2023년 1학기 운영체제 과제 #3

마지막 수정: 2023-06-06

과제 개요

1. 과제 내용

✓ 메모리 관리 기법을 확인할 수 있는 커널 시뮬레이터를 개발한다.

2 과제 목적

✓ 기본적인 커널의 메모리 관리 기법을 이해한다

3 제출 기한

✓ 2023년 6월 15일 23:55 까지 (Hard deadline)

4 제출 방법

✓ (1) pdf 형식의 보고서 파일과 (2) tar 형식의 실습 과제 압축 파일을 LearnUs에 제출

5 제한 사항

✓ 1인 1프로젝트이며 각자 환경에서 작업한다. 표절 검사가 반드시 진행된다는 점에 유의한다.

✓ 프로그래밍 언어는 `C/C++` 또는 `Rust`를 사용한다.

- 세가지 프로그래밍 언어(C, C++, Rust) 중에서 본인이 가장 편한 언어로 과제를 구현하면 되며, 프로그래밍 언어에 따른 채점의 유불리는 존재하지 않는다. 이 세 언어 외의 다른 언어(Java, Python 등)의 사용은 허용하지 않는다.
- Rust 언어는 기존의 C언어 외에 리눅스 커널 개발 언어로 새롭게 추가된 시스템 프로그래밍 언어로, 메모리 안전성이 주요한 특징이다. 수업시간에 이 언어를 다루지 않으므로, 관심있는 학생만 사용하면 된다.
- 표준 라이브러리는 자유롭게 사용 가능하나, 그 외의 외부 라이브러리의 사용은 금지한다.
- Makefile로 컴파일을 할 시, C/C++는 gcc로 컴파일, Rust는 rustc로 컴파일한다.

✓ 과제는 리눅스 환경에서 진행한다. 리눅스 종류와 버전 및 커널 버전은 자유롭게 설정할 수 있으나, 채점 환경을 참고하여 과제를 진행하는 것을 권장한다. 채점 환경에서 동작하지 않으면 점수를 받기 어렵다.

- OS: Ubuntu 20.04.5 LTS, locale: UTF-8 (euc-kr 사용 금지)
- gcc/g++: 9.4.0 (C/C++), rustc: 1.68.2 (Rust)

✓ 실습 과제 프로그램 구조 (주의: 구조가 다르면 채점이 진행이 안되어 0점 처리될 수 있음)

- 제출물의 압축을 해제하면 생성되는 폴더 내에 `Makefile`과 구현한 프로그램의 소스 코드가

존재해야 한다.

- `Makefile`에 필요한 옵션(라이브러리 옵션, C++14/17 옵션 등)은 반드시 정확하게 작성하여, 제출한 프로젝트가 채점 환경에서 컴파일이 문제없이 될 수 있도록 해야 한다.
- `make` 명령을 실행하면 각각의 소스 코드들이 컴파일되어, `project3`라는 이름의 실행 가능한 바이너리가 생성되어야 한다. 즉, 터미널에서 '`./project3`' 명령을 통해 프로그램이 실행된다.
- `make clean` 명령을 실행하면 `make`를 통해 생성된 빌드 결과물이 삭제되어야 한다.

6 유의 사항

✓ 늦은 제출은 절대 받지 않음

✓ 타 수강생의 보고서 및 과제를 카피 또는 수정하여 제출하였을 경우 모두 0점 처리

✓ 그림, 코드 조각을 포함한 모든 참고 자료는 반드시 **출처**를 명시. 과제에 참고 문헌이 없으면 감점.

✓ 제출한 소스 코드에는 반드시 **주석**을 달 것. 주석의 내용이 불분명하거나 없을 경우 감점.

✓ **과제 제출 방법, 프로그램 구조(특히 프로그램 이름)**를 지키지 않을 경우 감점 또는 0점.

✓ 과제에 대한 문의 사항이 있을 경우 **LeanUS 과목 게시판에 공개 질문**으로 문의한다.

- 단, 단순한 프로그래밍 언어의 사용과 관련된 질문이나 컴파일 관련 질문은 답변하지 않을 수 있다.

✓ 이 과제의 내용을 웹 사이트에 배포하거나 질문한 것이 발각되는 경우 0점 처리

✓ 본 과제에서 설명하는 파일에는 **확장자**가 대부분 존재하지 않는다. 임의로 *.txt와 같은 확장자를 붙이지 않도록 해야 한다.

보고서 과제

보고서는 자유 형식으로 작성하되, 아래의 내용을 반드시 포함하여 pdf 문서로 제출한다.

✓ 작성한 프로그램의 동작 과정과 구현 방법

- 소스 코드를 line by line으로 설명하기 보다는, 전체 큰 구조 관점에서 설명
- 각 알고리즘에 대하여 **순서도나 다이어그램 등 도식화 자료**를 반드시 활용할 것

✓ 개발 환경 명시

- uname -a 결과, 운영체제 및 컴파일러 정보, CPU 정보 등

✓ 결과 화면 및 토의 (중요)

- 결과 화면을 캡처하여 반드시 첨부하고, 과제 구현의 완성도가 얼마나 되는지 반드시 명시할 것
- 자신이 만든 프로그램이 얼마나 잘 동작하는지 성의 있게 분석할 것
- 직접 다양한 입력 케이스를 제작해보고 이를 바탕으로 본인의 구현물을 테스트해볼 것
- 서로 다른 페이지 교체 알고리즘의 성능 비교를 진행해 볼 것

✓ 과제 수행 시 겪었던 어려움과 해결 방법

✓ 과제 진행 중 참고한 문헌

실습 과제

실습 과제에서는 운영체제의 메모리 관리 기법을 확인할 수 있는 **커널 시뮬레이터**를 개발한다. 구체적으로, 이번 과제에서는 2차 과제에서 개발하였던 커널 시뮬레이터를 기반으로 하여, 메모리 관리와 관련된 기능이 추가된 커널 시뮬레이터를 개발하는 것을 목적으로 한다. 이 커널 시뮬레이터가 다루는 주요한 개념은 (1) 물리 메모리와 가상 메모리, (2) 페이지 교체 알고리즘, (3) Copy-on-Write이다. 본 과제 명세는 2차 과제에서 추가되거나 변경된 점을 중심으로 설명되어 있으니 유의할 것.

