OS Term Project 보고서

고려대학교 정보대학 컴퓨터학과

2023320064 유륜규

목차

1. 서론

1.1. CPU 스케줄러의 개념 및 중요성

현대의 컴퓨터 시스템은 동시에 여러 작업을 처리하는 멀티프로그래밍(Multiprogramming) 환경을 기반으로 합니다. 이러한 환경에서 CPU 는 여러 프로세스(작업)가 공유하는 핵심 자원이며, 이 자원을 효율적으로 관리하는 것은 운영체제의 가장 중요한 역할 중 하나입니다. CPU 스케줄링은 여러 프로세스가 CPU 를 사용하기 위해 경쟁할 때, 어떤 프로세스에 CPU 를 할당할지, 그리고 얼마 동안 할당할지를 결정하는 메커니즘입니다.

효율적인 CPU 스케줄링은 시스템의 전반적인 성능에 직접적인 영향을 미칩니다. 이는 응답 시간(Response Time) 최소화, 처리율(Throughput) 최대화, CPU 활용률(CPU Utilization) 증대, 그리고 공정성(Fairness) 유지와 같은 다양한 목표를 동시에 달성하고자 합니다. 예를 들어, 웹 브라우징이나 문서 편집과 같이 사용자 상호작용이 중요한 작업에서는 빠른 응답 시간이 필수적이며, 대규모 데이터 처리와 같은 배치(Batch) 작업에서는 처리율 극대화가 중요합니다. 따라서 운영체제는 시스템의 목적과 특성에 따라 적절한 스케줄링 알고리즘을 선택하고 적용하여 자원을 최적으로 분배합니다.

1.2. 본인이 구현한 스케줄러 요약

본 보고서는 C 언어를 사용하여 구현한 CPU 스케줄러 시뮬레이터에 대한 내용을 다룹니다. 이 시뮬레이터는 운영체제에서 가장 널리 사용되는 스케줄링 알고리즘인 **FCFS (First-Come, First-Served), SJF (Shortest Job First) 비선점형 및 선점형, 우선순위(Priority) 비선점형 및 선점형, 그리고 라운드 로빈(Round Robin)**을 모두 시뮬레이션하고 비교할 수 있도록 설계되었습니다.

시뮬레이터는 사용자로부터 프로세스 개수를 입력받아, 각 프로세스의 PID, 도착 시간(Arrival Time), CPU 버스트 시간(CPU Burst Time), 우선순위(Priority) 등의 속성을 무작위로 생성하거나, 사용자가 원하는 경우 미리 정의된 프로세스 집합을 활용할 수 있도록 유연성을 제공합니다. 각 스케줄링 알고리즘은 가상의 시간을 1 단위씩 증가시키며 프로세스의 CPU 할당 과정을 시뮬레이션하고, 이때 발생하는 문맥 교환(Context

Switching) 및 프로세스 상태 변화를 간트 차트(Gantt Chart) 형태로 출력하여 시각적으로 쉽게 이해할 수 있도록 했습니다.

시뮬레이션이 완료되면, 각 알고리즘별로 핵심 성능 지표인 **평균 대기 시간(Average Waiting Time)**과 **평균 반환 시간(Average Turnaround Time)**을 정량적으로 계산하고 출력합니다. 이를 통해 각 알고리즘이 주어진 작업 부하에서 어떻게 동작하며, 어떤 성능 특성을 보이는지 다각적으로 분석하고 비교할 수 있습니다. 본 보고서에서는 구현 환경, 각 스케줄링 알고리즘의 상세 원리, 시뮬레이터의 시스템 구성 및 각 모듈의 동작 방식, 그리고 실제 실행 결과 분석 및 향후 발전 방향에 대해 자세히 다룰 예정입니다.

