include <sys/ipc.h>`: IPC(Inter-Process Communication) 기능 (`ftok`)
- `#include <sys/msg.h>`: 메시지 큐 기능 (`msgget`, `msgsnd`, `msgrcv`)
- `#include <sys/time.h>`: 시간 관련 기능 (`gettimeofday`, `itimerval`)

- `#include <queue>`: 큐 자료 구조 (queue)
- `#include <ctime>`: 시간 관련 함수 (time, srand)
- `#include <sys/wait.h>`: 프로세스 종료 대기 기능 (wait, waitpid)
- `#include <vector>`: 벡터 자료 구조 (vector)
- `#define PROCESS_NUM 10`: 생성할 자식 프로세스 수
- `#define TIMETICK 100`: 타임틱의 시간(ms)
- `#define AGING_INTERVAL 5000`: Aging이 일어나는 시간 간격(ms)
- `#define MIN_BURST 6000`: CPU 버스트의 최소 값(ms)
- `#define MAX_BURST 10000`: CPU 버스트의 최대 값(ms)
- `#define EXECUTION_LIMIT 60000`: 최대 실행 시간(ms)

b. 변수 및 구조체 정의

1) 전역 변수 선언

```
1 unsigned short process_count = 0;
2 unsigned int last_aging_time = 0;
3 FILE* outputFile;
4 int msgid;
5 int QUANTUM = 500;
```

- `unsigned short process_count`: 생성된 자식 프로세스의 수를 저장하는 변수
- `unsigned int last_aging_time`: 마지막으로 프로세스에 대해 에이징(aging)을 적용한 시간을 저장하는 변수. Aging은 CPU 시간을 할당받지 못한 프로세스가 우선순위를 높여 기아 상태에 빠지지 않도록 하기 위한 기법이다. 이 값은 마지막 aging이 발생한 시점을 기준으로 일정 시간 간격마다 aging을 수행한다.
- `FILE* outputFile`: 실행 결과를 저장할 파일을 가리키는 포인터
- `int msgid`: 메시지 큐의 ID를 저장하는 변수
- `int QUANTUM = 500`: 타임 슬라이스(퀀텀) 값으로, 프로세스가 CPU를 사용할 수 있는 최대 시간을 정의한다. 라운드 로빈 스케줄링에서 각 프로세스가 실행될 시간을 결정하며, 기본값은 500ms로 설정되어 있다. QUANTUM의 값을 조정하여 다양한 성능 테스트를 수행할 수 있다.

2) 구조체 정의

- Process 구조체

```
1 struct Process {
2     pid_t pid;
3     unsigned int cpu_burst;
4     unsigned int remaining_cpu_burst;
5     unsigned int io_burst = 0;
6     unsigned int remaining_io_burst = 0;
7     unsigned int io_start_time = 0;
8     unsigned int priority;
9     unsigned int response_time = 0;
10    unsigned int finish_time = 0;
11 };
```

각 프로세스의 세부 정보를 관리하기 위해 설계된 구조체이다. 이 구조체는 프로세스에 대한 다양한 정보를 저장하며, 프로세스의 생애 주기 동안 필요한 데이터를 기록한다. 주요 필드로는 pid가 있으며, 이는 각 프로세스를 고유하게 식별하는 데 사용된다. 또한, cpu_burst와 remaining_cpu_burst는 프로세스가 CPU를 사용해야 하는 시간과 남은 시간을 관리하는 데 사용된다. 이 정보는 프로세스가 CPU를 얼마나 오랫동안 사용할 수 있는지를 결정하는 데 중요한 역할을 한다. priority 필드는 프로세스의 우선순위를 나타내며, 라운드 로빈 스케줄링에서 중요한 요소로 작용한다. 이 외에도, 프로세스의 I/O 버스트 시간, 응답 시간, 종료 시간을 기록하는 필드도 포함되어 있다.

- Message 구조체

```
1 struct Message {
2     pid_t mtype;
3     Process process_info;
4 };
```

프로세스 간 통신을 위해 사용되는 구조체로, 두 개의 필드를 가지고 있다. mtype은 메시지를 송신할 대상 프로세스의 PID를 식별하는 데 사용되며, 메시지가 어떤 프로세스와 연관되는지를 결정하는 중요한 역할을 한다. process_info는 Process 구조체의 인스턴스로, 프로세스에 대한 모든 세부 정보를 담고 있다. 메시지 큐를 통해 각 프로세스의 상태 및 데이터를 교환할 수 있으며, 부모 프로세스는 자식 프로세스의 상태를 모니터링하고 제어할 수 있다.

● Statistics 구조체

```
1 struct Statistics {
2     unsigned int total_response_time = 0;
3     unsigned int avg_response_time = 0;
4     unsigned short context_switch = 0;
5     unsigned int total_execution_time = 0;
6     unsigned int total_turnaround_time = 0;
7     unsigned int avg_turnaround_time = 0;
8     vector<pid_t> completed_processes;
9
10    void calculateAverages(int process_count) {
11        if (process_count > 0) {
12            avg_response_time = total_response_time / process_count;
13            avg_turnaround_time = total_turnaround_time / process_count;
14        }
15    }
16
17    void printSummary(FILE* outputFile, int quantum) {
18        fprintf(outputFile, "<Statistics> (TIME QUANTUM: %d)\n", quantum);
19        fprintf(outputFile, "Process Count: %d\n", process_count);
20        fprintf(outputFile, "Average Response Time: %dms\n", avg_response_time);
21        fprintf(outputFile, "Average Turnaround Time: %dms\n", avg_turnaround_time);
22        fprintf(outputFile, "Context Switches: %hu\n", context_switch);
23        fprintf(outputFile, "Total Execution Time: %dms\n", total_execution_time);
24        fprintf(outputFile, "Completed Process Order: ");
25        for (pid_t pid : completed_processes) {
26            fprintf(outputFile, "%d ", pid);
27        }
28        fprintf(outputFile, "\n");
29    }
30 };
```

시스템의 성능 및 실행 결과를 요약하는 데 사용되는 구조체이다. 이 구조체는 시스템의 전체 응답 시간, 회전 시간, 문맥 교환 횟수와 같은 중요한 성능 지표를 계산하고 저장한다. `total_response_time`과 `total_turnaround_time`은 각각 모든 프로세스의 응답 시간과 회전 시간을 합산한 값으로, 전체 시스템의 효율성을 평가하는 데 사용된다. `avg_response_time`과 `avg_turnaround_time`은 각 프로세스의 평균 응답 시간과 회전 시간을 계산하여 시스템 성능을 평가하는 데 중요한 역할을 한다. 또한, `context_switch`는 문맥 교환 횟수를 기록하여, 시스템에서 발생한 문맥 교환의 빈도를 보여준다. `completed_processes`는 완료된 프로세스들의 PID를 저장하는 벡터로, 실행 순서를 확인할 수 있다. `calculateAverages` 함수는 평균 값을 계산하고, `printSummary` 함수는 실행 결과를 출력 파일에 기록하여 사용자가 시스템의 성능을 평가할 수 있도록 돕는다.

