2023학년도 운영체제 조별 과제

스케줄링 알고리즘 시뮬레이터 제작 및 결과 보고서

TEAM CORE

2018136045 박형진
2018136035 박동진
2018136068 오수환
2018140050 박순용

차 례

I. 개요

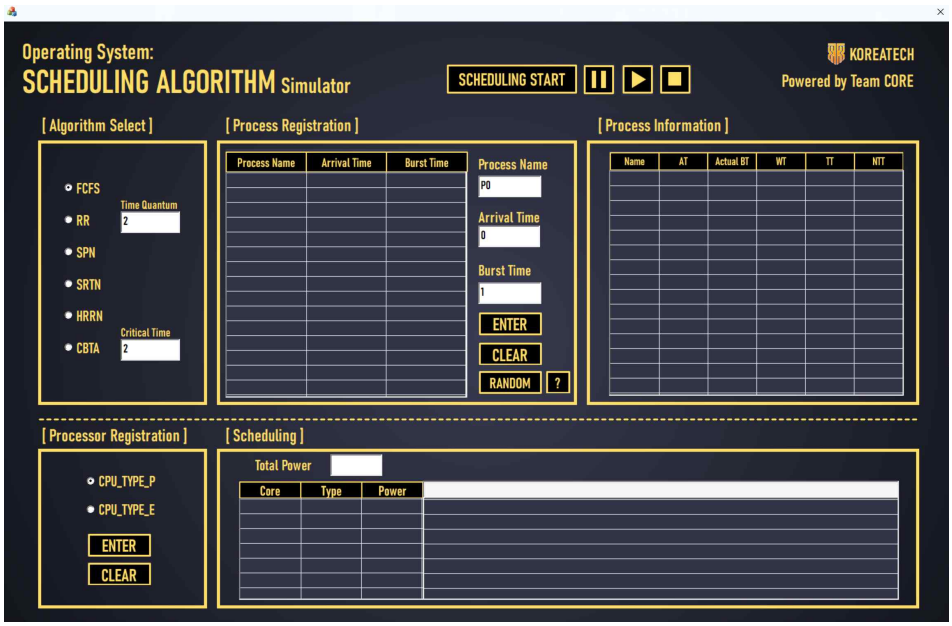
본 보고서는 2023학년도 한국 기술교육대학교 컴퓨터 공학부 운영체제 교과목에서 제시된 과제 “스케줄링 알고리즘 시뮬레이터 구현”(이하 과제로 칭한다)에 대하여, 그 수행 과정과 결과를 상세히 알리는 것을 목적으로 한다.

과제 요구사항으로 제시된 시스템 속성은 다음과 같다.

- E core는 1초에 1의 일을 처리하며 1초에 전력 1W를 소비한다. (시동 전력 0.1W)
- P core는 1초에 2의 일을 처리하며 1초에 전력 3W를 소비한다. (시동 전력 0.5W)
- 시동 전력은 미사용 중이던 코어를 사용하는 경우에 발생한다.
- 스케줄링은 1초 단위로 이루어진다. (P코어에 할당된 작업의 남은 양이 1이어도, 1초를 소모)

1. 결과 요약

본 팀에서 제작한 프로그램의 형태는 다음과 같다:



본 프로그램은 MFC를 기반으로 작성한 응용 프로그램으로, 과제에 제시된 부가적인 수행 목표를 모두 수행할 수 있도록 UI를 구성하였다. 위 그림에서 보이는 것처럼, 간단한 UI를 통하여 사용자가 스케줄링 방식과 프로세스 목록을 직접 조정할 수 있으며, 이를 토대로 초 단위로 프로세스 스케줄링을 수행할 수 있다.

한편, 직접 정의한 스케줄링 알고리즘으로는 기존의 SRTN을 응용한 CBTA(Critical Burst Time Allocation) 방식을 고안하였다. CBTA 알고리즘에서는 사용자가 직접 임계시간(cirtical time)을 지정할 수 있다. 스케줄러는 임계 시간 이하 BT를 갖는 프로세스들을 최우선으로 처리하며, 임계시간보다 긴 BT를 갖는 프로세스에 대해서는 가장 큰

BT를 갖는 프로세스를 우선적으로 처리한다.
이 스케줄링 방식은 슈퍼 컴퓨터와 같이 무거운 프로세스를 주로 처리하는 특수 목적 컴퓨터에서 간헐적으로 요청되는 가벼운 프로세스를 기준에 따라서 먼저 처리하여 일부 프로세스의 응답성을 높이는 데에 사용할 수 있다.

2. 프로젝트의 주요 특징 요약

본 프로그램의 차별점으로는 다음과 같은 3가지를 생각할 수 있다:

- 전체적인 프로젝트 관리를 위한 방법론(레이어드 플로우차트 방식)을 도입하였다:
레이어드 플로우차트 방식이란, 팀에서 독자적으로 개발한 흐름도 활용 방식을 말하며, 모듈성을 극대화하기 위하여 흐름도를 여러 단계(레이어)로 분할하여 작성하는 방법론을 말한다. 흐름도의 각 도형은 실제 코드와 일대일로 대응한다는 특징을 갖는다. 이는 각 기능의 명확한 구별과 작업 분배를 가능케 한다.

- 스케줄링 알고리즘에 대한 프레임워크를 구상하여, 일괄적인 프로젝트 관리를 가능하게 하였다:
스케줄링 알고리즘이 모두 유사한 구조를 갖는다는 점에서 착안하여, 하나의 변수만으로도 스케줄링 알고리즘을 간단히 지정할 수 있도록 프레임워크를 구성하였다. 모든 스케줄링 알고리즘은 하나의 구성 원리를 따르며, 전체적인 프로세스 관리가 일괄적으로 이루어진다. 뿐만 아니라, 팀원들에게 프로젝트에 대한 통일된 안목을 심어주어 보다 원활한 협업을 가능케 한다.

- Windows와의 친화성이 우수하다:
본 프로그램의 기획 단계에서 알고리즘의 안정성은 매우 중요한 요소로 평가되었고, 이를 위해 가능한 한 windows와 친화적인 라이브러리를 활용하도록 계획이 진행되었다. 프로그램 본체는 MFC를 기반으로 설계하고, Win32 API를 사용하여 스레드를 구현하는 등 동작의 안정성을 꾀하는 한편, 객체 간의 정보 교류가 활발한 프로그램의 특성을 반영하여 C++ STL 자료구조 중 하나인 Vector를 활용하였다.
이러한 팀원들의 노력은 프로그램에서 발생하는 오류의 양을 최소화하여 본래의 목적을 달성하는 데에 큰 도움이 되었고, 더불어 공개 라이브러리를 활용하는 것으로서 모듈성을 극대화하면서도 유지보수 비용을 크게 경감할 수 있었다.

II. 본론

이하에서는 프로젝트 설계 방법과, 이를 본 프로젝트에 적용한 결과, 그리고 그 결과로서 작성된 프로그램의 각 코드에 대하여 설명하고 시연한다. 직접 정의한 스케줄링 알고리즘을 설명한다.

1. 프로젝트 설계 방법론

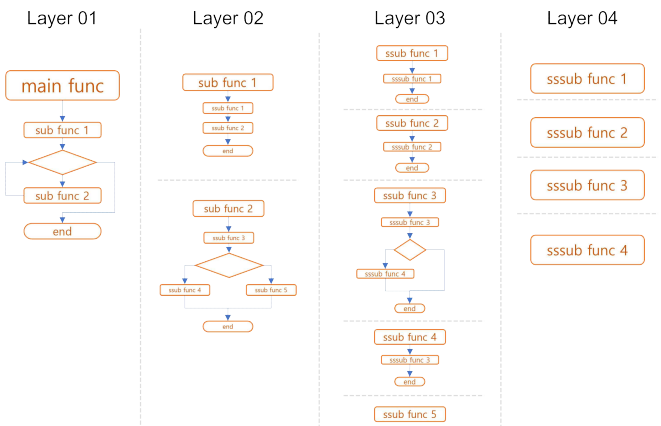
프로젝트 관리 기법 및 계획의 방법론으로는 폭포수 모형, 애자일 등 다양한 방식이 존재하지만, 본 프로젝트에서는 프로젝트의 규모와 복잡성에 근거하여 자체 개발한 방법론을 선택하여 프로젝트를 진행하였다.

i) 레이어드 플로우차트 방식

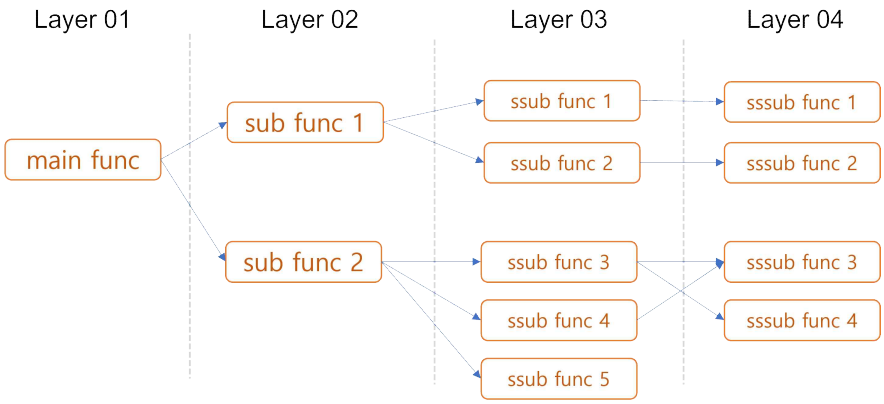
레이어드 플로우차트 방식이란 요구 분석의 결과를 토대로 흐름도를 수준별(계층별)로 작성하는 방식이다. 요구되는 기능을 논리적인 단위로 분리하고, 각 기능을 구현할 함수의 이름, 입력값, 반환값, 기능을 명시하는 방식으로 진행한다. 기본적인 사항은 단순한 의사 코드의 작성과 다르지 않지만, 입력값, 반환값, 기능이 명시된다는 시점에서 강한 모듈성을 가지며, 이를 토대로 역할 분배가 명확히 이루어진다는 특성 때문에 프로젝트 관리 방법으로도 유효하게 활용할 수 있다.

- 플로우차트의 계층화:

레이어드 플로우차트 작성의 핵심은 기능 분할이다. 1번째 레이어에서는 전체적인 기능의 수행을 위하여 각 기능을 논리적으로 구분 가능한 규모로 분할하여 순서도로 나타낸다. 이때 분할된 각각의 기능은 새로운 함수가 되며, 이 함수를 정의할 때에는 입력값, 반환값, 기능이 명확하게 정의되어야 한다. 이렇게 정의된 함수는 다음 레이어에서 동일한 방식으로 순서도로 표현된다. 이러한 방식을 반복하여, 순서도가 충분히 간단하여 순서도 없이 구현 가능한 수준으로 간략화되는 경우 혹은 더 이상 분할할 기능이 없는 경우 반복을 멈춘다.



- 함수 종속 관계도의 활용



함수 종속 관계도란 호출 계층을 레이어 별로 나타낸 것이다. 위 그림은 함수 종속 관계도의 예시를 나타낸 것이다. 각 화살표 (a,b)는 함수(프로시저) a의 수행 중 함수 b가 호출됨을 나타낸다. 예를 들자면 main func의 실행 중에는 sub func1과 sub func2가 호출되며, ssub func2의 수행 중에는 sssub func2가 호출되는 식이다.

함수 종속 관계도의 활용 방안은 크게 두 가지 존재한다. 하나는 호출 관계의 확인이다. 위 그림을 통해서 각 함수가 연결된 방식을 확인할 수 있으며, 순환적으로 정의된 함수를 확인하거나 예방하는 데에 도움이 된다. 두 번째로, 함수 구현 순서와 디버깅 순서를 정할 때 그 기준으로서 활용할 수 있다. 구현은 top down 방식도 가능하고 bottom up 방식도 가능하다. 또한 디버깅 순서 또한 top down 방식과 bottom up 방식이 가능하다. top down 방식이란, 최고 수준 루틴을 실행하면서 발생하는 오류를 탐색하는 방식인데, 이때 발생하는 오류는 해당 루틴의 구조가 문제이거나, 하위 레이어의 문제이므로, 함수 종속 관계도를 토대로 디버깅할 루틴의 범위를 좁혀나갈 수 있다. bottom up 방식은 최하위 레이어의 루틴을 하나씩 디버깅하는 방식인데, 각 루틴의 안정성을 높일 수 있어 상위 레이어의 루틴에서 발생할 문제를 최소화하거나 미연에 방지할 수 있다.

이상에서 소개한 것과 같이, 레이어드 플로우차트 방식은 다양한 장점을 갖고 있다. 하지만 단점도 일부 갖고 있는데, 가장 큰 단점으로는 전체적인 계획이 종료된 후에야 개발이 가능하다는 점을 꼽을 수 있다. 또한, 도중에 계획이 변경되는 경우가 많기 때문에 프로젝트가 복잡해질 수 있다. 전자의 경우 소규모 프로젝트에서는 큰 문제가 되지 않지만 대규모 프로젝트에서는 설계 기간이 매우 길어질 우려가 있다. 한편 후자의 경우, 계획 변경 전까지 제작한 모듈을 다시 활용하는 것으로 이러한 문제를 경감할 수 있다.

이러한 분할 방법의 장점은 전체적인 프로세스의 진행 양상을 의사 코드에 가깝게 인식할 수 있는 형태로 변환하는 동시에 다른 함수의 동작 상세 사항과 무관하게 각 루틴이 정의될 수 있기 때문에, 프로젝트 참가자들이 전반적으로 공통된 인식을 가지는 데에 도움이 됨과 동시에 각 인원별로 담당 업무를 분리하는 데에도 도움이 된다.

레이어드 플로우차트 방식의 또 다른 장점으로는 프로그램에 필요한 리소스와 객체들이 비교적 명확하게 정의된다는 점이다. 상위 레이어에서 객체를 정의하면 하위 레이어로 내용을 상세화하면서 각 객체가 수행해야 할 기능들이 서브루틴으로서 추가되기 때문에, 계층화가 끝나는 시점에는 각 객체에 필요한 기능(멤버 함수)들이 모두 정의된다.

ii) 협업을 위한 변수 교환 구조

각 모듈 별로 서로 다른 헤더파일을 보유하기 때문에, 서로 다른 모듈 사이에서 정보를 교환하기 위해서는 별도의 변수를 정의하여야 한다. 본 프로젝트에서는 이를 의식하여, 서로 무관한 모듈 사이에서 정보를 주고 받을 때에는 배열을 활용하는 방식을 취하였다.

iii) MFC의 활용

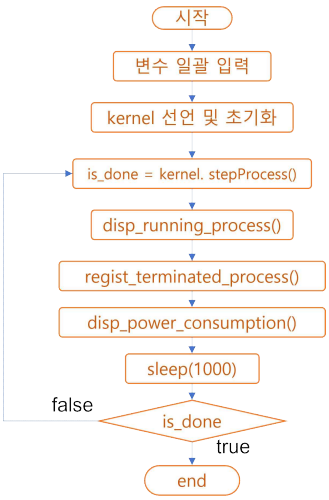
프로그램 설계 시 버그를 찾는 과정은 필수적인 사항이다. MFC의 클래스 멤버들을 활용하면서 디버깅 과정을 수행할 수 있다. 스케줄링 알고리즘 기초 개발은 main함수 내에서 디버깅을 통해 확인하며 설계하되, 이후 설계된 스케줄링 알고리즘을 기반으로 main함수 내에서 진행했던 스케줄링 과정을 MFC 이벤트 처리기 내에서 진행하도록 한다. 이때 버그 및 오류 수정은 AfxMessageBox()를 오류가 발생하는 코드 영역에 넣어주는 방식을 활용함으로써 원활하게 디버깅 과정을 수행할 수 있다.

2. 프로그램 구성

i) 메인 루틴

- 메인 루틴의 흐름도

메인 루틴의 흐름도는 레이어드 플로우차트의 첫 번째 레이어의 흐름도와 동일하다.



함수명	기능	입력값	반환값	관련 상수
kernel. stepProcess()	실행할 때마다 프로세스 스케줄링을 수행하여 프로세서를 할당 및 프로세서의 작업을 진행한다. 매 차례에서 프로세스의 상태와 새로 종료된 프로세스, 재시동된 cpu의 번호를 갱신한다.	int* running_process 현재 프로세서별로 할당받은 프로세스의 번호를 나타내는 배열. int* cpu_restarted 현재 프로세서별로 재시동 여부를 나타내는 배열 int* newly_terminated_process 현재 프로세서 진행 후 각 프로세스의 종료 여부를 나타내는 배열. double* weight 각 프로세스를 정렬할 때의 가중치를 지정하는 배열.	bool is_done 커널에 등록된 모든 프로세스가 종료 여부를 나타내는 불린.	CPU_NO_PROCESS -1
disp_running _process()	현재 실행중인 프로세스의 번호를 GUI에 표시한다.	int* running_process 현재 프로세서별로 할당받은 프로세스의 번호를 나타내는 배열	없음	없음
regist_termi	현재 프로세스	int*	없음	없음

nated_proce ss()	실행으로 인해 종료된 프로세스를 GUI에 표시한다.	newly_terminated_proc ess 현재 프로세서 진행 후 각 프로세스의 종료 여부를 나타내는 배열.		
disp_power_ consumption ()	현재까지의 프로세스 실행으로 인해 소모된 전력의 총량을 GUI에 표시함	int* power_usage 각 프로세서가 소모한 전력의 총량을 나타내는 변수	없음	없음

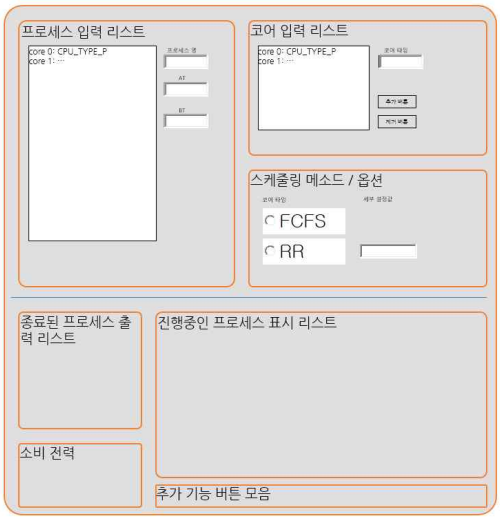
위의 과정에서 새롭게 정의한 함수를 토대로, 새롭게 정의해야 할 객체 혹은 그 멤버 변수, 멤버 함수의 목록을 다음과 같이 추출할 수 있다.

객체명		원형	상세
kernel	추가할 멤버 함수	bool stepProcess()	실행할 때마다 프로세스 스케줄링을 수행하여 프로세서를 할당 및 프로세서의 작업을 진행한다. 매 차례에서 프로세스의 상태와 새로 종료된 프로세스, 재시동된 cpu의 번호를 갱신한다.

kernel: os의 커널을 묘사한 클래스.

첫 번째 단계는 1번째 레이어를 구상하는 단계로, 메인 루틴의 동작 방식을 결정하는 단계이기도 하다. 커널이라는 객체의 동작 방식을 정하는 것이 핵심이었으며, 실시간성을 중요시하는 팀원들의 의견을 반영하여, 모든 프로세스를 일괄적으로 처리하는 형식이 아닌, 명령어를 실행할 때마다 각 프로세스의 상태를 새로 갱신하는 방식으로 커널을 구성하기로 하였다.

한편, GUI는 MFC의 다이얼로그 객체를 통해 구현하기로 하였으며, 다음과 같은 인터페이스를 구상하였다.