기본적인 입출력 구조는 2차 과제와 동일하다. 커널 시뮬레이터 위에서 존재하는 각 프로세스의 유저 모드는 커널 시뮬레이터 실행 시 인자로 주어지는 디렉토리 안에 존재하는 **가상 프로그램 파일**을 바탕으로 동작한다. 가상 프로그램 파일은 실제 코드가 아닌 간략화된 여러 줄의 **명령어**로 구성되어 있다.

```
memory_allocate 4
memory_allocate 3
memory_read 0
memory_write 1
memory_release 0
fork_and_exec program1
memory_release 1
wait
exit
```

2차 과제와 가장 큰 차이점은 메모리 관련 명령어가 추가되었다는 점이다. 이번 과제에서 구현하는 커널 시뮬레이터는 **프로세스마다 페이지 단위로 동작하는 가상 메모리가 존재**하며, **시스템 전체에 프레임 단위로 동작하는 물리 메모리가 하나 존재**한다. 가상 메모리의 1 페이지는 물리 메모리의 1 프레임에 대응되고, 가상 메모리는 32 페이지의 크기를 가지고 있고, 물리 메모리는 16 프레임의 크기를 가지고 있다. 유저 프로세스는 가상 메모리 전체를 사용할 수 있으며, 각 프로세스는 각 페이지가 물리 메모리의 어디를 참조하고 있는지를 나타내는 페이지 테이블을 가지고 있다. 단, 이번 과제에서의 페이지 테이블은 매우 단순한 형태로 구현하여, 페이지 아이디와 프레임 아이디에 대한 정보 정도만을 담고 있다. 또한, 이번 과제에서는가상 메모리의 공간이 부족한 경우는 고려하지 않는다. 만약, 현재 참조하고자 하는 페이지가 물리 메모리에 없거나, 할당하고자 하는 페이지가 있을 때 물리 메모리의 공간이 부족한 경우에는 **페이지 교체 알고리즘**(lru, fifo, lfu, mfu)을 사용한다. init 프로세스를 제외한 모든 프로세스는 생성 시점의 부모 프로세스의 가상 메모리를 복사해오는데, **Copy-on-Write**의 형식으로 복사한다. 즉, 부모 프로세스와 자식 프로세스의 페이지는 동일한 물리 메모리의 프레임을 가리키고 있다가 자식 프로세스의 페이지에 쓰기 명령이 들어올 때, 새로운 프레임으로 복사된다.

커널 시뮬레이터의 입출력

- 컴파일 결과로 실행되는 프로그램의 이름은 project3이다.
- project3 파일은 실행 시 반드시 두 인자를 받아 실행한다. (참고: argc, argv)
  - 인자 1. **가상 프로그램**들의 파일이 존재하는 디렉토리가 주어진다.
    - 인자 2. 페이지 교체 알고리즘이 주어진다: `lru, fifo, lfu, mfu`
  - 두 인자는 공백 한칸으로 구분되며, 반드시 '인자1' '인자2' 순서대로 주어진다.
  - 예시 입력.

```
$ ./project3 ../input lru
$ ./project3 /home/user/programs fifo
$ ./project3 ~/test/input lfu
```

- 이 시뮬레이터는 project3 파일과 같은 디렉토리에 result라는 파일을 생성하여 결과를 출력한다.

가상 프로그램 명령어

- 시스템 콜: `memory_allocate, memory_release, fork_and_exec, wait, exit`
  - 부모 프로세스는 반드시 fork_and_exec을 실행한 만큼 wait을 실행한다고 가정한다.
  - 2차 명세에 있었던 sleep은 이번 과제에서는 구현하지 않는다.
- 일반 유저 코드 명령어: `memory_read, memory_write`

memory_allocate arg1

- arg1: 16 이하의 양의 정수 (즉, 물리 메모리의 크기보다 큰 메모리 공간을 할당하지는 않음)
  - arg1로 주어진 페이지의 수만큼 연속적인 가상 메모리 공간을 할당하고, 비연속적으로 물리 메모리 공간도 할당한다. 연속적인 가상 메모리 공간이 부족한 경우는 고려하지 않으며, 물리 메모리 공간이 부족한 경우에는 페이지 교체 알고리즘을 통해 하나 이상의 페이지를 물리 메모리에서 제거한다.
    - (1)먼저, 페이지 교체 알고리즘을 통해 arg1로 주어진 만큼의 빈 프레임을 확보한다.
    - (2) 확보된 빈 프레임에 새로운 메모리를 할당한다.
  - 모든 메모리 할당은 하위 주소에서부터 시작한다.
  - Page ID는 0부터 1씩 계속 증가하는 방식으로 할당되며, 한 프로세스 안에서 고유하고, 재사용되지 않는다. 새로운 자식 프로세스는 부모 프로세스로부터 가상 메모리를 복사해오기에 부모 프로세스의 마지막 Page ID에서부터 새롭게 시작한다고 생각하면 된다.
  - Page ID는 한 페이지 단위로 부여된다. 즉, 마지막 Page ID가 10이고 이번에 10개의 페이지를 할당한다면, 새롭게 생성되는 페이지는 ID가 11부터 20까지가 된다. 하위 주소부터 1씩 증가하는 방식으로 부여된다고 생각하면 된다.
  - Allocation ID는 0부터 시작해서 memory_allocate 명령어가 실행될 때 마다 1씩 증가하며, 재사용되지 않는다. Allocation ID는 memory_release 명령어를 위해 존재한다. 자식

프로세스는 생성 시점에 부모 프로세스로부터 Allocation ID 정보를 복사해오기에, 해당 ID부터 시작해야 한다.

- 이 명령어는 시스템 콜로, 커널 모드로의 모드 스위칭이 발생한다.
  - 즉, 위에서 설명한 메모리 할당 작업은 커널 모드에서 이루어진다.