3) 큐 및 포인터 변수 정의

```
1 queue<Process*> ready_queues[3];
2 queue<Process*> io_wait_queue;
3 Process* current_process = nullptr;
4 Statistics statistics;
```

- queue<Process*> ready_queues[3]; 우선순위에 따라 프로세스를 분류하는 3개의 준비 큐를 저장하는 배열. 각 큐는 우선순위가 다른 프로세스를 대기시킨다.
- queue<Process*> io_wait_queue: I/O 작업을 기다리는 프로세스를 저장하는 큐. I/O 작업이 끝나면 해당 프로세스는 다시 준비 큐로 이동한다.
- Process* current_process = nullptr: 현재 실행 중인 프로세스를 가리키는 포인터.
- Statistics statistics: 시스템의 성능 통계를 저장하는 구조체로, 응답 시간, 회전 시간, 문맥 교환 횟수 등을 추적한다.

c. main 함수

```
1 int main(int argc, char *argv[]) {
2     if (argc == 2) {
3         char* endptr;
4         QUANTUM = strtol(argv[1], &endptr, 10);
5         if (*endptr != '\0') {
6             fprintf(stderr, "Invalid Time Quantum: %s\n", argv[1]);
7             exit(EXIT_FAILURE);
8         }
9     }
10
11     key_t key = ftok("/tmp", 'P');
12     msgid = msgget(key, IPC_CREAT | 0666);
13     if (msgid == -1) {
14         perror("msgget failed");
15         exit(EXIT_FAILURE);
16     }
17
18     outputFile = fopen("schedule_dump.txt", "w");
19
20     printf("Time Quantum set to: %d\n", QUANTUM);
21     fprintf(outputFile, "Time Quantum set to: %d\n\n", QUANTUM);
22
23     initializeChildProcesses();
24
25     int current_time = 0;
26     int quantum_timer = 0;
27
28     struct sigaction sa;
29     sa.sa_handler = signalHandlerStartProcess;
30     sa.sa_flags = SA_RESTART;
31     sigaction(SIGALRM, &sa, nullptr);
32
33     struct itimerval timer;
34     timer.it_interval.tv_sec = 0;
35     timer.it_interval.tv_usec = TIMETICK * 1000;
36     timer.it_value = timer.it_interval;
37     setitimer(ITIMER_REAL, &timer, nullptr);
38
39     struct timeval start_time, now;
40     gettimeofday(&start_time, nullptr);
```

```

42 while (!(current_process == nullptr && ready_queues[0].empty() && ready_queues[1].empty()
43        && ready_queues[2].empty() && io_wait_queue.empty())) {
44     gettimeofday(&now, nullptr);
45     int elapsed_time = (now.tv_sec - start_time.tv_sec) * 1000 + (now.tv_usec - start_time.tv_usec) / 1000;
46     Message msg;
47     msgrcv(msgid, &msg, sizeof(msg.process_info), 0, IPC_NOWAIT);
48
49     if (elapsed_time % TIMETICK == 0) {
50         processReadyQueues(quantum_timer, current_time);
51         processIoWaitQueue(quantum_timer, current_time);
52
53         displaySystemState(current_time);
54         fflush(outputFile);
55
56         if (!(current_process == nullptr && ready_queues[0].empty() && ready_queues[1].empty()
57                && ready_queues[2].empty() && io_wait_queue.empty())) {
58             current_time += TIMETICK;
59         }
60     }
61     usleep(TIMETICK * 10);
62 }
63
64 statistics.total_execution_time = current_time + statistics.context_switch;
65 statistics.calculateAverages(process_count);
66 statistics.printSummary(outputFile, QUANTUM);
67 fclose(outputFile);
68
69 msgctl(msgid, IPC_RMID, nullptr);
70 return 0;
71 }

```

프로그램의 전반적인 흐름을 관리하는 역할을 한다. 함수의 첫 번째 부분은 커맨드 라인 인수(argc, argv)를 처리하여, 사용자가 제공한 Time Quantum 값을 설정한다. 만약 두 번째 인수에 잘못된 값이 입력되면 오류 메시지를 출력하고 프로그램을 종료한다. 이렇게 설정된 Time Quantum 값은 이후 프로세스 스케줄링에 중요한 역할을 한다.

다음으로, 메시지 큐를 초기화한다. ftok 함수를 사용하여 고유한 키를 생성하고, 이를 통해 msgget 함수로 메시지 큐를 생성한다. 메시지 큐는 자식 프로세스들과의 통신을 담당하며, 생성된 메시지 큐의 ID는 msgid 변수에 저장된다. 만약 메시지 큐 생성에 실패하면, 오류 메시지를 출력하고 프로그램을 종료한다.

출력 파일을 준비하는 부분에서는 schedule_dump.txt라는 파일을 열어 실행 결과를 기록할 준비를 한다. Quantum 값이 출력 파일에 기록되며, 이는 스케줄링에서 각 프로세스가 실행되는 최대 시간(타임 슬라이스)을 정의한다. 이후 initializeChildProcesses 함수를 호출하여 자식 프로세스를 초기화하고, 프로세스와의 통신을 설정한다.

타이머 및 시그널 처리 부분에서는 sigaction을 이용해 SIGALRM 시그널을 처리하도록 설정한다. SIGALRM 시그널은 주기적으로 타이머가 발생할 때 프로세스 스케줄링을 실행하는 데 사용된다. 이를 위해 setitimer로 타이머를 설정하여 일정 간격(TIMETICK)마다 시스템 상태를 갱신하고, 프로세스들을 스케줄링 한다.

주요 스케줄링 루프에서는 while문을 사용해 프로세스의 상태를 주기적으로 점검한다. 각 프로세스는 준비 큐(ready_queues)와 I/O 대기 큐(io_wait_queue)에서 관리되며, 타이머 간격에 맞춰 큐를 처리하고 시스템의 상태를 출력 파일에 기록한다. 이때 메시지

큐로부터 프로세스 정보를 수신하여 프로세스를 실행하거나 대기 상태로 유지한다.