2. 본론

2.1. 다른 CPU 스케줄링 시뮬레이터 소개

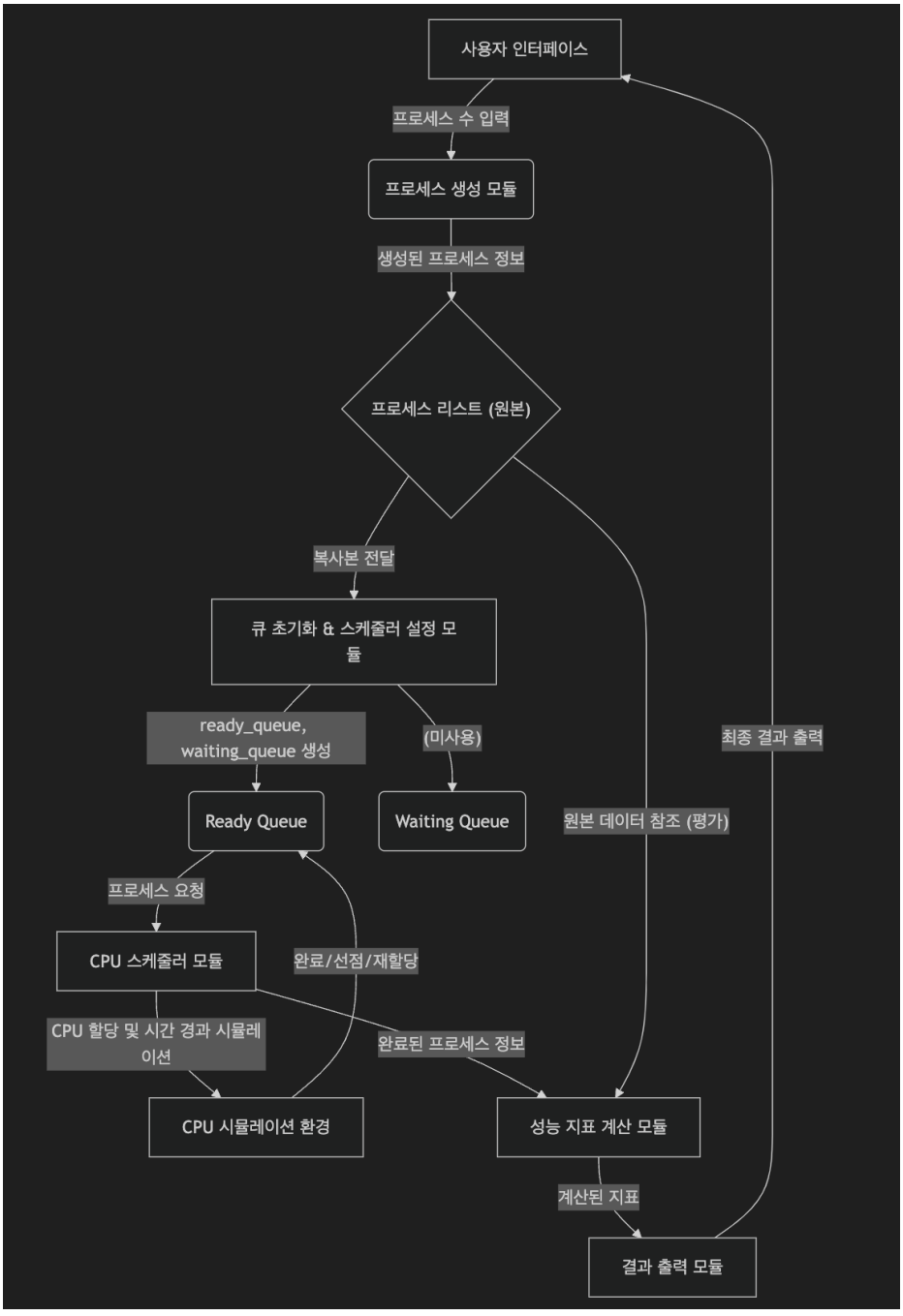
운영체제 교육 및 연구 분야에서는 CPU 스케줄링 알고리즘의 복잡한 동작 방식을 이해하고 성능을 분석하기 위해 다양한 시뮬레이터가 활용되고 있습니다. 이들은 주로 이론적인 개념을 시각적으로 보여주거나, 특정 시나리오에서 알고리즘의 효율성을 정량적으로 평가하는 데 목적을 둡니다.