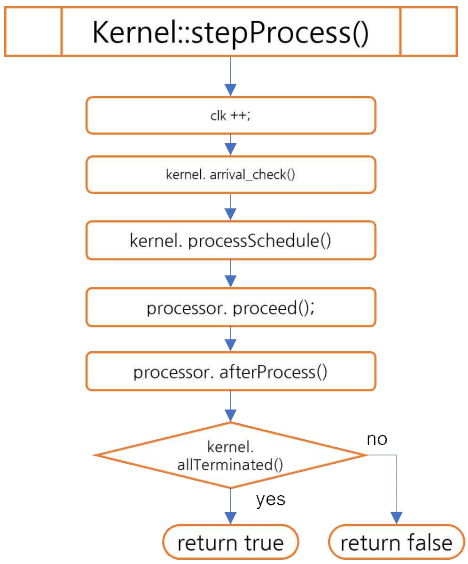
또한, 예상되는 문제로 MFC와 커널의 연결이 용이하지 않을 것을 대비하여, 커널의 연산 결과를 기본 자료형인 배열의 형태로 반환할 수 있도록 계획하였다.

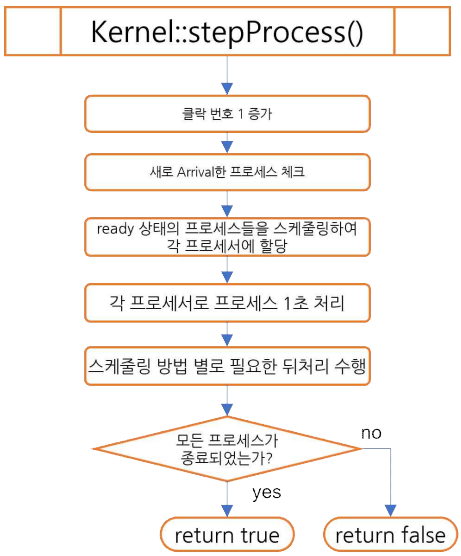
- 인원 할당

전체적인 알고리즘 프레임워크 구상 및 제작은 모든 팀원이 함께 진행하였고, MFC를 취급할 수 있는 박형진, 박동진 학생이 GUI를 집중적으로 담당하기로 하였으며, 박순용, 오수환 학생이 커널의 구현을 집중적으로 담당하기로 하였다.

ii) 커널(프레임)의 구상

커널 구상에 있어서는 각 스케줄링 알고리즘의 유사성을 감안하여 프레임워크의 형식으로 구현한 후, 스케줄링 알고리즘만 선택할 수 있도록 코드를 작성하기로 하였다. 또한, 교과 학습 시간에 학습한 내용의 연관성을 고려하여, 가능한 한 실제 커널의 동작 방식과 유사한 움직임을 보일 수 있도록 구상하기로 하였다. 프레임워크의 플로우차트(= 2번째 레이어)는 다음과 같다.





함수명	기능 상세	입력	출력	관련 상수
Kernel::arrival Check()	kernel. process_table에 등록된 각 프로세스의 AT를 검사하여 created 상태에서 -> ready 상태로 전환함. - 각 PCB의 WT, TT, NTT을 갱신 참고: - 아직 Arrival 되지 않은			PROCESS_STATE_ CREATED 0 PROCESS_STATE_ READY 1

	프로세스의 상태는 created - Arrival 된 후:경우에 따라 ready, running, terminated 등의 상태를 가짐(asleep 상태는 고려하지 않음) > ready: 프로세스가 할당되지 않은 상태 > running: 프로세스가 할당된 상태 > terminated: 프로세스가 종료된 상태			PROCESS_STATE_RUNNING 2 PROCESS_STATE_TERMINATED 3
Kernel::allTerminated()	프로세스 테이블의 각 프로세스가 전부 종료됐는지의 여부를 판단		bool result: 프로세스가 모두 종료된 경우 True, 그렇지않은 경우 False	
Kernel::processSchedule()	스케줄링 메소드는 Kernel.smethod를 참조한다. cpu_restart의 모든 성분을을False으로 초기화한다. CPU 코어에 할당되어 있던 프로세스가 종료된 경우 PCB에서 해당프로세스의 상태를 terminated로변경 레디큐를정렬(smethod를 참조) 레디큐에서디큐한프로세스 번호를 할당한다. 동시에 새로 할당한 프로세스의 상태를 running으로 변경한다(레디큐가비어있는경우 -1을 할당한다). Elseif 스케줄링 메소드가 선점 방식이면서 레디큐최상단프로세스의 가중치가 현재 프로세스보다 더 높은 경우 PCB에서 기존에 수행하던 프로세스의 상태를 ready로 변경 레디큐정렬(smethod참조) 레디큐에서디큐한프로세스를 할당한다. 새 프로세스의 상태를 running으로 변경한다. (레디큐가비어있는경우 -1을 할당한다.) #기존에 할당되었던 프로세스를 레디큐에인큐한다. Elseif 프로세스가 할당되어있지않은 CPU가 존재하는 경우 레디큐정렬(smethod참조) 레디큐에서디큐한프로세스를 할당한다. 새 프로세스의 상태를	int*running_process: 프로세스 테이블에 running 상태로 정의된 프로세스의 목록. 길이는 P와 같다. running_process[i] : i번째 CPU 코어에 할당된 프로세스의 번호 int* cpu_restarted: 코어 시동을 나타내는 길이 P의 배열. 프로세스가 할당되어있지않던 코어에 프로세스가 할당된 경우 해당 성분을 True를 변경. cpu_restated[i]==True ->i번 코어가 새로 시동함		CPU_NO_PROCESS -1

	running으로 변경한다. (레디큐가비어있는경우 -1을할당한다). 프로세스가 할당되어 있지 않던 CPU에 새로운 프로세스가 할당된 경우cpu_restarted[i] = True를 할당한다. 프로세스 테이블을 참조하여 현재 동작 상태의 프로세스의 번호를 배열로 편집			
Processor::proceed()	현재 할당된 프로세스와 이전에 할당되었던 프로세스를 토대로 시동 전력을 전력 소비 총량에 추가. 프로세서의각 CPU 코어에 할당된 프로세스가 존재하는 경우,각코어의 타입(P/E)에 따라서 프로세스의 Remaining Time을 줄이고, 각 cpu의 동작 전력 소비 총량을 갱신한다.	int*running_process: 프로세스 테이블에 running 상태로 정의된 프로세스의 목록. 길이는 P와 같다. running_process[i] : i번째 CPU 코어에 할당된 프로세스의 번호		CPU_NO_PROCESS -1
		int* cpu_restarted: 코어 시동을 나타내는 길이 P의 배열. cpu_restated[i]==True -> i번 코어가 새로 시동함		
Processor::afterProcess()	스케줄링 메소드의 after_process()를 호출함	미정		

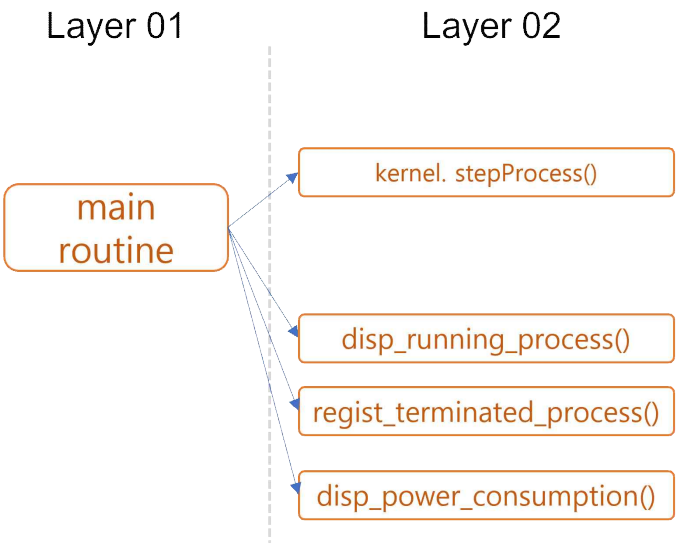
위의 과정에서 새롭게 정의한 함수를 토대로, 새롭게 정의해야 할 객체 혹은 그 멤버 변수, 멤버 함수의 목록을 다음과 같이 추출할 수 있다.

객체명		원형	상세
kernel	추가할 멤버 변수	int clk;	kernel.stepProcess의 실행 횟수를 측정하는 변수. 인가된 클럭의 개수로서의 의미도 갖는다.
		PCB* process_table	여러개의 PCB로 구성된 프로세스 관리용 테이블
		Processor processor	프로세서를 묘사한 클래스. cpu 코어의 개수를 지정할 수 있다.
	추가할 멤버 함수	void arrivalCheck()	kernel. process_table에 등록된 각 프로세스의 AT를 검사하여 created 상태에서 ready 상태로 전환함.
		bool allTerminated()	프로세스 테이블의 각 프로세스가 전부 종료됐는지의 여부를 판단
		void processSchedule()	프로세스 테이블을 참조하여 프로세서에 프로세스를 할당
		void afterProcess()	프로세스 처리 이후 스케줄링 방법에 따라 필요한 후처리를 진행함

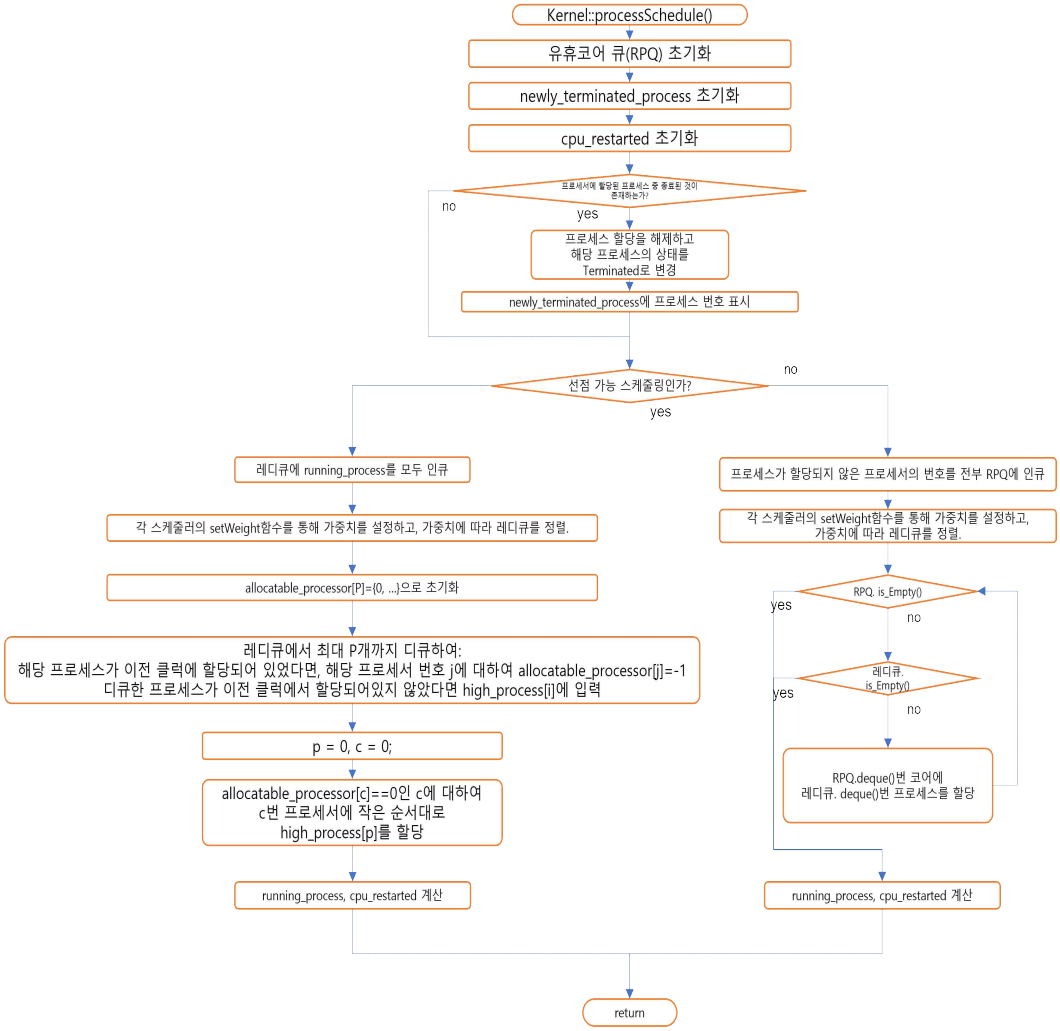
PCB	추가할 멤버 변수	int AT	프로세스의 수행이 요청된 시각(혹은 클럭 번호)
		int BT	E 코어로 프로세스를 수행하는데 소요되는 클럭의 수
		int WT	프로세스 요청 이후로 처리되지 않은 시간(클럭)의 총합
		int TT	프로세스 종료시 응답 시간의 값을 저장하는 변수
		double NTT	프로세스 종료시 정규화 응답 시간의 값을 저장하는 변수
		int state	프로세스의 상태를 저장하는 변수
processor	추가할 멤버 함수	proceed()	프로세서에 할당된 프로세스를 처리함.

PCB: 각 프로세스별로 관리를 위한 정보를 담고 있는 클래스.
processor: 여러개의 cpu 코어로 구성된 연산 유닛.

여기까지가 2번째 레이어의 구성이다. 함수 종속 관계도로 지금까지의 진행 상황을 표현하면 다음과 같이 나타낼 수 있다.



이상의 알고리즘은 스케줄링 방법과 무관하게 항상 동일한 구조를 취하며, 따라서 프레임 워크는 위 구조를 토대로 구성되었다. 스케줄링 방법에 따라 달라지는 부분은 kernel.processScheduling()에 한정되며, 모듈화를 위하여 scheduler라는 가상 클래스를 지정하고, 각 스케줄링 방법에 따라서 서로 다른 자식 클래스가 존재한다.
아래는 이를 구현하기 위한 흐름도(3번째 레이어)를 나타낸 것이다.



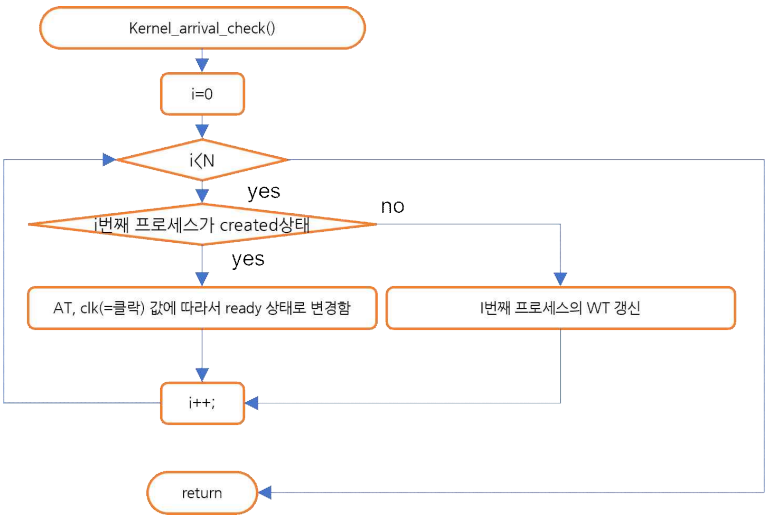
함수명	기능 상세	입력	출력	관련 상수
Scheduler::set_weight()	프로세스 테이블의 각 PCB에 weight를 지정하고, 그 값을 배열의 형태로 반환	PCB* processtable: 프로세스 테이블. int N: 프로세스 개수 double* weight 가중치를 저장하는 배열. weight[i]= i번 프로세스의 가중치		
Que::enque()	큐에 원소를 인큐함	int x		
Que::deque()	큐에서 원소를 디큐함	없음	int x: 큐의 머리 부분에 위치했던 원소	
Que::sort()	큐의 원소를 주어진 weight에 따라 정렬함	double* weight 가중치를 저장하는 배열. weight[i]: 원소 I의 가중치		

이상의 구조에서 추출한 객체 목록은 다음과 같다:

객체명		원형	상세
Que	추가할 멤버 변수	int body	큐에서 자료를 실제로 저장하는 배열
	추가할 멤버 함수	void enque()	큐에서 원소를 인큐함
		void deque()	큐에서 원소를 디큐함
		void clear()	큐의 내용물을 모두 비움
		bool is_empty()	큐의 내용물이 가득 찬 경우 true를 반환
		void sort()	큐의 원소를 주어진 weight에 따라 정렬함
scheduler	추가할 멤버 변수	int preemptive	프로세스의 수행이 요청된 시각(혹은 클럭 번호)
	추가할 멤버 함수	void setWeight()	큐의 내용물 정렬에 사용할 weight를 산정함

3번째 레이어의 다른 함수들에 대해서도 아래에 열거하였다.

a) Kernel::arrivalCheck()

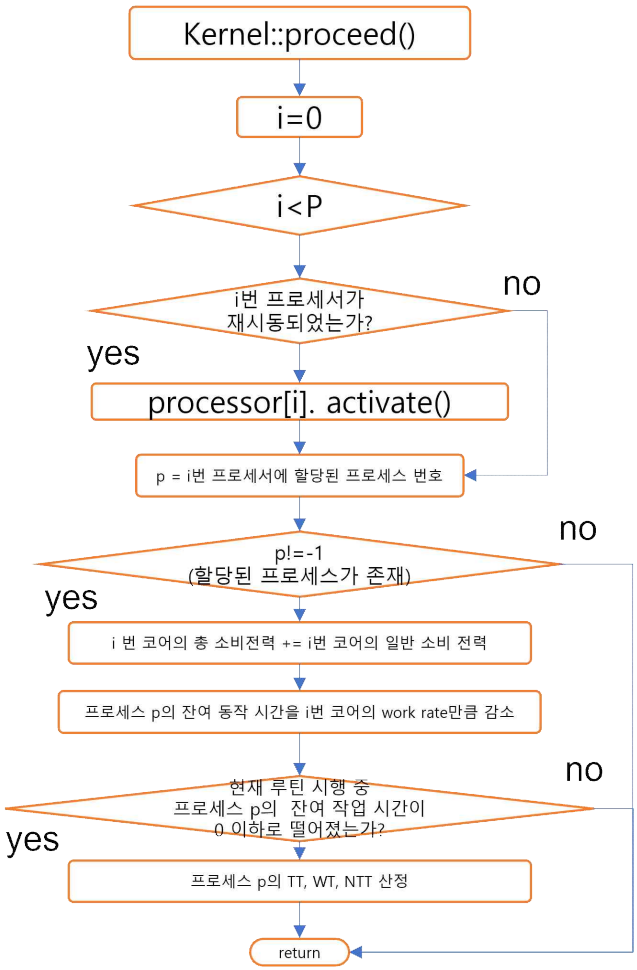


arrivalCheck()의 구현에 필요한 변수는 다음과 같다.