프로세스가 모두 종료되면, 프로그램은 총 실행 시간과 문맥 전환 횟수 등의 통계 정보를 계산하여 출력한다. calculateAverages와 printSummary 함수는 평균 응답 시간, 평균 대기 시간 등의 통계를 출력하여 실험 결과를 요약한다.

마지막으로, 프로그램은 자원을 정리한다. msgctl을 사용하여 메시지 큐를 제거하고, fclose로 출력 파일을 닫는다.

d. displaySystemState 함수

```
1 void displaySystemState(int time) {
2     fprintf(outputFile, "TIME: %dms\n", time);
3     if (current_process != nullptr) {
4         fprintf(outputFile, "Running: %d (CPU: %dms, I/O: %dms, I/O start: %dms)\n",
5             current_process->pid, current_process->remaining_cpu_burst,
6             current_process->remaining_io_burst, current_process->io_start_time);
7     } else {
8         fprintf(outputFile, "Running: -\n");
9     }
10    fprintf(outputFile, "<Ready Queues>");
11    for (int i = 0; i < 3; i++) {
12        fprintf(outputFile, "\nQueue %d: ", i);
13        queue<Process*> temp = ready_queues[i];
14        while (!temp.empty()) {
15            fprintf(outputFile, "%d (CPU: %dms, I/O: %dms) ",
16                temp.front()->pid, temp.front()->remaining_cpu_burst, temp.front()->remaining_io_burst);
17            temp.pop();
18        }
19    }
20    fprintf(outputFile, "\n");
21    fprintf(outputFile, "Waiting: ");
22    queue<Process*> temp_wait = io_wait_queue;
23    while (!temp_wait.empty()) {
24        fprintf(outputFile, "%d ", temp_wait.front()->pid);
25        temp_wait.pop();
26    }
27    fprintf(outputFile, "\n\n");
28 }
```

시스템의 현재 상태를 출력 파일에 기록하는 역할을 한다. 주어진 time 매개변수를 통해 현재 시간을 밀리초(ms) 단위로 표시하고, 현재 실행 중인 프로세스(current_process)의 상태를 출력한다. 실행 중인 프로세스가 있다면, 해당 프로세스의 ID(pid), 남은 CPU 버스트 시간(remaining_cpu_burst), 남은 I/O 버스트 시간(remaining_io_burst), 그리고 I/O 시작 시간(io_start_time)을 출력한다. 만약 실행 중인 프로세스가 없다면, "Running: -"를 출력하여 현재 실행 중인 프로세스가 없음을 표시한다.

다음으로, 준비 큐(ready_queues)의 상태를 출력한다. 준비 큐는 세 개의 우선순위로 나뉘어 있으며, 각 큐의 상태를 순차적으로 출력한다. 각 큐의 상태에서는 큐에 있는 각 프로세스의 ID와 해당 프로세스의 CPU 버스트 및 I/O 버스트 시간을 출력한다. 큐에 있는 프로세스는 출력 후 큐에서 제거된다.

마지막으로, I/O 대기 큐(io_wait_queue)의 상태를 출력한다. 큐에 있는 모든 프로세스의 ID를 출력하며, 큐에 프로세스가 없으면 출력이 생략된다.

이 함수는 시스템의 각 요소(실행 중인 프로세스, 준비 큐, I/O 대기 큐)의 상태를 주기적으로 기록하여, 스케줄링 과정에서 발생하는 이벤트들을 확인할 수 있게 한다.

e. processReadyQueues 함수

```
1 void processReadyQueues(int& quantum_timer, int& current_time) {
2     printf("Quantum Timer: %d, QUANTUM: %d\n", quantum_timer, QUANTUM);
3     if (current_process != nullptr) {
4         current_process->remaining_cpu_burst -= TIMETICK;
5
6         quantum_timer += TIMETICK;
7
8         if (current_process->remaining_cpu_burst <= 0) {
9             current_process->finish_time = current_time;
10            statistics.total_turnaround_time += current_process->finish_time;
11            statistics.completed_processes.push_back(current_process->pid);
12
13            printf("Process Completed (PID: %d)\n", current_process->pid);
14            fprintf(outputFile, "---- Process Completed (PID: %d) ----\n\n", current_process->pid);
15
16            current_process = nullptr;
17            statistics.context_switch++;
18        } else if (current_process->remaining_io_burst > 0 &&
19            (quantum_timer == current_process->io_start_time || current_process->io_start_time == 0)) {
20            io_wait_queue.push(current_process);
21            current_process = nullptr;
22            quantum_timer = 0;
23        }
24    }
25
26    if (current_process == nullptr || quantum_timer >= QUANTUM) {
27        if (current_process != nullptr && current_process->remaining_cpu_burst > 0) {
28            ready_queues[current_process->priority].push(current_process);
29        }
30
31        for (int i = 0; i < 3; i++) {
32            if (!ready_queues[i].empty()) {
33                current_process = ready_queues[i].front();
34                ready_queues[i].pop();
35                if (rand() % 3 == 0) {
36                    current_process->io_burst = (rand() % ((2000 - 1000) / 100 + 1) * 100) + 1000;
37                    current_process->remaining_io_burst = current_process->io_burst;
38                    current_process->io_start_time = (rand() % (QUANTUM / 100)) * TIMETICK;
39                }
40                quantum_timer = 0;
41                if (current_process->response_time == 0) {
42                    current_process->response_time = current_time;
43                    statistics.total_response_time += current_process->response_time;
44                }
45                statistics.context_switch++;
46                Message new_msg;
47                new_msg.mtype = current_process->pid;
48                new_msg.process_info = *current_process;
49                msgsnd(msgid, &new_msg, sizeof(new_msg.process_info), 0);
50                break;
51            }
52        }
53    }
```



```

55     unsigned int now_time = current_time;
56     if (now_time - last_aging_time >= AGING_INTERVAL) {
57         last_aging_time = now_time;
58         fprintf(outputFile, "----- Aging Occurred -----\\n\\n");
59         while (!ready_queues[1].empty()) {
60             Process* aged_process = ready_queues[1].front();
61             ready_queues[1].pop();
62             ready_queues[0].push(aged_process);
63         }
64         while (!ready_queues[2].empty()) {
65             Process* aged_process = ready_queues[2].front();
66             ready_queues[2].pop();
67             ready_queues[1].push(aged_process);
68         }
69     }
70 }

```

라운드로빈(Round-Robin) 스케줄링 알고리즘에서 준비 큐와 관련된 작업을 처리하는 핵심 함수이다. 이 함수는 각 타임틱마다 실행되어 프로세스의 CPU 버스트, I/O 버스트, 그리고 프로세스 상태를 관리한다. 함수의 주요 흐름은 다음과 같다.