대표적인 예시와 그 특징은 다음과 같습니다:

- **CPU-OS Simulator (Teach-Sim):** 이 시뮬레이터는 비교적 완성도 높은 그래픽 사용자 인터페이스(GUI)를 제공하여 FCFS, SJF, Priority, Round Robin 등 다양한 스케줄링 알고리즘의 동작을 시각적으로 보여줍니다. 프로세스의 생성, 상태 변화, CPU 할당 및 문맥 교환 과정을 애니메이션으로 확인할 수 있으며, CPU 활용률, 처리율, 대기 시간, 반환 시간 등의 성능 지표를 그래프 형태로 실시간으로 제공하여 사용자가 직관적으로 알고리즘의 특성을 파악할 수 있도록 돕습니다. 실제 운영체제의 주요 구성 요소들을 함께 시뮬레이션하여 전체적인 시스템 동작을 이해하는 데 유용합니다.
- **Visual Algo (visualgo.net):** 웹 기반의 알고리즘 시각화 도구로, CPU 스케줄링뿐만 아니라 다양한 자료구조 및 알고리즘을 인터랙티브하게 시각화합니다. CPU 스케줄링 섹션에서는 FCFS, SJF, Priority, Round Robin 등의 알고리즘에 대해 사용자가 직접 프로세스 데이터를 입력하고, Gantt 차트와 함께 각 프로세스의 완료 시간, 대기 시간, 반환 시간 등을 단계별로 보여주며 시뮬레이션 과정을 시각적으로 추적할 수 있도록 지원합니다. 간단하고 직관적인 UI 로 알고리즘의 동작 원리를 빠르게 이해하는 데 효과적입니다.

2.2. 본인이 구현한 시뮬레이터의 시스템 구성도

본 CPU 스케줄러 시뮬레이터는 모듈화된 설계를 통해 각 기능이 명확하게 분리되어 상호작용하도록 구성되었습니다. 이는 스케줄러의 복잡성을 관리하고, 유지보수 및 확장을 용이하게 합니다. 시뮬레이터의 전체적인 시스템 구성은 다음 다이어그램과 같습니다.



각 구성 요소에 대한 설명:

- **사용자 인터페이스 (User Interface):** main 함수에서 사용자와 상호작용하는 부분입니다. 프로세스 개수를 입력받고, 시뮬레이션 시작 및 종료, 최종 결과 출력을 담당하는 콘솔 기반 인터페이스입니다.
- **프로세스 생성 모듈 (create_process, print_processes):**
    - create_process 함수는 사용자 입력에 따라 지정된 개수만큼의 프로세스(process 구조체)들을 동적으로 할당하고 초기화합니다. 각 프로세스에는 고유한 PID, 무작위로 생성된 도착 시간, CPU 버스트 시간, 우선순위, 그리고 (현재 시뮬레이션에서는 사용되지 않지만) IO 관련 정보가 할당됩니다.
    - print_processes 함수는 생성된 초기 프로세스 목록을 사용자에게 보여주어 시뮬레이션 전 프로세스들의 상태를 확인할 수 있도록 합니다.
- **프로세스 리스트 (원본/복사본):**
    - process 구조체 배열로, create_process 함수에 의해 생성된 모든 프로세스의 원본 정보를 담고 있습니다.
    - 각 스케줄링 알고리즘 시뮬레이션을 수행할 때마다 이 원본 리스트를 backup_list 를 통해 복사하여 사용합니다. 이는 각 알고리즘이 독립적으로 동일한 초기 프로세스 데이터를 가지고 시뮬레이션될 수 있도록 보장합니다.
- **큐 초기화 & 스케줄러 설정 모듈 (config, init_queue, enqueue):**
    - init_queue 함수는 queue 구조체를 초기 상태로 만듭니다.
    - config 함수는 특정 스케줄링 알고리즘을 실행하기 전, 해당 알고리즘에 맞게 프로세스들의 remaining_time 등을 초기화하고, 복사된 프로세스들을 준비 큐(ready_queue)에 초기 상태로 enqueue 합니다.