`ex) memory_allocate 10`

- 100번째 cycle에서 이 명령어가 실행된 경우, 해당 cycle에서 모드 스위칭 발생 (유저 모드)
- 101번째 cycle: 커널 모드에서 시스템 콜 처리 (커널 모드)
  - 커널 모드의 처리 결과로, 연속된 10개의 페이지가 가상 메모리에 할당된다.
    - 10개의 페이지가 연속으로 할당될 수 있는 가장 하위의 가상 메모리 공간에 할당된다.
    - 10개의 페이지는 가장 하위에 할당되는 페이지부터 1씩 증가하는 ID를 부여받는다.
  - 할당된 10개의 페이지에 해당하는 10개의 프레임이 물리 메모리에 불연속적으로 할당된다.
    - 가장 하위에 있는 비어있는 공간부터 채우며, 불연속적이어도 상관이 없다.
  - 새롭게 할당된 페이지에 대한 정보가 페이지 테이블에 기록된다.
  - 이번에 할당된 페이지들에 대해서 하나의 Allocation ID가 부여된다. 이번이 두번째 memory_allocate 호출이라면, 이번 Allocation ID는 1이 된다. 단, Allocation ID는 부모 프로세스부터 계속 센다는 점에 유의한다.
  - 커널 모드 처리 이후 해당 프로세스는 바로 다시 ready queue에 삽입된다. 즉, 출력되는 상태가 Ready이다.
- 102번째 cycle: 스케줄러 동작 (커널 모드)
  - 커널 모드에서 스케줄러가 ready queue에서 다음 프로세스를 선택한다.

`memory_release arg1`

- arg1: 100 이하의 양의 정수
  - arg1로 주어진 Allocation ID에 해당하는 페이지들을 가상 메모리에서 해제한다. 만약 해당 페이지들이 물리 메모리에도 존재한다면 물리 메모리에서도 해제한다.
  - 해제하는 페이지들의 내용이 CoW로 인해 부모와 자식 프로세스들 사이에서 공유되고 있었다면 (‘R 권한), memory_release는 해당 페이지을 모두 ‘W’권한으로 변경하고 각 프로세스들로 복사한다. 즉, 이 시점 이후에는 부모 프로세스의 페이지가 없어지기에 해당 Page ID에 대한 페이지/프레임은 각자 자식 프로세스에서 독립적인 존재가 된다.
    - 이 과정에서 자식 프로세스의 해당 페이지는 부모 프로세스의 페이지와 독립적인 존재가 되고, 부모와 자식 프로세스의 가상 메모리에서 해당 페이지가 W 권한이 된다. 다만, 이 시점에 복사된 페이지가 바로 물리 메모리에 할당되지는 않는다. 즉, 이 시점에 폴트 핸들러까지 이어지지는 않는다. 추후 부모 또는 자식 프로세스에서 쓰기나 읽기 명령이 발생하면, 그때 폴트가 발생하고 물리 메모리 할당이 실행된다.
- 이 명령어는 시스템 콜로, 커널 모드로의 모드 스위칭이 발생한다.

- 즉, 위에서 설명한 메모리 할당 작업은 커널 모드에서 이루어진다.

`ex) memory_release 1`

- 100번째 cycle에서 이 명령어가 실행된 경우, 해당 cycle에서 모드 스위칭 발생 (유저 모드)
- 101번째 cycle: 커널 모드에서 시스템 콜 처리 (커널 모드)
  - 커널 모드의 처리 결과로, Allocation ID가 1에 해당하는 페이지들이 가상 메모리에서 해제된다.
  - 커널 모드 처리 이후 해당 프로세스는 바로 다시 ready queue에 삽입된다. 즉, 출력되는 상태가 Ready이다.
- 102번째 cycle: 스케줄러 동작 (커널 모드)
  - 커널 모드에서 스케줄러가 ready queue에서 다음 프로세스를 선택한다.

`fork_and_exec arg1`

- arg1: 실행하고자 하는 프로그램 이름 (문자열)
  - 프로세스 ID는 다르지만 이름이 같은 프로그램이 여러 개 실행될 수 있다.
- 이 명령어는 시스템 콜로, 커널 모드로의 모드 스위칭이 발생한다.
- 새로운 프로세스를 생성한다. 생성된 프로세스의 ID는 '마지막으로 생성된 프로세스 ID + 1'이 된다. 한 번 사용된 프로세스 ID는 재생되지 않는다.

page 포인터 변수로 바꿔서 주소를 가리키게 코드 수정

- 새롭게 생성되는 자식 프로세스는 생성되는 시점(즉, 시스템 콜 처리 과정)에 부모 프로세스의 가상 메모리와 페이지 테이블을 Copy-on-Write의 형식으로 복사한다. CoW로 복사된 부모 프로세스와 자식 프로세스의 페이지는 동일한 물리 메모리의 프레임을 가리키고 있다가 나중에 쓰기 명령이 들어올 때, 새로운 프레임으로 복사된다.
  - 부모 프로세스의 Page ID에 대한 내용들이 CoW 형식으로 복사된다.
  - 부모와 자식 프로세스가 공유하는 페이지들은 부모와 자식 프로세스 모두에서 R 권한이 된다.
  - 부모 프로세스의 Allocation ID에 대한 내용도 복사된다.
- 기타 자세한 내용은 2차 과제의 명세를 참고한다.
- 단, 이번 과제에서 이 명령어는 init 프로세스만 호출한다고 가정한다.

`ex) fork_and_exec hello`

- 100번째 cycle에서 이 명령어가 실행된 경우, 해당 cycle에서 모드 스위칭 발생 (유저 모드)
- 101번째 cycle: 커널 모드에서 시스템 콜 처리 (커널 모드)
  - 커널 모드의 처리 결과로, 새로운 프로세스가 생성된다.
    - 새로운 프로세스의 경우 출력되는 프로세스의 상태가 New이다.
  - 부모 프로세스는 바로 다시 ready queue에 삽입된다. 즉, 출력되는 상태가 Ready이다.
  - 이 시점에서 CoW 형식으로 메모리 복사가 진행된다.
- 102번째 cycle: 스케줄러 동작 (커널 모드)
  - New 상태인 자식 프로세스가 Ready로 전환되고, Ready queue에 삽입된다.
  - 커널 모드에서 스케줄러가 ready queue에서 다음 프로세스를 선택한다.

wait

- 모드 스위칭을 유발하는 시스템 콜로, 기타 자세한 내용은 2차 과제의 명세를 참고한다.
- (주의) wait 명령으로 인해 waiting하고 있는 부모 프로세스가 ready로 바뀌어 ready queue에 삽입되는 시점은 자식 프로세스의 exit 명령어에 대한 시스템 콜 처리 시점임에 주의한다.