먼저, 현재 실행 중인 프로세스가 있을 경우, 해당 프로세스의 남은 CPU 버스트(remaining_cpu_burst)를 타임틱만큼 차감한다. 타임틱에 해당하는 시간만큼 quantum_timer도 증가시킨다. 만약 현재 프로세스의 CPU 버스트가 0 이하로 떨어지면 해당 프로세스가 완료된 것으로 간주하고, 완료된 프로세스를 출력 파일에 기록한다. 또한, 완료된 프로세스의 종료 시간을 계산하여 statistics.total_turnaround_time에 더하고, 프로세스의 PID를 statistics.completed_processes에 추가한다. 이 후, 현재 프로세스는 nullptr로 설정되며, 문맥 교환이 발생하여 statistics.context_switch가 증가한다.

프로세스가 아직 종료되지 않았고, I/O 버스트 시간이 남아 있다면, 해당 프로세스를 I/O 대기 큐(io_wait_queue)로 이동시키고, current_process를 nullptr로 설정하여 새로운 프로세스를 준비할 수 있도록 한다. 이 경우, 타임틱에 맞춰 quantum_timer도 초기화된다.

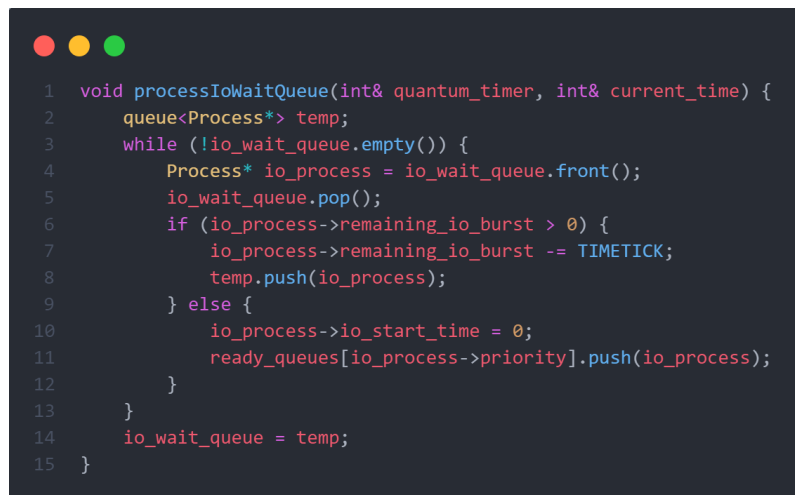
다음으로, 프로세스가 nullptr이거나 quantum_timer가 설정된 타임 슬라이스(QUANTUM)를 초과하면, 준비 큐에서 새로운 프로세스를 선택한다. 준비 큐는 우선순위에 따라 처리되며, 프로세스가 할당된 후에는 quantum_timer를 초기화하고, 만약 프로세스가 처음 실행되는 경우 응답 시간을 기록한다. 새로운 프로세스의 정보는 메시지 큐를 통해 다른 프로세스나 시스템에 전송된다.

마지막으로, AGING_INTERVAL 시간이 경과하면 프로세스 우선순위를 조정하는 aging 기법이 실행됩니다. 준비 큐에서 높은 우선순위 큐에 있는 프로세스를 낮은 우선순위 큐로 이동시키며, 이를 통해 CPU 자원을 할당 받지 못한 프로세스가 기아 상태에 빠지지 않도록 방지한다. Aging이 발생한 경우, 이를 출력 파일에 기록하여 해당 이벤트가 발생했음을 알려준다.

이 함수는 시스템의 동적인 상태를 실시간으로 관리하며, 프로세스 스케줄링을 효과

적으로 처리하기 위해 여러 큐를 활용하고 있다.

f. processIoWaitQueue 함수



```
1 void processIoWaitQueue(int& quantum_timer, int& current_time) {
2     queue<Process*> temp;
3     while (!io_wait_queue.empty()) {
4         Process* io_process = io_wait_queue.front();
5         io_wait_queue.pop();
6         if (io_process->remaining_io_burst > 0) {
7             io_process->remaining_io_burst -= TIMETICK;
8             temp.push(io_process);
9         } else {
10             io_process->io_start_time = 0;
11             ready_queues[io_process->priority].push(io_process);
12         }
13     }
14     io_wait_queue = temp;
15 }
```

I/O 대기 큐에서 대기 중인 프로세스들의 상태를 처리하는 함수이다. 이 함수는 각 프로세스의 I/O 버스트를 감소시키고, I/O 작업이 완료된 프로세스를 다시 준비 큐로 이동시키는 역할을 한다.

먼저, 임시 큐(temp)를 생성하여, I/O 대기 큐에 있는 프로세스를 하나씩 확인한다. 각 프로세스의 남은 I/O 버스트(remaining_io_burst)가 0보다 큰 경우, 해당 프로세스의 I/O 작업을 계속 진행할 수 있도록 I/O 버스트 시간을 TIMETICK만큼 감소시킨다. 이 프로세스는 임시 큐에 다시 추가된다.

반대로, I/O 버스트가 0 이하로 남아있는 프로세스는 I/O 작업이 완료된 것으로 간주하고, 프로세스를 준비 큐로 이동시킨다. 이때, I/O 시작 시간(io_start_time)을 0으로 설정하여 I/O 작업이 끝났음을 표시하고, 해당 프로세스를 우선순위에 맞는 준비 큐에 추가한다.

마지막으로, io_wait_queue는 임시 큐(temp)로 갱신된다. 이를 통해 I/O 대기 중이던 프로세스들의 상태를 처리하고, I/O가 끝난 프로세스는 준비 큐로 이동하여 CPU 자원을 기다릴 수 있게 된다.