- **Ready Queue (준비 큐):**
    - queue 구조체로 구현되며, CPU 할당을 기다리는 프로세스들의 집합입니다.
    - 스케줄링 알고리즘에 따라 이 큐에 프로세스가 삽입되고(enqueue), 제거(dequeue)되며, 내부적으로 재정렬(sort_ready_queue_by_arrival)되기도 합니다.
- **Waiting Queue (대기 큐):**
    - queue 구조체로 구현되어 있지만, 현재 버전에서는 IO 작업이 시뮬레이션되지 않으므로 실제로 사용되지는 않습니다. 향후 IO 시뮬레이션 기능이 추가될 경우, IO 를 수행 중인 프로세스들이 이 큐에서 대기하게 됩니다.
- **CPU 스케줄러 모듈 (FCFS, SJF_NONPREEM, SJF_PREEM, Priority_NONPREEM, Priority_PREEM, RoundRobin):**
    - 시뮬레이터의 핵심 엔진으로, 각 스케줄링 알고리즘에 대한 로직이 구현되어 있습니다.
    - current_time 변수를 기준으로 시간을 1 단위씩 진행시키면서, 현재 시간에 따라 준비 큐에서 다음으로 실행할 프로세스를 선택합니다.
    - 선택된 프로세스의 remaining_time 을 감소시키며 CPU 실행을 시뮬레이션하고, 필요한 경우(선점형 알고리즘, Round Robin) 문맥 교환을 수행합니다.
    - 각 스케줄러 함수는 시뮬레이션 진행 상황을 간트 차트 형태로 콘솔에 실시간으로 출력합니다.
- **CPU 시뮬레이션 환경:**

    - CPU 스케줄러 모듈 내부에서 current_time 을 증가시키고 프로세스의 remaining_time 을 감소시키는 방식으로 구현됩니다. 이는 실제 CPU 가 프로세스를 실행하고 시간을 소모하는 과정을 추상화하여 시뮬레이션합니다.

- **성능 지표 계산 모듈 (evaluate, evaluate_rr):**
    - FCFS, SJF_NONPREEM, SJF_PREEM, Priority_NONPREEM, Priority_PREEM 알고리즘은 각 스케줄러 함수 내에서 직접 total_waiting 과 total_turnaround 를 계산하고 평균을 출력합니다.
    - evaluate 및 evaluate_rr 함수는 각 스케줄러 함수를 호출하여 시뮬레이션을 실행한 후, 완료된 프로세스들의 waiting_time 과 turnaround_time 을 집계하여 최종 **평균 대기 시간**과 **평균 반환 시간**을 계산하고 출력합니다. 이 함수들은 각 시뮬레이션이 독립적으로 수행될 수 있도록 process_list 의 복사본을 전달하여 사용합니다.
- **결과 출력 모듈 (printf):**
    - print_processes 를 통해 초기 프로세스 정보를, 각 스케줄러 함수 내에서 간트 차트를, 그리고 evaluate 함수를 통해 최종 성능 지표를 콘솔에 출력합니다.

이러한 모듈화된 구성은 시뮬레이터가 다양한 스케줄링 알고리즘을 유연하게 적용하고, 각 알고리즘의 동작을 명확히 추적하며, 최종 성능을 효과적으로 분석할 수 있도록 합니다.

2.3. 각 모듈에 대한 설명 (알고리즘 표현)

본 시뮬레이터는 C 언어로 구현되었으며, 각 기능을 모듈화하여 관리하고 있습니다. 여기서는 시뮬레이터의 핵심 데이터 구조와 구현된 스케줄링 알고리즘들의 상세 동작 방식을 의사 코드 형태로 설명합니다.

