**Pintos Project 1: User Program (1)**

담당 교수 : 박성용 교수님

학번 / 이름 : 나호영

개발 기간 : 23.9.28 ~ 23.10.06

1. **개발 목표**

* 입력 받은 시스템 콜을 처리하기 위해 명령을 파싱한 후 stack에 저장하여 보내는 Argument Passing을 구현한다.
* 시스템 콜을 처리하는 시스템 콜 핸들러를 구현해 유저가 커널이 제공하는 기능을 활용할 수 있게 한다.
* Fibonacci, max\_of\_four\_int 와 같은 새로운 시스템 콜 함수와 서비스 제공 환경을 구현해본다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Argument Passing

유저가 입력한 명령어를 읽은 후 공백을 기준으로 파싱 한 후, 해당 토큰들을 스택에 정해진 순서대로 저장한다.

1. User Memory Access

유저 사이드에서 메모리 영역을 접근하려 할 때 커널 영역이나 할당되지 않는 영역에 침범하여 시스템에 영향을 끼치지 않도록 오류를 발생시켜 강제 종료시킨다.

1. System Calls

유저가 커널이 제공하는 기능을 사용할 수 있도록 시스템 콜에 해당하는 명령어를 입력받았을 때 Argument Passing 처리 후 시스템 콜 핸들러를 통하여 요청받은 서비스를 실행한다.

* 1. **개발 내용**
* Argument Passing

-> 입력받은 명령어를 strtok\_r 함수를 사용하여 파싱한다. 그 후 스택을 초기화하는 과정을 거친 뒤 토큰들을 정해진 순서대로 스택에 저장한다. 이 때 파라미터들은 가장 뒤부터 먼저 저장되고 실행할 파일이 가장 마지막에 저장된다. 그 후 메모리 주소 참조가 편하도록 word alignment를 4바이트 단위로 메모리에 저장시키기 위해 스택 데이터들을 4의 배수로 맞춰주는 작업을 진행한다. 그리고 데이터와 해당 데이터 주소값의 영역을 나누는 Null값을 넣은 후 데이터의 주소도 마찬가지로 데이터와 같은 순서대로 저장한다. 마지막으로 파싱된 토큰들의 총 개수를 나타내는 argc와 리턴 주소까지 스택에 저장한다.

* User Memory Access

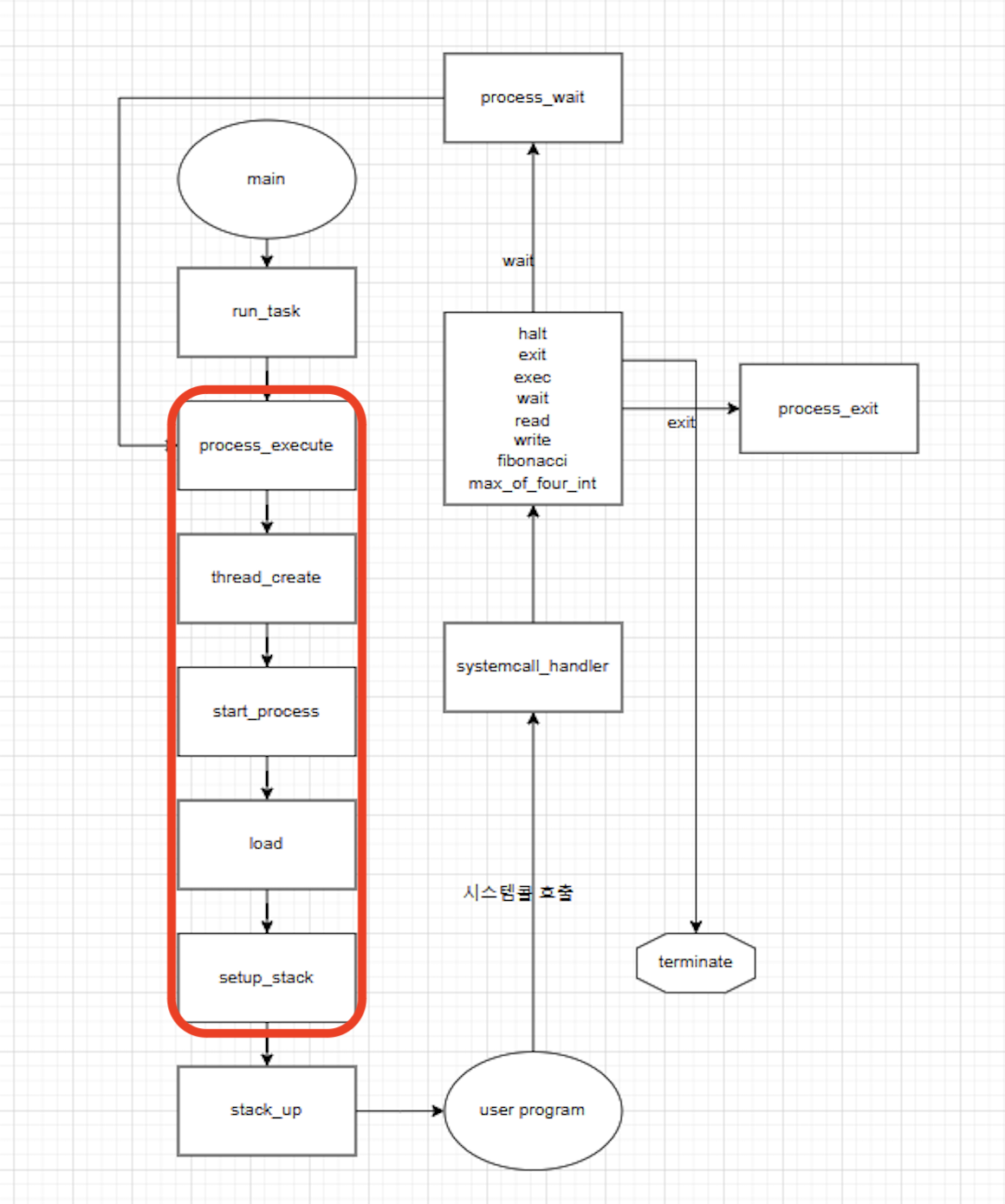
-> 핀토스 상에서 유저들이 메모리 접근을 시도할 때 해당 메모리가 유저 사이드에서 접근하면 안되는 주소일 때 invalid memory access라고 한다. 예를 들어 null 포인터거나 커널 주소 영역, 혹은 지정되지 않은 주소 영역일 때이다. 따라서 해당 영역에 접근을 시도한다면 시스템에 영향을 끼칠 수 있기 때문에 프로그램을 종료하는 등의 예외 처리가 필요하다. 이번 핀토스 프로젝트에서 invalid memory access를 막기 위해 한 방법은 주소를 넘겨 받을 때 null 여부를 검사와 pagedir\_get\_page 함수를 사용해 umapped virtual memory를 검사하였다. 또한 is\_user\_vaddr와 같은 함수를 사용해 커널 영역을 침범하지 않도록 예외처리를 진행하였다.