객체명		원형	상세
kernel	추가할 멤버 변수	int N	프로세스의 개수

충분히 간단하여 추가적으로 분할할 기능은 없다고 판단한다.

b) Kernel::proceed()



새로 정의할 함수는 다음과 같다:

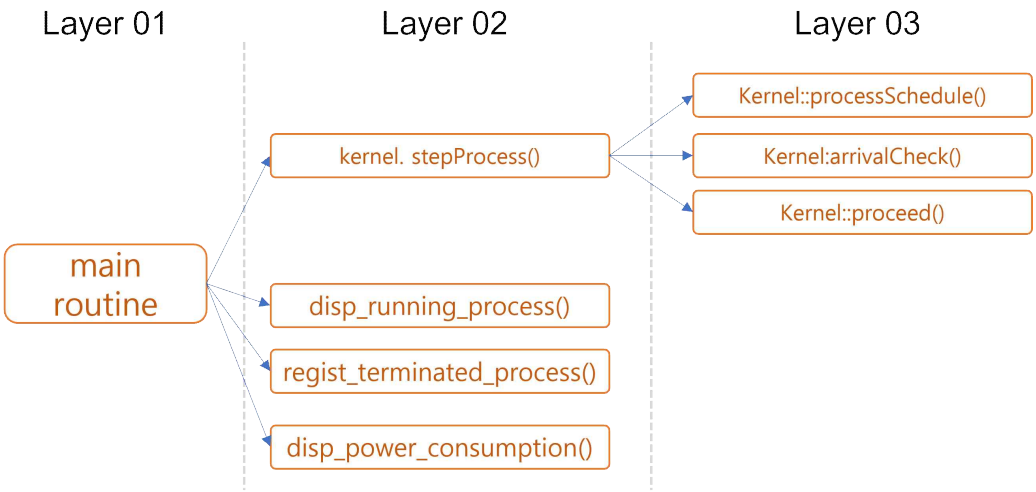
함수명	기능 상세	입력	출력	관련 상수
Core::activate()	(각 코어가 재시동 되었을 때) 에너지 소비량을 증가시킨다.	없음	없음	

새로 정의할 객체/멤버 요소는 다음과 같다:

객체명		원형	상세
kernel	추가할 멤버 변수	int P	프로세서의 개수. 최대 4까지 설정 가능하다.
	추가할 멤버 함수	proceed()	코어에 특정 프로세스를 할당하여 잔여 작업량을 감소시키고, 코어의 소모 전력을 증가시킨다.
Processor	추가할 멤버 변수	Core core[4]	cpu 코어를 묘사한 객체.

core	추가할 멤버 변수	int type	cpu 코어의 타입(P/E type)
		double regular_consumption	cpu 코어의 일반 소모전력
		double starting_consumption	cpu 코어의 시동 전력
		int work_rate	cpu 코어가 1클럭동안 처리하는 작업의 양
	추가할 멤버 함수	activate()	코어의 총 소모 전력을 시동 전력만큼 증가시킨다.

이상에서 3번째 레이어에서의 모든 함수를 살펴볼 수 있었다. 함수 종속 관계도를 활용하여 표현하면 다음과 같다:

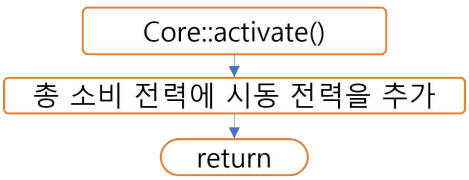


다음으로는 레이어 4의 내용을 살펴본다. que에 대한 내용을 제외하고 살펴보면 다음과 같다.

a) scheduler.setWeight():

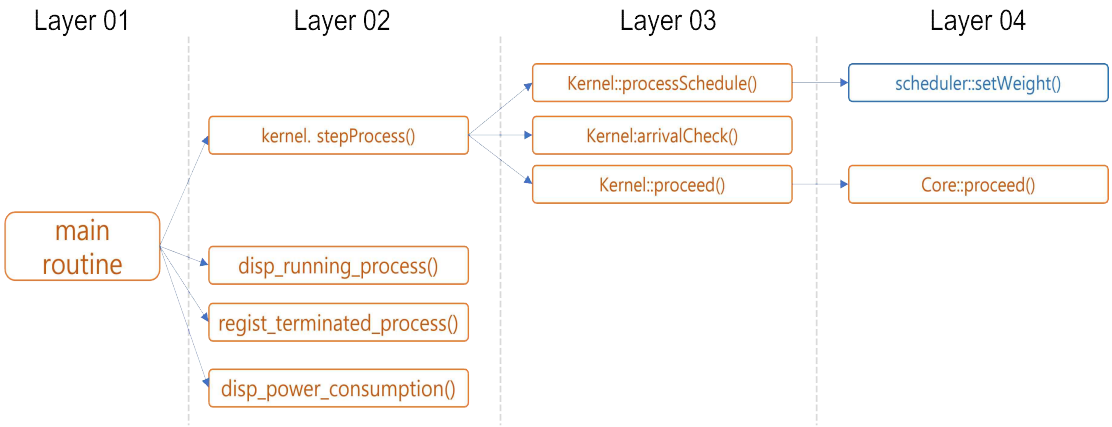
이 함수는 스케줄링 방법에 따라 다른 동작을 보이는 함수로, 상세 사항은 후술한다.

b) core.activate:



별도로 추가되는 내용은 없다.

이상에서, 4번째 레이어까지 포함하는 커널(프레임워크)의 함수 종속 관계도는 다음과 같다(scheduler::setWeight() 함수는 프레임 워크의 범위를 벗어나는 함수라는 의미에서 푸른색으로 표시하였다).



또한, 지금까지 선언한 객체 및 멤버 요소에 대한 리스트는 다음과 같이 종합할 수 있다:

객체명		원형	상세
kernel	추가할 멤버 변수	int N	프로세스의 개수
		int P	프로세서의 개수. 최대 4까지 설정 가능하다.
		int clk;	kernel.stepProcess의 실행 횟수를 측정하는 변수. 인가된 클럭의 개수로서의 의미도 갖는다.
		PCB* process_table	여러 개의 PCB로 구성된 프로세스 관리용 테이블
		Processor processor	프로세서를 묘사한 클래스. cpu 코어의 개수를 지정할 수 있다.
	추가할 멤버 함수	bool stepProcess()	실행할 때마다 프로세스 스케줄링을 수행하여 프로세서를 할당 및 프로세서의 작업을 진행한다. 매 차례에서 프로세스의 상태와 새로 종료된 프로세스, 재시동된 cpu의 번호를 갱신한다.
		void arrivalCheck()	kernel. process_table에 등록된 각 프로세스의 AT를 검사하여 created 상태에서 ready 상태로 전환함.
		bool allTerminated()	프로세스 테이블의 각 프로세스가 전부 종료되었는지의 여부를 판단
		void processSchedule()	프로세스 테이블을 참조하여 프로세서에 프로세스를 할당
		void afterProcess()	프로세스 처리 이후 스케줄링 방법에 따라 필요한 후처리를 진행함
		void proceed()	코어에 특정 프로세스를 할당하여 잔여 작업량을 감소시키고, 코어의 소모 전력을 증가시킨다.
PCB	추가할 멤버 변수	int AT	프로세스의 수행이 요청된 시각(혹은 클럭 번호)
		int BT	E 코어로 프로세스를 수행하는데 소요되는 클럭의 수
		int WT	프로세스 요청 이후로 처리되지 않은 시간(클럭)의 총합
		int TT	프로세스 종료 시 응답 시간의 값을 저장하는 변수
		double NTT	프로세스 종료 시 정규화 응답 시간의 값을 저장하는 변수
		int state	프로세스의 상태를 저장하는 변수
Processor	추가할 멤버 변수	Core core[4]	cpu 코어를 묘사한 객체.
	추가할 멤버 함수	proceed()	프로세서에 할당된 프로세스를 처리함.
core	추가할 멤버 변수	int type	cpu 코어의 타입(P/E type)

		double regular_consumption	cpu 코어의 일반 소모 전력
		double starting_consumption	cpu 코어의 시동 전력
	추가할 멤버 함수	int work_rate	cpu 코어가 1클럭 동안 처리하는 작업의 양
		activate()	코어의 총 소모 전력을 시동 전력만큼 증가시킨다.
scheduler	추가할 멤버 변수	int preemptive	프로세스의 수행이 요청된 시각(혹은 클럭 번호)
	추가할 멤버 함수	void setWeight()	큐의 내용물 정렬에 사용할 weight를 산정함
Que	추가할 멤버 변수	int body	큐에서 자료를 실제로 저장하는 배열
		int head	인큐 위치
		int tail	디큐 위치
		int cap	최대 원소 개수
		int n	원소 개수
	추가할 멤버 함수	void enque()	큐에서 원소를 인큐함
		void deque()	큐에서 원소를 디큐함
		void clear()	큐의 내용물을 모두 비움
		bool is_empty()	큐의 내용물이 가득 찬 경우 true를 반환
		bool is_full()	큐의 내용물이 가득 찬 경우 true를 반환
		void sort()	큐의 원소를 주어진 weight애 따라 정렬함

※ Que 클래스의 세부 사항에 대한 설명은 생략한다.

위 표에서는 각 클래스의 생성자와 소멸자는 표현하지 않았다.
코드로의 구현은 부록을 참조한다.

3. 스케줄러의 구성

스케줄러는 프레임워크 상에서 사용되는 일종의 펑터(functor)로, 온전히 setWeight() 함수 하나만을 위한 클래스이다. setWeight() 함수의 명확한 이해를 위해서는 Kernel의 생성자를 살펴보는 것이 좋다. 아래는 Kernel의 생성자의 코드의 일부를 나타낸 것이다.

```
#define SCHEDULING_FCFS     0
#define SCHEDULING_RR       1
#define SCHEDULING_SPN      2
#define SCHEDULING_SRTN     3
#define SCHEDULING_HRRN     4
#define SCHEDULING_UDSM     5
```

software.hpp의 일부

```
Kernel::Kernel(int n, int p, int sm, int* at, int* bt, int* core_types, int* settings): N(n), P(p), smet
hod(sm), clk(-1) {
    // 프로세스 테이블 생성 및 프로세스 등록
    process_table = new PCB[N];
    for (int i = 0; i < N; i++)
        process_table[i].registProcess(at[i], bt[i]);

    // 레디큐 생성
    RQ = new Que(N);
    // 유휴코어큐 생성
    RPQ = new Que(P);
    // 프로세서 생성
    processor = new Processor(P);
    processor->setType(core_types);
    // 스케줄러 생성

    switch (smethod) {
    case SCHEDULING_FCFS:
        scheduler = new FCFS();
        break;
    case SCHEDULING_RR:
        std::cout << "RR 초기화" << std::endl;
        scheduler = new RR(settings[0], settings[1], settings[2]); // tq, n, p
        scheduler->setAttr(settings);
        break;
    /*...생략...*/
    default:
        scheduler = new FCFS();
    }
    return;
}
```

software.cpp의 일부

이상에서 확인할 수 있듯이, kernel을 정의할 때 입력하는 인수 sm의 값에 따라서 kernel.scheduler의 클래스가 달라진다. 또한 실제로 kernel.scheduler가 참조되는 부분은 kernel.processSchedule() 함수로 한정된다.

더 자세한 이해를 위하여 코드를 참조한다. 아래는 software.cpp의 void Kernel:: processSchedule() 함수의 정의이다.

```
void Kernel::processSchedule(int* running_process, int* cpu_restarted, int* newly_terminate
d_process, double* weight) {
    // running_process는 이전 단계에서의 각 cpu의 동작 상태를 나타내는 배열
    // cpu_restart는 이번 단계에서 cpu의 재시동 여부를 나타낼 배열
    int p, c;
    RPQ->clear();                                   // 유휴코어 큐 초기화
    for (int i = 0; i < N; i++)                     // 종료프로세스 배열 초기화                    (A)
        newly_terminated_process[i] = 0;

    // 종료된 프로세서 유무 확인하여 cpu, PCB 정리
    for (int i = 0; i < P; i++) {  // 프로세서의 0~(P-1)번 코어에 대하여
        cpu_restarted[i] = 0;

        p = processor->core[i].allocated_process; // p: i번째 코어에 할당되어있던 프로세스 번호
        //p번 프로세스가 끝난 상황이면                                                            (B)
        if ((process_table[p].RT <= 0)&&(p!=CPU_NO_PROCESS) ){
            process_table[p].state = PROCESS_STATE_TERMINATED;
            newly_terminated_process[p] = 1;
            processor->core[i].allocated_process = CPU_NO_PROCESS;
        }
    }
    if (scheduler->preemptive == true) {
    // i.선점 방식의 스케줄링 알고리즘

        // 레디큐에 running 상태의 프로세스를 모두 인큐
        for (int i = 0; i < P; i++) {
            if (processor->core[i].allocated_process != CPU_NO_PROCESS) {
                RQ->enque(processor->core[i].allocated_process);
            }
        }
        scheduler->setWeight(process_table, weight, N);
        RQ->sort(weight);
        // 프로세스 스케줄링: 프로세서별 가중치를 계산하고 가중치에 따라 레디큐를 정렬함

        // allocatable_processor, high_process 초기화
        int* allocatable_processor = new int[P];
        int* high_process = new int[P];
        int hp = 0;                                 // high_process 원소 개수

        for (int i = 0; i < P; i++) {                                                           (C)
            allocatable_processor[i] = 0;
            high_process[i] = -1;
        }

        // 레디큐에서 최대 P개까지 디큐하여 allocatable_processor, high_process를 채움
        bool allocated;
        for (int i = 0; i < P; i++) {
            allocated = false;
            if (!RQ->is_empty()) {
                p = RQ->deque();

                for (int j = 0; j < P; j++) {
                    // 해당 프로세스가 이전 클럭에 할당되어 있었다면,
                    // j번째 프로세서는 재할당을 수행하지 않음을 표시
                    if (processor->core[j].allocated_process == p) {
                        allocatable_processor[j] = -1;
                        allocated = true;
                        break;
```

```
                    }
                }
                // 해당 프로세스가 이전 클럭에 할당되어 있지 않았다면
                // i번째 프로세스는 새로 할당해야하는 프로세스 목록에 추가
                if (not allocated) {
                    high_process[hp] = p;
                    hp++;
                }
            }
        }
        std::cout << "hp=" << hp << std::endl;
        std::cout << "allocatable_processor: |";
        for (int i = 0; i < P; i++)
            std::cout << allocatable_processor[i]<<"|";
        std::cout << std::endl << "high_process: |";
        for (int i = 0; i < hp; i++)
            std::cout << high_process[i]<< "|";
        std::cout << std::endl;

        p = 0;
        for (c = 0; c < P; c++) {
            if (p >= hp)
                break;
            if (allocatable_processor[c] == -1)
                continue;
            if (processor->core[c].allocated_process!=CPU_NO_PROCESS)
                process_table[processor->core[c].allocated_process].state = PROCESS_STATE_
READY;
            process_table[high_process[p]].state = PROCESS_STATE_RUNNING;
            processor->core[c].allocated_process = high_process[p];
            p++;
        }

        delete[] allocatable_processor;
        delete[] high_process;

    }
    else {
    // ii. 비선점 방식의 스케줄링 알고리즘
        for (int i = 0; i < P; i++) {
            if (processor->core[i].allocated_process == CPU_NO_PROCESS) {
                RPQ->enque(i);
            }
        }

        scheduler->setWeight(process_table, weight, N);
        RQ->sort(weight);                                                    (D)

        while (!RPQ->is_empty()) {
            if (RQ->is_empty()) { break; }
            c = RPQ->deque();       // c: 주목하는 코어의 번호
            p = RQ->deque();        // p: 할당할 프로세스 번호

            processor->core[c].allocated_process = p;
            process_table[p].state = PROCESS_STATE_RUNNING;
        }
    }
```

```
    // 재시동된 CPU 번호와 실행할 프로세스를 계산
    for (int i = 0; i < P; i++) {
        if ((running_process[i] == CPU_NO_PROCESS) && (processor->core[i].allocated_proces
s != CPU_NO_PROCESS))
            cpu_restarted[i] = 1;                                                    (E)
        running_process[i] = processor->core[i].allocated_process;
    }
    return;
}
```

위 코드에서 확인할 수 있듯이, 함수 kernel::processScheduling()은 5개의 영역으로 구분된다.

(A) 영역은 newly_terminated_process , 유휴 코어 큐 등을 초기화하는 영역이다.

(B) 영역에서는 스케줄링에 앞서, 종료된 프로세스들을 해당 프로세서로부터 할당 해제하는 작업이 이루어진다.

(C) 영역에서는 선점 방식 스케줄러에 대한 프로세스 스케줄링이 이루어진다. Running 프로세스와 ready 프로세스를 통틀어 가중치를 다시 산정한 후, 가장 높은 점수를 받는 프로세스를 다시 할당한다. 이때, 이전 클럭에서 동작한 프로세스와 일치하는 프로세스가 존재하는 경우 별도의 동작은 하지 않으며, 이전 클럭에서 동작하지 않았던 프로세스에 대해서만 새로운 프로세스 할당이 이루어진다. 이러한 방식을 통해서 무의미한 코어 교환을 피할 수 있다.

(D) 영역에서는 비선점 방식 스케줄러에 대한 프로세스 스케줄링이 이루어진다. 마찬가지로 (B)에서 할당이 해제된 프로세서들을 포함하여, 프로세서가 할당되지 않은 프로세서들을 RPQ에 인큐하고, RPQ와 RQ 중 어느 하나의 내용물이 비어있지 않는 한 계속해서 RPQ.deque()번 프로세서에 RQ.deque()번 프로세스를 할당한다.

(E) 영역에서는 cpu_restatred와 running_process를 확정한다.

이하에서는 setWeight() 함수의 동작 방식을 설명한다.