이 함수는 시스템에서 프로세스의 I/O 작업을 관리하며, CPU와 I/O 자원 간의 균형을 맞추는데 중요한 역할을 한다.

g. initializeChildProcesses 함수

```
1 void initializeChildProcesses() {
2     for (int i = 0; i < PROCESS_NUM; i++) {
3         Process* new_process = new Process;
4         new_process->pid = fork();
5         if (new_process->pid < 0) {
6             perror("fork failed");
7             exit(EXIT_FAILURE);
8         }
9         else if (new_process->pid == 0) {
10             srand(getpid());
11             Message msg;
12             while (true) {
13                 if (msgrcv(msgid, &msg, sizeof(msg.process_info), getpid(), 0) != -1) {
14                     Process* received_process = &msg.process_info;
15
16                     if (received_process->remaining_cpu_burst > 0) {
17                         usleep(TIMETICK * 1000);
18
19                         received_process->remaining_cpu_burst -= TIMETICK;
20                         if (received_process->remaining_cpu_burst <= 0) {
21                             received_process->io_burst = (rand() % ((2000 - 1000) / 100 + 1) * 100) + 1000;
22                             received_process->remaining_io_burst = received_process->io_burst;
23                             msg.mtype = getpid();
24                             msgsnd(msgid, &msg, sizeof(msg.process_info), 0);
25                         }
26                         else {
27                             msg.mtype = getpid();
28                             msgsnd(msgid, &msg, sizeof(msg.process_info), 0);
29                         }
30                     }
31                     if (received_process->remaining_cpu_burst <= 0 && received_process->remaining_io_burst > 0) {
32                         usleep(received_process->remaining_io_burst * 1000);
33                         received_process->remaining_io_burst = 0;
34                         msg.mtype = getpid();
35                         msgsnd(msgid, &msg, sizeof(msg.process_info), 0);
36                     }
37                 }
38             }
39         }
40         new_process->cpu_burst = (rand() % ((MAX_BURST - MIN_BURST) / 100 + 1) * 100) + MIN_BURST;
41         new_process->remaining_cpu_burst = new_process->cpu_burst;
42         new_process->priority = i % 3;
43         ready_queues[new_process->priority].push(new_process);
44         Message msg;
45         msg.mtype = new_process->pid;
46         msg.process_info = *new_process;
47         process_count++;
48         msgsnd(msgid, &msg, sizeof(msg.process_info), 0);
49     }
50 }
```

자식 프로세스를 생성하고, 각 프로세스에 대해 초기 설정을 수행하는 역할을 한다. 이 함수는 전체 프로세스 수(PROCESS_NUM)만큼 자식 프로세스를 생성하여 준비 큐에 삽입하고, 각 프로세스에 대한 정보를 메시지 큐를 통해 부모 프로세스와 공유한다.

먼저, 함수는 PROCESS_NUM만큼 반복문을 실행하여 자식 프로세스를 생성한다. 각 자식 프로세스는 fork() 시스템 호출을 통해 생성되며, 자식 프로세스는 pid == 0인 경우에만 while (true) 루프를 통해 무한히 실행된다. fork() 호출 후 자식 프로세스에서는 무한 루프 내에서 메시지 큐에서 자신에게 전송된 메시지를 수신하고, 프로세스의 상태를 업데이트하며 부모 프로세스에게 결과를 다시 전송한다.

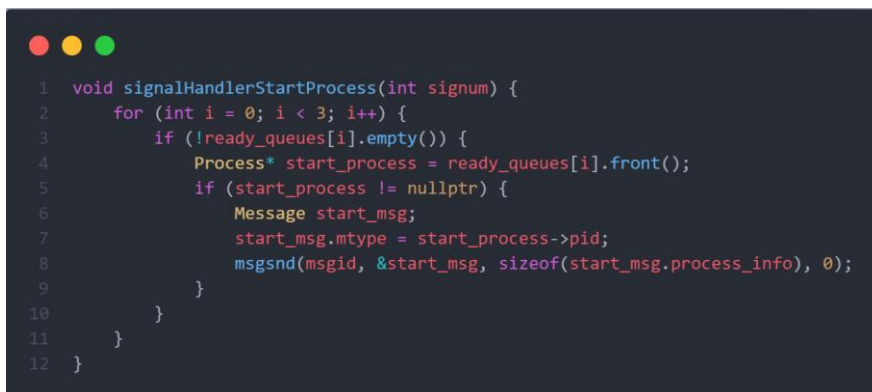
각 자식 프로세스는 메시지 큐에서 수신한 프로세스 정보를 바탕으로

remaining_cpu_burst 값을 감소시킨다. CPU 버스트가 끝나면, 해당 프로세스는 I/O 버스트를 시작하고, 이를 메시지 큐를 통해 부모 프로세스에게 알린다. 또한 I/O가 완료되면 다시 CPU 버스트를 진행할 수 있도록 부모에게 알린다.

부모 프로세스는 자식 프로세스들의 생성 시 rand() 함수를 사용해 CPU 버스트를 랜덤하게 설정하고, 프로세스의 우선순위(priority)도 0, 1, 2로 나누어 각 준비 큐에 삽입한다. 각 프로세스의 PID와 상태 정보는 Message 구조체에 담아 메시지 큐를 통해 부모 프로세스로 전송된다. 이때 프로세스의 정보는 자식 프로세스가 실행될 때마다 갱신된다.

이 함수는 시스템 시작 시 모든 자식 프로세스를 초기화하고, 준비 큐에 넣어 라운드-robin 스케줄링을 시작할 준비를 완료한다. process_count는 생성된 프로세스의 수를 추적하고, 메시지 큐를 통해 부모와 자식 프로세스 간의 정보를 전달하는 핵심적인 역할을 한다.

h. signalHandlerStartProcess 함수



```
1 void signalHandlerStartProcess(int signum) {
2     for (int i = 0; i < 3; i++) {
3         if (!ready_queues[i].empty()) {
4             Process* start_process = ready_queues[i].front();
5             if (start_process != nullptr) {
6                 Message start_msg;
7                 start_msg.mtype = start_process->pid;
8                 msgsnd(msgid, &start_msg, sizeof(start_msg.process_info), 0);
9             }
10        }
11    }
12 }
```

신호를 처리하는 함수로, 프로세스의 시작을 제어하는 역할을 한다. 이 함수는 SIGALRM 시그널이 발생했을 때 호출된다.

함수의 동작은 다음과 같다:

1. 우선순위 큐 순회: 먼저 준비 큐(ready_queues)에 들어있는 프로세스를 확인한다. 준비 큐는 3개의 우선순위 레벨(0, 1, 2)로 나뉘어 있으며, 각 큐에는 실행 대기 중인 프로세스가 저장되어 있다.
2. 프로세스 시작: 각 우선순위 큐에서 첫 번째 프로세스를 꺼내(front()), 해당 프로세스가 유효한지 확인한다. 유효한 프로세스가 있으면, Message 구조체를 만들어 해당 프로세스에 시작 메시지를 전송한다. 이 메시지는 start_msg.mtype = start_process->pid;로 프로세스의 PID를 설정한 후 메시지 큐(msgid)를 통해 부모 프로세스에게 전달된다.

3. 메시지 전송: 메시지가 전송되면, 해당 프로세스는 CPU에서 실행을 시작할 수 있다. 프로세스는 부모 프로세스가 제공한 타임 슬라이스(QUANTUM)에 따라 실행된다.