* System Calls

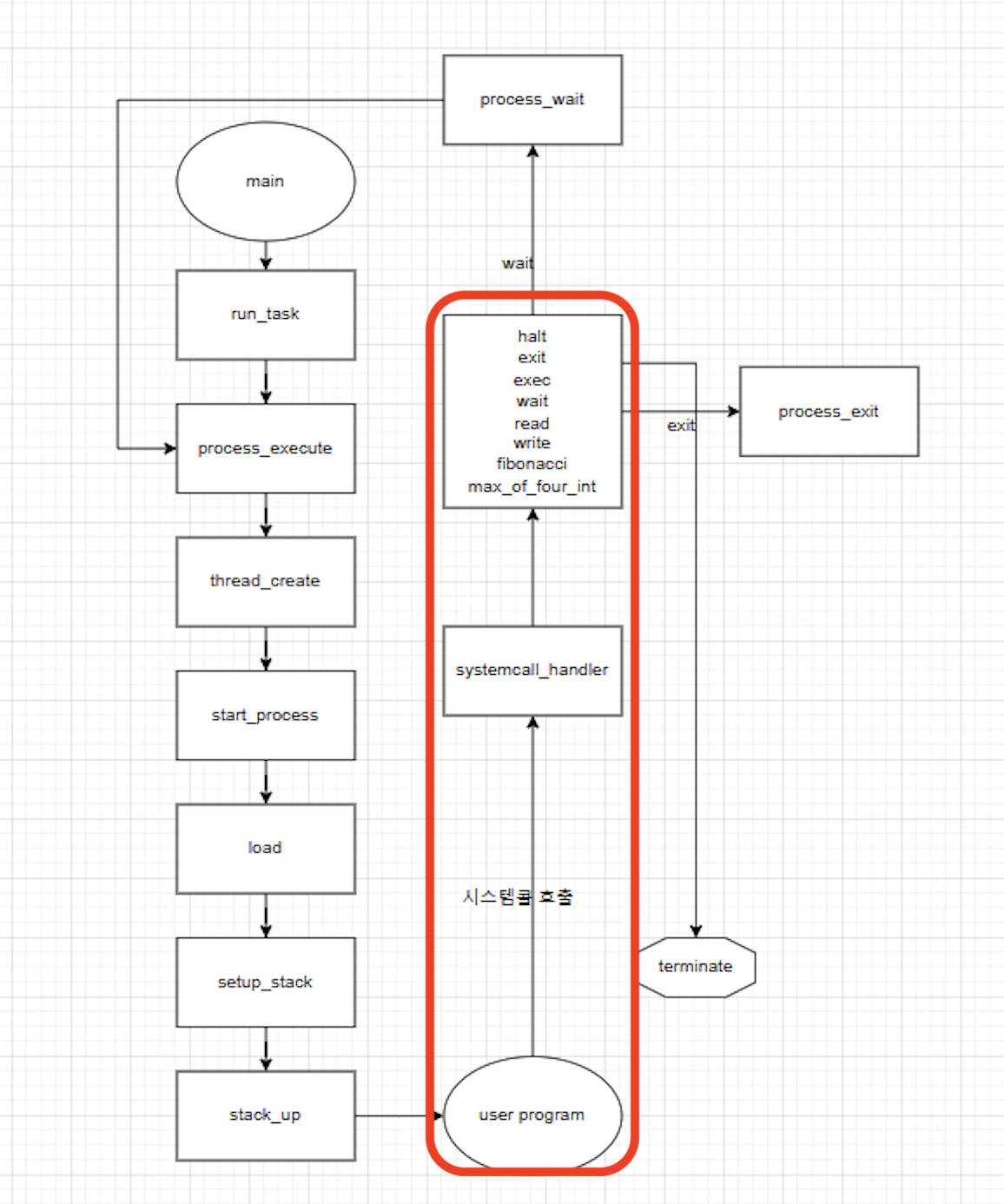
-> 유저의 프로그램이 실행되면 메모리 접근과 같은 시스템에 큰 영향을 끼칠 수 있는 행위들이 필요할 때가 있다. 따라서 대부분의 운영체제에서는 시스템에 악영향을 막고 유저 프로그램이 안전하게 실행될 수 있도록 유저 사이드와 커널 사이드를 나누어 관리한다. 이 때 유저 사이드에서 안전하게 커널의 기능들을 사용하기 위해 필요한 것이 시스템 콜이다. 이번 프로젝트에서 개발할 시스템 콜들은 다음과 같다.  
  
(1) halt : 핀토스 운영체제를 종료시킨다.  
(2) exit : 현재 유저 프로그램을 종료하고 상태를 반환한다.  
(3) exec : 자식 프로세스를 생성하고 해당 id를 리턴한다.  
(4) wait : 자식 프로세스가 종료될 때까지 부모 프로세스가 기다리게 한다.  
(5) read : STDOUT에 출력 하는 기능을 수행한다  
(6) write : STDIN으로부터 입력 받는 기능을 수행한다.  
(7) additional : 첫번째 인자의 fibonacci 수와 네가지 수 중에 가장 큰 수를 반환한다.  
  