스케줄러

본 프로그램에서의 스케줄링은 앞에서 확인하였듯이 레디 프로세스의 리스트를 가중치에 따라 정렬하고, 해당 클럭 주기에 사용할 프로세스를 선택한 후 각 프로세서에 할당하는 방식으로 진행된다. Scheduler는 각 프로세스별로 가중치를 부여하는 함수 setWeight()를 위한 가상 클래스이며, 다른 스케줄러 FCFS, RR, SPN, ... 등은 Scheduler를 상속받아 만들어낸다. 아래는 Scheduler의 코드이다.

```
class Scheduler {
public:
    // preemptive는 선점 가능 여부를 나타내는 속성입니다. 선점 가능한 스케줄링 메소드의 경우 True로 지정합니다.
    bool preemptive;

public:
    // 스케줄러 클래스의 생성자입니다. 필요한 경우 오버라이딩 하여 사용하십시오..
    Scheduler();
    // 속성 설정자입니다. 필요한 경우, 아래에서 속성 초기화를 수행합니다.
    virtual void setAttr(int* settings) {}
    // 스케줄링에서 사용할 가중치의 배열인 weights를 결정하는 함수입니다.
    virtual void setWeight(PCB* process_table, double* weights, int n, int clk) {}
    // Kernel::afterProcess에서 사용하는 함수입니다. 적절히 오버라이딩해서 사용하시면 됩니다.
    void afterProcess();
};
```

1. FCFS

FCFS는 First Come First Service의 약자로, 프로세스의 Arrival이 빠른 순서로 동작 순서를 결정하며, 이는 프로세스 진행에 따라서 달라지는 값이 아니기 때문에, 가중치는 단 한 번만 계산한다. 이때 가중치는 -AT와 같이 계산하며, 동일한 시점에 arrival한 프로세스 사이에서 순서가 뒤바뀌는 것을 막기 위해서 (i+1)/(n+1)만큼을 가중치에서 더 빼주었다. 아래는 FCFS 클래스의 코드이다.

```
class FCFS : public Scheduler {
public:
    bool done_once;

public:
    // FCFS 클래스의 생성자입니다. preemptive 설정, time quantum 등,
    // 필요한 속성을 만들거나 초기화합니다.
    FCFS() {
        this->preemptive = false;
        this->done_once = false;
    };

    // FCFS의 가중치 설정 함수입니다.
    // process_table: PCB 로 구성된 배열입니다. 즉, process_table[i]는 i번째 프로세스의 PCB를 나타냅니다.
    // process_table[i].AT 등의 속성을 자유롭게 사용할 수 있습니다.
    // 다만, process_table[i]의 값은 변경하지 않도록 주의합니다.
    // weight: weight[i]는 i번째 프로세스의 가중치를 나타냅니다. 참조에 의해 호출되며,
    // 본 함수의 목적은 이 weight의 원소를 적절하게 변경해주는 것입니다.
    // n: PCB의 개수(= 프로세스의 개수)입니다.
    void setWeight(PCB* process_table, int* weight, int n, int clk) {
        // 가중치를 이미 계산한 바가 있는 경우 아무 동작도 하지 않습니다.
        if (done_once == true)
            return;
```

```
        for (int i = 0; i < n; i++) {
            weight[i] = -double(process_table[i].AT) - (double)(i+1) / (n+1);
            // (1+i)/(n+1)을 빼주는 이유는 동일한 AT를 갖는 두 프로세스의 순서가
            // 버블 정렬에 의하여 매번 뒤바뀌는 상황을 막기 위함입니다.
        }
        done_once = true;
        return;
    }
};
```

2. RR

RR 방식은 Round Robin 방식의 약자로, 각 프로세스의 NTT를 비교적 일정하게 수행하기 위하여 고안된 프로세싱 방식이다. 각 프로세스가 사용자가 지정한 time quantum만큼의 시간만큼 프로세서를 돌아가면서 사용하는 방식을 말한다. 대표적인 선점 방식의 스케줄링 기법이다.

```
class RR : public Scheduler {
public:
    int N;                          // 프로세스 개수
    int P;                          // 프로세서 개수
    int* tc;                       // 타임 카운터. 프로세스 연속 실행 횟수 확인
    int tq;                        // 타임 퀀텀
    Que* SRQ;                       // sub_readyqueue

public:
    RR(int delta, int n, int p): N(n), P(p){
        this->preemptive = true;        // 선점 가능성
        this->tc = new int[N];          // 각 프로세스별 처리 횟수 확인용 배열
        this->tq = delta;               // 타임 퀀텀
        for (int i = 0; i < N; i++)
            tc[i] = 0;

        this->SRQ = new Que(N);
        return;
    }
    ~RR() {
        delete[] tc;
        delete SRQ;
        return;
    }
    void setWeight(PCB* process_table, double* weight, int n, int clk) {
        int i = 0;
        int rp = 0;

        for (i = 0; i < N; i++) {

            // 새로 arrival한 프로세스를 SRQ에 인큐
            if (clk == process_table[i].AT)
                SRQ->enque(i);

            // 프로세서에 할당된 프로세스의 tc ++
            if (process_table[i].state == PROCESS_STATE_RUNNING)
                tc[i]++;

            // rp = 프로세스 할당을 재고할 필요가 없는 프로세서의 개수.
            // 이전 클럭에서 종료된 프로세스도 고려해줘야 한다.
            if ((tc[i] < tq)  && (process_table[i].RT>=0) && (process_table[i].state==PROCESS_STAT
```

```
E_RUNNING))
                rp++;
        }
        // rp = 프로세스 할당을 재고할 프로세서의 개수.
        rp = P - rp;

        for (i = 0; i < N; i++) {
            // 종료된 프로세스의 가중치를 0으로 지정
            if (process_table[i].state == PROCESS_STATE_TERMINATED) {
                tc[i] = 0;
                weight[i] = 0;
            }
            // tq번 이상 연속으로 실행된 프로세스의 가중치를 0으로 지정하고 SRQ에 인큐
            if (tc[i] >= tq) {
                tc[i] = 0;
                weight[i] = 0;
                SRQ->enque(i);
            }
        }

        // 새로 할당해야 하는 프로세스의 개수만큼의 프로세스의 가중치를 1로 설정
        for (i = 0; i < rp; i++) {
            if (SRQ->is_empty())
                return;
            weight[SRQ->deque()] = 1;
        }
        return;
    }
};
```

프레임워크의 형태로 인해 가중치를 지정하는 형태로만 동작해야 한다는 제한 때문에, 스케줄러 내부에도 SRQ라는 형태의 큐를 만들어내야 하는 다소 비효율적인 형태를 취하게 되었지만, 동작 방식은 동일하다. 크게 3가지 동작으로 구분된다.

① SRQ 채우기: sub ready queue의 약자. 새로 인큐된 프로세스 혹은 나중에 다시 수행해야 하는 프로세스를 저장한다.
② 새로 할당해야 하는 프로세서의 개수 확인한다.
③ 프로세서 할당을 해제해야 하는 프로세스의 개수를 0으로 바꾸고, 새로 할당해야 하는 프로세스의 가중치를 1로 바꾼다.

불필요한 코어 교환에 대한 우려는 scheduler에서는 고려할 필요가 없다.

3. SPN

SPN은 가장 작업량이 적은 프로세스를 우선적으로 수행하는 비선점 방식 스케줄링 기법이다. 다음은 SPN 방식 스케줄러의 코드이다.

```
class SPN :public Scheduler {
public:
    SPN() {
        this->preemptive = false;
    }
```

```
    void setWeight(PCB* process_table, double* weight, int n, int clk) {
        for (int i = 0; i < n; i++) {
            weight[i] = -process_table[i].RT;
        }
    }
};
```

4. SRTN

SRTN은 매 순간 순간 가장 작업량이 적은 프로세스를 우선적으로 수행하는 선점 방식의 스케줄링 기법이다.
선점 방식이라는 특징을 제외하면 SPN과 다를 바가 없으며, 선점, 비선점의 특성은 프레임워크에서 처리해주기 때문에, SRTN의 setWeight() 함수는 SPN의 setWeight() 함수와 완전히 동일한 형태를 갖는다.

```
class SRTN : public Scheduler {
public:
    SRTN() {
        this->preemptive = true;
    }

    void setWeight(PCB* process_table, double* weight, int n, int clk) {
        for (int i = 0; i < n; i++) {
            weight[i] = -process_table[i].RT;
        }
    }
};
```

5. HRRN

HRRN 또한 비선점 방식으로 SPN과 매우 유사한 구조를 갖는다. 가중치만이 (AT+BT)/BT로 상이하기 때문에, 코드는 마찬가지로 매우 유사하다.

```
class HRRN : public Scheduler {
public:
    HRRN() {
        this->preemptive = false;
    }

    void setWeight(PCB* process_table, double* weight, int n, int clk) {
        for (int i = 0; i < n; i++) {
            weight[i] = (double)(clk - process_table[i].AT + process_table[i].BT) /
            (double)process_table[i].BT;
        }
    }
};
```

clk는 kernel에서 전달받는, 스케줄링이 이뤄지는 클럭의 번호이다.

6. CBTA

CBTA 방식은 본 팀에서 새로 기획한 새로운 방식의 스케줄링 방식이다. Critical Burning Time Allocation 의 약자로, 임계 시간(= critical time)을 지정하여, 그보다 잔여 작업량이 적은 프로세스는 BT가 작을수록 우선적으로 실행하고, 그렇지 않은 프로세스는 BT가 클수록 우선적으로 수행하는 방식이다. SRTN 방식의 응용이라고도 볼 수 있다.

```
class CBTA : public Scheduler {
public:
    int N;                  // 프로세스 개수
    int P;                  // 프로세서 개수
    int ct;                 // 임계 시간
    bool done_once;

public:
    CBTA(int n, int p, int critical_time) {
        this->preemptive = true;
        this->N = n;
        this->P = p;
        this->ct = critical_time;
        this->done_once = false;
    }

    ~CBTA() {
    }

    void setWeight(PCB* process_table, double* weight, int n, int clk) {
        if (done_once)
            return;

        int largest_BT = 0;
        for (int i = 0; i < N; i++) {
            if (process_table[i].BT > largest_BT)
                largest_BT = process_table[i].BT;
        }

        for (int i = 0; i < N; i++) {
            if (process_table[i].BT <= ct)
                weight[i] = ct-process_table[i].BT - double(i + 1) / double(n + 1);
            else
                weight[i] = process_table[i].BT -largest_BT - double(i + 1) / double(n + 1);
        }
        done_once = true;
        return;
    }
};
```

본 스케줄링 메소드의 주요 활용 방안은, 슈퍼컴퓨터 등, 무거운 작업을 주로 담당하는 특수 컴퓨터에서 간헐적으로 요청되는 가벼운 프로세스의 처리 등이 되겠다. 또한, 가벼운 프로세스가 간헐적으로 요구되는 실시간 시스템에서도, 임계 시간을 기준으로 해당 프로세스의 실행 여부를 결정하는 데에도 사용될 수 있다.

- 일상생활 속의 CBTA

1) 응급실에서 환자의 상태가 심각할수록 치료시간이 길지만 더 큰 우선순위를 가진다고 가정하자 이런 경우 중증 환자가 계속해서 쌓이게 된다면 짧은 치료시간을 요구하는 경증 환자에게 극심한 starvation 현상이 발생할 수 있다. 이를 극복하기 위해서 임계 치료시간(Critical Time)을 기준으로 BT가 이 임계 시간보다 낮다면 더 우선적으로 처리할 수 있

도록 한다.

2) 요리할 때 시간이 많이 소모되는 음식은 미리 물을 올려놓고 준비를 해놔야 한다고 가정하자 이 경우 음식을 만들어야 하는 순서와 상관없이 시간이 많이 소모되는 음식이 먼저 수행되므로 BT가 큰 프로세스가 더 높은 우선순위를 갖는다고 할 수 있다. 하지만 이때 극단적인 예시를 들어 설명하자면 곰국을 끓이겠다고 5초 동안 오징어 데치는 요리를 뒤로 미루게 된다. 이를 극복하기 위해서 임계 요리시간(Critical Time)을 기준으로 BT가 이 임계 시간보다 낮으면 더 우선적으로 처리할 수 있도록 한다.

- CBTA의 장단점 분석

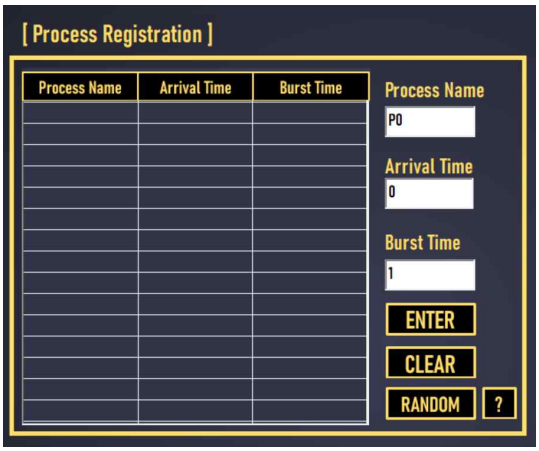
이 알고리즘은 SPN, SRTN, HRRN과 마찬가지로 각 프로세스의 BT를 안다고 가정해야 한다. 이 알고리즘은 BT가 높을수록 가중치가 높은 시스템 상황에 적용될 수 있는 알고리즘이다. 이 알고리즘 방식에 따라 BT가 큰 프로세스부터 처리할 때 BT가 Critical Time (임계 시간)보다 낮다면, 즉 빨리 끝낼 수 있는 프로세스라면 starvation을 예방할 수 있는 효과를 지닌다. 하지만 BT가 임계 시간보다 약간만 높은 프로세스라면 이 프로세스는 우선 순위가 끝없이 밀려 더욱 큰 starvatoin 현상이 발생하게 된다는 단점이 있다. 시스템의 프로세스 상황에 맞춰 임계 시간을 잘 설정하는 것이 이 알고리즘의 포인트이고 일정 주기로 임계 시간을 변동시켜주는 방법 또한 하나의 해결책이 될 수 있다.

4. GUI 구성

i) 입출력 객체

입출력은 크게 5영역으로 나눠진다. 프로세스 등록 박스, 스케줄링 메소드 선택 박스, 프로세서 상태 표시 박스, 간트 차트, 프로세스 상태표가 있다.

1. 프로세스 등록 박스



사용자는 프로세스의 이름과 AT, BT를 직접 입력하거나 RANDOM 버튼을 눌러 15개의 AT, BT가 랜덤인 프로세스를 입력한다. 여기서 입력하는 프로세스 정보는 GuiProcess를 객체로 가지는 vector(vector<GuiProcess> processList)에 입력된다. 이 vector가 곧바로 스케줄링 알고리즘으로 활용되는 것은 아니고 Scheduling Start 버튼을 누르면 이 정보를 토대로 스케줄링에 필요한 프로세스 정보를 추출한다. GuiProcess의 클래스 구성은 다음과 같다.

```
GuiProcess::GuiProcess(CString name, int arrivalTime, int burstTime) {
 random_device rd;
 mt19937 gen(rd());
 this->name = name;
 this->arrivalTime = arrivalTime;
 this->burstTime = burstTime;

 // 흰색 글씨가 안보일 수도 있는 것을 염려해 모두 255, 255, 255 보다 50 전으로 색을 선택해줍니다.
 uniform_int_distribution<> dist(0, 205);
 processColor = RGB(dist(gen), dist(gen), dist(gen));
}
```

guiProcess.cpp 중 일부

2. 스케줄링 메소드 선택 박스

사용자는 이 영역에서 스케줄링 알고리즘을 선택한다. 이때 유의해야 할 점은 RR을 선택할 시 time quantum을 설정해주어야 하고 CBTA를 선택할 시 Critical Time을 설정해주어야 한다. m_algorithm은 사용자가 선택하는 알고리즘의 변수이다. 0 ~ 5까지 각각의 알고리즘을 나타낸다. Scheduling Start 버튼을 누를 시 이 정보가 scheduling_method 변수에 반영되는 코드는 다음과 같다.

```
/* RR 전용 time_quantum, CBTA 전용 critical_time이므로 나머지 알고리즘의 이용 시 1으로 대입해줍니다.*/
    int time_setting = 1;
    if (pDlg->m_algorithm == SCHEDULING_RR) {
        if (pDlg->m_time_quantum < 1) {
            m_eThreadWork = THREAD_STOP;
            AfxMessageBox(_T("Time Quantum은 1초 이상으로 설정해주세요."));
            m_pThread->SuspendThread();
            DWORD dwResult;
            ::GetExitCodeThread(m_pThread->m_hThread, &dwResult);
            delete m_pThread;
            m_pThread = NULL;
            return 0;
        }
        time_setting = pDlg->m_time_quantum;
    }
    if (pDlg->m_algorithm == SCHEDULING_CBTA) {
        if (pDlg->m_time_quantum2 < 1) {
            m_eThreadWork = THREAD_STOP;
            AfxMessageBox(_T("Critical Time은 1초 이상으로 설정해주세요."));
            m_pThread->SuspendThread();
            DWORD dwResult;
            ::GetExitCodeThread(m_pThread->m_hThread, &dwResult);
            delete m_pThread;
            m_pThread = NULL;
            return 0;
        }
        time_setting = pDlg->m_time_quantum2;
    }

int scheduling_method = pDlg->m_algorithm;          // 스케줄링 방법 결정
```

MFCApplication1Dlg.cpp의 UINT SchedulingThread(LPVOID pParam) 함수 중 일부

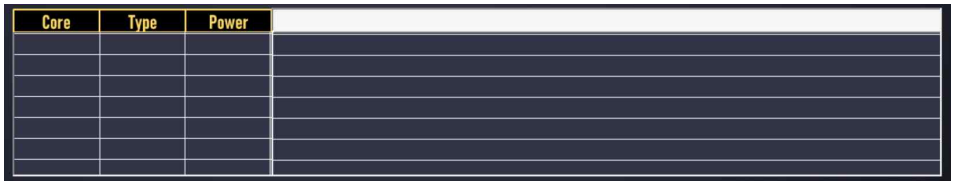
3. 프로세서 상태 표시 박스

사용자가 위의 왼쪽 사진에서 코어를 선택해서 입력하면 코어 리스트(vector<int> coreList) 추가되면서 오른쪽 사진의 코어 리스트 컨트롤(CListCtrl m_coreList)에도 추가된다. Scheduling Start 버튼을 누르면 실시간으로 전력 사용량과 합산 전력량이 출력된다. 스케줄링 stepProcess 과정이 한 번 수행된 이후(한 번의 클럭 주기) 각 코어의 power_usage vector를 함수의 매개변수로 전달해주고 다음의 함수에서 GUI에 정보를 출력해준다.

```
void DispPowerConsumption(std::vector<double>& power_usage, int p) {
    // 다이어로그 객체를 받아와서 이 함수에서 전력 정보를 출력해주는 작업을 합니다.
    CMFCApplication1Dlg* pDlg = (CMFCApplication1Dlg*)AfxGetApp()->GetMainWnd();
    double m_total_power_ = 0.0;    // 전체 전력 사용량 변수
    CString power_usage_str;
    /* 각각의 코어별로 전력 사용량이 GUI에 출력하면서 전체 전력에 합산됩니다. */
    for (int i = 0; i < p; i++) {
        power_usage_str.Format("%0.2f", power_usage[i]);
        power_usage_str += 'J';
        pDlg->m_coreList.SetItemText(i, 2, power_usage_str);
        m_total_power_ += power_usage[i];
    }
    CString total_power_str;
    total_power_str.Format("%.2f", m_total_power_);
    total_power_str += 'J';
    pDlg->m_total_power.SetWindowTextA(total_power_str);
}
```

simple_gui.cpp 중 일부

4. 간트 차트



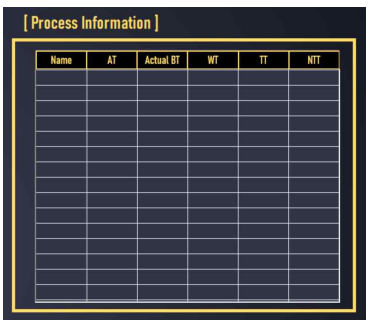
Scheduling Start 버튼을 누르면 실시간으로 실행 중인 프로세스를 리스트 컨트롤(CListCtrl m_running_processList)에 출력한다. 스케줄링 stepProcess 과정이 한 번 수행된 이후(한 번의 클럭 주기) 각 코어에서 실행 중인 프로세스 정보를