2.3.1. 프로세스(작업) 구조체 (process)

시뮬레이션에 참여하는 각 프로세스(작업)는 다음과 같은 process 구조체를 통해 그 속성이 정의됩니다. 이 구조체는 프로세스의 식별 정보, CPU 사용량, 우선순위뿐만 아니라 시뮬레이션 중 상태 변화 및 성능 측정을 위한 다양한 필드를 포함합니다.

```
typedef struct {
  int pid;          // 프로세스 고유 식별자 (Process ID)
  int arrival_time;   // 프로세스가 시스템에 도착한 시간
  int cpu_burst;     // 프로세스가 CPU 를 사용하는 총 시간 (Initial CPU burst time)
  int priority;      // 프로세스의 우선순위 (낮은 숫자 = 높은 우선순위)
  // (현재 버전에서는 미사용되는 IO 관련 필드)
  int io_request_times[max_io]; // IO 요청이 발생할 CPU 사용 시간 지점
  int io_bursts[max_io];      // 각 IO 작업에 소요되는 시간
  int io_count;             // 총 IO 요청 횟수
  int current_io_index;       // 현재 처리 중인 IO 요청 인덱스
  int in_io;                // IO 중인지 여부
  int io_remaining_time;      // 남은 IO 시간
```

```
    int remaining_time;  // 남은 CPU 버스트 시간 (동적으로 변화)
    int completion_time; // 프로세스 완료 시간
    int waiting_time;    // 프로세스가 준비 큐에서 대기한 총 시간
    int turnaround_time; // 프로세스의 총 반환 시간 (도착 ~ 완료)
    int is_completed;    // 프로세스 완료 여부 플래그 (0: 미완료, 1: 완료)
    int is_enqueued;     // Round Robin 에서 큐에 중복 추가 방지 플래그
} process;
```

2.3.2. 큐(Queue) 관리 모듈

CPU 스케줄링에서 필수적인 준비 큐(Ready Queue) 및 **대기 큐(Waiting Queue)**는 queue 구조체와 관련된 함수들을 통해 구현됩니다. 이 큐는 process 구조체의 포인터를 저장하며, FIFO(First-In, First-Out) 방식으로 작동합니다.

2.3.3. 스케줄러 핵심 로직 (간단한 의사 코드)

시뮬레이터는 다양한 CPU 스케줄링 알고리즘을 구현하여 그 동작을 시뮬레이션하고 성능을 측정합니다. 각 알고리즘은 **현재 시간(current_time)**을 기준으로 프로세스를 선택하고 실행하며, 간트 차트를 통해 시뮬레이션 과정을 시각적으로 보여줍니다.

A. FCFS (First-Come, First-Served) 스케줄링

가장 먼저 도착한 프로세스에게 CPU 를 할당하는 **비선점형** 알고리즘입니다.

```
Function FCFS(ready_queue, total_processes):
  current_time = 0
  Sort ready_queue by arrival_time // 도착 시간 순으로 정렬

  While some processes are not completed:
    p = Dequeue from ready_queue
    If p.arrival_time > current_time:
      current_time = p.arrival_time // 프로세스 도착까지 대기 (IDLE)

    p.waiting_time = current_time - p.arrival_time
    current_time = current_time + p.cpu_burst // CPU 실행
    p.completion_time = current_time
    p.turnaround_time = p.completion_time - p.arrival_time
    Mark p as completed
    Print "| P" + p.pid

  Calculate and Print Average Waiting Time, Average Turnaround Time
```