유저 레벨에서 시스템 콜을 호출하면 운영체제는 유저 사이드에서 커널 사이드로 제어권을 넘기기 위해 mode bit을 바꾸어 커널 모드로 전환시킨다. 시스템 콜은 운영체제에서 인터럽트의 일종으로 커널 내부의 인터럽트 테이블을 확인해 해당하는 번호를 확인하여 인터럽트 핸들러가 작동되게 한다. 시스템 콜을 담당하는 인터럽트 핸들러가 작동되면 시스템 콜 핸들러가 작동하여 시스템 콜 번호를 확인해 해당하는 시스템 콜에 대한 처리를 한 후 다시 mode bit을 바꾸어 유저 사이드로 복귀한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**9.28 ~ 9.29 : 과제 설명 동영상 시청 후 배경 지식 공부  
      9.30 ~ 10.3 : 개발 시작 후 Arugment Passing / System call Handler 구현  
      10.4 ~ 10.5 : invalid memory access 처리 및 additional system call 구현  
      10.5 ~ 10.6 : 보고서 작성
   2. **개발 방법**  
      **Argument Passing**프로그램에서 실행될 때 많은 부분을 담당하는 process.c에서 이루어진다. 파싱을 위하여 strtok\_r 함수를 주로 사용해 토큰화를 시킨다. 이 때 load 함수에서 파일을 실행할 때와 start\_process로 넘어가 쓰레드의 이름이 파라미터로 넘어온 file\_name으로 그대로 넘어가는 문제를 해결하기 위해 일차적으로 파일을 의미하는 첫 번째 토큰을 분리하여 command\_name으로 처리한다. file\_name은 const로 관리되는 함수도 있고 아닌 함수도 있기 때문에 통일성을 위하여 파일 이름을 의미하는 첫번 째 토큰은 command\_name으로 한다. 공백을 기준으로 파싱된 토큰들은 스택 초기화 후 esp stack pointer를 이용하여 각각 순서에 맞게 저장한다.  
        
      **User Memory Access**syscall.c 파일에서 유저 사이드에서 메모리에 접근할 때 invald한지 검사하기 위해 핀토스에서 제공하는 is\_user\_vadder 함수를 감싸는 wrapper 함수인 checking\_valid\_address 함수를 따로 정의하여 시스템 콜 핸들러가 호출될 때 invalid하면 종료되도록 한다. 또한 이 null인지도 확인한다. exception.c 파일에서도 page\_fault 함수에서 is\_kernel\_vadder 함수를 추가하여 fault\_addr를 받아 예외처리하는 코드를 추가한다.  
        
      **System Calls**syscall.c에서 시스템 콜 인터럽트 발생 시 실행되는 시스템 콜 핸들러를 구현한다. 시스템 콜 번호들은 각각 syscall-nr.h 에서 ENUM으로 관리되고 해당하는 번호를 switch문으로 구분하여 처리한다. 시스템 콜 번호는 파라미터로 들어온 intr\_frame\* f의 esp를 참조해 확인할 수 있다. esp에서 워드 사이즈인 4를 필요한만큼 더하여 스택에 저장되어 있는 토큰들을 읽어와서 해당하는 처리를 진행한다. 명령 수행시 결과값이 반환된다면 f의 eax에 저장한다.  
        
      **Fibonnaci & Max\_of\_four\_int**새로운 시스템 콜을 추가하기 위해서 위의 상황에서는 변경하지 않았던 lib/user/syscall.c 와 lib/user/syscall.h 파일을 변경해야 한다. 시스템 콜 함수를 작성하기 위해 Fibonnaci는 기존의 syscall1을 사용하면 되지만 max\_of\_four\_int는 넘어오는 파라미터가 4개이기 때문에 syscall4라는 새로운 함수를 작성해야 한다. 또한 syscall-nr.h 파일의 SYSNUM에 새로운 시스템 콜을 위한 번호를 할당한다. 그 후 userprog/syscall.c에서 새로운 함수들을 작성하는데 이 때 반환값이 발생하기 때문에 eax에 넣어준다. example에서 additional.c을 만들어 additional 파일로 실행할 수 있게 한다.

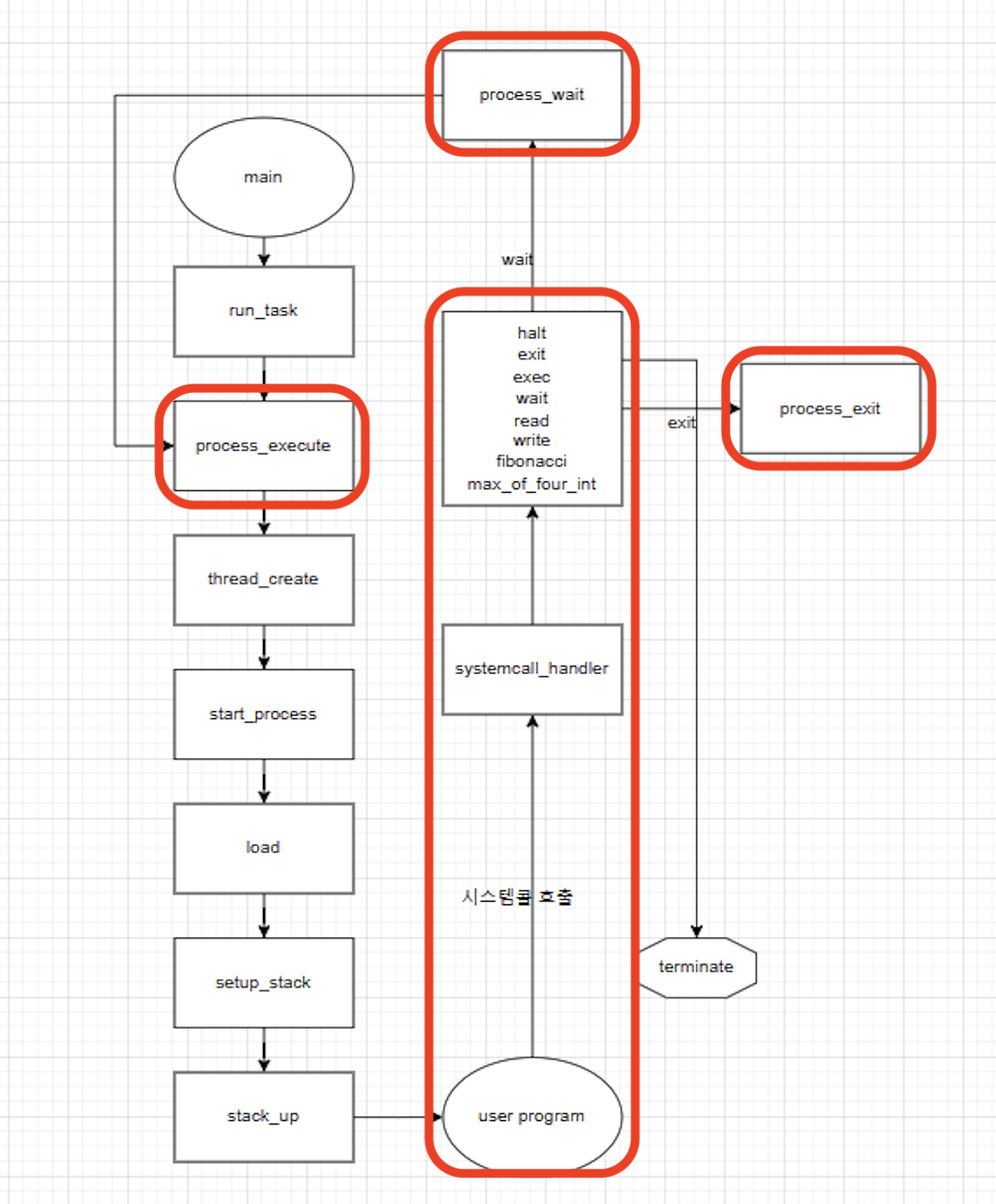
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing



1. User Memory Access

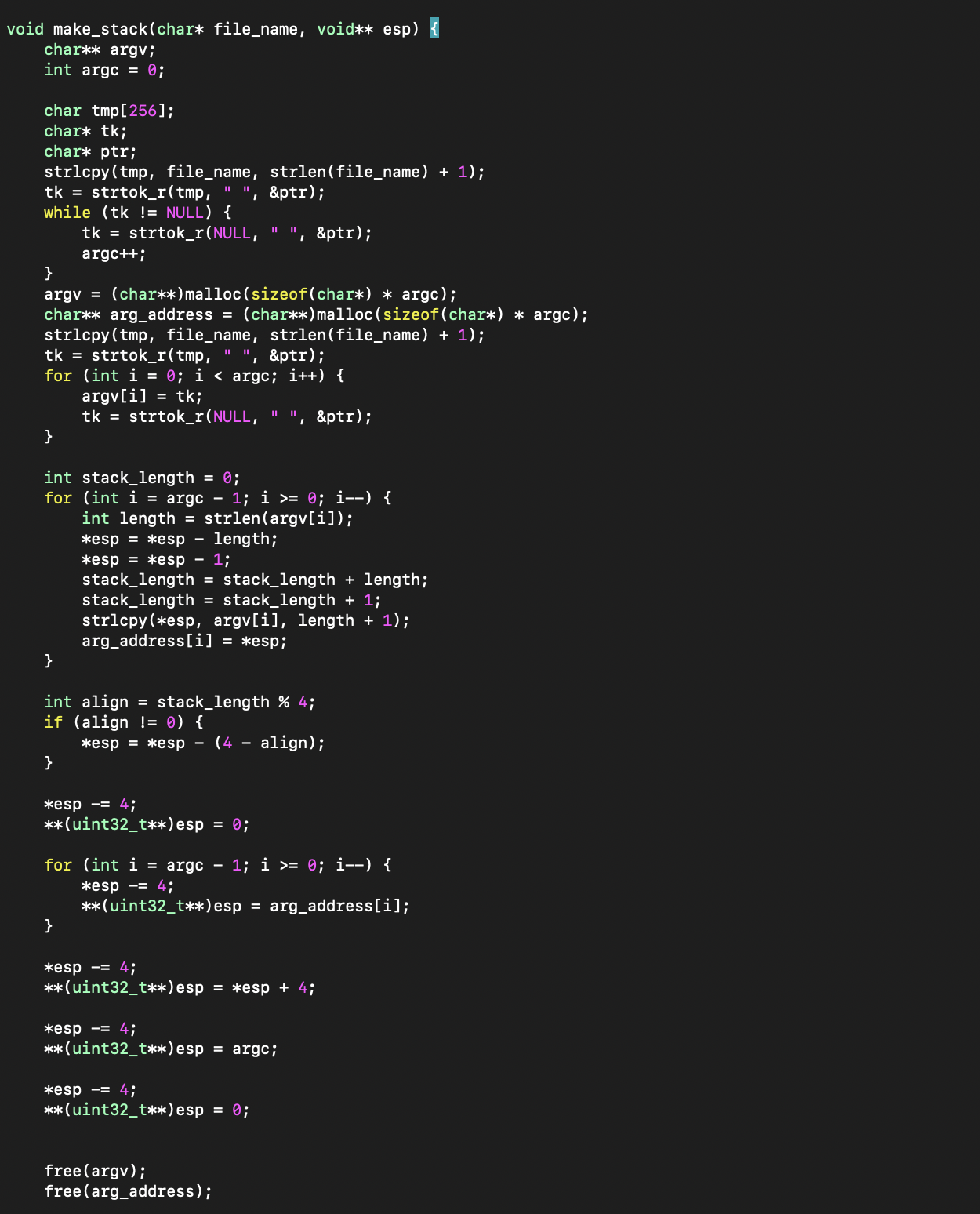


1. System Calls

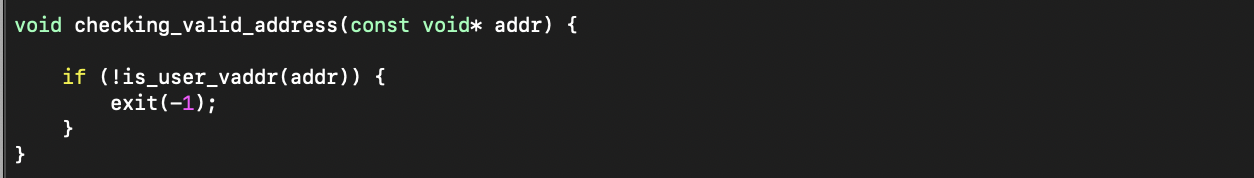


* 1. **제작 내용**

**1. Argument Passing**

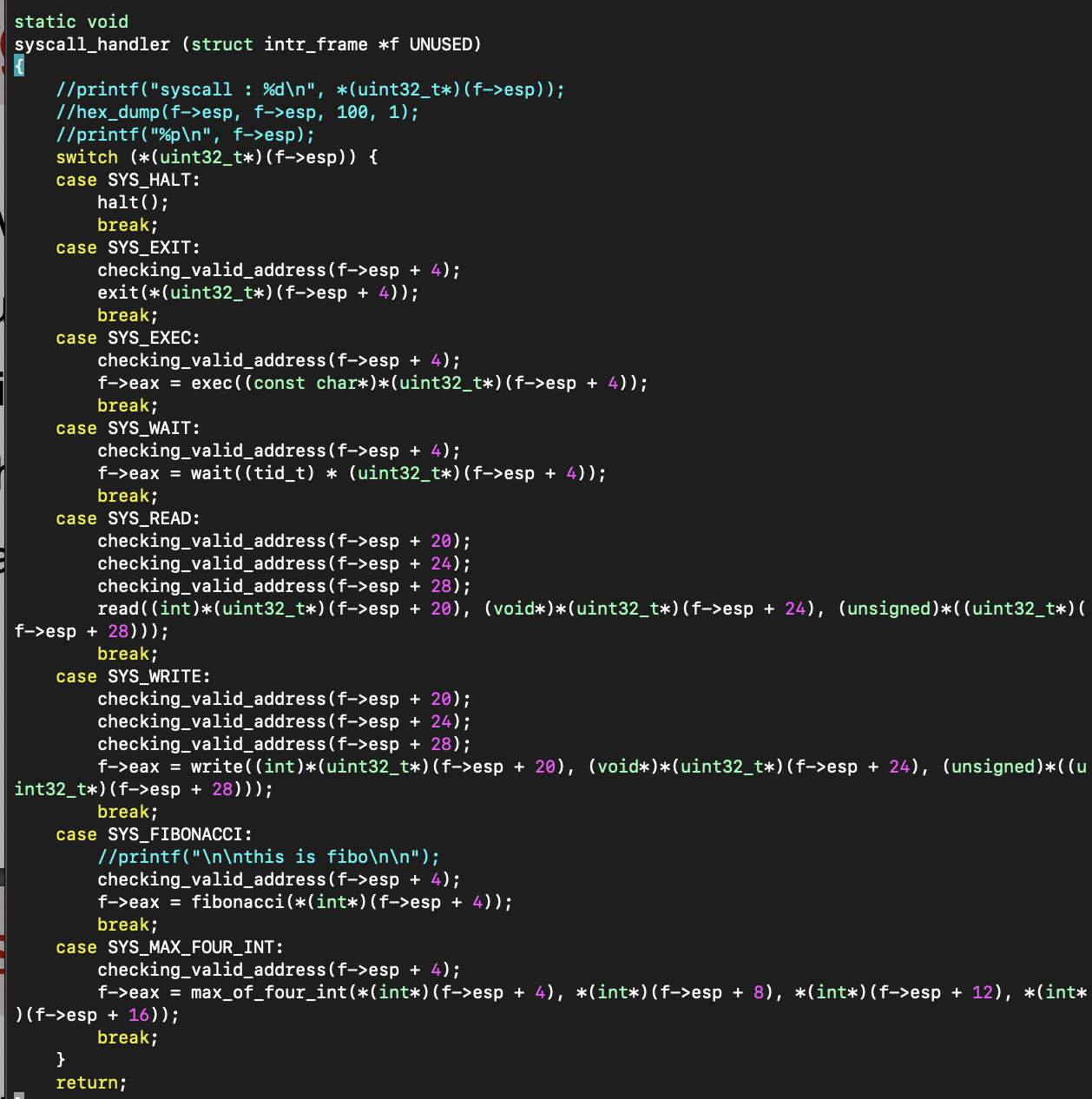
****

이전 계획에서와 마찬가지로 load 함수에서 파일 실행과 thread의 이름이 입력받은 모든 명령어들 그대로 되지 않게 미리 첫번째 토큰만 파싱하여 command\_name으로 처리한다. Argument Passing 파트에서 가장 중요한 역할을 하는 함수인 make\_stack 함수이다. 해당 함수는 전체 명령어를 의미하는 file\_name을 받아와서 strtok\_r 함수를 이용해 먼저 공백 수를 기준으로 몇개인지 파악한 뒤 argc를 초기화한다. 각각 분리하기 위해 토큰들을 저장할 argv와 해당 토큰들의 주소를 저장할 arg\_address를 argc만큼 malloc으로 동적 한후 토큰들과 해당 주소들을 argv와 arg\_address에 저장한다. 