운영체제
최종보고서

김인호**(20192580)** 류태원**(20192587)** 이호연**(20150498)** 최지예**(20211318)**

3팀

목차

1. 서론

1.1 프로젝트 개요

본 프로젝트의 주제는 MiniOS 환경에서 프로세스 스케줄러를 구현하는 것이다. MiniOS는 Userspace 환경에서 동작하는 OS 구현체로, 실제 커널과 유사한 기능을 수행하도록 설계되었다. 비록 MiniOS가 실제 하드웨어 제어나 리소스 제어와 같은 기능을 완벽하게 수행하지는 않지만, 커널이 담당하는 프로세스 관리, 스케줄러 구현, 리소스 제어와 같은 핵심 기능을 구현하는 것을 목표로 한다.

이번 프로젝트에서는 리눅스 커널의 프로세스 관리 및 스케줄러 기능을 MiniOS에서 유사하게 구현하고자 한다. 이를 위해 리눅스의 struct task_struct와 유사한 프로세스 관리 구조체를 MiniOS에서 구현하고, 이 구조체를 바탕으로 실제 프로세스 스케줄러와 유사한 기능을 수행하도록 할 예정이다. MiniOS는 실제 리눅스 커널처럼 Ring 0에서 동작하지 않기 때문에 CPU 리소스를 시분할하여 프로세스를 스케줄링하는 방식은 아니지만, 커널의 개념을 차용하여 MiniOS 환경에서 유사하게 동작하는 시스템을 구현하는 것을 목표로 한다.

또한, 이러한 프로세스 관리 메커니즘을 사용자가 쉽게 이해할 수 있도록 시각화하는 것이 중요한 목표이다. 이를 위해 현재 스케줄러의 Run Queue에 어떤 프로세스가 존재하는지, 해당 프로세스가 time quantum을 모두 소모했을 때 어떤 동작을 하는지 등을 시각화할 것이다. 이 외에도, 프로세스별 메모리 사용 정보 등 세부 정보를 함께 제공하여 실제 운영체제가 어떻게 동작하는지에 대한 명확한 이해를 돕고자 한다.

1.2 관련 기술 소개

1.2.1 Round Robin Scheduler

Round Robin 스케줄러는 시분할 시스템에서 널리 사용되는 프로세스 스케줄링 알고리즘 중 하나다. 이 알고리즘은 각 프로세스에 동일한 우선순위를 부여하고, 정해진 시간 간격(time quantum) 동안 프로세스를 실행한 후, 다음 프로세스로 넘어가는 방식으로 동작한다. 이 과정을 모든 프로세스가 종료될 때까지 반복하며, 프로세스가 일정 시간 동안 실행되고 나면 다음 프로세스로 전환되기 때문에 공평하게 CPU 시간을 배분할 수 있다. Round Robin 스케줄러의 주요 장점은 간단하고 공평하다는 점이다. 모든 프로세스가 동일한 시간만큼 CPU를 사용할 수 있기 때문에 특정 프로세스가 과도하게 CPU를 점유하는 것을 방지할 수 있다. 이러한 특징 덕에 대기 시간이 짧고, 시스템 응답 시간이 예측 가능하다는 장점이 있다. 다만, 유의할 점은 Round Robin 스케줄러는 time quantum을 설정하는 것이 중요한 고려 사항이다. 이 간격이 너무 짧으면 프로세스 전환이 자주 일어나

오버헤드가 커질 수 있고, 너무 길면 각 프로세스의 응답 시간이 길어질 수 있다. 따라서 적절한 time quantum을 각 프로세스에게 분배하는 것이 핵심이다.

1.2.2 SIGNAL

시그널은 운영체제에서 프로세스 간의 통신을 위한 수단으로 사용되는데, 특정 이벤트가 발생했을 때 커널이 프로세스에게 알리는 방법으로, 비동기적으로 동작하며 프로세스의 실행 흐름을 제어하는 데 유용하다. 대표적인 시그널로는 `SIGINT`, `SIGKILL`, `SIGTERM`, `SIGSEGV` 등이 있다.

SIGINT는 사용자가 Ctrl+C를 누를 때 발생하는 시그널로, 일반적으로 프로세스를 중단하고 종료시키는 역할을 한다. SIGTSTP는 사용자가 Ctrl+Z를 누를 때 발생하는 시그널로, 현재 실행 중인 프로세스를 일시 중지시키며, SIGCONT는 일시 중지된 프로세스를 다시 실행시키는 시그널로, fg 또는 bg 명령어와 함께 사용된다. 이러한 시그널들은 프로세스 관리에 유용하게 사용되는 시그널이다.

특히 SIGSTOP, SIGTSTP와 SIGCONT 시그널을 활용하면 프로세스 관리를 효과적으로 수행할 수 있다. 예를 들어, 특정 프로세스를 실행시키다가 일시적으로 중단시키고 다른 작업을 수행하고 싶을 경우, Ctrl+Z를 눌러 SIGTSTP 시그널을 보내 일시 중지시킬 수 있다. 이후 필요한 조치를 취한 후 fg 명령어를 사용하여 SIGCONT 시그널을 보내 프로세스를 재개할 수 있다.

또한 앞서 설명한 이미 정의된 동작 말고도 signal handler를 이용하여 원하는 signal동작을 수행하도록 할 수 있다. 이처럼 시그널은 운영체제에서 프로세스 간 통신과 제어를 위한 중요한 도구로, 프로세스 관리와 통신에 있어 유용한 기능을 제공한다.

1.2.3 Reptyr

Reptyr는 터미널 세션에서 실행 중인 프로세스를 다른 터미널로 이동시키는 유용한 도구이다. 보통 SSH세션을 통해 작업을 할 때 세션이 끊기면 실행 중인 프로세스도 종료된다. 하지만 Reptyr을 이용하면 원래 터미널에서 시작된 프로세스를 중단하지 않고 다른 터미널로 옮겨 작업을 계속할 수 있다. 예를 들어, 긴 시간 동안 실행되는 작업을 한 터미널에서 시작했더라도, 다른 터미널로 이동하여 작업을 이어나갈 수 있다. 이는 특히 원래의 터미널이 종료되거나 접근할 수 없을 때 매우 유용하다.

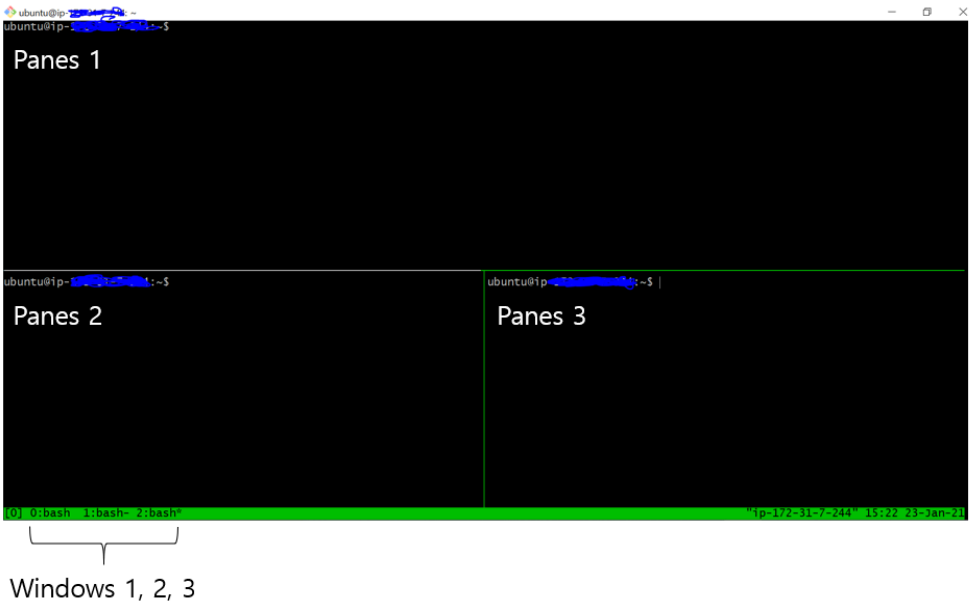
Reptyr를 사용하는 과정은 간단하다. 먼저 터미널에서 실행 중인 프로세스의 PID를 확인한 다음, 다른 터미널에서 `reptyr [PID]` 명령어를 입력하여 프로세스를 이동시킨다. 이러한 방법을 통해 사용자는 프로세스를 종료하지 않고도 계속해서 작업을 수행할 수 있다.

Reptyr는 프로세스를 다른 터미널로 이동시키기 위해 ptrace 시스템 콜을 사용한다. ptrace는 주로 디버거에서 사용하는 시스템 콜로, 한 프로세스가 다른 프로세스의 실행을 제어할 수 있게 해준다. 이외에도 Reptyr는 SIGTSTP와 SIGCONT 신호를 이용해 프로세스를 관리한다. 먼저, SIGTSTP 신호를 보내 대상 프로세스를 일시 중지시키고, 이후 ptrace를 통해 프로세스를 새로운 터미널로 이동시킨 다음, SIGCONT 신호를 보내 프로세스를 다시 실행시킨다. 이러한 방식으로 프로세스는 새로운 터미널에서 계속 실행될 수 있다.