담고 있는 running_process vector와 코어 개수(p), index(clock)를 함수의 매개변수로 전달해주고 다음의 함수에서 GUI에 출력한다. 코드는 다음과 같다.

```
void DispRunningProcess(std::vector<int>& running_process, int p, int index) {
    // 다이어로그 객체를 받아와서 이 함수에서 실행중인 프로세스 정보를 출력해주는 작업을 합니다.
    CMFCApplication1Dlg* pDlg = (CMFCApplication1Dlg*)AfxGetApp()->GetMainWnd();

    CString sec;
    sec.Format(_T("%d"), index + 1);
    pDlg->m_running_processList.InsertColumn(index + 1, sec, LVCFMT_RIGHT, 80, index + 1);

    for (int i = 0; i < p; i++) {
        if (running_process[i] != -1) {
            pDlg->m_running_processList.SetItemText(i, index, pDlg->processList[running_process[i]].name);
        }
    }
    return;
}
```

simple_gui.cpp 중 일부

5. 프로세스 상태표



Scheduling 수행 과정 중 1클럭 주기로 종료되는 프로세스가 있으면 해당 프로세스의 정보를 리스트 컨트롤(CListCtrl m_processResult)에 출력한다. 스케줄링 stepProcess 과정이 한 번 수행된 이후(한 번의 클럭 주기) 종료되는 프로세스와 관련된 정보를 매개변수로 받아서 출력해주는 코드는 다음과 같다.

```
void RegistTerminatedProcess(std::vector<int>& newly_terminated_process,
    std::vector<int>& terminated_process,
    std::vector<int>& actual_burst_time,
    std::vector<int>& waiting_time,
    std::vector<int>& turn_around_time,
    std::vector<double>& normalized_turn_around_time,
    int n)
{
    // 다이어로그 객체를 받아와서 이 함수에서 종료되는 프로세스 정보를 출력해주는 작업을 합
```

```
니다.
    CMFCApplication1Dlg* pDlg = (CMFCApplication1Dlg*)AfxGetApp()->GetMainWnd();
    for (int i = 0; i < n; i++) {
        if (newly_terminated_process[i]) {
            terminated_process[i] = 1;
            CString str;
            int nIndex = pDlg->m_processResult.InsertItem(pDlg->m_processResult.GetItem
Count(), pDlg->processList[i].name);
            str.Format(_T("%d"), pDlg->processList[i].arrivalTime);
            pDlg->m_processResult.SetItemText(nIndex, 1, str);
            str.Format(_T("%d"), actual_burst_time[i]);
            pDlg->m_processResult.SetItemText(nIndex, 2, str);
            str.Format(_T("%d"), waiting_time[i]);
            pDlg->m_processResult.SetItemText(nIndex, 3, str);
            str.Format(_T("%d"), turn_around_time[i]);
            pDlg->m_processResult.SetItemText(nIndex, 4, str);
            str.Format(_T("%0.2f"), normalized_turn_around_time[i]);
            pDlg->m_processResult.SetItemText(nIndex, 5, str);
        }
    }
    pDlg->m_processResult.EnsureVisible(pDlg->m_processResult.GetItemCount() - 1, FALS
E);
    return;
}
```

simple_gui.cpp 중 일부

ii) 스레드 루틴

스케줄링이 한 번에 수행되는 것이 아니라 실시간 클락 단위로 수행되기 때문에 스케줄링의 수행과 GUI 출력이 스레드 내부에서 이루어지도록 구성했다. 스레드가 실행 중일 때는 THREAD_RUNNING 상태, 일시 정지 상태에는 THREAD_PAUSE 상태로 두고 이 경우는 스케줄링이 진행 중인 상황이기 때문에 사용자가 다른 작업을 시도할 경우 할 수 없도록 예외처리를 두었다. 스케줄링 작업을 포기하거나 작업이 완료된 경우 해당 스레드는 THREAD_STOP 상태가 된다.

```
#define THREAD_STOP     0
#define THREAD_RUNNING 1
#define THREAD_PAUSE    2

CWinThread* m_pThread;
int m_eThreadWork = THREAD_STOP;  // 스레드는 돌아가고 있지 않는 상태로 초기화 해줍니
다.
```

MFCApplication1Dlg.cpp 중 일부

스케줄링 시작 버튼을 누르면 스레드가 실행되며 스케줄링이 시작된다. 이때 스케줄링에 필요한 설정용 변수들을 초기화시키는 작업이 필요하다. 앞서 말한 GUI를 통해 사용자에게 입력받은 정보들을 이용하여 초기화한다. 코드는 다음과 같다. time_setting과 scheduling_method도 설정용 변수로써 받아주어야 하지만 앞서 소개한 입출력 객체에서 설명한 코드가 있기에 이하 생략한다.

```
/*설정용 변수... 이 변수들은 GUI로부터 전달받아야 합니다.*/
int N = pDlg->processList.size();                    //프로세스 개수
int P = pDlg->coreList.size();                       // 코어 개수

vector<int> at(N);                                    // 도착 시간 배열
for (int i = 0; i < N; i++)
    at[i] = pDlg->processList[i].arrivalTime;
vector<int> bt(N);                                    // 버스트 타임 배열
for (int i = 0; i < N; i++)
    bt[i] = pDlg->processList[i].burstTime;
vector<int> core_types(P);                            // cpu 코어 타입 지정 배열
for (int i = 0; i < P; i++)
    core_types[i] = pDlg->coreList[i];

// time_setting과 scheduling_method은 생략
int settings[3] = { time_setting, N, P };             // 스케줄링 프로세스 설정
```

MFCApplication1Dlg.cpp의 UINT SchedulingThread(LPVOID pParam) 함수 중 일부

설정용 변수를 초기화했다면 다음은 커널의 동작을 위한 동작용 변수와 GUI 출력을 위한 출력용 변수, 커널과 같은 스케줄링 구현에 사용되는 변수를 초기화시켜준다. 코드는 다음과 같다.

```
/*동작용 변수=> 임의 변경시 커널의 동작에 영향을 줄 수 있습니다.*/
vector<int> running_process(P, CPU_NO_PROCESS);     // 현재 i번 코어에서 동작중인 프로세스
running_process[0] = CPU_NO_PROCESS;

vector<int> process_state(N);                        // 프로세스 상태 배열
for (int i = 0; i < N; i++)
    process_state[i] = -1;

vector<int> cpu_restarted(P, 0);                     // 재시작한 코어의 번호
vector<double> weight(N, 0);                         // 스케줄링용 가중치 배열

/*출력용 변수 => 값을 변경하더라도 커널의 동작에 영향을 주지 않습니다.*/
vector<int> terminated_process(N, 0);                    // i번 프로세스는 이미 종료된 바 있음.
vector<int> newly_terminated_process(N, 0);              // i번 프로세스가 이번 동작에서 종료됨.
vector<double> power_usage(P, 0.0);                      // i번 코어에서 사용한 전력의 총량
vector<int> waiting_time(N, 0);                          // i번 프로세스의 WT
vector<int> actual_working_time(N, 0);                   // i번 프로세스의 BT
vector<int> turn_around_time(N, 0);                      // i번 프로세스의 TT
vector<double> normalized_turn_around_time(N, 0);        // i번 프로세스의 NTT
vector<int> remaing_time(N, 0);                          // i번 프로세스의 RT

    /*스케줄링 구현에 사용되는 변수*/
```

```
    bool process_is_done = false;
    Kernel kernel(N, P, scheduling_method, at, bt, core_types, settings);    // 커널 초기화
    int index = 0;
```

MFCApplication1Dlg.cpp의 UINT SchedulingThread(LPVOID pParam) 함수 중 일부

구현과 출력에 필요한 모든 변수를 초기화했다면 실제 스케줄링을 시작한다. 과정은 앞서 언급했던 메인 루틴의 흐름도와 같다. 이 코드 속에 있는 출력함수들도 앞서 입출력 객체에서 언급한 함수이다. break문에 의해 while문을 빠져나온다면 스케줄링이 완료된 상태이므로 스레드 상태를 THREAD_STOP으로 만들어놓고 빠져나온다. 코드는 다음과 같다.

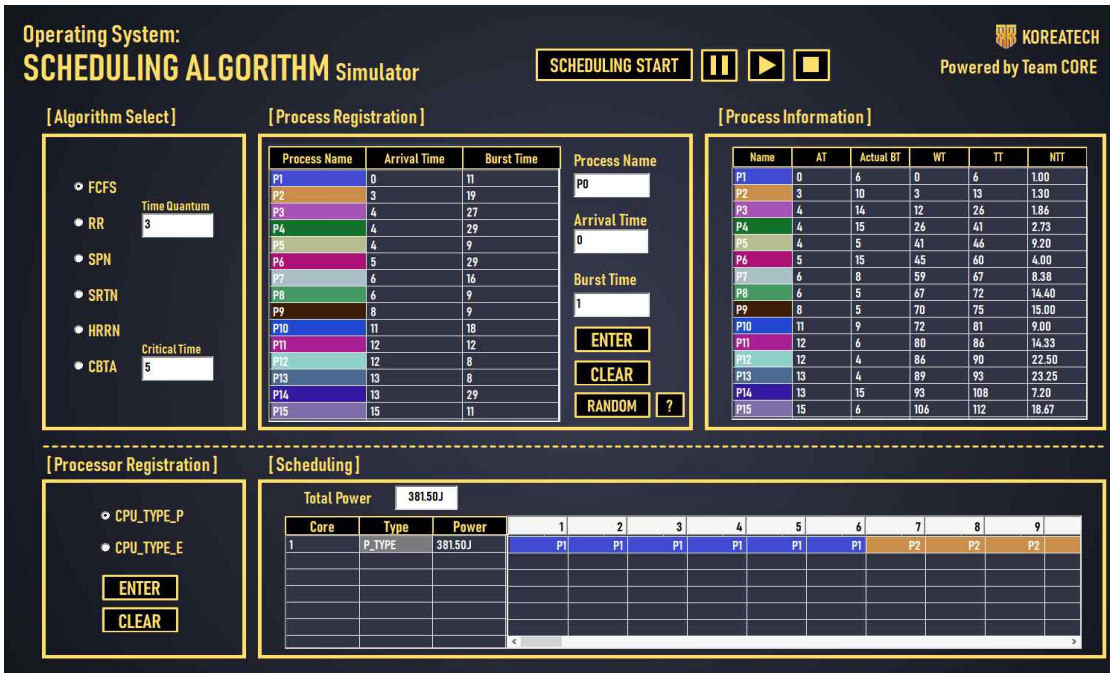
```
while (m_eThreadWork == THREAD_RUNNING) {
        /*스케줄링을 포함한 CPU의 연산을 한 차례 수행합니다. */
        process_is_done = kernel.stepProcess(running_process, cpu_restarted, newly_terminat
ed_process, weight);
        /*화면에 프로세싱 정보를 출력합니다.*/
        /*프로세스별 속성 확인*/
        kernel.get_wating_time(waiting_time);
        kernel.get_actual_working_time(actual_working_time);
        kernel.get_remaining_time(remaing_time);
        kernel.get_turn_around_time(turn_around_time);
        kernel.get_normalized_turn_around_time(normalized_turn_around_time);
        kernel.processor->get_power_usage(power_usage);

        /* 각 코어의 전력사용량과 총 전력사용량을 GUI에 출력합니다. */
        DispPowerConsumption(power_usage, P);

        /* 종료되는 process들을 GUI에 출력합니다. */
        RegistTerminatedProcess(newly_terminated_process, terminated_process,
                                actual_working_time, waiting_time, turn_around_time,
                                normalized_turn_around_time, N);

        if (process_is_done) {
            m_eThreadWork = THREAD_STOP;
            break;
        }

        /* 각 코어별로 현재 처리중인 프로세스를 GUI에 출력합니다. */
        DispRunningProcess(running_process, P, index);

        pDlg->movingScroll();
        index++;

        Sleep(200);

    }
    AfxMessageBox("완료");
```

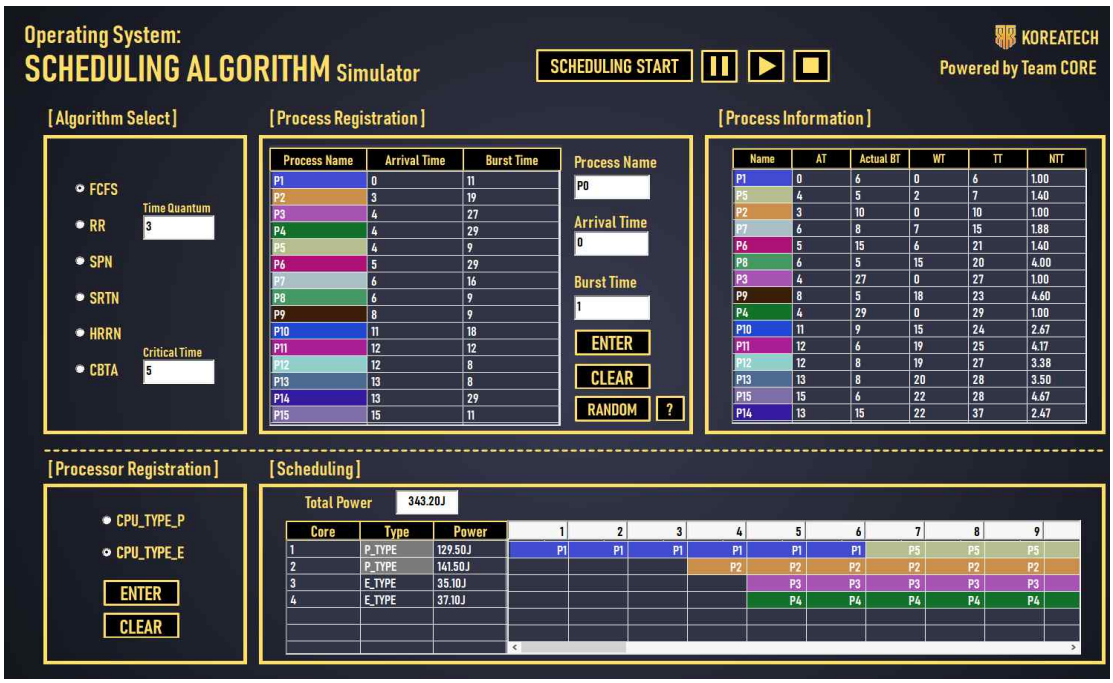
MFCApplication1Dlg.cpp의 UINT SchedulingThread(LPVOID pParam) 함수 중 일부

5. 시연

- FCFS

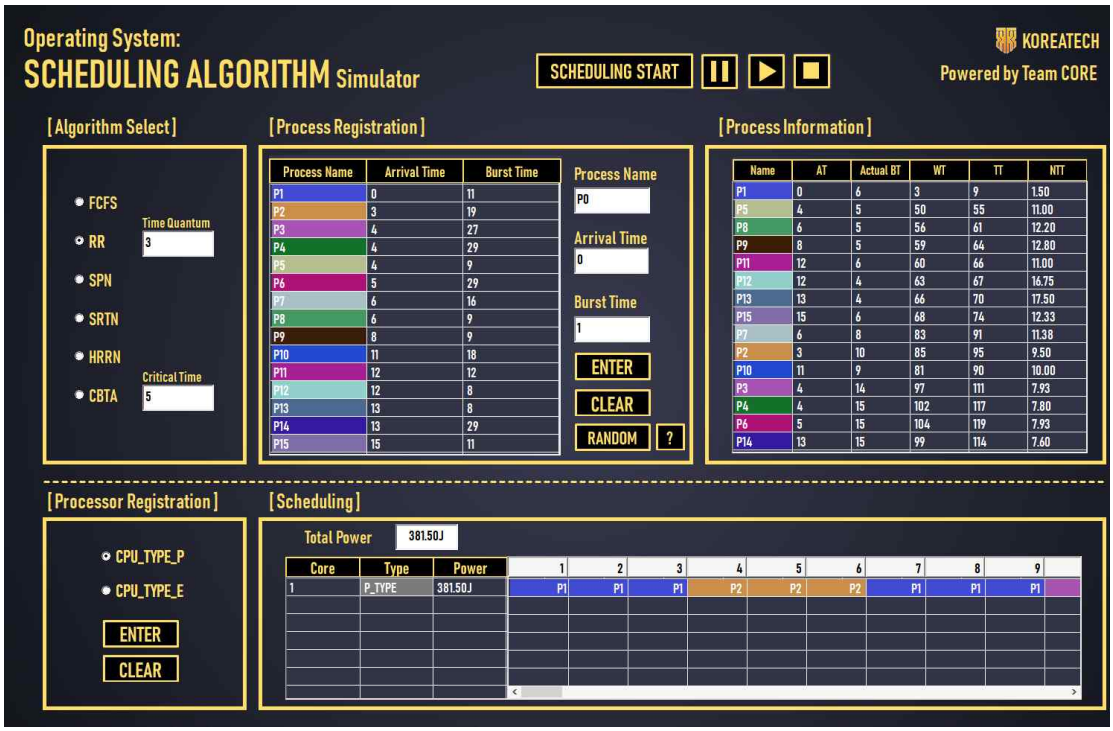
스케줄링 알고리즘 시뮬레이터를 통하여 시연해본 FCFS의 모습이다.
아래의 사진들을 통하여 FCFS 알고리즘이 코어가 1개인 상황과 4개인 상황에서 정상적으로 구현이 되는 것을 볼 수 있다.

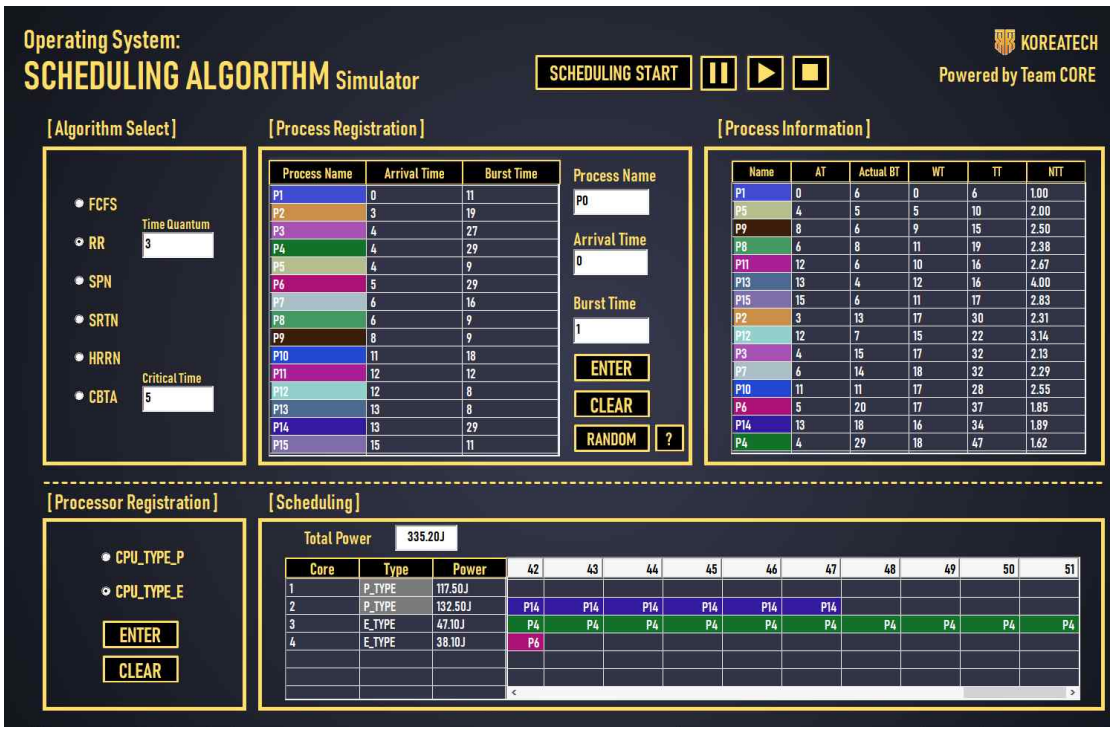




- RR (Time Quantum = 3)

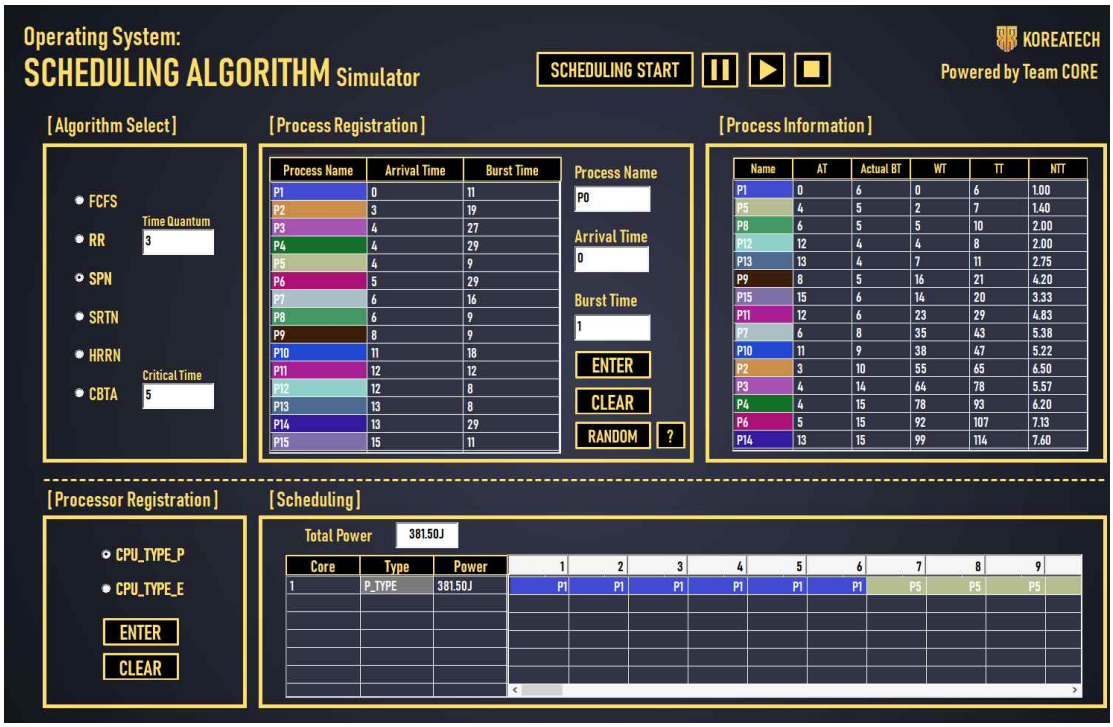
스케줄링 알고리즘 시뮬레이터를 통하여 시연해본 time quantum이 3일 때 RR의 모습이다. 아래의 사진들을 통하여 RR 알고리즘이 코어가 1개인 상황과 4개인 상황에서 정상적으로 구현이 되는 것을 볼 수 있다.