그 후엔 매뉴얼에서 제공된 순서대로 먼저 argv를 스택에 esp을 이용해 저장한 후 word alignment를 위해 총 스택 데이터 길이를 저장한 후 해당 길이를 4로 나누어 남은 나머지를 4에서 뺀만큼 esp에서 뺄셈한다. 스택 데이터와 스택 데이터 주소를 저장할 영역을 나누는 것을 의미하는 0을 넣은 후 다시 데이터를 저장할 때와 마찬가지 순서로 arg\_address를 스택에 저장한다. 그 후 arg\_address가 시작하는 주소를 저장한 후 argc, 리턴 주소를 차례로 저장한다.사용했던 argv와 arg\_address를 free로 해제하면 Argument Passing이 끝난다.

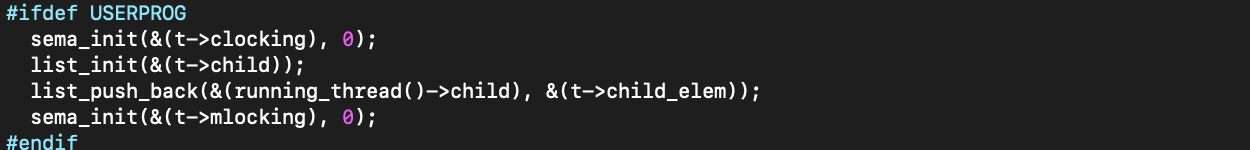
**2. User Memory Access**  
checking\_valid\_address 함수를 선언하여 is\_user\_vaddr을 이용해 해당 주소가 해당 유저의 영역이 아니라면 바로 종료 되도록 예외처리를 진행하였다. 이 함수는 switch문 내부에 있는 모든 시스템 콜에서 호출되어 예외처리를 진행하였다.