1.2.4 Tmux

Tmux는 터미널 멀티플렉서로, 하나의 터미널 창에서 여러 개의 세션을 열고 관리할 수 있게 해주는 도구이다. 사용자는 Tmux를 통해 여러 개의 터미널 창을 생성하고, 각 창에서 독립적인 작업을 수행할 수 있다. 이는 특히 한 화면에서 여러 터미널을 확인해야 하는 환경에서 유용하게 사용될 수 있다.

Tmux는 3개의 계층인 session, windows, panes으로 구성된다. Session은 하나 이상의 윈도우가 있는 독립된 작업 공간이다. Windows는 동일한 세션에서 시각적으로 분리된 부분이다. Pane는 동일한 윈도우에서 분리된 부분이다. 다음 그림은 tmux를 통해 한개의 Session 내부의 세 개의 Windows와 세개의 Pane을 생성한 모습이다.



Tmux의 주요 기능인 창 분할 기능은 하나의 터미널 창을 여러 개의 창으로 분할하여 동시에 여러 작업을 모니터링하고 수행할 수 있다. 단순 터미널 화면만을 출력하는 것이 아니라, 특정프로세스를 분할된 화면에서 실행할 수도 있는데, 이 기능이 이번 프로젝트에서 활용하는 주요 기능 중 하나이다.

Tmux와 앞서 설명한 Reptyr를 사용하는 것이 이번 프로젝트의 핵심인데, 이를 이용해서 miniOS의 각 컴포넌트를 tmux로 UI화 시키고 이를 통해 사용자에게 쉽게 시각화해주는 것이 가능하다. 가령, Tmux의 창 분할 기능과 Reptyr를 결합하면 특정 프로세스를 다른 창으로 이동시킬 수 있다. Tmux와 Reptyr를 함께 사용하는 방법의 예시는 다음과 같다. 먼저, Tmux 세션을 생성한다 (`tmux new-session -s miniOS`). 생성된 세션 내에서 여러 작업을 수행하며, tmux split-window를 통해 창을 수직이나 수평으로 분할할 수 있다. 그 후, 분할된 창 중 하나에서 `reptyr [PID]` 명령어를 입력하여 특정 프로세스를 해당 창으로 이동시킬 수 있다.

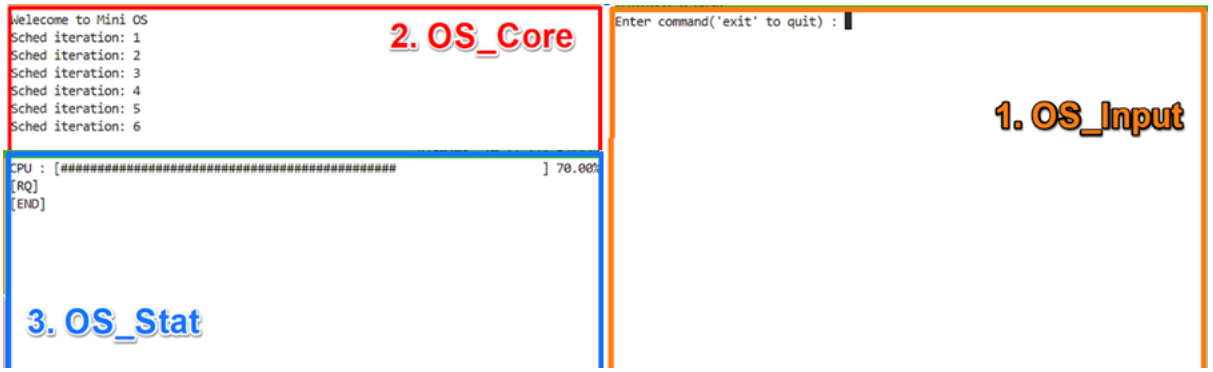
이와 같이 Tmux와 Reptyr를 함께 사용하면, 한 터미널에서 여러 작업을 효율적으로 관리하고, 프로세스를 유연하게 이동시킬 수 있다. 이는 특히 이번 프로젝트에서 miniOS 프로세스의 관리를 시각화하고, 스케줄러 동작을 시각화함으로써 커널의 세부 동작 과정을 시각화해 보여주고자 한다.

Tmux관련 명령어

- **Session**
  - tmux : 새로운 세션 시작
  - tmux new -s NAME : 세션 이름과 함께 새로운 세션 시작
  - tmux ls : 현재 세션 목록 나열
  - tmux 실행 중 ctrl+b -> d 를 누르고 현재 세션에서 빠져나오기
  - tmux a : 마지막 세션으로 들어가기
  - tmux a -t 세션 이름 : 특정 세션으로 들어가기
- **Window**
  - ctrl+b -> c : 새로운 윈도우 생성
  - ctrl+d : 윈도우 닫기
  - ctrl+b -> p,n : 이전, 다음 윈도우로 이동
  - ctrl+b -> , : 현재 윈도우 이름 바꾸기
  - ctrl+b -> w : 현재 윈도우 목록 나열
- **Pane**
  - ctrl+b -> ", % : 현재 창을 가로, 세로로 나누기
  - ctrl+b -> 방향키 : 방향키 방향의 창으로 이동
  - ctrl+b -> z : 현재 창 확대/축소 전환
  - ctrl+b -> [: space를 누르면 선택을 시작하고 enter를 누르면 선택 내용이 복사
  - ctrl+b -> space : 창 배열을 순환

2. 본론

2.1 MiniOS 구성



[그림 1]. Mini OS의 UI 구성

MiniOS의 사용자 인터페이스는 크게 세 가지 컴포넌트로 구성되어 있다. 이들 각각의 컴포넌트는 사용자의 입력을 받아들이고, 시스템 전반을 관리하며, 프로세스의 상태와 스케줄러의 동작을 시각화하는 역할을 한다.

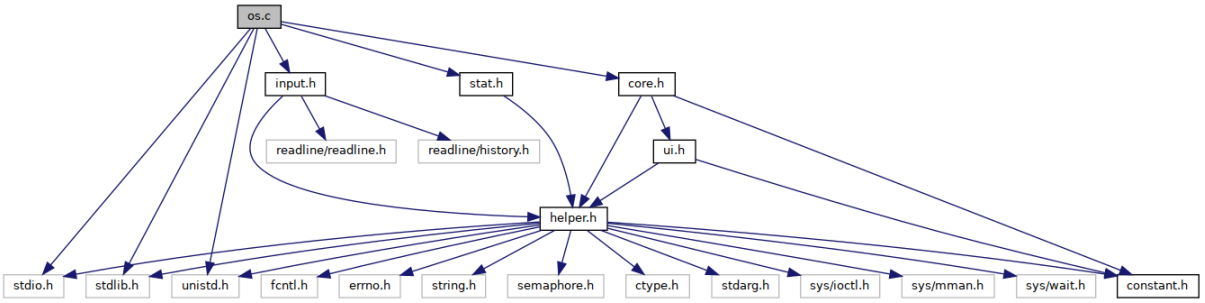
첫 번째 컴포넌트는 **OS_Input**이다. 이 컴포넌트는 사용자의 명령어 입력을 받는 역할을 한다. 사용자는 이 입력 창을 통해 MiniOS에 명령어를 전달하며, 이를 통해 다양한 작업을 수행할 수 있다.

두 번째 컴포넌트는 **OS_Core**이다. OS_Core는 MiniOS 시스템 전반에 대한 관리 기능을 수행한다. 이 컴포넌트는 시스템의 상태를 모니터링하고, 필요한 관리 작업을 수행하며, 스케줄러의 동작을 관리한다. OS_Core는 시스템의 핵심적인 기능을 담당하는 중심 컴포넌트이다.

세 번째 컴포넌트는 **OS_Stat**이다. 이 컴포넌트는 프로세스의 관리 상태와 스케줄러의 동작, 그리고 각 프로세스의 상세 정보를 시각적으로 보여준다. 사용자는 이 정보를 통해 현재 시스템에서 어떤 프로세스가 실행 중인지, 각 프로세스가 어느 정도의 자원을 사용하고 있는지 등을 확인할 수 있다.

이들 개별 컴포넌트는 앞서 기술 소개에서 언급했던 Reptyr와 Tmux를 이용해 구성된다. 각 컴포넌트는 개별 프로세스를 Tmux에 attach 한 형태로 되어 있으며, 마우스 클릭을 통해 프로세스 간 창 전환을 수행할 수 있다. 이러한 방식은 사용자가 다양한 작업을 동시에 수행하고 모니터링할 수 있게 해준다.

구성된 MiniOS의 모습은 최상단 파일에서 각 컴포넌트에 대해 의존성을 가지며, 각 컴포넌트별로 수행되어야 하는 함수들이 정의되어 있다. 최상단 OS 컴포넌트에서는 이 함수들을 불러들여 실행할 수 있도록 구성되어 있다. 이를 통해 MiniOS는 사용자 입력, 시스템 관리, 프로세스 모니터링을 유기적으로 결합하여 통합적인 운영체제 환경을 제공한다.