exit

- 기타 자세한 내용은 2차 과제의 명세를 참고한다.
- (주의) exit 시스템 콜이 커널 모드에서 처리되는 과정에서 wait 명령어로 인해 waiting 상태인 부모 프로세스가 ready queue에 삽입된다.
- 시스템 콜 처리 과정에서 해당 프로세스의 모든 페이지가 물리 메모리에서 해제된다. 정확히는 해당 프로세스의 memory_release 연산들이 모두 실행되는 결과와 같다. 즉, read 권한만 있는 경우에는 물리 메모리의 해당 프레임을 해제하지는 않는다.

memory_read arg1

- arg1: 읽고자 하는 페이지의 Page ID (1000 이하의 양의 정수)
- 이 명령어는 일반 유저 코드 명령어로 유저 모드에서 실행된다.
- arg1에 해당하는 페이지가 물리 메모리에 할당되어 있지 않은 경우, 페이지 폴트가 발생하고 커널 모드로 전환되어 페이지 폴트 핸들러가 해당 페이지를 물리 메모리에 할당한다. 필요한 경우 페이지 교체 알고리즘을 통해 페이지를 교체한다.
- Protection 폴트는 발생하지 않는다.

ex) memory_read 1

- 100번째 cycle에서 이 명령어가 실행된 경우, 해당 cycle에서 1번 Page ID에 읽기 시도
- 해당 페이지가 물리 메모리에 존재하는 경우,
  - 101번째 cycle에서 다음 유저 명령어를 실행함.
- 해당 페이지가 물리 메모리에 존재하지 않는 경우, 모드 스위치 발생.
  - 101번째 cycle: 커널 모드에서 페이지 폴트 핸들러 처리
    - 페이지 교체 알고리즘에 의해 해당 페이지가 물리 메모리에 할당되고, 해당 프로세스는 ready 상태가 되어 ready queue에 삽입된다.
  - 102번째 cycle: 커널 모드에서 스케줄러 처리

memory_write arg1

- arg1: 쓰고자 하는 페이지의 Page ID (1000 이하의 양의 정수)
- 이 명령어는 일반 유저 코드 명령어로 유저 모드에서 실행된다.
- arg1에 해당하는 페이지가 물리 메모리에 할당되어 있지 않은 경우, 페이지 폴트 핸들러가 해당 페이지를 물리 메모리에 할당한다. 필요한 경우 페이지 교체 알고리즘을 통해 페이지를 교체한다.
- arg1에 해당하는 페이지가 W 권한으로 되어 있는 경우, Protection fault가 발생하지 않는다.

- arg1에 해당하는 페이지가 R 권한으로 되어 있는 경우, 부모 프로세스와 자식 프로세스에서의 동작이 다르다.
  - 부모 프로세스: Protection fault가 발생하여 커널 모드로 전환되고, fault handler에 의해 부모와 자식 프로세스 모두에서 해당 페이지가 'W 권한'으로 변경된다. 부모 프로세스는 새로운 프레임을 생성하지는 않고, 기존의 프레임을 그대로 활용한다. 이 과정에서 자식 프로세스들의 새로운 물리 프레임 할당까지 이어지지는 않는다.
  - 자식 프로세스: Protection fault가 발생하여 커널 모드로 전환되고, fault handler에 의해 새로운 프레임이 생성되고 물리 메모리에 할당된다. 이때, 물리 메모리에 할당할 공간이 없다면 protection fault handler가 처리되고 있는 같은 cycle에 페이지 교체 알고리즘에 의한 페이지 교체까지 진행한다. 이 새로운 프레임 할당은 해당 자식 프로세스에 대해서만 이루어지고, 또다른 자식 프로세스가 해당 페이지를 공유하고 있었다면, 다른 자식 프로세스에 대한 프레임 할당을 이 과정에서 하지는 않는다.

`ex) memory_write 1`

- 100번째 cycle에서 이 명령어가 실행된 경우, 해당 cycle에서 1번 Page ID에 쓰기 시도
- 해당 페이지가 'R'에 해당하는 경우, 폴트 발생
  - 101번째 cycle: Protection fault handler가 실행되고, 해당 프로세스는 ready queue에 삽입된다.
  - 102번째 cycle: 스케줄러 동작
- 해당 페이지가 'W'에 해당하는 경우,
  - 해당 페이지가 물리 메모리에 있는 경우, 101번째 cycle에서 다음 유저 명령어 실행
  - 해당 페이지가 물리 메모리에 없는 경우, 모드 스위치가 발생. 101번째 cycle에서 page fault handler를 커널 모드에서 실행하여 페이지 교체 알고리즘에 의한 페이지 교체 실행. 해당 프로세스는 ready queue에 삽입된다.

위 규칙으로 작성된 가상 프로그램을 실행하는 커널 시뮬레이터의 상세한 동작은 다음과 같다.

- 상세한 내용은 2차 과제 명세를 참조하며, 현재 명세는 차이점을 중심으로 설명되어 있다.
- 유저 프로그램에서 **시스템 콜**을 호출하거나 **폴트**가 발생하면, 커널 모드로 **모드 스위칭**이 발생한다.
- **유저 모드**는 가상 프로그램의 명령어를 실행하고 있는 맥락이다.
  - 각 유저모드의 명령어는 1 cycle을 소비한다.
- **커널 모드**는 커널이 필요한 기능을 처리하고 있는 맥락이다.
  - 커널 모드의 하나의 동작도 1 cycle을 차지하며, 유저 모드 명령어와 동시에 처리되지 않는다.
  - **커널 모드의 동작은 다음 다섯가지 중 하나**이다.
    - (1) 부팅 동작 (`boot`): 0번째 cycle에서만 존재하는 동작으로, 시스템을 부팅한다.
    - (2) 시스템 콜 처리 (`system call`): 유저 모드의 시스템 콜 명령을 처리한다.