B. SJF (Shortest Job First) 비선점형 스케줄링

현재 시간까지 도착한 프로세스 중 CPU 버스트 시간이 가장 짧은 프로세스를 선택하여 완료될 때까지 실행하는 비선점형 알고리즘입니다.

```
Function SJF_NONPREEM(ready_queue, total_processes):
  current_time = 0

  While some processes are not completed:
    shortest_job = Find process in ready_queue with shortest cpu_burst
          (among those arrived by current_time)

    If no such process:
      current_time = current_time + 1 // IDLE
      Continue

    p = Remove shortest_job from ready_queue
    p.waiting_time = current_time - p.arrival_time
    current_time = current_time + p.cpu_burst
    p.completion_time = current_time
    p.turnaround_time = p.completion_time - p.arrival_time
    Mark p as completed
    Print "| P" + p.pid

  Calculate and Print Average Waiting Time, Average Turnaround Time
```

C. SJF (Shortest Remaining Time First) 선점형 스케줄링

현재 실행 중인 프로세스보다 **남은 CPU 버스트 시간이 더 짧은** 프로세스가 도착하면 CPU 를 선점하여 할당하는 선점형 알고리즘입니다.

```
Function SJF_PREEM(all_processes_list, total_processes):
  current_time = 0
  While some processes are not completed:
    shortest_remaining = Find process with shortest remaining_time
                (among those arrived by current_time and not completed)
    If no such process:
      current_time = current_time + 1 // IDLE
      Continue
    If shortest_remaining is different from previously running process:
      Print "| P" + shortest_remaining.pid + " "

    shortest_remaining.remaining_time = shortest_remaining.remaining_time - 1
    current_time = current_time + 1

    If shortest_remaining.remaining_time == 0:
      shortest_remaining.completion_time = current_time
      shortest_remaining.turnaround_time = current_time -
shortest_remaining.arrival_time
      shortest_remaining.waiting_time = shortest_remaining.turnaround_time -
shortest_remaining.cpu_burst
      Mark shortest_remaining as completed

  Calculate and Print Average Waiting Time, Average Turnaround Time
```

D. Priority (우선순위) 비선점형 스케줄링

현재 시간까지 도착한 프로세스 중 **가장 높은 우선순위(숫자가 낮을수록 높음)**를 가진 프로세스를 선택하여 완료될 때까지 실행하는 비선점형 알고리즘입니다.

```
Function Priority_NONPREEM(ready_queue, total_processes):
  current_time = 0

  While some processes are not completed:
    highest_priority_job = Find process in ready_queue with highest priority
                (among those arrived by current_time)

    If no such process:
      current_time = current_time + 1 // IDLE
      Continue

    p = Remove highest_priority_job from ready_queue
    p.waiting_time = current_time - p.arrival_time
    current_time = current_time + p.cpu_burst
    p.completion_time = current_time
    p.turnaround_time = p.completion_time - p.arrival_time
    Mark p as completed
    Print "| P" + p.pid

  Calculate and Print Average Waiting Time, Average Turnaround Time
```

E. Priority (우선순위) 선점형 스케줄링

현재 실행 중인 프로세스보다 더 높은 우선순위를 가진 프로세스가 도착하거나 큐에 있으면 CPU 를 선점하여 할당하는 선점형 알고리즘입니다.

```
Function Priority_PREEM(all_processes_list, total_processes):
  current_time = 0
  While some processes are not completed:
    highest_priority_process = Find process with highest priority
                    (among those arrived by current_time and not completed)
    If no such process:
      current_time = current_time + 1 // IDLE
      Continue

    If highest_priority_process is different from previously running process:
      Print "| P" + highest_priority_process.pid + " "

    highest_priority_process.remaining_time = highest_priority_process.remaining_time - 1
    current_time = current_time + 1

    If highest_priority_process.remaining_time == 0:
      highest_priority_process.completion_time = current_time
      highest_priority_process.turnaround_time = current_time -
highest_priority_process.arrival_time
      highest_priority_process.waiting_time = highest_priority_process.turnaround_time -
highest_priority_process.cpu_burst
      Mark highest_priority_process as completed
  Calculate and Print Average Waiting Time, Average Turnaround Time
```