[그림 2]. Mini OS의 핵심 컴포넌트 의존성 그래프

2.1.1 사용자 프롬프트 (os_input)

OS_Input 컴포넌트는 사용자의 명령어 입력을 처리하는 역할을 담당한다. 이 컴포넌트는 사용자가 입력한 명령어를 읽고, 이를 OS_Core 컴포넌트로 전달하여 해당 명령어가 실행되도록 한다.

이 컴포넌트의 핵심 동작은 먼저, 필요한 입력 인자들을 파싱하고, 무한 루프를 통해 사용자 입력을 지속적으로 받는다. 이 과정에서 `readline` 함수를 사용하여 입력 프롬프트를 표시하고 사용자의 명령어를 입력받는다. 이후 입력된 명령어가 유효한지 확인하기 위해 입력을 검증하고, 만일 입력이 공백으로만 이루어진 경우, 해당 입력은 무시되고 루프의 처음으로 돌아간다. 유효한 입력인 경우, 입력을 히스토리에 추가하고, 입력 버퍼에 복사한 후, OS_Core 컴포넌트로 입력을 전달한다.

이외에도 일반적인 프롬프트의 동작처럼 구현하기 위해 Ctrl-C SIGINT에 대해 핸들러를 등록해 놨는데, 사용자가 Ctrl-C를 눌렀을 때 "Ctrl-C pressed. Press 'Ctrl-D' to quit." 메시지를 출력하고, readline의 입력 라인을 초기화하도록 구현하였다. 종료는 사용자가 Ctrl-D를 누르면 루프가 종료된다.

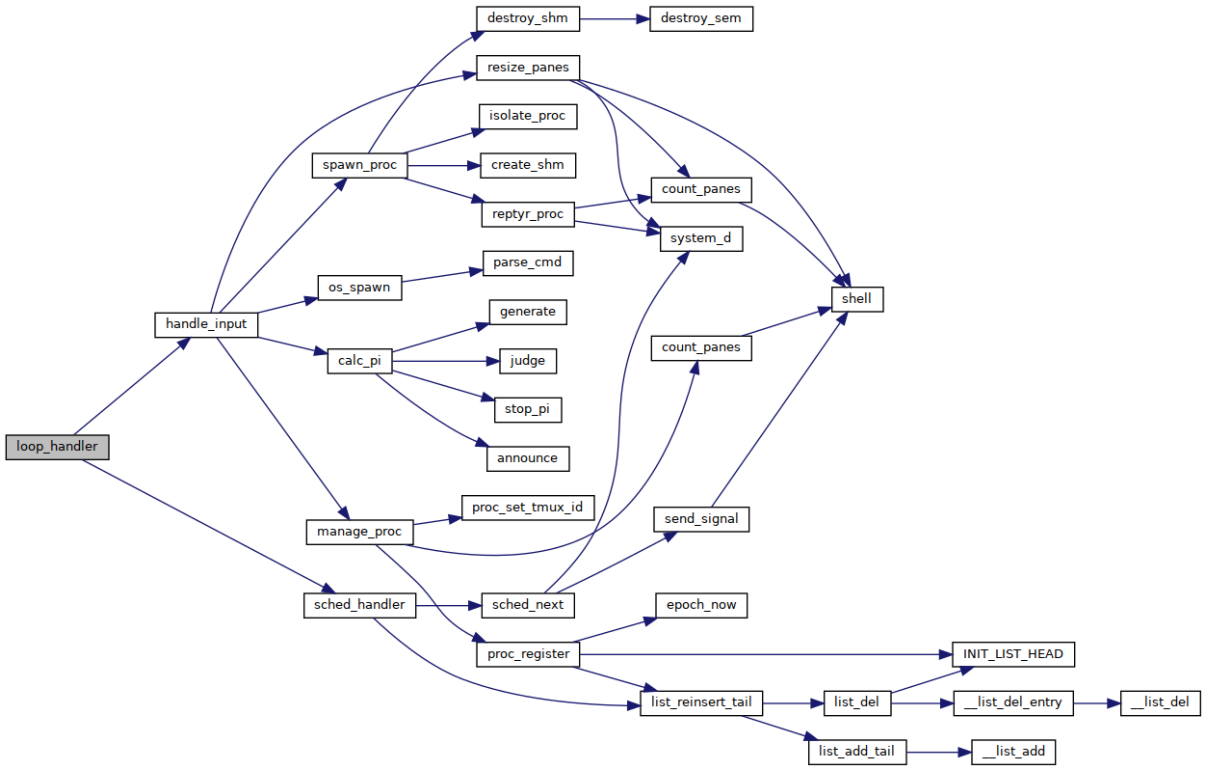
readline 라이브러리는 명령어 입력을 간편하게 처리하고, 입력 히스토리를 관리하는 데 사용된다. 이를 통해 사용자는 편리하게 명령어를 입력하고, 이전 명령어를 쉽게 재사용할 수 있다. 또한, 입력된 명령어는 공유 메모리를 통해 OS_Core 컴포넌트로 안전하게

전달된다. 이 과정에서 세마포어 락을 사용하여 여러 프로세스가 동시에 자원에 접근하는 상황에서 발생할 수 있는 경쟁 상태를 방지하고, 입력이 올바르게 전달되도록 보장한다

2.1.2 MiniOS 관리 (os_core)

OS_Core 컴포넌트는 MiniOS의 핵심 관리 기능을 담당하며, 기본적인 프로세스 생성 이벤트와 스케줄러 동작을 관리한다. 이 컴포넌트는 사용자 입력에 따라 새로운 프로세스를 생성하고, 생성된 프로세스들을 효율적으로 스케줄링하여 관리하는 역할을 한다.

프로세스 생성은 fork/exec 계열의 함수를 통해 이루어진다. 다만, 특이한 점은 프로세스를 생성하는 함수(spawn_proc)는 명령어를 파싱하여 해당 명령어를 실행하는 프로세스를 생성하고, 부모 프로세스와 자식 프로세스 간의 실행 순서를 보장하기 위해 공유 메모리 기반 세마포어를 사용한다. 자식 프로세스는 자신의 프로세스 그룹을 설정하고, 부모 프로세스가 준비될 때까지 일시 중지 상태로 대기한다. 부모 프로세스는 자식 프로세스가 준비된 후에야 실행을 시작한다. 이로 인해 부모와 자식 간의 실행 순서가 올바르게 유지된다.



[그림 3]. Mini OS의 스케줄러 루프 핸들러 구조

스케줄러 구현은 타이머를 기반으로 동작하도록 설계되었다. 타이머 시작 함수(init_timer)는 일정 시간마다 시그널을 발생시키는 타이머를 설정한다. 타이머가 설정된 주기마다 시그널이 발생하면, 루프 관리 함수(loop_handler)가 호출되어 스케줄링 작업을 수행한다. 이

함수는 현재 실행 중인 프로세스들을 관리하고, 새로운 입력이 있을 경우 이를 처리하여 프로세스를 적절히 스케줄링한다. 이러한 구조를 통해 MiniOS는 프로세스 관리와 스케줄링을 효율적으로 수행할 수 있다. 이외에도 프로세스 관리를 위해 프로세스 테이블(`ptable`)을 사용하여 현재 실행 중인 프로세스들의 정보를 관리하며, 이는 스케줄러가 각 프로세스의 상태를 추적하고 관리하는 데 도움을 준다.

2.1.3 사용량 정보 모니터링 (os_stat)

OS_Stat 컴포넌트는 MiniOS의 상태를 모니터링하고 시각화하는 역할을 담당한다. 이 컴포넌트는 CPU와 메모리 사용량을 계산하고, 각 프로세스의 상세 정보를 출력하여 시스템의 상태를 실시간으로 사용자에게 제공한다. OS_Stat 컴포넌트는 MiniOS의 상태를 모니터링하기 위해 타이머를 설정하고, 주기적으로 타이머 핸들러 함수(stat_hdlr)를 호출하여 시스템 상태를 갱신한다.

이 타이머 핸들러 함수는 CPU와 메모리 사용량을 계산하고, 이를 시각적으로 표시하는 역할을 한다. CPU 사용량 계산은 `/proc/stat` 파일을 읽어 이전과 현재의 CPU 시간을 비교하여 수행되며, 메모리 사용량 계산은 `/proc/meminfo` 파일을 읽어 전체 메모리와 사용 가능한 메모리를 비교하여 수행된다. 계산된 사용량은 막대 그래프로 시각화되어 화면에 출력된다.

이 함수는 또한 프로세스 테이블(`ptable`)에 저장된 각 프로세스의 정보를 출력한다. 프로세스 테이블은 현재 실행 중인 프로세스들의 정보를 관리하며, 이를 통해 각 프로세스의 상태를 추적하고 관리한다. 스케줄러의 동작 상황을 시각화하기 위해 스케줄러에 위치한 실행 대기열과 완료된 대기열의 프로세스 목록을 출력하며, 하단에는 각 프로세스의 PID, 상태, 가상 메모리 크기 등의 상세 정보를 출력한다.