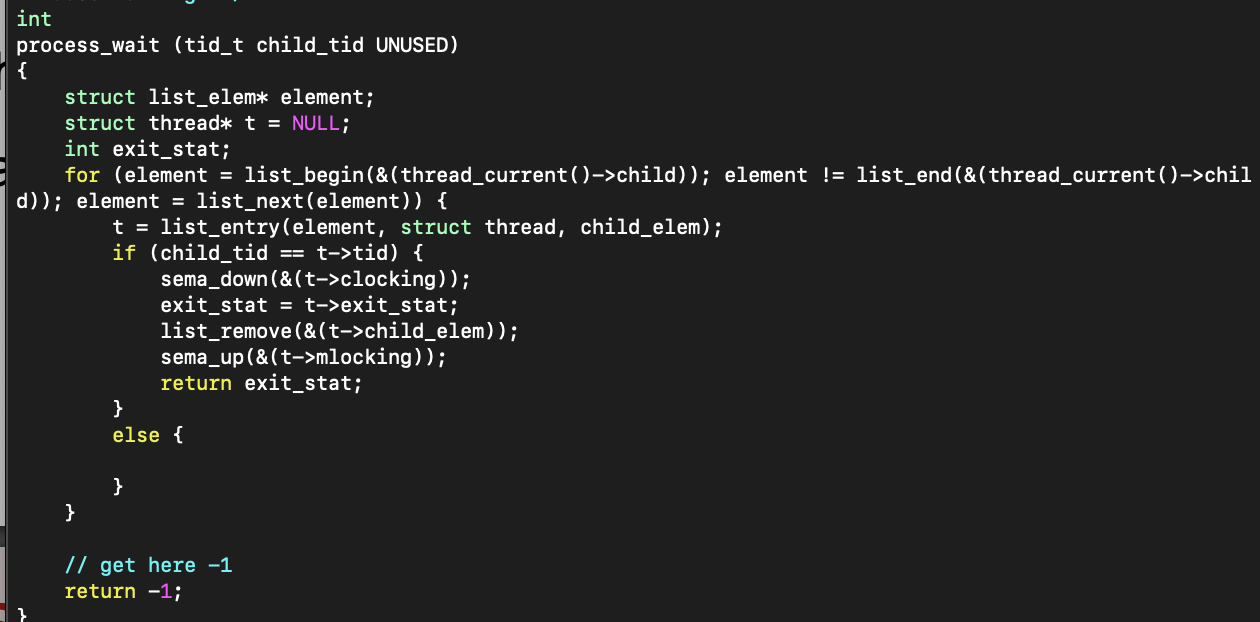
위의 조건문은 userprog/exception.c에 page\_fault 함수에서 존재한다. user가 null인지 체크하고 is\_kernel\_vaddr을 이용해 커널 공간을 접근하려고 하는 게 아닌지 체크하여 맞다면 종료시키는 예외처리를 진행하였다.