F. Round Robin (RR) 스케줄링

각 프로세스에게 미리 정해진 타임 퀀텀(time quantum) 동안 CPU 를 할당하고, 시간이 만료되면 CPU 를 선점하여 다음 프로세스에게 넘겨주는 선점형 알고리즘입니다.

```
Function RoundRobin(all_processes_list, total_processes, time_quantum):
  current_time = 0
  rr_queue = new Queue() // 실행을 위한 큐

  While some processes are not completed:
    Add all newly arrived processes (at current_time) to rr_queue

    If rr_queue is empty:
      current_time = current_time + 1 // IDLE
      Continue

    p = Dequeue from rr_queue
    exec_time = MIN(p.remaining_time, time_quantum) // 실행 시간은 남은 시간과 퀀텀 중 짧은 것

    For t from 0 to exec_time - 1:
      If p is different from previously running process:
        Print "| P" + p.pid + " "

      current_time = current_time + 1
      Add any newly arrived processes (at current_time) to rr_queue // 중간에 도착한 프로세스 추가
```

p.remaining_time = p.remaining_time - exec_time

If p.remaining_time == 0:

p.completion_time = current_time

p.turnaround_time = current_time - p.arrival_time

p.waiting_time = p.turnaround_time - p.cpu_burst

Mark p as completed

Else:

Enqueue p to rr_queue // 남은 시간 있으면 큐 맨 뒤에 다시 삽입

Calculate and Print Average Waiting Time, Average Turnaround Time

2.4. 시뮬레이터 실행 결과 화면, 평가

본 섹션에서는 구현된 CPU 스케줄러 시뮬레이터의 실제 실행 결과를 제시합니다. 시뮬레이션은 사용자가 입력한 프로세스 개수에 따라 무작위로 생성된 프로세스 집합을 대상으로 수행되었습니다. 각 스케줄링 알고리즘이 CPU 를 할당하고 프로세스를 처리하는 과정은 **간트 차트(Gantt Chart)** 형태로 시각화되며, 최종적으로는 **평균 대기 시간**과 **평균 반환 시간**이 계산되어 출력됩니다.

```
Last login: Wed Jun 11 17:25:23 on ttys027
/Users/reilly/Desktop/Korea_univ_3_1/OS/cpu_scheduler ; exit;
➜  ~ /Users/reilly/Desktop/Korea_univ_3_1/OS/cpu_scheduler ; exit;
ENTER NUMBER OF PROCESS: 5

[📋] 생성된 프로세스 목록:
PID     AT      CPU     Pri     IO_Count        IO_Times        IO_Bursts
 1       7       4      10       1               0               2
 2       0       7       5       2               5 1             1 3
 3       2       6      10       2               4 3             5 5
 4       4       7       2       1               6               2
 5       5       7      10       1               2               3
FCFS
Gantt Chart
| P2| P3| P4| P5| P1
Average Waiting Time: 9.80
Average Turnaround Time: 16.00


SJF_NONPREEM
Gantt Chart
| P2| P1| P3| P4| P5|

Average Waiting Time: 8.20
Average Turnaround Time: 14.40


SJF_PREEM
Gantt Chart
| P2 | P1 | P3 | P4 | P5 |

Average Waiting Time: 8.20
Average Turnaround Time: 14.40


PRIORITY_NONPREEM
Gantt Chart
| P2| P4| P3| P5| P1|

Average Waiting Time: 10.00
Average Turnaround Time: 16.20


PRIORITY_PREEM
Gantt Chart
| P2 | P4 | P2 | P1 | P3 | P5 |

Average Waiting Time: 9.80
Average Turnaround Time: 16.00


RR
Gantt Chart
| P2 | P3 | P2 | P4 | P3 | P5 | P2 | P1 | P4 | P3 | P5 | P2 | P1 | P4 | P5 | P4 | P5 |

Average Waiting Time: 16.00
Average Turnaround Time: 22.20
```

3. 결론

3.1. 구현한 시뮬레이터에 대한 정리

본 프로젝트를 통해 C 언어를 사용하여 다양한 CPU 스케줄링 알고리즘을 시뮬레이션하는 도구를 성공적으로 구현하였습니다. 구현된 시뮬레이터는 FCFS, SJF(비선점/선점), 우선순위(비선점/선점), 그리고 Round Robin 의 5 가지 핵심 스케줄링 알고리즘을 포함하며, 각 알고리즘의 동작 방식을 간트 차트(Gantt Chart) 형태로 시각화하여 보여주고, 평균 대기 시간(Average Waiting Time) 및 **평균 반환 시간(Average Turnaround Time)**이라는 정량적인 성능 지표를 제공합니다.