OS_Stat 컴포넌트는 MiniOS의 상태를 시각적으로 표현하여 사용자에게 유용한 정보를 제공한다. CPU와 메모리 사용량을 실시간으로 모니터링하고, 프로세스의 상세 정보를 출력함으로써 시스템의 상태를 쉽게 파악할 수 있다. 이를 통해 사용자는 MiniOS의 성능과 리소스 사용 현황을 효과적으로 관리할 수 있다.

2.2 MiniOS 동작 방식 (내부 통신 구조)

MiniOS는 내부 통신을 위해 공유 메모리와 세마포어를 사용하여 각 컴포넌트 간의 효율적인 데이터 교환과 동기화를 보장한다. 이 시스템은 세 개의 주요 공유 메모리 영역인 "COMMON", "INPUT", "PTABLE"을 사용하여 OS_INPUT, OS_CORE, 그리고 OS_STAT 컴포넌트 간의 통신을 관리한다.

먼저, 공유 메모리와 세마포어를 생성하고 관리하는 부분인데, 공유 메모리 생성 함수(create_shm)는 공유 메모리 파일을 생성하고 매핑하며, 세마포어 생성 함수(create_sem)는 세마포어를 생성한다. 각각의 공유 메모리는 데이터 접근의 동기화를 위해 세마포어를 포함한다. 이 공유 메모리와 세마포어는 MiniOS의 최상위 컴포넌트인 boot.c 파일의 init_global에서 초기화된다. 여기서는 "COMMON", "INPUT", "PTABLE" 세 가지 공유 메모리를 생성하고 이를 초기화한다.

초기화된 공유 메모리는 boot 과정에서 각 컴포넌트로 전달된다. `spawn_proc` 함수는 각 컴포넌트를 새로운 프로세스로 실행하고, `os_args` 구조체를 통해 공유 메모리의 참조를 전달한다. 각 컴포넌트는 전달받은 공유 메모리를 사용하여 데이터를 읽고 쓴다. 예를 들어, OS_INPUT 컴포넌트는 사용자의 입력을 받아 `input_buf`에 저장하고, 세마포어를 통해 OS_CORE에게 이를 알린다.

이와 같이, MiniOS의 내부 통신 구조는 공유 메모리와 세마포어를 사용하여 각 컴포넌트 간의 데이터를 안전하게 교환하고 동기화하며, 타이머를 통해 주기적으로 스케줄러를 동작시켜 프로세스를 효율적으로 관리한다.

2.2.1 명령어 실행 과정 **(os_input -> os_core)**

MiniOS에서 사용자가 입력한 명령어는 OS_Input 컴포넌트에서 받아 OS_Core 컴포넌트로 전달되고 실행된다. 이 과정은 공유 메모리와 세마포어를 통해 각 컴포넌트 간의 효율적인 데이터 교환과 동기화를 보장한다.

먼저, 사용자가 터미널에서 명령어를 입력하면 OS_Input 컴포넌트가 이를 처리한다. `readline` 함수를 사용하여 입력된 명령어를 받아들이고, 유효성을 검사한다. 유효한 명령어는 `input_buf`에 저장된다. 명령어가 저장된 후, 세마포어를 통해 OS_Core 컴포넌트로 전달된다.

OS_Core 컴포넌트는 주기적으로 입력된 명령어를 확인하고 처리한다. `sem_trywait` 함수를 사용하여 세마포어가 해제될 때까지 대기한 후, `input_buf`에 저장된 명령어를 읽어와 실행한다. 입력된 명령어에 따라 적절한 함수를 호출하여 명령어를 처리하며, 특정 명령어는 별도의 처리 과정을 거친다. 예를 들어, "exit" 명령어는 MiniOS를 종료하고, "pi" 명령어는 π 값을 계산하는 함수가 호출된다.

이 과정에서 OS_Core는 공유 메모리와 세마포어를 사용하여 데이터의 일관성과 동기화를 유지한다. 입력된 명령어가 처리된 후, 결과는 OS_Stat 컴포넌트로 전달되어 시각화된다.

OS_Stat 컴포넌트는 주기적으로 CPU와 메모리 사용량을 계산하고, 각 프로세스의 상태를 출력하여 시스템의 상태를 실시간으로 모니터링할 수 있도록 한다.

2.2.2 프로세스 테이블 전달 과정 (os_core -> os_stat)

MiniOS에서 프로세스 테이블은 프로세스의 상태와 정보를 관리하는 핵심 데이터 구조이다. OS_Core 컴포넌트는 이 프로세스 테이블을 관리하고, OS_Stat 컴포넌트는 이를 통해 시스템 상태를 시각화한다. 프로세스 테이블 전달 과정은 다음과 같은 순서로 이루어진다.

OS_Core 컴포넌트는 프로세스 테이블을 주기적으로 업데이트한다. 이 테이블은 현재 실행 중인 프로세스, 대기 중인 프로세스, 완료된 프로세스 등의 상태 정보를 포함한다. OS_Core는 새로 생성된 프로세스를 프로세스 테이블에 등록하고, 프로세스의 상태가 변경될 때마다 이를 갱신한다.

프로세스 테이블은 공유 메모리 영역인 "PTABLE"에 저장된다. OS_Core는 이 공유 메모리를 통해 프로세스 테이블의 데이터를 읽고 쓴다. 공유 메모리를 사용하는 이유는 데이터의 일관성을 유지하고, 여러 컴포넌트가 동시에 접근할 수 있도록 하기 위해서다. 또한, 세마포어를 사용하여 데이터 접근을 동기화하고, 경쟁 상태를 방지한다.

OS_Stat 컴포넌트는 주기적으로 프로세스 테이블의 상태를 읽어와 화면에 시각화한다. stat을 관리하는 타이머 이벤트가 주기적으로 발생하면 CPU와 메모리 사용량을 계산하고, 프로세스 테이블에 저장된 각 프로세스의 상태를 출력한다. 이를 통해 사용자는 현재 시스템에서 어떤 프로세스가 실행 중인지, 각 프로세스가 어느 정도의 자원을 사용하고 있는지 등을 실시간으로 확인할 수 있다.

이와 같이, MiniOS는 공유 메모리와 세마포어를 활용하여 프로세스 테이블을 효과적으로 관리하고, 이를 통해 시스템의 상태를 실시간으로 모니터링할 수 있다. OS_Core는 프로세스 테이블을 업데이트하고, OS_Stat은 이를 기반으로 시스템 상태를 시각화하여 사용자가 시스템의 동작을 명확하게 파악할 수 있도록 한다.

2.3 MiniOS 프로세스 생성 방법

2.3.1 프로세스 생성 과정 (fork + 프로세스 격리)

MiniOS에서 프로세스 생성은 OS_Core 컴포넌트가 담당하며, 이를 통해 새로운 작업을 시작하고 관리할 수 있다. 프로세스 생성은 주로 `spawn_proc` 함수에 의해 이루어진다. 이

함수는 새로운 프로세스를 생성하고, 부모와 자식 간의 실행 순서를 보장하도록 설계되었다.

먼저, OS_Core는 새로운 프로세스를 생성하기 위해 필요한 준비를 한다. 이 과정에서 공유 메모리와 세마포어를 사용하여 부모 프로세스와 자식 프로세스 간의 상태를 공유하고 동기화한다. `create_shm` 함수를 사용하여 새로운 공유 메모리를 생성하고, 이를 통해 부모와 자식 간의 상태 정보를 주고받는다.

`spawn_proc` 함수는 `fork` 시스템 호출을 사용하여 부모 프로세스에서 자식 프로세스를 생성한다. `fork` 호출이 성공하면 두 개의 프로세스가 존재하게 되며, 반환 값에 따라 부모와 자식 프로세스가 서로 다른 코드를 실행한다. 자식 프로세스는 자신의 프로세스 그룹을 설정하고, 부모 프로세스가 준비될 때까지 일시 중지 상태로 대기한다. 이는 자식 프로세스가 독립적인 작업 단위로 동작하도록 하기 위함이다. 자식 프로세스는 이후 `SIGTSTP` 시그널을 받아 일시 중지 상태로 대기하게 되며, 이 시점에서 부모 프로세스가 준비되면 자식 프로세스는 작업을 시작할 준비가 된다.

부모 프로세스는 자식 프로세스가 준비된 후 실행을 시작한다. 부모 프로세스는 자식 프로세스를 `reptyr_proc` 함수를 통해 관리하며, 이를 통해 자식 프로세스의 상태를 모니터링하고 제어할 수 있다. 부모 프로세스는 또한 `manage_proc` 함수를 호출하여 자식 프로세스를 프로세스 테이블에 등록하고, 이를 통해 시스템 전체에서 프로세스의 상태를 관리할 수 있도록 한다.