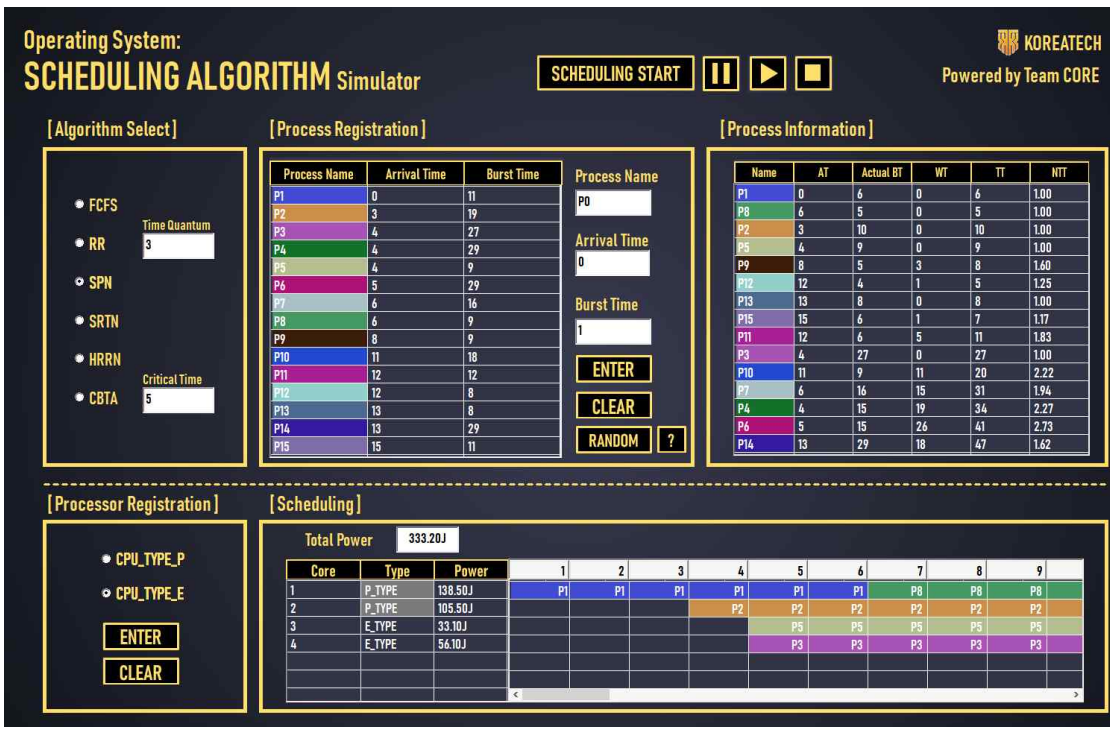




- SPN

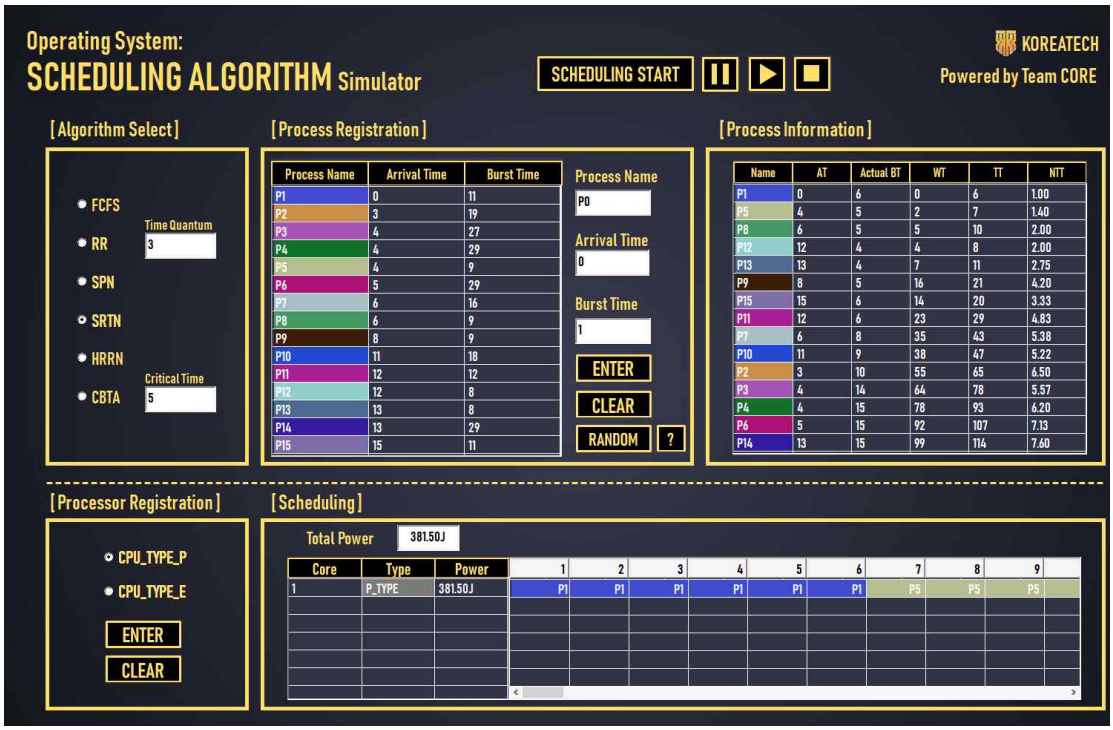
스케줄링 알고리즘 시뮬레이터를 통하여 시연해본 SPN의 모습이다.
아래의 사진들을 통하여 SPN 알고리즘이 코어가 1개인 상황과 4개인 상황에서 정상적으로 구현이 되는 것을 볼 수 있다.

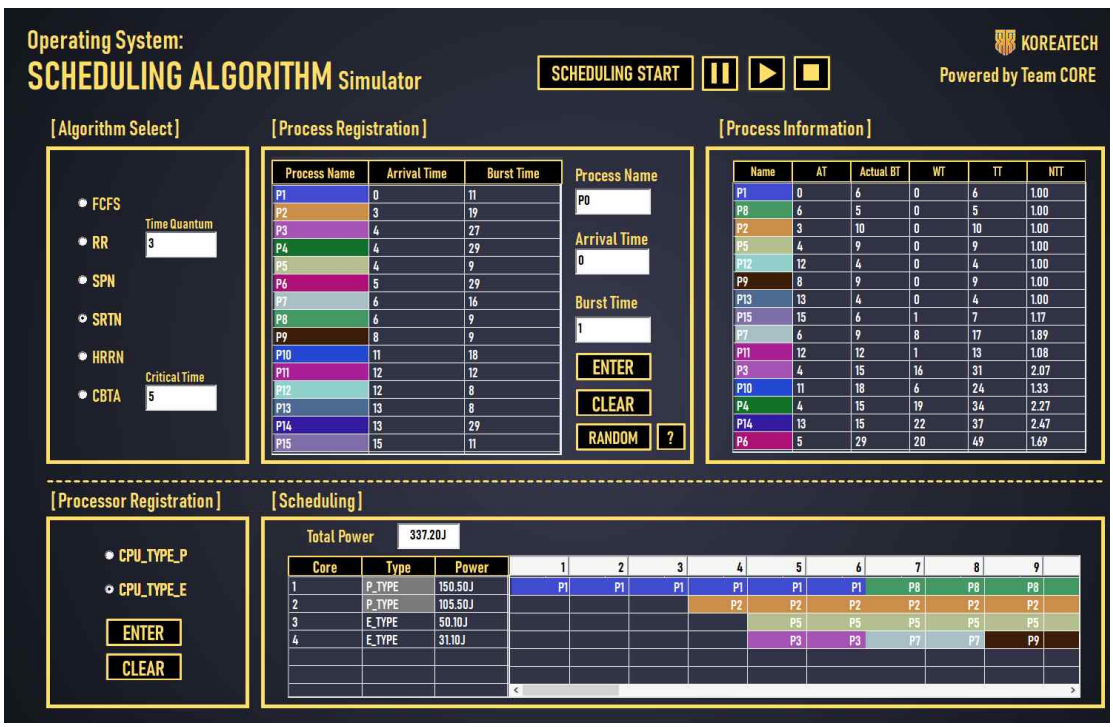




- SRTN

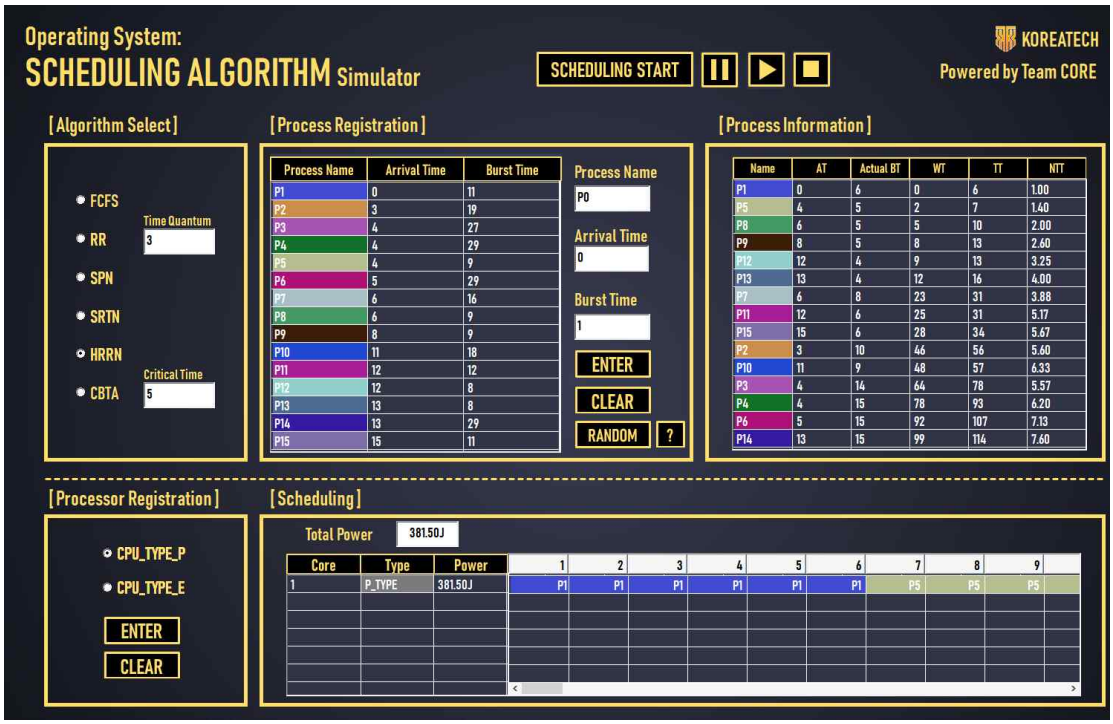
스케줄링 알고리즘 시뮬레이터를 통하여 시연해본 STRN의 모습이다.
아래의 사진들을 통하여 SRTN 알고리즘이 코어가 1개인 상황과 4개인 상황에서 정상적으로 구현이 되는 것을 볼 수 있다.

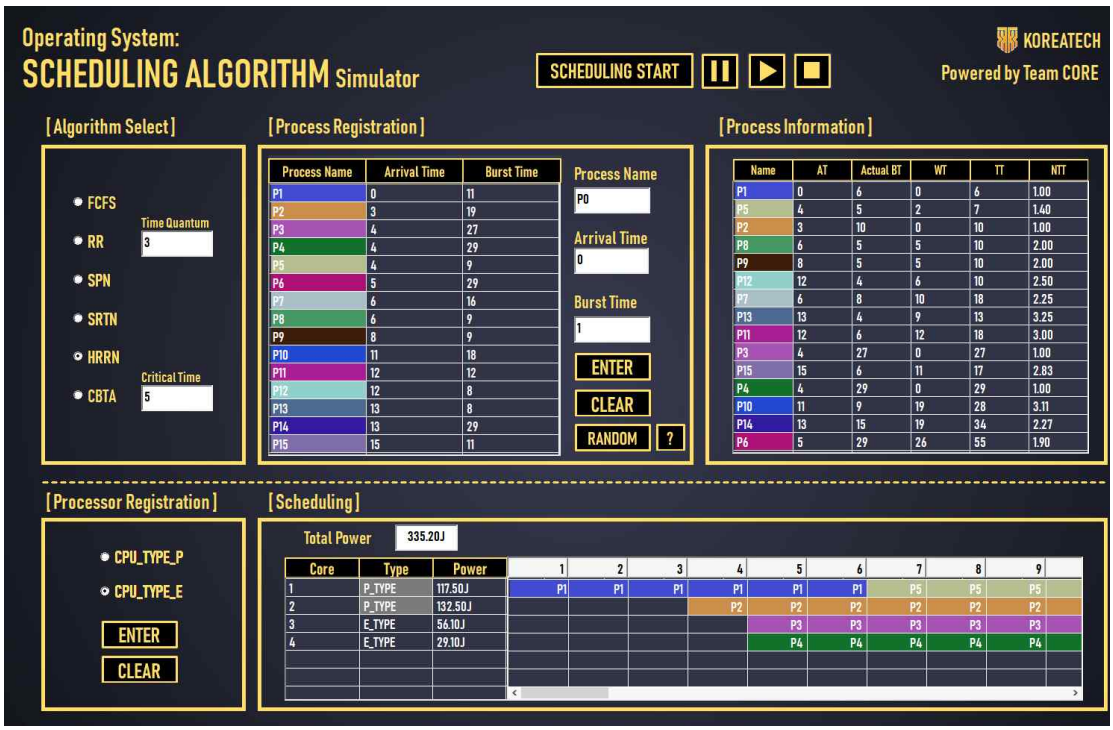




- HRRN

스케줄링 알고리즘 시뮬레이터를 통하여 시연해본 HRRN의 모습이다.
아래의 사진들을 통하여 HRRN 알고리즘이 코어가 1개인 상황과 4개인 상황에서 정상적으로 구현이 되는 것을 볼 수 있다.



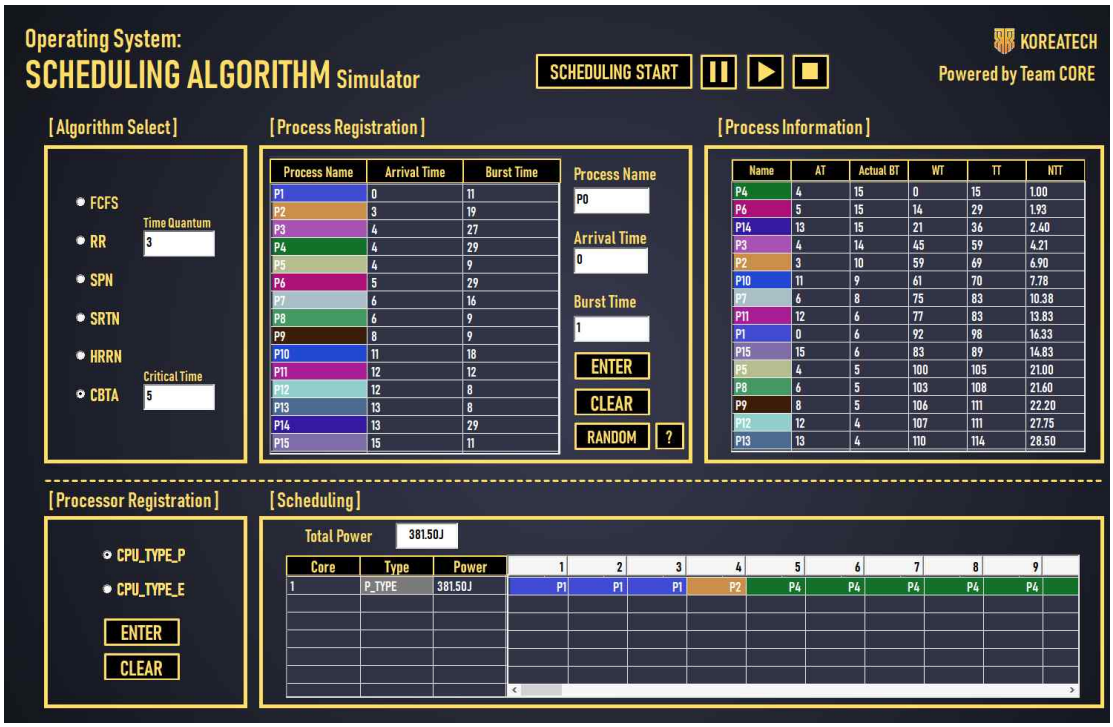


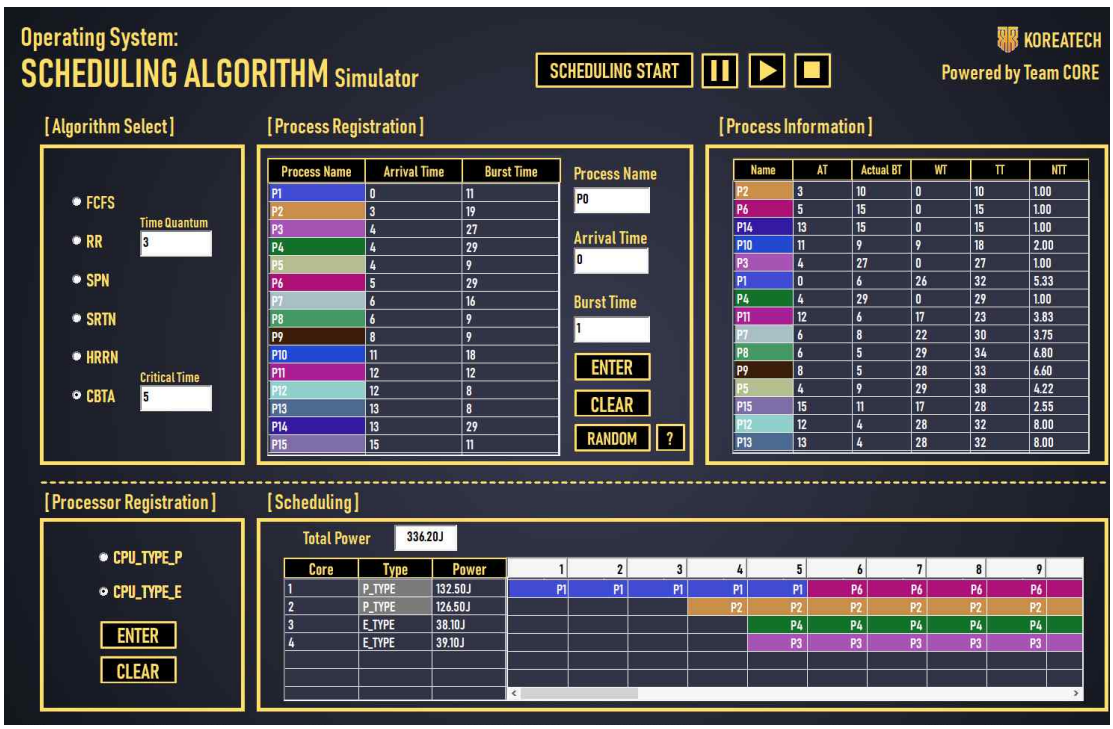
- CBTA

스케줄링 알고리즘 시뮬레이터를 통하여 시연해본 CBTA모습이다.