이 함수는 라운드로빈 스케줄링의 실행을 시작하는 중요한 부분으로, 준비 큐에서 프로세스를 하나씩 실행시키는 역할을 한다. 각 프로세스는 CPU에서 실행될 준비가 되면 시작 메시지를 받아 실행을 시작한다.

i. 전체 코드

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <cstdio>
4  #include <unistd.h>
5  #include <signal.h>
6  #include <sys/types.h>
7  #include <sys/ipc.h>
8  #include <sys/msg.h>
9  #include <sys/time.h>
10 #include <queue>
11 #include <ctime>
12 #include <sys/wait.h>
13 #include <vector>
14
15 #define PROCESS_NUM 10
16 #define TIMETICK 100
17 #define AGING_INTERVAL 5000
18 #define MIN_BURST 6000
19 #define MAX_BURST 10000
20 #define EXECUTION_LIMIT 60000
21
22 using namespace std;
23
24 unsigned short process_count = 0;
25 unsigned int last_aging_time = 0;
26 FILE* outputFile;
27 int msgid;
28 int QUANTUM = 500;
29
30 struct Process {
31     pid_t pid;
32     unsigned int cpu_burst;
33     unsigned int remaining_cpu_burst;
34     unsigned int io_burst = 0;
35     unsigned int remaining_io_burst = 0;
36     unsigned int io_start_time = 0;
37     unsigned int priority;
38     unsigned int response_time = 0;
39     unsigned int finish_time = 0;
40 };
41
42 struct Message {
43     pid_t mtype;
44     Process process_info;
45 };
46
47 struct Statistics {
48     unsigned int total_response_time = 0;
49     unsigned int avg_response_time = 0;
50     unsigned short context_switch = 0;
51     unsigned int total_execution_time = 0;
52     unsigned int total_turnaround_time = 0;
53     unsigned int avg_turnaround_time = 0;
54     vector<pid_t> completed_processes;
```

```

55
56     void calculateAverages(int process_count) {
57         if (process_count > 0) {
58             avg_response_time = total_response_time / process_count;
59             avg_turnaround_time = total_turnaround_time / process_count;
60         }
61     }
62
63     void printSummary(FILE* outputFile, int quantum) {
64         fprintf(outputFile, "<Statistics> (TIME QUANTUM: %d)\n", quantum);
65         fprintf(outputFile, "Process Count: %d\n", process_count);
66         fprintf(outputFile, "Average Response Time: %dms\n", avg_response_time);
67         fprintf(outputFile, "Average Turnaround Time: %dms\n", avg_turnaround_time);
68         fprintf(outputFile, "Context Switches: %hu\n", context_switch);
69         fprintf(outputFile, "Total Execution Time: %dms\n", total_execution_time);
70         fprintf(outputFile, "Completed Process Order: ");
71         for (pid_t pid : completed_processes) {
72             fprintf(outputFile, "%d ", pid);
73         }
74         fprintf(outputFile, "\n");
75     }
76 };
77
78 queue<Process*> ready_queues[3];
79 queue<Process*> io_wait_queue;
80 Process* current_process = nullptr;
81 Statistics statistics;
82
83 void displaySystemState(int time);
84 void processReadyQueues(int& quantum_timer, int& current_time);
85 void processIoWaitQueue(int& quantum_timer, int& current_time);
86 void initializeChildProcesses();
87 void signalHandlerStartProcess(int signal);
88
89 int main(int argc, char *argv[]) {
90     if (argc == 2) {
91         char* endptr;
92         QUANTUM = strtol(argv[1], &endptr, 10);
93         if (*endptr != '\0') {
94             fprintf(stderr, "Invalid Time Quantum: %s\n", argv[1]);
95             exit(EXIT_FAILURE);
96         }
97     }
98
99     key_t key = ftok("/tmp", 'P');
100     msgid = msgget(key, IPC_CREAT | 0666);
101     if (msgid == -1) {
102         perror("msgget failed");
103         exit(EXIT_FAILURE);
104     }
105
106     outputFile = fopen("schedule_dump.txt", "w");
107
108     printf("Time Quantum set to: %d\n", QUANTUM);
109     fprintf(outputFile, "Time Quantum set to: %d\n\n", QUANTUM);
110
111     initializeChildProcesses();
112
113     int current_time = 0;
114     int quantum_timer = 0;
115
116     struct sigaction sa;
117     sa.sa_handler = signalHandlerStartProcess;
118     sa.sa_flags = SA_RESTART;
119     sigaction(SIGALRM, &sa, nullptr);
120
121     struct itimerval timer;
122     timer.it_interval.tv_sec = 0;
123     timer.it_interval.tv_usec = TIMETICK * 1000;
124     timer.it_value = timer.it_interval;
125     setitimer(ITIMER_REAL, &timer, nullptr);
126
127     struct timeval start_time, now;
128     gettimeofday(&start_time, nullptr);

```

```

129
130 while (!(current_process == nullptr && ready_queues[0].empty() && ready_queues[1].empty()
131         && ready_queues[2].empty() && io_wait_queue.empty())) {
132     gettimeofday(&now, nullptr);
133     int elapsed_time = (now.tv_sec - start_time.tv_sec) * 1000 + (now.tv_usec - start_time.tv_usec) / 1000;
134     Message msg;
135     msgrcv(msgid, &msg, sizeof(msg.process_info), 0, IPC_NOWAIT);
136
137     if (elapsed_time % TIMETICK == 0) {
138         processReadyQueues(quantum_timer, current_time);
139         processIoWaitQueue(quantum_timer, current_time);
140
141         displaySystemState(current_time);
142         fflush(outputFile);
143
144         if (!(current_process == nullptr && ready_queues[0].empty() && ready_queues[1].empty()
145             && ready_queues[2].empty() && io_wait_queue.empty())) {
146             current_time += TIMETICK;
147         }
148     }
149     usleep(TIMETICK * 10);
150 }
151
152 statistics.total_execution_time = current_time + statistics.context_switch;
153 statistics.calculateAverages(process_count);
154 statistics.printSummary(outputFile, QUANTUM);
155 fclose(outputFile);
156
157 msgctl(msgid, IPC_RMID, nullptr);
158 return 0;
159 }
160
161 void displaySystemState(int time) {
162     fprintf(outputFile, "TIME: %dms\n", time);
163     if (current_process != nullptr) {
164         fprintf(outputFile, "Running: %d (CPU: %dms, I/O: %dms, I/O start: %dms)\n",
165             current_process->pid, current_process->remaining_cpu_burst,
166             current_process->remaining_io_burst, current_process->io_start_time);
167     }
168     else {
169         fprintf(outputFile, "Running: -\n");
170     }
171     fprintf(outputFile, "<Ready Queues>");
172     for (int i = 0; i < 3; i++) {
173         fprintf(outputFile, "\nQueue %d: ", i);
174         queue<Process*> temp = ready_queues[i];
175         while (!temp.empty()) {
176             fprintf(outputFile, "%d (CPU: %dms, I/O: %dms) ",
177                 temp.front()->pid, temp.front()->remaining_cpu_burst, temp.front()->remaining_io_burst);
178             temp.pop();
179         }
180     }
181     fprintf(outputFile, "\n");
182     fprintf(outputFile, "Waiting: ");
183     queue<Process*> temp_wait = io_wait_queue;
184     while (!temp_wait.empty()) {
185         fprintf(outputFile, "%d ", temp_wait.front()->pid);
186         temp_wait.pop();
187     }
188     fprintf(outputFile, "\n\n");
189 }
190
191 void processReadyQueues(int& quantum_timer, int& current_time) {
192     printf("Quantum Timer: %d, QUANTUM: %d\n", quantum_timer, QUANTUM);
193     if (current_process != nullptr) {
194         current_process->remaining_cpu_burst -= TIMETICK;
195
196         quantum_timer += TIMETICK;
197
198         if (current_process->remaining_cpu_burst <= 0) {
199             current_process->finish_time = current_time;
200             statistics.total_turnaround_time += current_process->finish_time;
201             statistics.completed_processes.push_back(current_process->pid);
202

```