OS_Core는 입력된 명령어에 따라 적절한 처리를 수행한다. 예를 들어, 사용자가 입력한 명령어가 "pi"일 경우 `calc_pi` 함수를 호출하여 pi 값을 계산하는 프로세스를 생성한다. 이 과정에서 `os_spawn` 함수가 사용되어 명령어를 실행하고, 결과를 출력한다.

프로세스가 종료되면 `cleanup_hdlr` 함수를 호출하여 관련 리소스를 정리하고, 공유 메모리와 세마포어를 해제한다. 이를 통해 시스템의 일관성과 안정성을 유지한다.

2.3.2 프로세스에 대한 터미널 연결 **(Reptyr)**

MiniOS에서 프로세스를 생성하고 관리하는 과정에서 Reptyr 도구는 프로세스를 다른 터미널로 유연하게 이동시키는 데 중요한 역할을 한다. Reptyr는 특정 터미널에서 실행 중인 프로세스를 다른 터미널로 이동시키는 도구로, 사용자는 이를 통해 프로세스를 종료하지 않고도 다른 터미널에서 작업을 계속할 수 있다.

MiniOS에서 Reptyr는 `reptyr_proc` 함수를 통해 구현된다. 이 함수는 특정 프로세스를 새로운 터미널 창에 연결하여 해당 프로세스를 다른 터미널에서 계속 실행할 수 있도록 한다. 먼저, `count_panes` 함수는 현재 Tmux 세션에서 열려 있는 창의 개수를 반환한다. 이 정보는 새로운 프로세스를 어느 창에 연결할지를 결정하는 데 사용된다. `reptyr_proc` 함수는 현재 열려 있는 창의 개수를 확인한 후, 새로운 창을 생성하여 프로세스를 이동시킨다.

구체적으로, `reptyr_proc` 함수는 다음과 같은 단계로 이루어진다. 먼저, `count_panes` 함수를 호출하여 현재 창의 개수를 확인한다. 만약 창의 개수가 3개 미만이면, 프로세스를 첫 번째 창에 연결하고, 3개인 경우에는 세 번째 창에 연결한다. 창의 개수가 3개를 초과하면 마지막 창에 프로세스를 연결한다.

이후, system 함수를 사용하여 Tmux 명령어를 실행한다. 새로운 창을 생성하고, `reptyr [PID]` 명령어를 실행하여 해당 PID를 가진 프로세스를 새로운 창에 연결한다. 이와 함께 `tmux select-pane` 명령어를 사용하여 새 창의 제목을 프로세스 ID로 설정한다.

또한, `resize_panes` 함수는 Tmux 창의 크기를 조정하는 역할을 한다. 이 함수는 창의 높이를 설정하여 프로세스가 효율적으로 표시되도록 한다. Tmux의 창 크기를 조정하기 위해 `window_height`를 가져오고, 각 창의 높이를 계산하여 적절하게 크기를 조정한다. 이 과정에서 첫 번째와 세 번째 창의 높이는 고정되어 있으며, 나머지 창의 높이는 윈도우 높이에 따라 동적으로 조정된다.

결론적으로, MiniOS에서 Reptyr는 프로세스를 다른 터미널 창으로 이동시켜 miniOS의 UI를 구성하고, 효율적인 시스템 관리를 가능하게 한다. 사용자는 Reptyr를 통해 프로세스를 종료하지 않고도 터미널을 전환하여 작업을 계속할 수 있으며, Tmux와 결합하여 강력한 관리 기능을 제공받을 수 있다.

2.4 MiniOS 스케줄러 구현

MiniOS는 운영체제의 기본적인 기능을 모방하여 프로세스를 생성하고 관리하는 시스템을 제공한다. 이 시스템은 주로 OS_Core와 proc 모듈을 통해 구현된다. MiniOS의 프로세스 관리 매커니즘과 라운드 로빈 스케줄링 방법은 다음과 같다.

MiniOS의 프로세스 관리 매커니즘은 프로세스 테이블(proc_table) 구조체를 중심으로 작동한다. 이 테이블은 시스템 내 모든 프로세스의 상태를 관리하며, 각 프로세스는 `task` 구조체로 표현된다. `task` 구조체는 프로세스 ID, 상태, 메모리 사용량, 우선순위, 시간 정보 등을 포함하고 있다.

프로세스가 생성되면, OS_Core는 `proc_register` 함수를 호출하여 새로운 프로세스를 프로세스 테이블에 등록한다. 이 함수는 프로세스의 기본 정보를 초기화하고, 프로세스를 실행 대기 큐(rq)에 추가한다. 또한, 각 프로세스는 Tmux 창과 연결되어 있어, 사용자는 Tmux 창을 통해 각 프로세스의 출력을 실시간으로 확인할 수 있다.

프로세스 테이블은 공유 메모리 영역 "PTABLE"에 저장되며, OS_Core와 OS_Stat 컴포넌트가 이를 통해 데이터를 주고받는다. 이를 통해 여러 컴포넌트가 동시에 프로세스 정보를 읽고 쓸 수 있으며, 세마포어를 사용하여 데이터 접근을 동기화하고 경쟁 상태를 방지한다.

MiniOS는 라운드 로빈(Round Robin) 스케줄링 알고리즘을 사용하여 프로세스를 관리한다. 이 알고리즘은 모든 프로세스에 동일한 실행 시간을 할당하여 공평하게 CPU 시간을 배분한다. MiniOS에서 라운드 로빈 스케줄링은 `sched_handler`와 `sched_next` 함수에 의해 구현된다.

`sched_handler` 함수는 타이머에 의해 주기적으로 호출되어 실행 대기 큐에 있는 프로세스를 관리한다. 이 함수는 현재 실행 중인 프로세스를 대기 큐의 끝으로 이동시키고, 다음 실행할 프로세스를 선택하여 실행을 시작한다. 선택된 프로세스는 `sched_next` 함수를 통해 실행되며, 이 함수는 해당 프로세스에 `SIGCONT` 시그널을 보내 실행을 재개시키고, 다른 모든 프로세스에는 `SIGTSTP` 시그널을 보내 일시 중지시킨다.

라운드 로빈 스케줄링은 각 프로세스가 일정 시간 동안 실행된 후, 다음 프로세스가 실행되도록 하여 공평한 자원 분배를 보장한다. MiniOS는 이러한 스케줄링 메커니즘을 통해 프로세스 간의 공평한 실행을 보장하며, 시스템의 성능과 응답성을 향상시킨다.

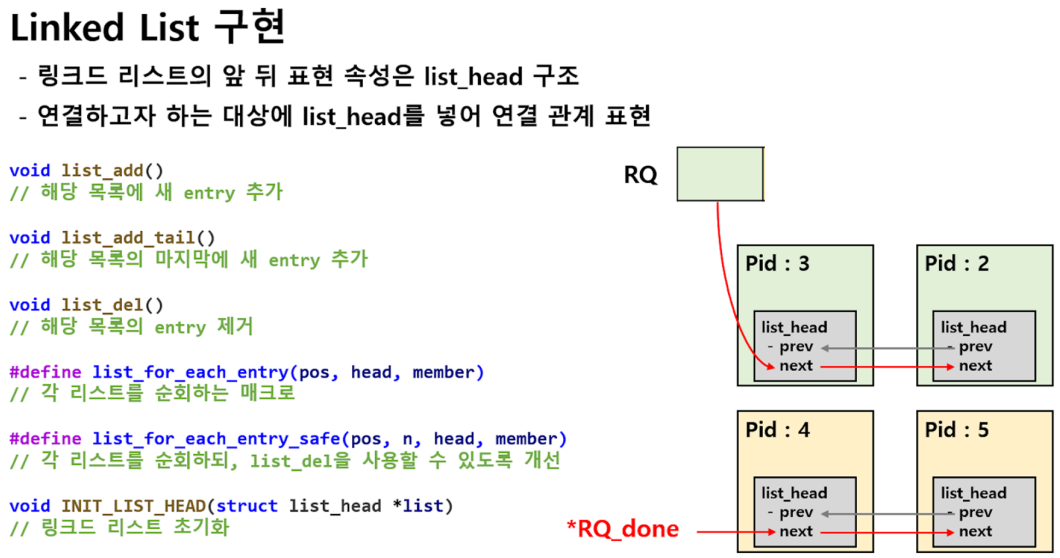
2.4.1 프로세스 관리 구조체 **(struct task, rq)** 구현

MiniOS의 프로세스 관리 시스템은 프로세스의 상태와 정보를 효율적으로 관리하기 위해 `struct task`와 실행 대기 큐(rq)를 중심으로 구성되어 있다.

`struct task`는 각 프로세스의 상태와 정보를 저장하는 구조체다. 여기에는 프로세스 명령어, Tmux 창 ID, 프로세스 ID, 프로세스 식별자, 프로세스 상태, 총 메모리 사용량, 프로세스 우선순위, 프로세스 시작 및 종료 시간, 작업에 필요한 전체 시간, 남은 시간, 대기 시간,

소모된 시간 등의 정보가 포함된다. 이 구조체는 각 프로세스의 중요한 정보를 저장하며, 프로세스의 상태를 관리하는 데 핵심적인 역할을 한다.