CBTA 알고리즘에서는 사용자가 지정한 임계 시간(Critical Time)을 기준으로 BT를 분류하여 우선순위를 부여한다. 여기서는 임계 시간을 5로 지정하고 시연하였다.

아래의 사진들을 통하여 CBTA 알고리즘이 코어가 1개인 상황과 4개인 상황에서 정상적으로 구현이 되는 것을 볼 수 있다.





Ⅲ. 결론

1. 프로젝트 자체에 대한 안타까운 점 및 개선 사항

-동적 레이아웃을 해결하지 못했다.

윈도우에서 실행중인 다이어그램의 크기를 받아와서 각 객체의 위치와 크기를 받아온 다이어그램 정보에 비례해서 변수로 지정해주면 해결할 수 있지만, 버튼과 제목표시줄을 gui 개선을 위해 bmp image파일로 만들었고, 이 bmp파일은 크기 조절이 불가능한 점을 미쳐 알지 못했다. (제목표시줄과 버튼 모두 bmp파일을 안넣고 할 수는 있었지만 주변 배경과 어울리는 색을 위해 bmp파일을 쓰는것을 선택했다.) 이를 해결하기 위해 리스트 컨트롤의 제목표시줄을 바꿀 수 있는 방법과 CButton의 색을 변경할 수 있는 방법을 찾아본 결과 각 객체를 부모 클래스로부터 상속받아 다시 그려줘야 한다는 해결책을 알았지만 실제로 해본 결과 원하던 대로 결과반영이 제대로 되지 않았다. MFC에서 보통 버튼이나 UI를 꾸밀경우 bmp파일로 넣고 타 실행환경에서 해상도 변경이 되지 않도록 속성에서 DPI 인식을 없음으로 설정해야 한다는 것을 알아냈고 이 프로그램 또한 해상도 변경을 할 수 없도록 설정했다. 해상도 변경 이슈와 동적 레이아웃 이슈는 추후 개선해야 할 사항이다.

- 프로그램의 거대화

MFC의 구조 상 이벤트 처리기와 같은 여러 함수들을 같은 파일 안에서 사용하면서 유지보수에 어려움을 겪었다. 비슷한 예로 들자면 운영체제 강의시간에 배웠던 단일커널 구조의 문제점 즉, 프로그램의 크기가 커지면 커질수록 양이 너무 많아져 유지보수가 어려워진다는 점을 몸소 느끼며 프로젝트를 진행했다. 변수들을 넘겨주고 다른 소스파일의 함수에서 다이어로그로 접근한 코드가 있기는 했지만, 이는 일반적인 상황은 아니었다.

- 자료구조와 보안성

프로세스 테이블을 처음 구상할 때, 다른 객체를 참고하지 않더라도 프로세스 테이블만으로 각 프로세스에 대한 정보를 충분히 얻을 수 있도록 구상해야 했다. 핸들러 클래스에 익숙하지 않은 탓이었다.

각 클래스 간에 정보 교환 방법으로는 배열(vector)을 사용했는데, 이는 매우 부적절한 선택이었다. 본래 각 배열들이 대부분 동일한 크기를 갖기 때문에, 배열 포인터와 그 크기로 이루어진 구조체를 구성했다면 훨씬 효과적으로, 또한 간단하게 정보를 교환할 수 있었을 것이다.

더불어, 구상 난이도를 핑계로 정보 은닉과 캡슐화를 배제하는 바람에 상당히 불안정한 코드가 만들어졌다는 점이 아쉬운 점으로 남는다.

2. 개인적 감상

- 박형진

알고리즘 프레임워크 자체를 만들어놓고 스케줄링 알고리즘에 접근하는 과정을 경험하면서 프레임워크의 중요성에 대해 느낄 수 있었다. 또 스케줄링 알고리즘에 대해 고민하고 생각해보는 시간을 가지면서 시험공부와는 다른 느낌으로 공부하는 기분을 느낄 수 있었고 실제로 개념을 활용해보면서 보다 확실하게 알고리즘을 숙지할 수 있게 되었다. 운영체제 시간에 배웠던 스레드 등도 MFC를 통해 사용해보면서 강의시간에 배웠던 여러 내용을 다시 한번 인지해보게 되는 계기가 되었다.

깃허브의 중요성 또한 깨닫게 되었다. 팀원들이 깃을 사용하지 않는 팀원들이었고 이에 코드들을 일일이 주고받으며 팀 프로젝트 진행에 다소 불편함을 느꼈다. 팀원들이 깃허브를 사용하지 않는다면 혼자라도 GIT에 올려두고 하는 편이 백업이나 진행 상황 체크 등의 면에서 좋다는 것을 프로젝트를 진행하면서 깨달았다.

- 박순용

고찰을 팀워크 차원에서의 고찰과 진행 방법론 차원에서의 고찰로 구분할 수는 있겠지만, 그 감상을 구분하여 작성하기는 어려울 듯하다.

전체적인 프로젝트 진행 순서를 결정한 사람으로서, 프로젝트의 진행 자체에 관해 고민한 부분이 많았다. 본디 타 학과 전공생으로서 팀원들에게 신뢰감을 심어주는 동시에 프로젝트에 적극적으로 참여하기 위해서는 프로젝트에 대해 어디까지 담당할 수 있는지 보여줄 필요가 있었고, 이를 위해 전공 공부 중 독자적으로 발전시킨 프로그래밍 방법론을 도입하여 전체적인 프로세스 진행 방식을 제시하는 방안을 선택했다. 물론 여기에는 단순히 작업 효율성을 증진 시키겠다는 의미만이 담겨있던 것은 아니었다. 여기에는 본인이 이전에 사용하던 방법론을 팀 활동에 접목시키기 위해 고안한 여러 방법들을 실제로 활용해보려는 의도가 담겨있었다. 하지만, 실제 팀원들은 본인의 예상과 달리, 특정한 방법론을 도입하지 않고, 단순히 소스 코드를 전부 읽어서 전체를 이해하려는 경향을 보였다. 아무래도 기존 방식에 대한 익숙함이 가장 큰 원인이었을 것이다. 한편, 새로운 방법론에 의구심을 표하는 이도 있었다. 때문에 새로운 방법론의 실용성에 대한 근거를 제시하기 위하여 개인적으로 이전 프로젝트에서 만들었던 로봇의 프로그램을 직접 보여주기도 했다.

하지만 당초의 불안이 무색하게, 프로젝트는 예상 이상으로 순조롭게 진행되었다. 팀원들 개개인이 새로 제시한 방법론에 빠르게 익숙해지지는 못했다는 점은 예상과 다르지 않았으나, 그럼에도 프로젝트에 매우 의욕적으로 임해주었다. 새로운 방법론을 긍정적으로 받아들였는지, 프로젝트 후반에 가까워질수록 전체적인 구조를 이해하려는 시도를 보여주기도 했다.

한편, 프로젝트의 성패와는 무관하게, 방법론 자체에 대해서도 생각해볼 주제가 생겼다. 논리적인 구성 자체에 한해서는 레이어드 플로우차트라는 방법론은 이미 여러 차례에 걸쳐 본인에게 그 쓸모를 입증한 바 있었고, 이번에도 마찬가지로 복잡한 생각을 구조화하는 데에 큰 도움이 되었다. 그러나 이번 프로젝트를 진행하던 중, 코드가 전체적으로 가독성이 부족하다는 평을 들었다. 본래 이 방법론을 구상할 때, 가독성 문제는

플로우차트를 통해 해소될 것이라고 생각하여 코드 자체의 가독성은 깊게 고민하지 않았다. 어찌 보면 새로운 방법론을 도입하는 입장으로서, 그 방법론의 의미와 의도에 대해 충분히 설명하지 않은 것이 문제인지도 모르겠다. 물론 코드 자체의 가독성이 떨어진다는 점도 충분히 숙고할 문제 중 하나라고는 인지하는 바이다. 한편, 가독성이 떨어지는 가장 큰 이유로서 함수, 객체 사이의 호출 관계에 대한 인식을 형성하기가 어렵다는 팀원의 의견을 들을 수 있었다. 이를 근거로, 이후에 작성할 프로그램에 대해서는 함수 종속 관계도 이상으로, 객체와 멤버 변수 및 함수의 분포 양상에 대해 일괄적으로 이해할 수 있도록 하는 그래프를 그리는 방법에 대해 고민해볼 필요가 있겠다.

- 오수환

수업 내용이였던 스케줄링 알고리즘과 자원 관리 등 다양한 개념을 사용하여 코드를 구현해보면서 개념을 한번 더 확인하고 이해할 수 있었다. 또한, 프로젝트를 수행하면서 협업을 통하여 문제를 해결할 수 있었다. 팀원들과 함께 고민하고, 문제를 해결해 나가는 과정에서 다양한 아이디어와 지식을 공유할 수 있었으며, 이를 통해 더 나은 결과물을 만들어 낼 수 있었다. 스케줄러를 구현하는 것은 어려웠지만, 매우 유익하고 의미있는 경험이었다. 이를 통해 프로그래밍 기술뿐만 아니라, 문제 해결 능력과 협업 능력도 크게 향상시킬 수 있었으며, 이후에도 다양한 프로젝트에서 이러한 경험을 살려나갈 수 있을 것으로 느껴진다.

- 박동진

프로그램을 개발할 때 크게보면 실질적인 동작을 수행하는 알고리즘과 그 알고리즘을 어떻게 하면 사용자에게 효율적으로 출력할 수 있을지를 고민하는 UI 프로그래밍으로 구분할 수 있다. 알고리즘이 훌륭해도 UI를 개발하지 못하면 무용지물이 될 수 있겠지만, 그래도 결국 프로그램은 어떠한 알고리즘을 가지는지가 핵심이라는 생각이 들었다. 결국 알고리즘을 구현하지 못하거나 효율적이지 못한 코드라면 그 프로그램은 알맹이가 없는 껍데기 뿐인 프로그램이 될 것이라는 생각이 들었다.

평소 혼자서 백준문제풀이를 하며 공부를 진행할 때도 도식화를 하며 문제의 원리를 이해하려고 하기보다는 공식과 코드를 외우며 빠르게 문제를 해결해나가기 바빴던 나를 반성하게 되었다. 정작 이번 팀프로젝트를 통해 실질적인 알고리즘에 대한 코드를 설계하려니까 벽에 막힌 느낌이었고 초등학생 시절부터 컴퓨터 공학 3학년까지 진행하면서 해왔던 공부방식이 틀렸다 라고 느낀 계기가 되었다. 팀원인 순용님이 알고리즘을 먼저 도식화하고 프레임워크를 설계하시는 과정에서 함께 설계해보며 논리적인 사고를 하는 방법 및 짜임새있게 자신의 생각을 정리하며 순서도를 작성해나가는 법 등을 배울 수 있었고, 역량이 부족해서 알고리즘 설계에 도움이 많이 되어드리지 못한 부분에 있었기에 더욱 분발하고자 주어진 역할 내에서 최선을 다하기 위해 노력했던 것 같다. 앞으로의 공부방식을 배울 수 있었기에 의미 있는 팀 프로젝트였다고 생각한다.

Ⅳ. 부록

software.h

```
#pragma once
#ifndef __SOFTWARE_HPP__
#define __SOFTWARE_HPP__

#define SCHEDULING_FCFS    0
#define SCHEDULING_RR      1
#define SCHEDULING_SPN     2
#define SCHEDULING_SRTN    3
#define SCHEDULING_HRRN    4
#define SCHEDULING_CBTA    5

#define PROCESS_STATE_NOT_LOADED     -1
#define PROCESS_STATE_CREATED         0
#define PROCESS_STATE_READY           1
#define PROCESS_STATE_RUNNING         2
#define PROCESS_STATE_TERMINATED      3

#include "hardware.h"

class PCB {
public:
    int AT;    int BT;
    // Arrival Time / Bust Time
    int WT;    int TT;       int RT;    double NTT;
    // Wating Time / Turn-around Time / Remaining Time(잔여 작업 시간) / Nomalized Turn-ar
ound Time
    int actual_working_time;
    // 실제로 처리된 시간의 총합을 나타냅니다. (= actual burst time)
    int state;
    // 프로세스 상태를 나타냅니다. PROCESS_STATE_CREATED 따위의 상수를 사용하여 나타냅니다.

public:
    // PCB 클래스 생성자입니다.
    PCB();
    // PCB에 프로세스의 정보를 등록합니다.
    void registProcess(int at, int bt);
};

class Que {
public:
    int capacity;       // 배열 크기
    int num;             // 배열에 저장된 자료의 크기
    int* body;           // 자료를 저장할 배열
    int head;            // 다음에 디큐할 정수의 인덱스
    int tail;           // 다음에 인큐할 정수를 저장할 인덱스
```

```
public:
    Que(int cap);
    ~Que(void);

    // 큐가 가득 찬 경우 true를 반환합니다.
    bool is_full();
    // 큐가 비어있으면 true를 반환합니다.
    bool is_empty();
    // 주어진 weights(= 각 원소의 가중치를 배열로 정리한 것)에 따라 큐의 내용을 정렬합니다.
    void sort(std::vector<double>& weights);
    void sort(double* weights);
    // 주어진 정수를 인큐합니다.
    void enque(int p);
    // 큐의 tail에서 원소를 디큐합니다.
    int deque();
    // 큐 안에 지정한 값이 있는지 판단합니다.
    bool has(int p);
    // 큐의 내용물을 비움
    void clear();
};

class Scheduler {
public:
    // preemptive는 선점 가능 여부를 나타내는 속성입니다. 선점 가능한 스케줄링 메소드의 경우 True로
지정합니다.
    bool preemptive;

public:
    // 스케줄러 클래스의 생성자입니다. 필요한 경우 오버라이딩 하여 사용하십시오..
    Scheduler();
    // 속성 설정자입니다. 공교롭게도, 아래에서 속성 초기화를 하게 됐습니다.
    virtual void setAttr(int* settings) {}
    // 스케줄링에서 사용할 가중치의 배열인 weights를 결정하는 함수입니다.
    virtual void setWeight(PCB* process_table, std::vector<double>& weights, int n, int clk) {}
    // Kernel::afterProcess에서 사용하는 함수입니다. 적절히 오버라이딩해서 사용하시면 됩니다.
    void afterProcess();
};

class FCFS : public Scheduler {
    // 멤버 변수는 여기에서 정의합니다.
public:
    bool done_once;



    // 멤버 함수는 여기에서 정의합니다.
public:
    // FCFS 클래스의 생성자입니다. preemptive 설정, time quantum 등,
    // 필요한 속성을 만들거나 초기화합니다.
    FCFS() {
        this->preemptive = false;
```

```
        this->done_once = false;
    };

    // FCFS의 가중치 설정 함수입니다.
    // process_table: PCB 로 구성된 배열입니다. 즉, process_table[i]는 i번째 프로세스의 PCB를 나타냅니다. process_table[i].AT 등의 속성을 자유롭게 사용할 수 있습니다.
    //                  다만, process_table[i]의 값은 변경하지 않도록 주의합니다.
    // weight: weight[i]는 i번째 프로세스의 가중치를 나타냅니다. 참조에 의해 호출되며, 본 함수의 목적은 이 weight의 원소를 적절하게 변경해주는 것입니다.
    // n: PCB의 개수(= 프로세스의 개수)입니다.
    void setWeight(PCB* process_table, std::vector<double>& weight, int n, int clk) {
        if (done_once == true)
            return;

        for (int i = 0; i < n; i++) {
            weight[i] = -double(process_table[i].AT) - (double)(i + 1) / (n + 1);
            // i/n을 빼주는 이유는 동일한 AT를 갖는 두 프로세스의 순서가 버블 정렬에 의하여 매번 뒤바뀌는 상황을 막기 위함입니다.
        }
        done_once = true;
        return;
    }
};

class RR : public Scheduler {
public:
    int N;                      // 프로세스 개수
    int P;                      // 프로세서 개수
    int* tc;                    // 타임 카운터. 프로세스 연속 실행 횟수 확인
    int tq;                     // 타임 퀀텀
    Que* SRQ;                   // sub_readyqueue

public:
    RR(int delta, int n, int p) : N(n), P(p) {
        this->preemptive = true;        // 선점 가능성
        this->tc = new int[N];          // 각 프로세스별 처리 횟수 확인용 배열
        this->tq = delta;               // 타임 퀀텀
        for (int i = 0; i < N; i++)
            tc[i] = 0;

        this->SRQ = new Que(N);
        return;
    }

    ~RR() {
        delete[] tc;
        delete SRQ;
        return;
    }
```

```
    void setWeight(PCB* process_table, std::vector<double>& weight, int n, int clk) {
        int i = 0;
        int rp = 0;

        for (i = 0; i < N; i++) {

            // 새로 arrival한 프로세스를 SRQ에 인큐
            if (clk == process_table[i].AT)
                SRQ->enque(i);

            // 프로세서에 할당된 프로세스의 tc ++
            if (process_table[i].state == PROCESS_STATE_RUNNING)
                tc[i]++;

            // rp=프로세스 할당을 재고할 필요가 없는 프로세서의 개수.
            // 이전 클럭에서 종료된 프로세스도 고려해줘야 한다.
            if ((tc[i] < tq) && (process_table[i].RT >= 0) && (process_table[i].state == PROCESS_ST
ATE_RUNNING))
                rp++;
        }
        rp = P - rp;

        for (i = 0; i < N; i++) {
            if (process_table[i].state == PROCESS_STATE_TERMINATED) {
                tc[i] = 0;
                weight[i] = 0;
            }
            std::cout << tc[i] << std::endl;
            if (tc[i] >= tq) {
                tc[i] = 0;
                weight[i] = 0;
                SRQ->enque(i);
            }
        }

        for (i = 0; i < rp; i++) {
            if (SRQ->is_empty())
                return;

            weight[SRQ->deque()] = 1;
        }
        return;
    }
};

class SPN :public Scheduler {
public:
    SPN() {
        this->preemptive = false;
    }
```

```
    void setWeight(PCB* process_table, std::vector<double>& weight, int n, int clk) {
        for (int i = 0; i < n; i++) {
            weight[i] = -process_table[i].RT;
        }
    }
};

class SRTN : public Scheduler {
public:
    SRTN() {
        this->preemptive = true;
    }

    void setWeight(PCB* process_table, std::vector<double>& weight, int n, int clk) {
        for (int i = 0; i < n; i++) {
            weight[i] = -process_table[i].RT;
        }
    }
};

class HRRN : public Scheduler {
public:
    HRRN() {
        this->preemptive = false;
    }

    void setWeight(PCB* process_table, std::vector<double>& weight, int n, int clk) {
        for (int i = 0; i < n; i++) {
            weight[i] = (double)(clk - process_table[i].AT + process_table[i].BT) / (double)process_t
able[i].BT;
        }
    }
};


class CBTA : public Scheduler {
public:
    int N;              // 프로세스 개수
    int P;              // 프로세서 개수
    int ct;             // 임계 시간
    int done_once;

public:
    CBTA(int critical_time, int n, int p) {
        this->preemptive = true;
        this->N = n;
        this->P = p;
        this->ct = critical_time;
        this->done_once = false;
```

```
    }

    ~CBTA() {
    }

    void setWeight(PCB* process_table, std::vector<double>& weight, int n, int clk) {
        if (done_once)
            return;

        double largest_BT = 0;
        for (int i = 0; i < N; i++) {
            if (process_table[i].BT > largest_BT)
                largest_BT = process_table[i].BT;
        }

        for (int i = 0; i < N; i++) {
            if (process_table[i].BT <= ct)
                weight[i] = 1 + (double)ct - process_table[i].BT - double(i + 1) / double(n + 1);
            else
                weight[i] = process_table[i].BT - largest_BT - double(i + 1) / double(n + 1);

        }
        done_once = true;
        return;
    }
};

class Kernel {
public:
    int N;
    int P;
    int smethod;
    int clk;
    Que* RQ;                    //레디큐
    Que* RPQ;                   // 유휴코어 큐: Resting Processor Que
    PCB* process_table;
    Processor* processor;
    Scheduler* scheduler;

public:
    // 커널 클래스의 생성자입니다.
    // n: 프로세스의 개수 / p: cpu 코어의 개수 / sm: 스케줄링 메소드
    // at[n]: 프로세스별 어라이벌 타임
    // bt[n]: 프로세스별 cpu 버스트 타임
    // core_types: 코어 타입
    // settings: 스케줄러별 옵션
    Kernel(int n, int p, int sm, std::vector<int>& at, std::vector<int>& bt, std::vector<int>& core_t
ypes, int* settings);

    ~Kernel();
```

```
    // 프로세스 테이블에 등록된 PCB에서 현재 클락과 프로세스별 AT를 비교합니다.
    // clk >=AT 이면서 CREATED 상태인 프로세스를 Ready 상태로 전환합니다.
    void arrivalCheck();

    // 프로세스 테이블에 등록된 모든 프로세스가 terminated 상태이면 True를 반환합니다.
    bool allTerminated();

    // 스케줄링 펑터를 통해 각 프로세스별 가중치를 정하고,
    // 가중치에 따라 레디큐를 정렬한 후 프로세스를 CPU 코어에 할당합니다.
    void processSchedule(std::vector<int>& running_process, std::vector<int>& cpu_restarted, std::
vector<int>& newly_terminated_process, std::vector<double>& weight);

    // 각 CPU가 할당된 프로세스를 실행합니다.
    void proceed(std::vector<int>& running_processes, std::vector<int>& cpu_restarted);

    // 스케줄링 펑터 별로 필요한 뒷처리를 수행합니다.
    void afterProcess() { return; }

    // 아직 종료되지 않은 프로세스가 존재하는 경우 다음 펄스를 추가하여 프로세스를 1번 진행합니다.
    // 종료되지 않은 프로세스가 존재하는 경우 true를, 그렇지 않은 경우 false를 반환합니다.
    bool stepProcess(std::vector<int>& running_process, std::vector<int>& cpu_restarted, std::vect
or<int>& newly_terminated_process, std::vector<double>& weight);

    // 각 프로세스별 상태를 확인합니다.
    void get_process_state(std::vector<int>& process_state);
    // RT를 확인합니다.
    void get_remaining_time(std::vector<int>& remaining_time);
    // BT를 확인합니다.
    void get_actual_working_time(std::vector<int>& actual_working_time);
    // WT를 확인합니다.
    void get_wating_time(std::vector<int>& waiting_time);
    // TT를 확인합니다.
    void get_turn_around_time(std::vector<int>& turn_around_time);
    // NTT를 확인합니다.
    void get_normalized_turn_around_time(std::vector<double>& normalized_turn_around_time);
};

#endif
```