```

203     printf("Process Completed (PID: %d)\n", current_process->pid);
204     fprintf(outputFile, "---- Process Completed (PID: %d) ----\n\n", current_process->pid);
205
206     current_process = nullptr;
207     statistics.context_switch++;
208 } else if (current_process->remaining_io_burst > 0 &&
209           (quantum_timer == current_process->io_start_time || current_process->io_start_time == 0)) {
210     io_wait_queue.push(current_process);
211     current_process = nullptr;
212     quantum_timer = 0;
213 }
214 }
215
216 if (current_process == nullptr || quantum_timer >= QUANTUM) {
217     if (current_process != nullptr && current_process->remaining_cpu_burst > 0) {
218         ready_queues[current_process->priority].push(current_process);
219     }
220
221     for (int i = 0; i < 3; i++) {
222         if (!ready_queues[i].empty()) {
223             current_process = ready_queues[i].front();
224             ready_queues[i].pop();
225             if (rand() % 3 == 0) {
226                 current_process->io_burst = (rand() % ((2000 - 1000) / 100 + 1) * 100) + 1000;
227                 current_process->remaining_io_burst = current_process->io_burst;
228                 current_process->io_start_time = (rand() % (QUANTUM / 100)) * TIMETICK;
229             }
230             quantum_timer = 0;
231             if (current_process->response_time == 0) {
232                 current_process->response_time = current_time;
233                 statistics.total_response_time += current_process->response_time;
234             }
235             statistics.context_switch++;
236             Message new_msg;
237             new_msg.mtype = current_process->pid;
238             new_msg.process_info = *current_process;
239             msgsnd(msgid, &new_msg, sizeof(new_msg.process_info), 0);
240             break;
241         }
242     }
243 }
244
245 unsigned int now_time = current_time;
246 if (now_time - last_aging_time >= AGING_INTERVAL) {
247     last_aging_time = now_time;
248     fprintf(outputFile, "----- Aging Occurred ----- \n\n");
249     while (!ready_queues[1].empty()) {
250         Process* aged_process = ready_queues[1].front();
251         ready_queues[1].pop();
252         ready_queues[0].push(aged_process);
253     }
254     while (!ready_queues[2].empty()) {
255         Process* aged_process = ready_queues[2].front();
256         ready_queues[2].pop();
257         ready_queues[1].push(aged_process);
258     }
259 }
260 }
261
262 void processIoWaitQueue(int& quantum_timer, int& current_time) {
263     queue<Process*> temp;
264     while (!io_wait_queue.empty()) {
265         Process* io_process = io_wait_queue.front();
266         io_wait_queue.pop();
267         if (io_process->remaining_io_burst > 0) {
268             io_process->remaining_io_burst -= TIMETICK;
269             temp.push(io_process);
270         } else {
271             io_process->io_start_time = 0;
272             ready_queues[io_process->priority].push(io_process);

```

```

273     }
274 }
275 io_wait_queue = temp;
276 }
277
278 void initializeChildProcesses() {
279     for (int i = 0; i < PROCESS_NUM; i++) {
280         Process* new_process = new Process;
281         new_process->pid = fork();
282         if (new_process->pid < 0) {
283             perror("fork failed");
284             exit(EXIT_FAILURE);
285         }
286         else if (new_process->pid == 0) {
287             srand(getpid());
288             Message msg;
289             while (true) {
290                 if (msgrcv(msgid, &msg, sizeof(msg.process_info), getpid(), 0) != -1) {
291                     Process* received_process = &msg.process_info;
292
293                     if (received_process->remaining_cpu_burst > 0) {
294                         usleep(TIMETICK * 1000);
295
296                         received_process->remaining_cpu_burst -= TIMETICK;
297                         if (received_process->remaining_cpu_burst <= 0) {
298                             received_process->io_burst = (rand() % ((2000 - 1000) / 100 + 1) * 100) + 1000;
299                             received_process->remaining_io_burst = received_process->io_burst;
300                             msg.mtype = getppid();
301                             msgsnd(msgid, &msg, sizeof(msg.process_info), 0);
302                         }
303                     }
304                     else {
305                         msg.mtype = getppid();
306                         msgsnd(msgid, &msg, sizeof(msg.process_info), 0);
307                     }
308                     if (received_process->remaining_cpu_burst <= 0 && received_process->remaining_io_burst > 0) {
309                         usleep(received_process->remaining_io_burst * 1000);
310                         received_process->remaining_io_burst = 0;
311                         msg.mtype = getppid();
312                         msgsnd(msgid, &msg, sizeof(msg.process_info), 0);
313                     }
314                 }
315             }
316         }
317         new_process->cpu_burst = (rand() % ((MAX_BURST - MIN_BURST) / 100 + 1) * 100) + MIN_BURST;
318         new_process->remaining_cpu_burst = new_process->cpu_burst;
319         new_process->priority = i % 3;
320         ready_queues[new_process->priority].push(new_process);
321         Message msg;
322         msg.mtype = new_process->pid;
323         msg.process_info = *new_process;
324         process_count++;
325         msgsnd(msgid, &msg, sizeof(msg.process_info), 0);
326     }
327 }
328
329 void signalHandlerStartProcess(int signum) {
330     for (int i = 0; i < 3; i++) {
331         if (!ready_queues[i].empty()) {
332             Process* start_process = ready_queues[i].front();
333             if (start_process != nullptr) {
334                 Message start_msg;
335                 start_msg.mtype = start_process->pid;
336                 msgsnd(msgid, &start_msg, sizeof(start_msg.process_info), 0);
337             }
338         }
339     }
340 }

```

2. 실행 방법

a. 소스 코드 컴파일