실행 대기 큐(rq)는 링크드 리스트를 사용하여 구현되었다. `list_head` 구조체는 이전 및 다음 항목을 가리키는 포인터를 가지고 있어 이를 이용해 리스트 내의 각 항목을 연결한다. 이 접근법을 사용한 이유는 새로운 메모리를 할당하지 않고 기존의 메모리만을 사용하여 링크드 리스트를 구현함으로써 개발상의 부담을 줄이고 메모리 관리의 어려움을 해소할 수 있기 때문이다.



[그림 4]. Mini OS의 Linked List 구현

다음과 같은 모습으로, list_head 구조체는 prev와 next를 가지고 있어 해당 list_head 구조체를 연결하고자 하는 대상에 집어넣음으로써 연결 리스트를 구현하였음을 확인할 수 있다. 여기서 이와 같은 접근법을 사용한 이유는, 링크드 리스트를 구현함에 있어 새로운 메모리를 할당하지 않고, 기존의 메모리만을 가지고 연결 리스트를 구현하게 되면 개발상의 부담을 줄이고 메모리 관리의 어려움을 해소할 수 있기 때문에 다음과 같이 구현하였다.

이 list 구현체에서 제공하는 함수들은 크게 다음과 같은 것들이 있는데, list_add / list_add_tail / list_del 함수를 통해 엔트리를 리스트에 추가하거나 삭제할 수 있고, 링크드 리스트를 편하게 순회하고 코드를 직관적으로 하기 위해 다음과 같이 list_for_each_entry 함수와 list_for_each_entry_safe라는 매크로를 구현하였다.

두 함수의 차이점으로는 list_for_each_entry에서는 포인터를 관리하는 방식에 따라 이 매크로 안에서 list_del을 수행했을 때 잘못된 메모리를 참조하게 되서 오류가 나는데,

이러한 문제를 해결하기 위해 포인터를 2개 사용하는 list_for_each_entry 매크로를 추가하였다. 마지막으로 INIT_LIST_HEAD는 링크드 리스트를 초기화하는 구조이다.

2.4.2 Timer tick 기반 스케줄러 구현

MiniOS의 스케줄러는 Timer Tick을 기반으로 프로세스를 스케줄링하는 메커니즘을 사용한다. 이는 일정한 시간 간격마다 인터럽트를 발생시켜, 스케줄러가 현재 실행 중인 프로세스를 중단하고 다음 프로세스를 실행하도록 하는 방식이다. 이 타이머 기반 스케줄러를 이용해 MiniOS에서는 이를 통해 공평하고 효율적인 프로세스 관리가 이루어진다.

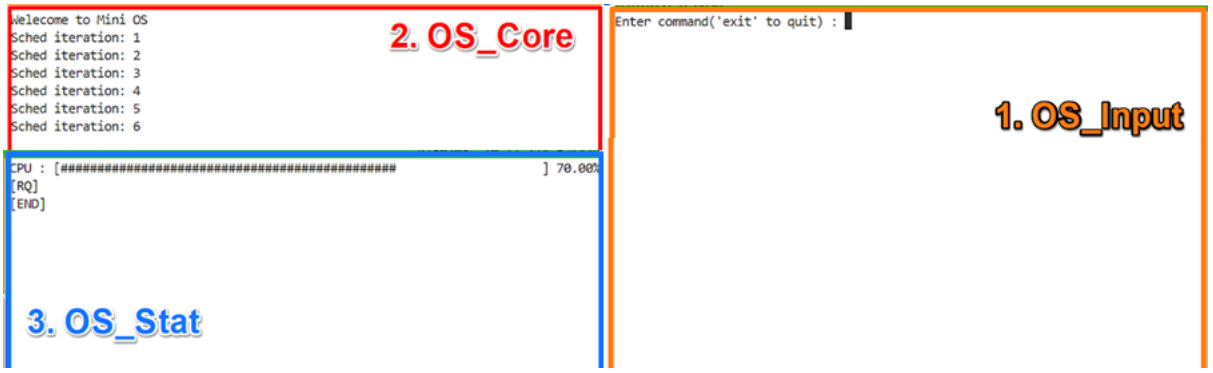
먼저, 타이머는 `init_timer` 함수에서 초기화되고 설정된다. 타이머가 설정된 주기마다 시그널이 발생하면, 해당 시그널에 등록된 핸들러가 호출되어 스케줄링 작업을 수행한다. 핸들러는 타이머에 의해 주기적으로 호출되며, 먼저 `handle_input` 함수를 호출하여 사용자의 입력을 확인하고 처리한다. 사용자가 명령어를 입력하면 이를 처리하여 새로운 프로세스를 생성하거나 종료한다. 이후 `sched_handler` 함수를 호출하여 현재 실행 중인 프로세스를 중단하고, 다음 프로세스를 실행한다.

MiniOS는 라운드 로빈 스케줄링 알고리즘을 사용하여 모든 프로세스에 공평하게 CPU 시간을 배분한다. `sched_handler` 함수는 현재 실행 중인 프로세스를 중단하고, 이를 실행 대기 큐의 끝으로 이동시킨다. 이후 실행 대기 큐에서 다음 프로세스를 선택하여 실행한다. 선택된 프로세스는 `sched_next` 함수를 통해 실행되며, 이 함수는 해당 프로세스에 `SIGCONT` 시그널을 보내 실행을 재개시키고, 다른 모든 프로세스에는 `SIGTSTP` 시그널을 보내 일시 중지시킨다. 각 프로세스의 실행 시간과 대기 시간도 이 과정에서 업데이트된다. 실행된 프로세스는 실행 시간이 증가하고, 대기 중인 프로세스는 대기 시간이 증가한다.

스케줄러는 프로세스의 상태를 관리하고, 실행 대기 큐와 완료된 작업 큐를 통해 프로세스의 실행을 제어한다. `proc_register` 함수는 새로운 프로세스를 생성하여 실행 대기 큐에 추가하고, `sched_next` 함수는 다음 실행할 프로세스를 선택하여 실행한다. 이 과정에서 프로세스의 상태가 변경되며, 각 프로세스의 실행 시간과 대기 시간이 지속적으로 업데이트된다.

3. 결론

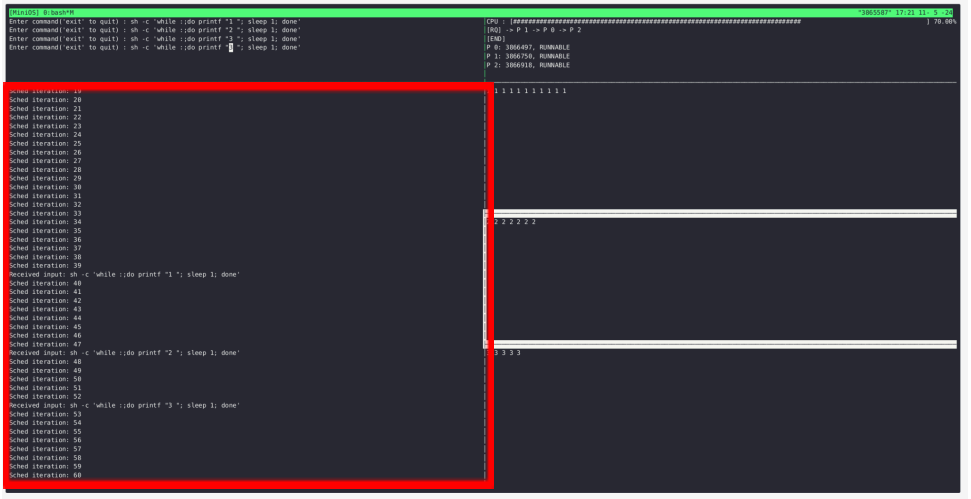
3.1 구현 화면



A. 사용자 입력 처리

- 명령어 입력: OS_Input 컴포넌트에서 사용자가 입력한 명령어를 확인.
readline 라이브러리를 사용하여 명령어를 입력받고, 입력을 검증한 후 처리 수행.
- 명령어 전달: 사용자 입력은 공유 메모리 "INPUT"을 통해 OS_Core 컴포넌트로 전달됨. 세마포어를 사용하여 데이터 접근을 동기화하고, 입력이 올바르게 전달되도록 보장함.