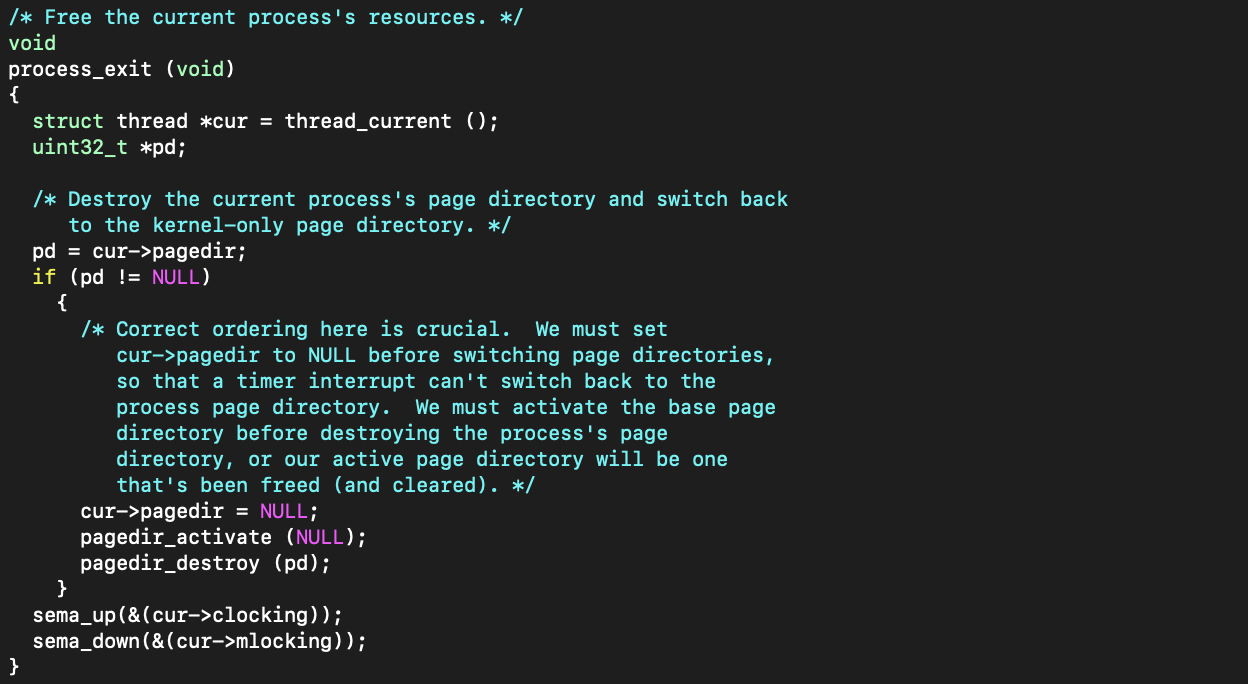
**3. System Calls**



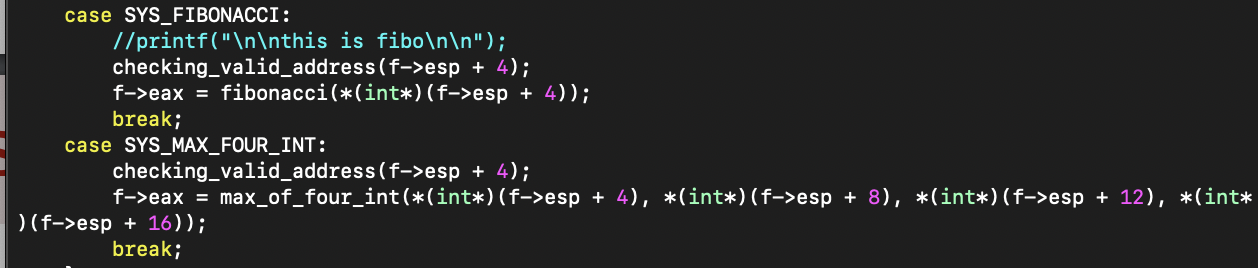
시스템 콜이 발생할 때 해당하는 시스템 콜 번호를 switch 문으로 나누어 처리하는 syscall\_handler 함수이다. f의 esp로 해당 시스템 콜을 확인하고 case로 가서 처리한다. valid한 메모리 접근인지 체크한 후 esp에 워드 사이즈만큼 더하여 해당하는 위치로 가서 인자로 넘긴다. SYS\_READ와 SYS\_WRITE는 다른 시스템 콜과 달리 esp에 4가 아닌 20을 더하여 시작하는 것을 볼 수 있는데 이는 hexa\_dump로 확인해보니 해당 위치로부터 fd가 시작하여 20으로 더하니 정상적으로 입출력되는 것을 확인할 수 있었다.

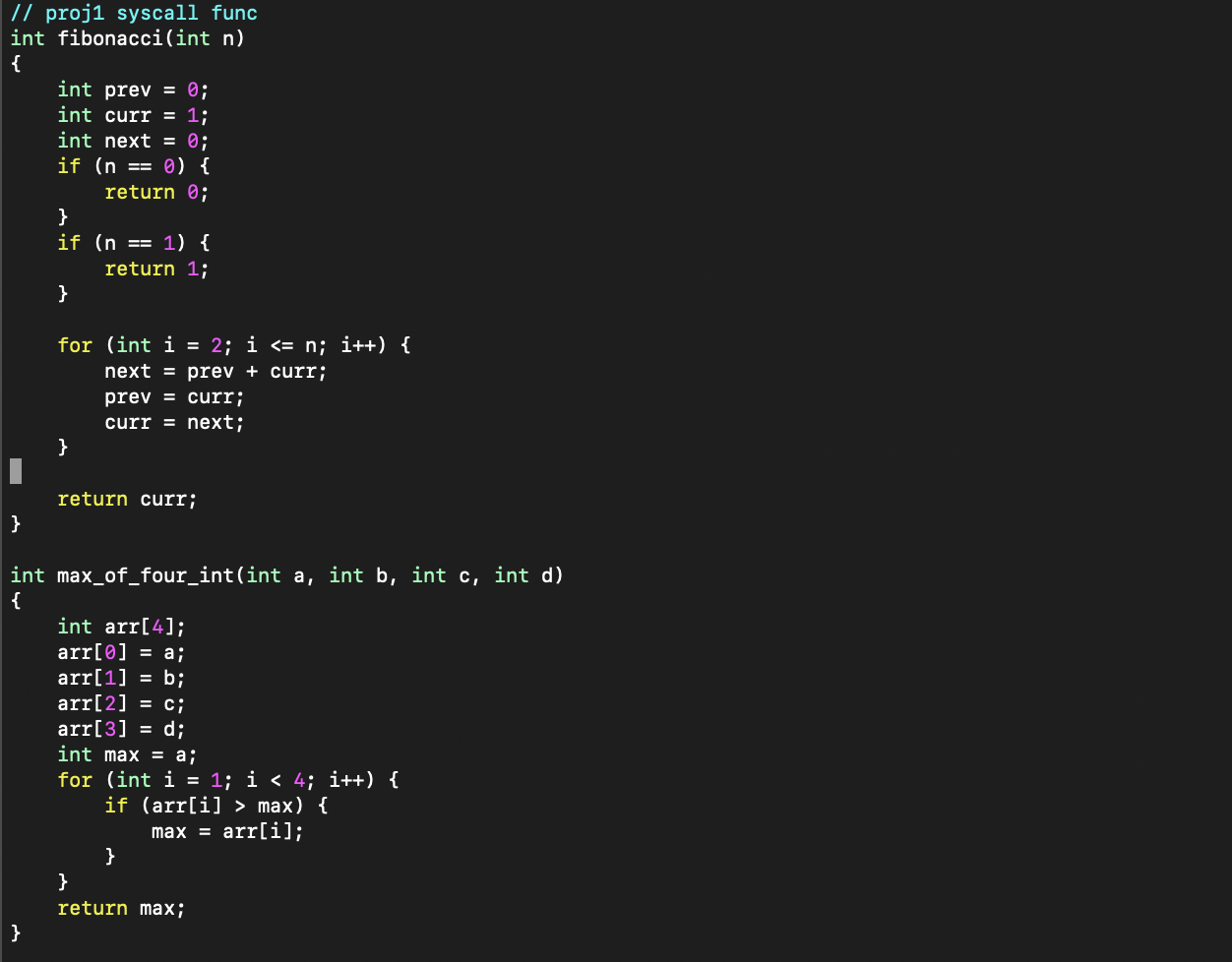
위의 첫번째 코드는 thread.h 함수에 thread 구조체에 추가한 코드이다. clocking과 mlocking은 각각 자식과 메모리에 대한 lock을 걸고 해제하는 semaphore 변수이다. child는 자식 스레드를 의미하고 child\_elem은 자식들의 elem을 의미한다. exit\_stat은 종료 당시 상태를 저장하는 변수를 의미한다. 두번째 코드는 새로 정의한 변수들을 위해 init 작업을 itit\_thread 함수에서 수행한다. 모든 변수들을 초기화하고 child\_elem은 running\_thread의 자식 리스트에 넣어주었다.