software.cpp

```
#include <iostream>
#include "software.h"
#include <vector>
//#include "Windows.h"

#define PCB_UNCERTAIN   -1

#define QUE_NO_ENTITY   -1
```

```
PCB::PCB() :AT(PCB_UNCERTAIN), BT(PCB_UNCERTAIN), RT(PCB_UNCERTAIN),
WT(PCB_UNCERTAIN), TT(PCB_UNCERTAIN), NTT(PCB_UNCERTAIN),
actual_working_time(PROCESS_STATE_NOT_LOADED), state(PROCESS_STATE_NOT_LOADED)
{ }

void PCB::registProcess(int at, int bt) {
    AT = at;
    BT = bt;
    RT = bt;
    actual_working_time = 0;
    state = PROCESS_STATE_CREATED;
    return;
}


Que::Que(int cap) : capacity(cap), num(0), head(0), tail(0) {
    Que::body = new int[cap];
}

Que::~Que(void) { delete[] Que::body; }

bool Que::is_full() { return num == capacity; }

bool Que::is_empty() { return num == 0; }

void Que::enque(int p) {
    if (is_full())
        return;
    body[tail] = p;
    num++;
    tail = (tail + 1) % capacity;
    return;
}

int Que::deque() {
    if (is_empty())
        return QUE_NO_ENTITY;
    int r = body[head];
    head = (head + 1) % capacity;
    num--;
    return r;
}

void Que::sort(std::vector<double>& weights) {
    int i, j;
    int temp;
    for (i = 0; i < num; i++) {
        for (j = 0; j < num - i - 1; j++) {
            if (weights[body[(head + j) % capacity]] < weights[body[(head + j + 1) % capacity]]) {
                temp = body[(head + j) % capacity];
```

```
                body[(head + j) % capacity] = body[(head + j + 1) % capacity];
                body[(head + j + 1) % capacity] = temp;
            }
        }
    }
    return;
}

void Que::sort(double* weights) {
    int i, j;
    int temp;
    for (i = 0; i < num; i++) {
        for (j = 0; j < num - i - 1; j++) {
            if (weights[body[(head + j) % capacity]] < weights[body[(head + j + 1) % capacity]]) {
                temp = body[(head + j) % capacity];
                body[(head + j) % capacity] = body[(head + j + 1) % capacity];
                body[(head + j + 1) % capacity] = temp;
            }
        }
    }
    return;
}

bool Que::has(int p) {
    for (int i = 0; i < num; i++) {
        if (body[(i + head) % capacity] == p)
            return true;
    }
    return false;
}

void Que::clear() {
    while (!is_empty())
        deque();
    return;
}

Scheduler::Scheduler() : preemptive(false) {
    return;
}

void Scheduler::afterProcess() {
    return;
}

Kernel::Kernel(int n, int p, int sm, std::vector<int>& at, std::vector<int>& bt, std::vector<int>& cor
e_types, int* settings) : N(n), P(p), smethod(sm), clk(-1) {
    // 프로세스 테이블 생성 및 프로세스 등록
    process_table = new PCB[N];
    for (int i = 0; i < N; i++)
```

```
            process_table[i].registProcess(at[i], bt[i]);

        // 레디큐 생성
        RQ = new Que(N);
        // 유휴코어큐 생성
        RPQ = new Que(P);
        // 프로세서 생성
        processor = new Processor(P);
        processor->setType(core_types);
        // 스케줄러 생성
        switch (smethod) {
        case SCHEDULING_FCFS:
            scheduler = new FCFS();
            break;
        case SCHEDULING_RR:
            scheduler = new RR(settings[0], settings[1], settings[2]);    // tq, n, p
            scheduler->setAttr(settings);
            break;
        case SCHEDULING_SPN:
            scheduler = new SPN();
            break;
        case SCHEDULING_SRTN:
            scheduler = new SRTN();
            break;
        case SCHEDULING_HRRN:
            scheduler = new HRRN();
            break;
        case SCHEDULING_CBTA:
            scheduler = new CBTA(settings[0], settings[1], settings[2]);
            break;
        default:
            scheduler = new SPN();
        }
        return;
}

Kernel::~Kernel() {
        delete[] process_table;
        delete RQ;
        delete RPQ;
        delete processor;
        delete scheduler;
        return;
}

void Kernel::arrivalCheck() {
        for (int i = 0; i < N; i++) {
            if ((process_table[i].state == PROCESS_STATE_CREATED) && (process_table[i].AT <= clk)) {
                process_table[i].state = PROCESS_STATE_READY;
                RQ->enque(i);
```

```
        }
    }
    return;
}

bool Kernel::allTerminated() {
    for (int i = 0; i < N; i++) {
        std::cout << i << ": ";
        if (process_table[i].state < PROCESS_STATE_TERMINATED) {
            std::cout << "not terminated" << std::endl;
            return false;
        }
        std::cout << "Terminated" << std::endl;
    }
    return true;
}

void Kernel::processSchedule(std::vector<int>& running_process, std::vector<int>& cpu_restarted,
std::vector<int>& newly_terminated_process, std::vector<double>& weight) {
    // running_process는 이전 단계에서의 각 cpu의 동작 상태를 나타내는 배열
    // cpu_restart는 이번 단계에서 cpu의 재시동 여부를 나타낼 배열
    int p, c;

    RPQ->clear();                                   // 유휴코어 큐 초기화
    for (int i = 0; i < N; i++)                    // 종료프로세스 배열 초기화
        newly_terminated_process[i] = 0;

    // 종료된 프로세서 유무 확인하여 cpu, PCB 정리
    for (int i = 0; i < P; i++) {    // 프로세서의 0~(P-1)번 코어에 대하여
        cpu_restarted[i] = 0;

        p = processor->core[i].allocated_process;              // p: i번째 코어에 할당되어있던 프로세
스 번호
        if ((process_table[p].RT <= 0) && (p != CPU_NO_PROCESS)) {    //p번 프로세스가 끝난 상황
이면
            process_table[p].state = PROCESS_STATE_TERMINATED;
            newly_terminated_process[p] = 1;
            processor->core[i].allocated_process = CPU_NO_PROCESS;
        }
    }

    if (scheduler->preemptive == true) {
        // i.선점 방식의 스케줄링 알고리즘

        // 레디큐에 running 상태의 프로세스를 모두 인큐
        for (int i = 0; i < P; i++) {
            if (processor->core[i].allocated_process != CPU_NO_PROCESS) {
                RQ->enque(processor->core[i].allocated_process);
            }
        }
```

```
// 프로세스 스케줄링: 프로세서별 가중치를 계산하고 가중치에 따라 레디큐를 정렬함
scheduler->setWeight(process_table, weight, N, clk);
RQ->sort(weight);

// allocatable_processor, high_process 초기화
int* allocatable_processor = new int[P];
int* high_process = new int[P];
int hp = 0;                              // high_process 원소 개수

for (int i = 0; i < P; i++) {
    allocatable_processor[i] = 0;
    high_process[i] = -1;
}


// 레디큐에서 최대 P개까지 디큐하여 allocatable_processor, high_process를 채움
bool allocated;
for (int i = 0; i < P; i++) {
    allocated = false;
    if (!RQ->is_empty()) {
        p = RQ->deque();

        for (int j = 0; j < P; j++) {
            // 해당 프로세스가 이전 클럭에 할당되어 있었다면,
            // j번째 프로세서는 재할당을 수행하지 않음을 표시
            if (processor->core[j].allocated_process == p) {
                allocatable_processor[j] = -1;
                allocated = true;
                break;
            }
        }

        // 해당 프로세스가 이전 클럭에 할당되어 있지 않았다면
        // i번째 프로세스는 새로 할당해야하는 프로세스 목록에 추가
        if (!allocated) {
            high_process[hp] = p;
            hp++;
        }
    }
}

p = 0;
for (c = 0; c < P; c++) {
    if (p >= hp)
        break;
    if (allocatable_processor[c] == -1)
        continue;
    if (processor->core[c].allocated_process != CPU_NO_PROCESS)
        process_table[processor->core[c].allocated_process].state = PROCESS_STATE_REA
```

```
DY;
                process_table[high_process[p]].state = PROCESS_STATE_RUNNING;
                processor->core[c].allocated_process = high_process[p];
                p++;
            }
            delete[] allocatable_processor;
            delete[] high_process;
        }
        else {
            // ii. 비선점 방식의 스케줄링 알고리즘
            for (int i = 0; i < P; i++) {
                if (processor->core[i].allocated_process == CPU_NO_PROCESS) {
                    RPQ->enque(i);
                }
            }

            scheduler->setWeight(process_table, weight, N, clk);
            RQ->sort(weight);

            while (!RPQ->is_empty()) {
                if (RQ->is_empty()) { break; }
                c = RPQ->deque();       // c: 주목하는 코어의 번호
                p = RQ->deque();        // p: 할당할 프로세스 번호

                processor->core[c].allocated_process = p;
                process_table[p].state = PROCESS_STATE_RUNNING;
            }
        }

        // 재시동된 CPU 번호와 실행할 프로세스를 계산
        for (int i = 0; i < P; i++) {
            if ((running_process[i] == CPU_NO_PROCESS) && (processor->core[i].allocated_process !=
CPU_NO_PROCESS))
                cpu_restarted[i] = 1;
            running_process[i] = processor->core[i].allocated_process;
        }
        return;
}

// RT, WT, TT, NTT 계산
void Kernel::proceed(std::vector<int>& running_process, std::vector<int>& cpu_restarted) {
        int p;
        for (int i = 0; i < P; i++) {
            if (cpu_restarted[i]) {
                processor->core[i].activate();
            }
            p = running_process[i];
            if (p != CPU_NO_PROCESS) {
                processor->core[i].total_consumption    +=    processor->core[i].regular_consumption;
// 기본 소비 전력 소모
```

```
            process_table[p].RT -= processor->core[i].work_rate;                    // 프로세스 처리(RT 감소)
            process_table[p].actual_working_time += 1;                              // 실제 실행 시간 증가
            if ((process_table[p].RT <= 0) && (process_table[p].TT == PCB_UNCERTAIN) && (process_table[p].WT == PCB_UNCERTAIN)) {
                process_table[p].TT = clk + 1 - process_table[p].AT;                // 처리가 끝난 프로세스의 TT를 계산
                process_table[p].WT = process_table[p].TT - process_table[p].actual_working_time;
                process_table[p].NTT = (double)process_table[p].TT / (double)process_table[p].actual_working_time;
                // 처리가 끝난 프로세스의 NTT를 계산
            }
        }
    }
    return;
}

bool Kernel::stepProcess(std::vector<int>& running_process, std::vector<int>& cpu_restarted, std::vector<int>& newly_terminated_process, std::vector<double>& weight) {
    clk++;
    arrivalCheck();

    processSchedule(running_process, cpu_restarted, newly_terminated_process, weight);
    proceed(running_process, cpu_restarted);
    afterProcess();
    if (allTerminated())
        return true;
    return false;
}

void Kernel::get_process_state(std::vector<int>& process_state) {
    for (int i = 0; i < N; i++) {
        process_state[i] = process_table[i].state;
    }
    return;
}

void Kernel::get_remaining_time(std::vector<int>& remaning_time) {
    for (int i = 0; i < N; i++)
        remaning_time[i] = process_table[i].RT;
    return;
}

void Kernel::get_wating_time(std::vector<int>& waiting_time) {
    for (int i = 0; i < N; i++)
        waiting_time[i] = process_table[i].WT;
    return;
}
```

```cpp
void Kernel::get_actual_working_time(std::vector<int>& actual_working_time) {
    for (int i = 0; i < N; i++)
        actual_working_time[i] = process_table[i].actual_working_time;
    return;
}

void Kernel::get_turn_around_time(std::vector<int>& turn_around_time) {
    for (int i = 0; i < N; i++)
        turn_around_time[i] = process_table[i].TT;
    return;
}

void Kernel::get_normalized_turn_around_time(std::vector<double>& normalized_turn_around_time)
{
    for (int i = 0; i < N; i++)
        normalized_turn_around_time[i] = process_table[i].NTT;
    return;
}
```

hardware.h

```cpp
#pragma once
#include <vector>
#ifndef __HARDWARE_HPP__
#define __HARDWARE_HPP__

#define CPU_TYPE_NONE   0
#define CPU_TYPE_P      1
#define CPU_TYPE_E      2

#define CPU_NO_PROCESS  -1

class Core{
public:
    int type;                           // cpu 타입 (Perfomance 중심)
    int work_rate;                      // 단위시간당 처리 속도 [작업/초]
    double regular_consumption;             // 일반 동작시 소비 전력
    double starting_consumption;        // 재시동시 소비 전력
    double total_consumption;           // 커널 구동 이래로 소비한 총 전력 크기
    int allocated_process;              // cpu에 할당된 프로세스 번호

public:
    // 코어 클래스의 생성자입니다. 멤버 속성은 전부 기본값으로 지정합니다.
    Core(void) : type(CPU_TYPE_NONE),
        work_rate(0),
        regular_consumption(0.0),
        starting_consumption(0.0),
        allocated_process(CPU_NO_PROCESS),
        total_consumption(0.0)
    {};
```

```
    // 지정된 type에 따라 멤버변수의 초기화를 진행합니다.
    // 총 소비 전력은 변경하지 않으므로 주의합니다.
    void setType(int type=CPU_TYPE_E);
    // 코어가 재시작하는 경우 총 소비 전력에 시동 전력을 추가.
    void activate();
};

class Processor {
public:
    int P;            // 사용 CPU 개수
    Core* core;       // CPU 코어

public:
    Processor(int p) :P(p) {
        core = new Core[P];
        return;
    }
    ~Processor() {
        delete[] core;
        return;
    }

    void setType(std::vector<int>& cpu_type) {
        for (int i = 0; i < P; i++)
            core[i].setType(cpu_type[i]);
        return;
    }
    Core& operator[](int idx) {return core[idx];}
    void get_power_usage(std::vector<double>& power_usage);
};

#endif
```

hardware.cpp

```
#include "hardware.h"
void Core::setType(int type) {
    switch (type) {
    case CPU_TYPE_E:
        this->type = CPU_TYPE_E;
        work_rate = 1;
        regular_consumption = 1;
        starting_consumption = 0.1;
        break;
    case CPU_TYPE_P:
        this->type = CPU_TYPE_P;
        work_rate = 2;
        regular_consumption = 3;
        starting_consumption = 0.5;
        break;
    case CPU_TYPE_NONE:
```

```
            this->type = CPU_TYPE_NONE;
            regular_consumption = 0;
            starting_consumption = 0;
            break;
        }
        return;
}

void Core::activate() {
        total_consumption += starting_consumption;
}

void Processor::get_power_usage(std::vector<double>& power_usage){
        for (int i = 0; i < P; i++)
            power_usage[i] = (double)core[i].total_consumption;
        return;
}
```