        - (3) 스케줄러 동작 (`schedule`): 다음 실행할 프로세스를 ready queue에서 선택한다.
            - schedule 동작이 끝난 cycle에는 반드시 running process가 있는 것으로 출력된다.
        - (4) 대기 상태 (`idle`): 스케줄러 동작이 발생할 때까지 대기한다.
            - 커널 모드 상태에서, 시스템 콜 처리하는 맥락이 아닐 때, 해당 cycle에 ready queue가 비어있으면 대기 상태이고, ready queue가 비어있지 않으면 스케줄러 동작이다.
        - (5) 폴트 처리 (`fault`): 페이지 폴트나 Protection 폴트를 처리한다.
- 이 시뮬레이터는 단순한 **FIFO 스케줄러**를 기반으로 동작한다.
- 같은 Cycle 내에서 작업 순서
    - (1) 프로세스 상태 갱신 (Waiting or New → Ready 전환, Terminated 삭제 처리) 및 Ready Queue 갱신
    - (2) 커널 모드 또는 유저 모드의 명령 실행
        - 부팅/시스템 콜이 아닌 커널 모드 진행 시, ready queue가 비어 있으면 대기 상태 (**`idle`**)
        - 부팅/시스템 콜이 아닌 커널 모드 진행 시, ready queue가 비지 않으면 스케줄링 (**`schedule`**)
        - 스케줄러 동작은 ready queue를 한 번 더 갱신하고, running 상태 프로세스 생성
            - 즉, schedule 동작의 cycle 결과에는 커널모드이지만 running process가 존재함.
    - (4) 해당 Cycle 실행 직후 결과 출력
- (주의) 특정 cycle에 동시에 ready 상태가 되는 여러 프로세스가 있는 경우, 다음 규칙을 따른다. 4월 16일에 런어스에 공지되었던 내용에서 조금 더 수정되었으므로, 현재 설명을 기준으로 구현할 것.
    - N번째 cycle에 ready가 되는 프로세스의 종류는 다음과 같이 두가지가 있다.
        - (1) 프로세스 상태 갱신 단계에서 ready가 되는 프로세스 (New → Ready)
        - (2) 시스템 콜 또는 폴트 핸들러 처리 과정에서 ready가 되는 프로세스
    - (1)과 (2)는 서로 다른 시점에 ready queue에 삽입되므로, '동시'가 아님에 유의한다.
    - 2차 과제와 달리 Sleep이 없기 때문에 (1)에 해당하는 케이스는 New에서 Ready가 되는 케이스밖에 없다. 즉, 동시에 여러 프로세스가 Waiting에서 Ready가 되는 경우가 3차 과제에서는 존재하지 않는다.
    - 마찬가지로, (2)에서도 exit 시스템 콜 핸들러의 처리 과정에서 부모 프로세스를 Ready queue에 삽입하거나, 기타 시스템 콜을 호출한 프로세스나 폴트가 발생한 프로세스가 해당 시스템 콜/폴트 처리 과정에서 ready queue로 삽입되는 경우들이 있는데, 해당 시스템 콜/폴트 핸들러가 둘 이상의 프로세스를 삽입하는 경우는 이번 과제에서 존재하지 않는다.
- 물리 메모리에 할당 및 참조 시 상황에 따라 적절한 페이지 교체 알고리즘을 사용한다.
    - 메모리 할당 명령어에 의해 같은 cycle에 여러 페이지가 할당되거나 교체되어 나갈 수 있다. 즉, cycle로 계산했을 때 동일하게 계산되는 페이지들이 있어 알고리즘에 의해 계산된 우선순위가 동일할 경우, 하위 메모리 주소에 있는 페이지부터 교체되어 나가는 것으로 한다. 메모리 할당 명령어에 의해 할당해야 하는 물리 메모리를 모두 할당할때까지 페이지 교체 알고리즘을 반복한다.

- 페이지 교체 알고리즘의 '참조' 시점은 memory_allocate, memory_read, 그리고 memory_write 명령어의 호출한 cycle이 종료되는 시점이다. 즉, 해당 명령어 처리가 끝난 다음 cycle부터 reference count가 1 증가한다. read 또는 write 명령어로 인해 폴트가 발생해서 새롭게 할당되는 경우에는, 폴트 처리 cycle 처리 과정 중에 '참조'된 것이고, 해당 cycle이 끝나면 reference count가 1 증가한다. 페이지 교체 알고리즘 등으로 인해 해당 페이지가 물리 메모리에서 해제되면 reference count가 초기화된다.

커널 시뮬레이터의 동작 결과는 다음과 같은 규칙으로 출력한다.

- 이 프로그램은 `project3`파일과 같은 디렉토리에 `result`라는 파일을 생성하여 결과를 출력한다.
- 출력하지 않는 정보들이 존재하지만, 시스템 내부에서는 해당 정보들을 가지고 있어야 한다.
- **매 cycle마다 명령이 실행된 직후의 시스템 상황**을 다음과 같은 정보를 포함하여 출력한다.
  - 각 cycle 마다의 정보는 개행 문자 두번으로 구분한다. (아래의 예시 코드 참고)
  - 맨 처음에는 몇번째 cycle인지 출력한다

(예시)	[cycle #0]

  - **1. 현재의 실행 모드:**
    - `user`: 유저 모드
    - `kernel`: 커널 모드. 현재 실행중인 프로세스가 없어도 커널 모드이다. (대기 상태)

(예시)	1. mode: kernel

  - **2. 현재 실행 명령어**
    - 유저 모드인 경우, 해당 cycle의 명령어와 인자 그대로 출력
    - 커널 모드인 경우, 다음 다섯가지 중 하나 출력: `boot`, `system call`, `schedule`, `idle`, `fault`

(예시1)	2. command: boot
(예시2)	2. command: run 30

  - **3. 현재 실행 프로세스**
    - 현재 Running인 프로세스가 있다면, '프로세스 ID(프로세스 이름, 부모 프로세스 ID)'의 형식으로 출력하고, 현재 Running인 프로세스가 없다면 none으로 출력한다.

(예시1)	3. running: none
(예시2)	3. running: 1(init, 0)

  - **4. 현재 물리 메모리 상황**
    - 4 프레임마다 | 기호를 넣어 구분한다. 맨 앞과 맨 뒤에도 | 기호를 넣는다. 그리고 각 항목 사이는 공백 한칸으로 구분하는데, |의 앞 뒤에는 공백을 삽입하지 않는다.
    - 현재 할당된 메모리는 프로세스 ID와 Page ID가 출력되며, 빈 메모리 공간은 -로 출력한다. 정확히는, '프로세스 ID(Page ID)' 형식으로 출력하는데, 프로세스 ID가 5고 Page ID가

3이면 5(3) 형식으로 출력하면 된다.

- CoW로 인해 둘 이상의 프로세스가 공유중인 프레임은 부모 프로세스의 ID를 출력하면 된다.