```
g++ -o OS_rr_term1 OS_rr_term1.cpp
```

: OS_rr_term1 파일을 컴파일하여 실행 파일 OS_rr_term1를 생성한다.

b. 실행 명령어

1) 기본 Time Quantum

```
./OS_rr_term1
```

: 프로그램이 `int QUANTUM = 500;`에서 선언된 기본 Time Quantum 값(500ms)으로 실행된다. 별도의 입력 없이 기본값으로 실행할 때 적합하다.

2) Time Quantum 직접 지정

```
./OS_rr_term1 300
```

```
./OS_rr_term1 500
```

```
./OS_rr_term1 700
```

: 명령어 뒤에 숫자를 입력하여 원하는 Time Quantum 값을 설정할 수 있다. 예를 들어, `./OS_rr_term1 300`은 Time Quantum을 300ms로 설정하여 프로그램을 실행한다. 이 방법은 다양한 Time Quantum을 테스트하거나 상황에 맞는 설정을 적용할 때 유용하다.

3. Time Quantum의 크기에 따른 성능 비교

Time Quantum의 크기는 CPU 스케줄링 성능에 중요한 영향을 미친다. Time Quantum이 짧으면 프로세스가 자주 전환되며 응답 시간이 빨라지지만, Context Switch가 빈번하게 발생하여 시스템 오버헤드가 증가할 수 있다. 반면, Time Quantum이 길면 각 프로세스가 CPU를 더 오랫동안 사용할 수 있어 오버헤드를 줄일 수 있지만, 응답 시간은 늘어나고 프로세스 간의 기아 상태나 대기 시간이 발생할 수 있다. 이를 통해 성능을 비교하기 위해 Time Quantum을 300ms, 500ms, 700ms로 각각 설정하여 테스트를 진행하였다.

- 1) 300ms (짧은 Time Quantum): 빠른 응답 시간을 제공하지만, Context Switch가 자주 발생하여 성능이 저하될 수 있다. 이는 특히 실시간 응답성 또는 인터랙티브한 프로세스에 적합하지만, CPU 자원을 효율적으로 사용하지 못할 수 있다.
- 2) 500ms (중간 Time Quantum): 짧고 긴 Quantum 간의 균형을 맞추어, 대부분의 시스템에서 최적의 성능을 제공한다. 이는 프로세스 간의 전환을 적당히 줄이고, CPU 자원을 효율적으로 사용할 수 있는 장점이 있다. 또한, 일반적인 컴퓨팅 환경에 가장 적합한 Time Quantum으로 평가된다.
- 3) 700ms (긴 Time Quantum): Context Switch 빈도를 줄여 CPU 효율성을 높일 수 있지만, 응답 시간이 길어지고, 특정 프로세스가 대기 상태에 오래 있을 수 있다. 이는 CPU 집약적인 작업에 적합하며, 응답 시간이 덜 중요한 경우에 유리하다.

이와 같은 세 가지 Time Quantum을 비교함으로써, 각 Time Quantum이 시스템 성능에 미치는 영향을 분석하고, 최적의 성능을 달성할 수 있는 Time Quantum의 크기를 평가할 수 있었다.

결과 분석

실험 결과를 통해 Time Quantum 크기에 따른 CPU 스케줄링의 성능 차이를 분석할 수 있었다. 각 설정에 대한 분석은 다음과 같다:

1) 300ms (짧은 Time Quantum)

```
<Statistics> (TIME QUANTUM: 300)
Process Count: 10
Average Response Time: 5190ms
Average Turnaround Time: 58890ms
Context Switches: 325
Total Execution Time: 90125ms
Completed Process Order: 1473 1467 1470 1476 1471 1468 1474 1469 1475 1472
```

평균 응답 시간은 5190ms로 가장 짧은 Time Quantum 덕분에 빠른 응답성을 제공했지만, Context Switch의 빈도가 325회로 매우 많아 시스템 오버헤드가 크게 발생했다. 이로 인해 총 실행 시간이 90125ms로 가장 길었다. 이는 자주 발생하는 프로세스 전환이 시스템 성능을 저하시킬 수 있음을 보여준다.

2) 500ms (중간 Time Quantum)

```
<Statistics> (TIME QUANTUM: 500)
Process Count: 10
Average Response Time: 5690ms
Average Turnaround Time: 57600ms
Context Switches: 203
Total Execution Time: 81503ms
Completed Process Order: 2380 2383 2389 2386 2384 2387 2381 2388 2382 2385
```

평균 응답 시간은 5690ms로 비교적 짧은 편이면서, 평균 Turnaround Time은 57600ms로 Time Quantum 300ms에 비해 개선되었다. Context Switch는 203회로 감소하여 시스템 오버헤드가 줄었고, 총 실행 시간은 81503ms로 가장 효율적인 결과를 보였다. 이는 500ms의 Time Quantum이 전반적으로 프로세스 전환과 CPU 사용 효율 간의 균형을 잘 맞춘다는 것을 보여준다.

3) 700ms (긴 Time Quantum)

```
<Statistics> (TIME QUANTUM: 700)
Process Count: 10
Average Response Time: 6480ms
Average Turnaround Time: 59240ms
Context Switches: 148
Total Execution Time: 79748ms
Completed Process Order: 3086 3092 3095 3089 3090 3093 3087 3091 3088 3094
```

평균 응답 시간은 6480ms로 가장 높았으며, 평균 Turnaround Time은 59240ms로 상대적으로 길었다. Context Switch는 148회로 가장 적었고, 총 실행 시간도 79748ms로 가장 짧았다. 이는 Context Switch 오버헤드는 최소화되었지만, 응답성이 떨어질 수 있음을 나타낸다. 따라서 이 설정은 CPU 집약적인 작업이나 응답성이 덜 중요한 경우에 적합하다.

결론

Time Quantum이 짧을수록 응답성은 개선되지만 Context Switch로 인한 시스템 오버헤드가 증가하고, Time Quantum이 길수록 오버헤드는 줄지만 응답성이 떨어진다. 이번 실험에서는 500ms의 Time Quantum이 가장 균형 잡힌 성능을 보여주었으며, 응답성과 효율성 간의 절충안으로 적합함을 확인할 수 있었다.