B. 프로세스 생성 관리

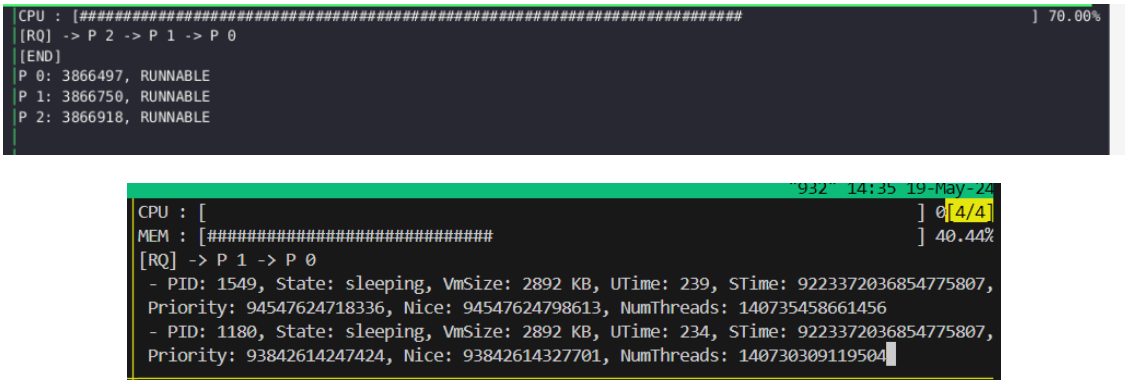


- 프로세스 생성: OS_Core는 spawn_proc 함수를 사용하여 새로운 프로세스를 생성함. fork를 사용하여 프로세스를 생성하고, 그룹을 설정하여 독립적인 작업 단위로 분리시킴
- 프로세스 등록: proc_register 함수를 통해 생성된 프로세스를 테이블에 등록. 이 프로세스 테이블은 모든 프로세스의 상태를 관리하고, 실행 대기 큐와 완료된 작업 큐를 사용함.
- 프로세스 관리: OS_Core는 프로세스 상태를 모니터링하고, 필요한 경우 프로세스를 중단하거나 재개한다. 이를 통해 각 프로세스간 할당 시간을 관리함.

C. 라운드 로빈 스케줄링

- 스케줄링 알고리즘: 라운드 로빈 스케줄링 알고리즘 통해 실행 시간을 할당. 각 프로세스는 일정 시간 동안 실행된 후, 다음 프로세스로 전환됨.
- 타이머 기반 스케줄링: init_timer 함수는 일정한 시간 간격마다 시그널을 발생시키는 타이머를 설정함. 타이머 핸들러인 loop_handler 함수는 주기적으로 호출되어 스케줄링 작업을 수행함.
- 프로세스 전환: sched_handler 함수는 현재 실행 중인 프로세스를 중단하고, 다음 프로세스 실행. sched_next 함수는 선택된 프로세스를 실행하고, 다른 모든 프로세스를 일시 중지시킴.

D. 시스템 상태 시각화



- 상태 모니터링: OS_Stat 컴포넌트는 주기적으로 CPU와 메모리 사용량을 계산하고, 이를 시각적으로 표시함. 이를 통해 사용자는 시스템의 상태를 실시간으로 모니터링할 수 있음.
- 프로세스 정보 출력: stat_hdlr 함수는 프로세스 테이블의 정보를 읽어와 각 프로세스의 상태를 출력함. 프로세스의 실행 시간, 대기 시간, 메모리 사용량 등을 확인할 수 있음.

3.2 챌린지

첫번째 문제는 SIGTSTP 명령어를 사용해 프로세스를 일시 중지시키려 했으나 예상대로 작동하지 않는 문제이다. SIGTSTP는 사용자가 일시적으로 작업을 중지하고 나중에 재개할 수 있도록 하는 터미널 명령어이나, 예상대로 작동하지 않았다. 이 문제를 해결하고자 SIGTSTP 대신 SIGSTOP 명령어를 사용하는 것으로 변경하였다. SIGSTOP의 경우, OS수준에서 프로세스를 완전히 중지시키는 신호로, SIGTSTP보다 더 강력하게 작동한다. SIGSTOP 명령어를 통해 프로세스가 정상적으로 일시 중지 하는 것을 확인할 수 있었다.

두 번째 문제는 프로세스의 어떠한 정보를 실시간으로 모니터링할지 결정하는 것이었다. 본 프로젝트의 주제에 맞게 적합한 정보를 보여주는 것이 가장 중요하다고 생각하였다. 따라서 CPU사용량, 메모리 사용량, 프로세스 시간, 대기 시간을 보여주는 것으로 회의를 통해 결정하였다. 이를 통해 프로세스 라운드 로빈 스케줄러에 대한 효과적인 모니터링을 보여줄 수 있었다.

3.3 향후 발전 방향

본 프로젝트를 통해서는 단순한 라운드 로빈 스케줄링만 구현하였고, 그에 대한 우선순위는 설정하지 않았다. 실제로 프로세스 사이에 우선순위를 염두해두는 것이 중요하기 때문에, 추후 프로젝트를 발전시킨다면 프로세스의 우선순위를 설정하여, 더 높은 우선순위를 가지는 프로세스를 우선적으로 실행하는 방법을 도입시킬 수 있을 것이다.

또한 프로세스의 실시간 정보를 계속해서 보여주고 있기 때문에, 프로세스의 상태와 Resource 사용량을 함께 고려하여 동적으로 우선순위를 조정하는 알고리즘을 추가하는 것도 향후 발전 방향으로 생각할 수 있다.

4. 회의록

4.1 7주차 회의

회의내용

1. 정기회의 요일 및 시간 고정

매주 수요일 6시 30분 대면 회의

2. 개인별 프로젝트 수행관련 역량 및 아이디어 공유

1) 리눅스 사용 경험 여부(본인 수준에 대한 디테일이 있으면 좋겠습니다.)

김인호
인강을 통해 리눅스 기초 독학 경험 및 연구실에서 몇 가지 명령어 및 커맨드 공부하였고, 컴퓨터 구조 과목에서 개인 프로젝트를 진행했으나 리눅스 공부라고 생각될 정도는 아닙니다. 간단한 코드를 보고 어느정도 이해는 할 수 있으나 쉘 스크립트를 능숙하게 다룰 수 있는 정도는 아닙니다. 기초공부정도 했던 수준입니다.

류태원
저는 현재 숭실대 AI보안연구센터에서 인턴을 하며 처음으로 리눅스를 다뤄보았고, sh파일을 통해 프로그램을 실행시키기 위해 리눅스의 명령어들을 공부해보았습니다. 주로 AI모델 학습을 위한 py코드와 함께 conda 환경 설정 및 모델 훈련, Evaluation을 위한 명령어들을 공부하였고 이번 운영체제 수업을 통해 virtualbox에서의 운영체제 생성 및 조작 방법에 대해 공부할 예정입니다. 또한 이번 프로젝트를 통해 하드웨어 및 소프트웨어의 작동 방식에 대해 지식을 더 쌓아보고 싶습니다.

이호연

파이썬 서버 배포할 때 기반 OS를 리눅스로 써 보았습니다. 주로 서버 프로그램 설치해서 사용해보는 것 위주로 사용해 보았습니다.

최지예
리눅스 사용 경험 여부: 전공 수업에서 architecture simulation을 사용한 application 성능 평가, 특성 분석 및 성능 최적화를 프로젝트로 진행하였습니다. 다만 단기로 진행한 개인 프로젝트여서 아직 리눅스에 대해 잘 알지 못하는 수준이라고 할 수 있습니다.

2) c언어 사용 경험 유무(본인 수준에 대한 디테일 함이 있으면 좋겠습니다.)

김인호

전공과목 및 멘토링 경험이 있습니다. 코드보고 이해 가능 및 기본 문법은 알고 있고 몇몇 라이브러리를 검색하여 어느정도 코딩을 할 수 있습니다.

류태원
C언어는 1학년 전공수업 때 기초를 다뤄보았고, 지난 학기 IoT실험 수업에서 아두이노를 사용하기 위해 C언어를 일부 다뤄보았습니다. 또한 2학년 2학기 신호 및 시스템 수업 프로젝트에서도 MATLAB 사용을 위해 조금 다뤄보았습니다.

이호연
C언어 수업을 15년도에 들었습니다. 그 외에는 과제 구현 경험 말고는 크게 없습니다.

최지예
C언어 사용 경험 여부: 마찬가지로 전공 수업에서 한학기 정도 공부하였습니다. 복잡하지 않은 수준의 코드를 읽고 해석할 수 있는 수준입니다.

3) 리눅스를 통해 자율주제로 프로젝트를 진행한다면 하고싶은 주제(디테일 하지는 않아도 대략적인 설명이 있으면 좋겠습니다.)

김인호
과제의 연장선으로 생각하여 우리 팀만의 minios를 완성해보는 것도 좋을 것 같습니다. minios의 사용 목적을 명확히 하여 해당 목적에서 사용할법한 함수들을 다양하게 구현하여 완성하면 좋을 것 같습니다.

ex) 공학용 계산기를 구현하겠다 -> 기본적인 연산 및 미적분 등 고급 연산 구현, 단 강의 내용에서 배운 fork, IPC등 내용을 자연스럽게 적용하여 프로젝트와 동시에 강의 내용공부까지 가능하게 하면 더 좋을 것 같습니다.

류태원
miniOS 환경에서 테트리스와 같은 쉬운 퍼즐게임을 만들어보는 것도 괜찮다고 생각합니다. 특히 OS환경에서 효율적으로 만드는 방법 등을 고민해보면서 만들면 공부에 더 도움이 될 것이라 생각합니다.