(예시)	4. physical memory: ¦0(0) 0(1) 1(2) 1(3)¦1(4) 1(5) 1(6) 1(7)¦- - - -¦- - - -¦

○ **5. 현재 실행중인 프로세스의 가상 메모리 상황**

- 현재 Running 상태의 프로세스가 없으면 이 정보는 출력되지 않는다.
- 4 프레임마다 | 기호를 넣어 구분한다. 맨 앞과 맨 뒤에도 | 기호를 넣는다.
- 현재 할당된 메모리는 Page ID가 출력되며, 빈 메모리 공간은 -로 출력한다.

(예시)	5. virtual memory: ¦0 1 2 3¦4 5 6 7¦8 9 - -¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦

○ **6. 현재 실행중인 프로세스의 페이지 테이블 상황**

- 5번 정보 (현재 실행중인 프로세스의 가상 메모리)와 위치를 맞추어서 두 줄을 출력한다.
- 첫번째 줄에는 해당 페이지의 물리 메모리에서의 위치를 출력한다. 물리 메모리에 현재 할당되어 있지 않은 경우에는 -로 출력한다.
- 두번째 줄에는 읽기/쓰기 권한을 출력한다. 기본적으로 현재 프로세스가 할당한 페이지는 읽기 권한과 쓰기 권한을 가지고 있어 W로 출력하고, 부모 프로세스와 자식 프로세스가 CoW로 인해 공유 중인 페이지인 경우에는 읽기 권한만 있으므로 부모와 자식 프로세스에서 모두 R로 출력한다. 이후, memory_release나 memory_write가 부모 또는 자식 프로세스에서 실행되면, 부모와 자식 모두 W가 된다.

(예시)	6. page table: ¦0 1 2 3¦4 5 6 7¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦ ¦R R W W¦W W W W¦W W - -¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦

입력 예시 (예시 가상 프로그램)

- `init, program1`가 주어진 디렉토리에 있다는 가정.
- 프로그램 `init`

```
memory_allocate 16
fork_and_exec program1
wait
memory_read 10
memory_write 10
memory_read 0
exit
```

- 프로그램 `program1`

```
memory_read 0
memory_write 0
memory_allocate 2
memory_release 1
memory_allocate 3
memory_release 0
memory_allocate 2
exit
```

출력 예시

바로 위의 예시 입력(가상 프로그램)이 있는 디렉토리가 실행 인자로 주어졌다는 가정. 페이지 교체 알고리즘은 `fifo`로 실행했다고 가정.

```
[cycle #0]
1. mode: kernel
2. command: boot
3. running: none
4. physical memory:
|- - - -|- - - -|- - - -|- - - -|

[cycle #1]
1. mode: kernel
2. command: schedule
3. running: 1(init, 0)
4. physical memory:
|- - - -|- - - -|- - - -|- - - -|
5. virtual memory:
|- - - -|- - - -|- - - -|- - - -|- - - -|- - - -|- - - -|- - - -|
6. page table:
|- - - -|- - - -|- - - -|- - - -|- - - -|- - - -|- - - -|- - - -|
|- - - -|- - - -|- - - -|- - - -|- - - -|- - - -|- - - -|- - - -|
```

```
[cycle #2]
1. mode: user
2. command: memory_allocate 16
3. running: 1(init, 0)
4. physical memory:
¦- - - -¦- - - -¦- - - -¦- - - -¦
5. virtual memory:
¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦
¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦- - - -¦

[cycle #3]
1. mode: kernel
2. command: system call
3. running: none
4. physical memory:
¦1(0) 1(1) 1(2) 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦

[cycle #4]
1. mode: kernel
2. command: schedule
3. running: 1(init, 0)
4. physical memory:
¦1(0) 1(1) 1(2) 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
¦W W W W¦W W W W¦W W W W¦W W W W¦- - - -¦- - - -¦- - - -¦- - - -¦

[cycle #5]
1. mode: user
2. command: fork_and_exec program1
3. running: 1(init, 0)
4. physical memory:
¦1(0) 1(1) 1(2) 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
¦W W W W¦W W W W¦W W W W¦W W W W¦- - - -¦- - - -¦- - - -¦- - - -¦

[cycle #6]
1. mode: kernel
2. command: system call
```

```
3. running: none
4. physical memory:
¦1(0) 1(1) 1(2) 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦

[cycle #7]
1. mode: kernel
2. command: schedule
3. running: 1(init, 0)
4. physical memory:
¦1(0) 1(1) 1(2) 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
¦R R R R¦R R R R¦R R R R¦R R R R¦- - - -¦- - - -¦- - - -¦- - - -¦

[cycle #8]
1. mode: user
2. command: wait
3. running: 1(init, 0)
4. physical memory:
¦1(0) 1(1) 1(2) 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
¦R R R R¦R R R R¦R R R R¦R R R R¦- - - -¦- - - -¦- - - -¦- - - -¦

[cycle #9]
1. mode: kernel
2. command: system call
3. running: none
4. physical memory:
¦1(0) 1(1) 1(2) 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦

[cycle #10]
1. mode: kernel
2. command: schedule
3. running: 2(program1, 1)
4. physical memory:
¦1(0) 1(1) 1(2) 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
¦R R R R¦R R R R¦R R R R¦R R R R¦- - - -¦- - - -¦- - - -¦- - - -¦
```

```
[cycle #11]
1. mode: user
2. command: memory_read 0
3. running: 2(program1, 1)
4. physical memory:
¦1(0) 1(1) 1(2) 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
¦R R R R¦R R R R¦R R R R¦R R R R¦- - - -¦- - - -¦- - - -¦- - - -¦

[cycle #12]
1. mode: user
2. command: memory_write 0
3. running: 2(program1, 1)
4. physical memory:
¦1(0) 1(1) 1(2) 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
¦R R R R¦R R R R¦R R R R¦R R R R¦- - - -¦- - - -¦- - - -¦- - - -¦

[cycle #13]
1. mode: kernel
2. command: fault
3. running: none
4. physical memory:
¦2(0) 1(1) 1(2) 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
```

여기서부터 변경해야함 : 메모리가 들어있을때 교체 코드 구현 안 함.
(프로세스 1에서 프로세스 2(program1) 걸로 물리메모리 0번 자리 교체해야함(fault 함수 만들기))

```
[cycle #14]
1. mode: kernel
2. command: schedule
3. running: 2(program1, 1)
4. physical memory:
¦2(0) 1(1) 1(2) 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
¦W R R R¦R R R R¦R R R R¦R R R R¦- - - -¦- - - -¦- - - -¦- - - -¦

[cycle #15]
1. mode: user
```