사용자로부터 프로세스 개수를 입력받아 무작위로 프로세스 속성(도착 시간, CPU 버스트, 우선순위)을 생성하여 유연한 시뮬레이션 환경을 제공하며, 각 알고리즘별 성능을 동일한 조건에서 비교 분석할 수 있도록 설계되었습니다. 시뮬레이션 결과, 특정 프로세스 집합에서는 우선순위 기반 스케줄링이 가장 효율적인 대기 및 반환 시간을 보이는 반면, Round Robin 은 공정성이라는 장점에도 불구하고 잦은 문맥 교환 오버헤드로 인해 성능 지표가 다소 높아질 수 있음을 확인했습니다. 이러한 분석을 통해 각 알고리즘의 이론적 특성이 실제 시뮬레이션에서 어떻게 나타나는지 실증적으로 이해할 수 있었습니다.

3.2. 프로젝트 수행 소감 및 향후 발전 방향

이번 CPU 스케줄러 시뮬레이션 프로젝트를 수행하면서 운영체제의 핵심 개념 중 하나인 CPU 스케줄링에 대한 깊이 있는 이해를 얻을 수 있었습니다. 특히, 단순히 이론을 학습하는 것을 넘어, C 언어로 직접 각 스케줄링 알고리즘의 논리를 구현하고 시스템 자원 할당 과정을 단계별로 시뮬레이션하면서, 프로세스의 상태 변화, 문맥 교환의 중요성, 그리고 각 알고리즘이 시스템 성능에 미치는 미묘한 영향들을 체감할 수 있었습니다. 초기에는 큐 관리와 시간 동기화, 그리고 선점형 알고리즘의 복잡한 로직 구현에서 어려움이 있었지만, 단계별로 기능을 추가하고 디버깅하는 과정을 통해 문제 해결 능력과 프로그래밍 역량을 향상시킬 수 있었습니다.

향후 이 시뮬레이터는 다음과 같은 방향으로 발전시킬 수 있을 것입니다.

- ➔ 다양한 프로세스 속성 및 상태 추가: 현재 IO 관련 필드는 존재하지만 시뮬레이션 로직에 반영되지 않고 있습니다. IO 대기 및 완료를 시뮬레이션하고, 이에 따른 대기 큐(Waiting Queue)의 실제 활용을 구현하여 더욱 현실적인 운영체제 환경을 모사할 수 있습니다.
- ➔ 교착 상태(Deadlock) 및 동기화 문제 시뮬레이션: 스케줄링과 함께 운영체제의 중요한 개념인 교착 상태 발생 조건 및 해결 기법을 시뮬레이터에 통합하여, 자원 할당이 프로세스 실행에 미치는 영향을 더욱 심층적으로 분석할 수 있습니다.
- ➔ 다른 스케줄링 알고리즘 추가 구현: 현재 구현된 기본 알고리즘 외에도, 멀티레벨 큐(Multi-Level Queue), 멀티레벨 피드백 큐(Multi-Level Feedback Queue), 비율 단조 스케줄링(Rate Monotonic Scheduling, RMS) 등과 같이 더욱 복잡하고 현대적인 스케줄링 알고리즘들을 추가로 구현하여 시뮬레이터의 범용성을 높이고 다양한 시나리오에서의 성능을 비교 분석할 수 있습니다.

4. 부록

소스코드 포함 Github 주소

https://github.com/Reilly-Yu/251RCOSE34102