이호연
프로그램 성능 분석 프로그램 같은 거 만들어 보면 재밌을 것 같습니다.

최지예
운영체제에 대해 이제 배우고 있어서 어느정도 수준으로 프로젝트를 수행할 수 있을지 잘 모르겠습니다. 다만 가능하다면 리눅스 환경에서 간단한 자동화 작업을 구현하면 좋을 것 같습니다. 예를 들어보자면 파일 백업, 로그파일에서 원하는 정보 추출을 구현하면 좋을 것 같다고 생각하였습니다.

4.2 8주차 회의

회의 날짜: 2024.04.24
시간: 15:00~18:00
회의 참여자: 김인호, 류태원, 이호연, 최지예
회의 내용

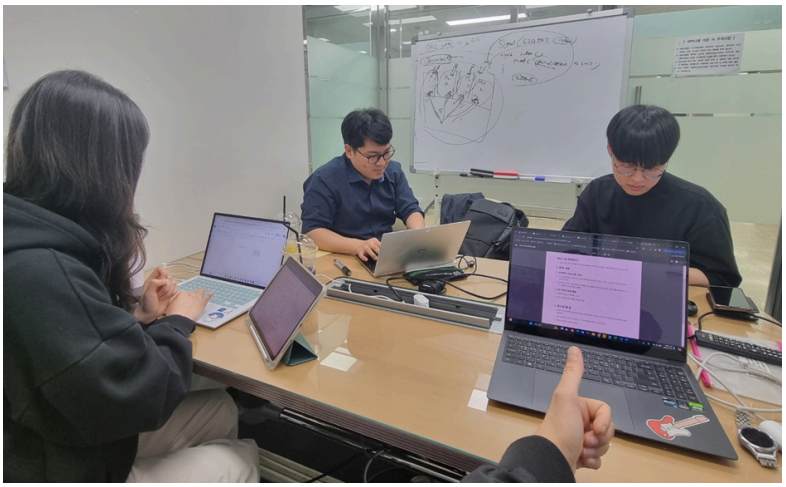
1. 프로젝트 주제 구체화

주제: miniOS환경에서 프로세스 스케줄러 구현

진행 현황에 따라 다양한 기능 추가 구현

2. 프로젝트를 위한 배경지식 조사

Round robin scheduler, fork, signal, IPC등 관련한 내용 복습 및 스터디



4.3 9주차 회의

1. 회의 일정

   2024.05.01 18:30~19:30

2. 회의 참여자

김인호, 류태원, ~~이호연,~~ 최지예
이호연(시험으로 인한 온라인 참여)

3. 회의 내용

1) 프로젝트 **UI** 구성

- miniOS 환경에서 프로세스 스케쥴러 구현을 위한 UI에 대해 토의

< 기본적인 UI 구상 형식 >

$ asdfasdf	Process Log		
scheduler 1	scheduler 2	scheduler 3	scheduler 4

2) UI 구성을 위한 프로그램

- TMUX(Terminal multiplexer)를 사용하기로 결정
- tmux 명령어에 대한 공부

Session
- tmux : 새로운 세션 시작
- tmux new -s NAME : 세션 이름과 함께 새로운 세션 시직
- tmux ls : 현재 세션 목록 나열
- tmux 실행 중 ctrl+b -> d 를 누르고 현재 세션에서 빠져나오기
- tmux a : 마지막 세션으로 들어가기
- tmux a -t 세션 이름 : 특정 세션으로 들어가기

● Window
- ctrl+b -> c : 새로운 윈도우 생성
- ctrl+d : 윈도우 닫기
- ctrl+b -> p : 이전 윈도우로 이동
- ctrl+b -> n : 다음 윈도우로 이동
- ctrl+b -> , : 현재 윈도우 이름 바꾸기
- ctrl+b -> w : 현재 윈도우 목록 나열

- Pane
    - ctrl+b -> " : 현재 창을 가로로 나누기
    - ctrl+b -> % : 현재 창을 세로로 나누기
    - ctrl+b -> 방향키 : 방향키 방향의 창으로 이동
    - ctrl+b -> z : 현재 창 확대/축소 전환
    - ctrl+b -> [: space를 누르면 선택을 시작하고 enter를 누르면 선택 내용이 복사
    - ctrl+b -> space : 창 배열을 순환

- tmux를 직접 사용해보고, 각자 다음주까지 UI생성하는 방법을 생각해오기

- 스케쥴러에 대해 개인 공부하기.

4.4 10주차 회의

1. 회의 일정
2024.05.08 18:00~19:30

2. 회의 참여자
김인호, 류태원, 최지예, 이호연

3. 회의 내용

1) 프로젝트 **UI** 구성
- miniOS 환경에서 프로세스 스케쥴러 UI 구성 확정
- UI예시)

<table>
<tr><td colspan="2">Terminal</td></tr>
<tr><td>Input
$</td><td>Status

CPU ############### 70%</td></tr>
<tr><td rowspan="2">Process Log</td><td>process 1

1 1 1 1 1 1 1</td></tr>
<tr><td>process 2

2 2 2 2 2 2 2</td></tr>
</table>

2) 운영체제에 대한 스터디
시그널, 프로세스, 그룹, KILL, PTY, TTY, terminal, Reptyr, 프로세스 테이블 학습.

3) 기본 **TMUX** 코드에 대한 스터디
split-window / select-window / kill-session / select-pane

4. 다음 회의까지 해올 것들
SIGTSTP 및 SIGCONT 등을 이용해 Round Robin 구현 및 통합
추가적인 명령어 공부
proc/ 밑의 관련 정보 확인하는 방법 공부

5. 추가 가능한 기능에 대해 브레인스토밍
Round Robin Priority 구현
Status에 들어갈 수 있는 추가 정보들
인터페이스 관련 아이디어 구상

4.5 11주차 회의

1. 회의 일정

2024.05.15 18:00~19:30

2. 회의 참여자

김인호, 류태원, 최지예, 이호연

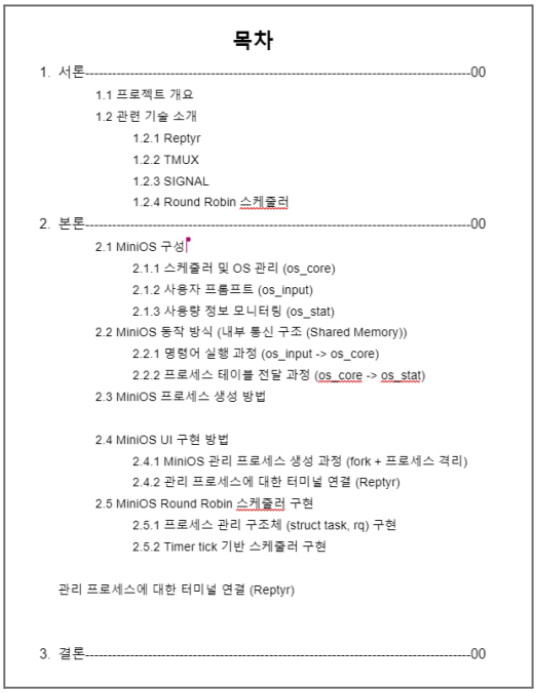
3. 회의 내용

1) 중간보고서 작성 내용 회의

- 보고서 포함 사항 및 현재 진행 현황 정리

2) 프로젝트 진행사항 challenge 내용 회의

- MiniOS에서 수행 중인 process가 signal로 중단되지 않는 문제
- 프로세스 종료 확인 불가 문제
- 수행 중인 프로세스 관련 세부 정보 확인 및 os_stat에 추가



목차

4. 다음 회의까지 해올 것들

- 각종 issue 해결방안 탐색
- 중간 보고서 작성

4.6 12주차 회의

1. 회의 일정

2024.05.19 14:00~17:00

2. 회의 참여자
김인호, 류태원, 최지예, 이호연

3. 회의 내용
1) 중간보고서 작성 내용 회의
- 중간보고서 개별 작성 및 내용 통합 검토

4. 다음 회의까지 해올 것들
- 각종 issue 해결방안 탐색
- 추가 기능에 대해 생각해오기

4.7 13주차 회의

1. 회의 일정
2024.05.29 18:00~20:00

2. 회의 참여자
김인호, 류태원, 이호연, 최지예
3. 회의 내용
1) minios UI 수정에 관한 회의
    a) CPU 사용량 구현했을 때에 관한 회의
    b) Mem 사용량 구현 관련 회의
2) Signal 처리 관련 수정
    a) SIGSTOP으로 프로세스 중단 명령 수정

3) UI 및 추가 기능 관련 회의

4. 다음 회의까지 해올 것들
- UI관련 사항 수정

4.8 14주차 회의

1. 회의 일정
2024.06.08 16:30~18:30

2. 회의 참여자

김인호, 류태원, 이호연, 최지예

3. 회의 내용

1) 최종 발표 자료 준비

2) 최종 보고서 작성

목차