1. lru는 마지막으로 참조(r/w)된 cycle을 page에 기록해야 만들 수 있다.
2. fifo는 page가 물리 메모리에 저장된(할당,해제) cycle을 기록해놔야 알 수 있다.
이거보다, physical_memory 객체 안에 fifo되야할 index를 저장하는 변수를 만드는게 더 편할듯
3. LFU, MFU는 물리메모리에 저장된 페이지가 참조된 횟수를 page에 기록해놔야 알 수 있다.
(자식이 공유중인 부모의 page를 참조하는것도 부모의 page 참조횟수가 늘어나는 것)
page에 변수 추가:int alloccycle, refercycle, refercount
물리 메모리에서 해제 될 땐 이 참조된 cycle/횟수를 0으로 초기화 시켜야한다.

```
2. command: memory_allocate 2
3. running: 2(program1, 1)
4. physical memory:
¦2(0) 1(1) 1(2) 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
¦W R R R¦R R R R¦R R R R¦R R R R¦- - - -¦- - - -¦- - - -¦- - - -¦

[cycle #16]
1. mode: kernel
2. command: system call
3. running: none
4. physical memory:
¦2(0) 2(16) 2(17) 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14)
1(15)¦

[cycle #17]
1. mode: kernel
2. command: schedule
3. running: 2(program1, 1)
4. physical memory:
¦2(0) 2(16) 2(17) 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14)
1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦16 17 - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 - - 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦1 2 - -¦- - - -¦- - - -¦- - - -¦
¦W R R R¦R R R R¦R R R R¦R R R R¦W W - -¦- - - -¦- - - -¦- - - -¦

[cycle #18]
1. mode: user
2. command: memory_release 1
3. running: 2(program1, 1)
4. physical memory:
¦2(0) 2(16) 2(17) 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14)
1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦16 17 - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 - - 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦1 2 - -¦- - - -¦- - - -¦- - - -¦
¦W R R R¦R R R R¦R R R R¦R R R R¦W W - -¦- - - -¦- - - -¦- - - -¦

[cycle #19]
1. mode: kernel
```

cycle 14까진 fifo 알고리즘으로는 완성 시킴
cycle 15부터 가득 찬 물리메모리에 메모리를 할당하려고 하므로
mem_allocate 명령에 페이지 교체 알고리즘을 반복해서 넣어서 빈 자리를 만들어야함 -여기서부터 하기

virtual memory에는 페이지가 있지만 실제 물리 메모리에는 index 1, 2는 존재하지 않는다. 즉 page table 관계에서도 물리 메모리의 1번 2번 인덱스는 날라가고 뒤에 1번 2번 프레임에 할당된 새로운 페이지가 테이블에 기록된다. 내 구현에서는 가상메모리와 페이지 테이블에 같은 페이지 포인터가 들어가 있는데 이걸 어떻게 바꿔내야 할까?
즉, 페이지 테이블에 있는 페이지 2개만 frame num을 -1로 만들 수 있을까?
->page table의 아래 권한 부분을 page table에서 가져오는게 아니라 virtual_memory 에서 출력하자!
결국 5번과 6번은 정렬되어 있고 맨 위부터 (vm index)->page_id(내 코드에선 pagenum)-frame num-r/w_bit 순서로 적혀있는 구조이므로, 사실 전부 virtual_memory의 페이지 정보에 들어있는 항목들이다.
즉, 가상메모리의 frame number가 1, 2인 page를 찾아서 -1로 바꿔주면(즉, page와 frame의 매핑을 끊어주면) 아래와 같이 표현이 가능할 것이다 !
+ 출력하는 부분도 바꿀 필요가 없다. 결국 가상메모리와 pagetable은 같은 page를 담고 있기 때문에 그냥 frame에서 빠질 page의 frame number만 -1로 바꾸면 된다.
위쪽 fault 명령에서 change_page로 뺄때 빠지는 page frame number도 -1로 바꿔줘야함
(지금 바꾸는 프로세스에 가려져서 보이지 않는 부모 프로세스의 page table을 수정)

```
2. command: system call
3. running: none
4. physical memory:
¦2(0) - - 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦

[cycle #20]
1. mode: kernel
2. command: schedule
3. running: 2(program1, 1)
4. physical memory:
¦2(0) - - 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 - - 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
¦W R R R¦R R R R¦R R R R¦R R R R¦- - - -¦- - - -¦- - - -¦- - - -¦

[cycle #21]
1. mode: user
2. command: memory_allocate 3
3. running: 2(program1, 1)
4. physical memory:
¦2(0) - - 1(3)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 - - 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
¦W R R R¦R R R R¦R R R R¦R R R R¦- - - -¦- - - -¦- - - -¦- - - -¦

[cycle #22]
1. mode: kernel
2. command: system call
3. running: none
4. physical memory:
¦2(0) 2(18) 2(19) 2(20)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14)
1(15)¦

[cycle #23]
1. mode: kernel
2. command: schedule
3. running: 2(program1, 1)
4. physical memory:
¦2(0) 2(18) 2(19) 2(20)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14)
1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦18 19 20 -¦- - - -¦- - - -¦- - - -¦
```