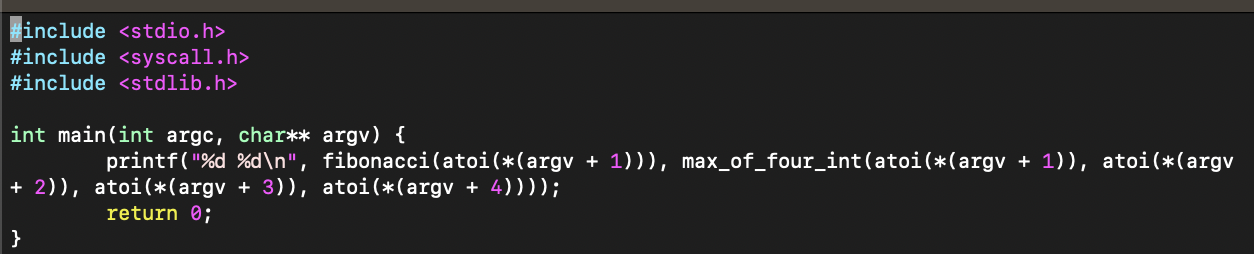
쓰레드에 대한 구조체 변수와 각 변수들의 추가 및 초기화 작업은 프로세스가 wait, exit할 수 있게 하기 위한 작업이다. 위의 process\_wait을 새로 구현하였는데 이는 부모 프로세스가 자식 프로세스가 종료될 때까지 기다렸다가 종료 상태인 exit\_stat을 받는다. 따라서 반복문으로 스레드의 자식 리스트를 돌며 clocking 의 sema\_down으로 자식 프로세스를 기다리고 mlocking 으로 메모리를 남겨둔다.

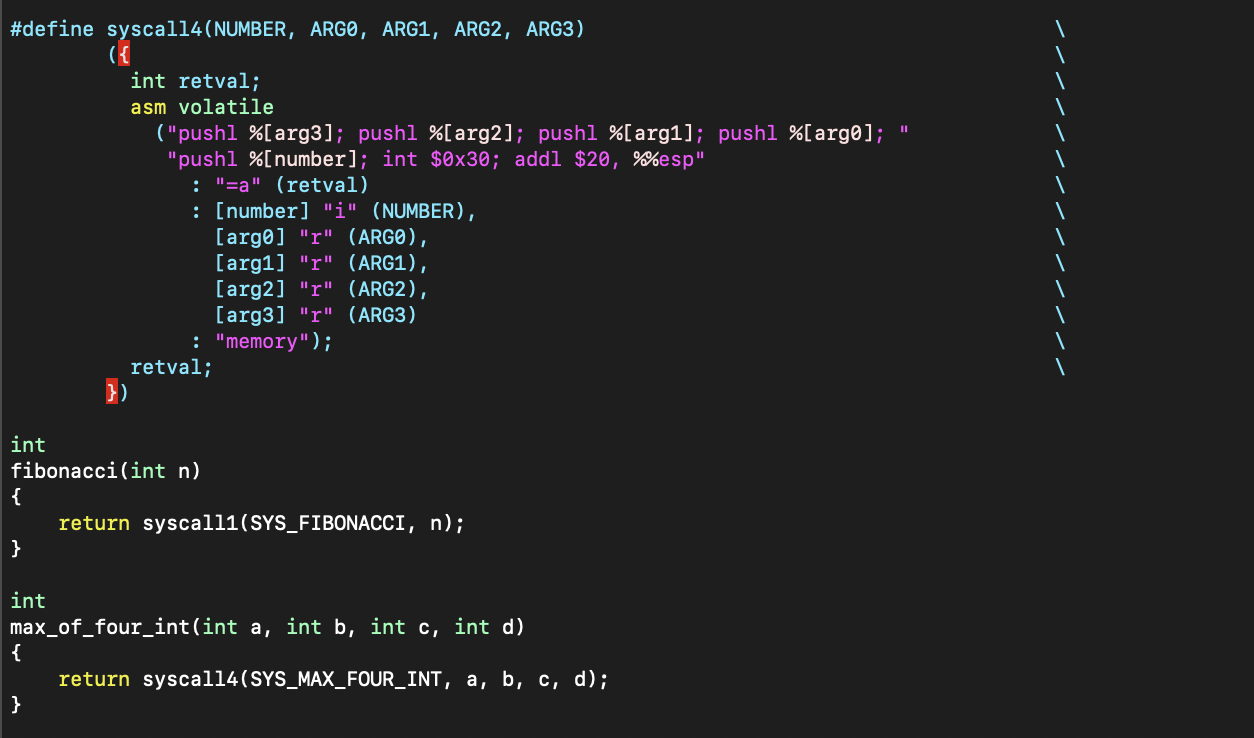
위는 process\_exit 함수로 밑에 sema\_up과 sema\_down만 추가하였다. exit할 때는 wait할 때와 반대로 clocking을 sema\_up 걸어두고 mlocking을 sema\_down한다.

**4.** Additional System calls



위의 코드들은 syscall.c 내부에 존재하고 시스템 콜 SYS\_FIBONACCI 와 SYS\_MAX\_FOUR\_INT 번호가 발생하면 case로 들어가 해당 함수가 호출된다. 여기서 함수의 파라미터로 정수형으로 형변환하여 받아서 계산하기 편리하도록 하였다.

위의 코드는 examples 파일에 존재하는 additional 파일을 위한 소스 코드이다.

lib/user/syscall.c에 만든 syscall4 함수이다. fibonacci함수는 argument가 1개로 기존에 존재하는 syscall1을 호출하지만 max\_of\_four\_int는 4개로 새로운 시스템 콜 함수를 정의하였다. 살펴보면 arg를 4개 pushl하는 것을 알 수 있다.

* 1. **시험 및 평가 내용**첫번 째는 pintos/src/userprog 에서 makc check한 결과이다. 명세서에 나와 있는 21가지 항목 모두 pass 한 것을 볼 수 있다.   
       
     두번 째는 additional을 수행한 결과이다. 결과가 55 62로 잘 나오고 있는 것을 확인할 수 있다.