page number 할당 방법 수정 -> 가장 큰 수 다음으로 넣는게 아니라 한 프로세스에서 alloc 명령이 실행된 갯수 +1부터 page number 할당하는거임

```
6. page table:
¦0 - - -¦4 5 6 7¦8 9 10 11¦12 13 14 15¦1 2 3 -¦- - - -¦- - - -¦- - - -¦
¦W R R R¦R R R R¦R R R R¦R R R R¦W W W -¦- - - -¦- - - -¦- - - -¦

[cycle #24]
1. mode: user
2. command: memory_release 0
3. running: 2(program1, 1)
4. physical memory:
¦2(0) 2(18) 2(19) 2(20)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14)
1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦18 19 20 -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 - - -¦4 5 6 7¦8 9 10 11¦12 13 14 15¦1 2 3 -¦- - - -¦- - - -¦- - - -¦
¦W R R R¦R R R R¦R R R R¦R R R R¦W W W -¦- - - -¦- - - -¦- - - -¦

[cycle #25]
1. mode: kernel
2. command: system call
3. running: none
4. physical memory:
¦- 2(18) 2(19) 2(20)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦

[cycle #26]
1. mode: kernel
2. command: schedule
3. running: 2(program1, 1)
4. physical memory:
¦- 2(18) 2(19) 2(20)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦- - - -¦- - - -¦- - - -¦- - - -¦18 19 20 -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦- - - -¦- - - -¦- - - -¦- - - -¦1 2 3 -¦- - - -¦- - - -¦- - - -¦
¦- - - -¦- - - -¦- - - -¦- - - -¦W W W -¦- - - -¦- - - -¦- - - -¦

[cycle #27]
1. mode: user
2. command: memory_allocate 2
3. running: 2(program1, 1)
4. physical memory:
¦- 2(18) 2(19) 2(20)¦1(4) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦- - - -¦- - - -¦- - - -¦- - - -¦18 19 20 -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦- - - -¦- - - -¦- - - -¦- - - -¦1 2 3 -¦- - - -¦- - - -¦- - - -¦
```

```
¦- - - -¦- - - -¦- - - -¦- - - -¦W W W -¦- - - -¦- - - -¦- - - -¦

[cycle #28]
1. mode: kernel
2. command: system call
3. running: none
4. physical memory:
¦2(21) 2(18) 2(19) 2(20)¦2(22) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14)
1(15)¦

[cycle #29]
1. mode: kernel
2. command: schedule
3. running: 2(program1, 1)
4. physical memory:
¦2(21) 2(18) 2(19) 2(20)¦2(22) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14)
1(15)¦
5. virtual memory:
¦21 22 - -¦- - - -¦- - - -¦- - - -¦18 19 20 -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 4 - -¦- - - -¦- - - -¦- - - -¦1 2 3 -¦- - - -¦- - - -¦- - - -¦
¦W W - -¦- - - -¦- - - -¦- - - -¦W W W -¦- - - -¦- - - -¦- - - -¦

[cycle #30]
1. mode: user
2. command: exit
3. running: 2(program1, 1)
4. physical memory:
¦2(21) 2(18) 2(19) 2(20)¦2(22) 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14)
1(15)¦
5. virtual memory:
¦21 22 - -¦- - - -¦- - - -¦- - - -¦18 19 20 -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 4 - -¦- - - -¦- - - -¦- - - -¦1 2 3 -¦- - - -¦- - - -¦- - - -¦
¦W W - -¦- - - -¦- - - -¦- - - -¦W W W -¦- - - -¦- - - -¦- - - -¦

[cycle #31]
1. mode: kernel
2. command: system call
3. running: none
4. physical memory:
¦- - - -¦- 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦

[cycle #32]
1. mode: kernel
2. command: schedule
```

```
3. running: 1(init, 0)
4. physical memory:
¦- - - -¦- 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦- - - -¦- 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
¦W W W W¦W W W W¦W W W W¦W W W W¦- - - -¦- - - -¦- - - -¦- - - -¦

[cycle #33]
1. mode: user
2. command: memory_read 10
3. running: 1(init, 0)
4. physical memory:
¦- - - -¦- 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦- - - -¦- 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
¦W W W W¦W W W W¦W W W W¦W W W W¦- - - -¦- - - -¦- - - -¦- - - -¦

[cycle #34]
1. mode: user
2. command: memory_write 10
3. running: 1(init, 0)
4. physical memory:
¦- - - -¦- 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦- - - -¦- 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
¦W W W W¦W W W W¦W W W W¦W W W W¦- - - -¦- - - -¦- - - -¦- - - -¦

                        이부분에서 page fault가 안나서 2사이클 밀림
[cycle #35]
1. mode: user
2. command: memory_read 0
3. running: 1(init, 0)
4. physical memory:
¦- - - -¦- 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦- - - -¦- 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
¦W W W W¦W W W W¦W W W W¦W W W W¦- - - -¦- - - -¦- - - -¦- - - -¦

[cycle #36]
```

```
1. mode: kernel
2. command: fault
3. running: none
4. physical memory:
¦1(0) - - -¦- 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦

[cycle #37]
1. mode: kernel
2. command: schedule
3. running: 1(init, 0)
4. physical memory:
¦1(0) - - -¦- 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 - - -¦- 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
¦W W W W¦W W W W¦W W W W¦W W W W¦- - - -¦- - - -¦- - - -¦- - - -¦

[cycle #38]
1. mode: user
2. command: exit
3. running: 1(init, 0)
4. physical memory:
¦1(0) - - -¦- 1(5) 1(6) 1(7)¦1(8) 1(9) 1(10) 1(11)¦1(12) 1(13) 1(14) 1(15)¦
5. virtual memory:
¦0 1 2 3¦4 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
6. page table:
¦0 - - -¦- 5 6 7¦8 9 10 11¦12 13 14 15¦- - - -¦- - - -¦- - - -¦- - - -¦
¦W W W W¦W W W W¦W W W W¦W W W W¦- - - -¦- - - -¦- - - -¦- - - -¦

[cycle #39]
1. mode: kernel
2. command: system call
3. running: none
4. physical memory:
¦- - - -¦- - - -¦- - - -¦- - - -¦
```

과제 제출 방법

보고서 pdf 파일과 소스 압축 파일을 별도의 파일로 제출한다. 즉, 총 제출 파일은 2개이다.

보고서 파일 (pdf)

PDF 파일로 변환하여 제출한다. 파일의 이름은 반드시 `hw3_학번.pdf`로 한다.

예시) 학번이 2023123123인 경우, 보고서 파일의 이름은 `hw3_2023123123.pdf`

소스 압축 파일 (tar)

반드시 다음 명령어를 참고하여 압축 파일을 생성한다. 다른 형식의 압축이나 이중 압축은 허용되지 않는다.

예시) 학번이 2023123123이고, 본인이 작업한 디렉토리가 `hw3`라고 가정.

```
$ cd hw3/..
$ mv hw3 2023123123
$ tar cvf hw3_2023123123.tar 2023123123
```

- 이후 해당 tar 파일을 압축 해제하였을 때, `2023123123`이라는 디렉토리 하나만 나타나야 한다.
- `2023123123`디렉토리 바로 아래에 `Makefile`이 존재하며, `make` 명령 실행 시 해당 디렉토리에 `project3`이라는 실행가능한 바이너리가 생성되어야 한다.
- 스크립트 기반으로 채점이 진행되므로, 이 제출 양식을 반드시 지켜야 불이익이 없다.