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

담당 교수 : 김영재 교수님

조 / 조원 : 20181618 김하늘

개발 기간 : 10-20 ~ 11-1

1. **개발 목표**

* 교육용 OS인 Pintos의 userprogram 기능을 구현하면서 OS가 작동하는 과정을 이해하고, system call 구현 과정에서 Synchronization과 같은 이론을 실제로 적용시켜본다.

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

- process 실행 시, 알맞은 argument를 건네주기 위해 입력받은 명령어를 토큰화 하여 stack에 쌓는다. 이는 esp라는 포인터를 이용하여 high address부터 low address 방향으로 접근하여 저장한다.

1. User Memory Access

- user process가 자신의 메모리 영역 밖을 접근하려 하거나 kernel 메모리 영역에 접근하려 하거나 할당되지 않은 메모리 영역을 접근하려 할 경우 차단되어 process가 종료되도록 구현한다.

1. System Calls

- user process가 IO와 같은 권한 밖의 일을 하려면 system call을 통해서 kernel이 대신 처리하도록 해야한다. 이번 프로젝트에서 구현한 system call은 부모 프로세스가 자식 프로세스에 대해 wait()을 호출시켜 자식 프로세스가 종료될때까지 기다리는 wait, 현재 프로세스를 종료시켜 status를 kernel로 리턴하는 exit, 스택에 쌓은 argument를 건네주어 process가 실행파일을 돌려 새로운 자식 프로세스가 수행하는 exec

* 1. **개발 내용**
* Argument Passing

프로세스 실행 시 해당 프로세스에 argument를 건네줘야한다. 이 argument는 스택에 저장되어서 해당 프로세스에 전달되는데, esp라는 포인터를 이용하여 구현한다. 각 argument를 토큰화하여 가장 높은 address로부터 낮은 address 방향으로 쌓이며, 4 \* 16bit의 블럭 단위로 저장되도록 조정된다.

* User Memory Access

사용자가 자신의 메모리 영역 밖을 접근하려 하거나, kernel 메모리 영역에 접근하려 하거나, 할당되지 않은 메모리에 접근하려 할 경우 차단되어야한다. 이러한 접근을 invalid memory access라고 하며, 이를 막기 위해 page dirtectory를 이용하여 유저 프로세스가 접근하려는 메모리 영역에 대한 page가 해당 프로세스에 할당되었는지 확인한다.

* System Calls

OS는 프로세스 수행을 위한 쓰레드를 작동시키는 영역이 두가지로 나뉜다. 실제 사용자가 접근하여 명령어를 조정하는 user 영역과 user 영역에서 해결할 수 없는 수행을 하는 kernel 영역이다. 시스템 콜은 이 두가지 영역을 이어주는 역할로, user program이 OS를 사용할 수 있도록 하는 기능이다. 시스템콜에서 system call API가 호출되면 arguement를 스택에 쌓고, system call handler에서 해당 스택을 이용해 명령어를 수행한다.

(1) Halt

핀토스를 종료하는 시스템콜로, 현재 쓰레드의 활동을 종료시킨다. Shutdown\_power\_off() 명령어를 통해 구현할 수 있다.

(2) Exit

현재 프로세스를 종료하는 시스템콜로, exit status를 현재 프로세의 thread에 저장한 뒤 thread\_eixt()을 호출하여 process가 종료되도록 구현한다.

(3) Exec

주어진 argument를 건네주고 실행파일을 돌려 새 자식 프로세스를 실행시키는 시스템 콜이다. 자식 프로세스가 완료될 때 까지 부모 프로세스는 기다리는 기능도 필요하다.

(4) Wait

현재 프로세스의 작동을 기다리게 하는 시스템 콜이다. 자식 프로세스가 실행되고 있는 동안 부모 프로세스가 사용하는 기능이다.

(5) Read

read 처리를 수행하는 시스템 콜로, 이 프로젝트에서는 콘솔에 입력하는 기능만을 구현하였다. input\_getc()를 사용한다.

(6) Write

write 처리를 수행하느 시스템 콜로, read와 마찬가지로 콘솔에 출력하는 기능만 구현하였다. purbuf() 함수를 사용한다.

(7) addritional

fibonacci 수열의 값을 return하는 fibonacc()함수와 4개의 값 중 최대값을 return하는 mas\_of\_four\_int()함수를 구현하여 시스템 콜에서 사용한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

2020.10.20 ~ 2020.10.21 핀토스 메뉴얼 정독 후 구현 구상

2020.10.22 ~ 2020.10.23 argument passing 및 system call 일부 구현

2020.10.24 ~ 2020.10.30 user access memory 처리 및 나머지 system call 구현

2020.10.31 ~ 2020.11.01 보고서 작성

* 1. **개발 방법**
* Argument Passing

프로세스를 실행할 때 argument를 건네주기 위해 process\_start -> load 의 함수를 거쳐 수행된다. 이때 process.c파일의 load함수의 void \*\*exp 포인터를 이용해 argument 정보가 쌓인 스택의 포인터를 제공한다. 이 스택에 쌓기 위해선 우선 하나의 문자열 형태로 들어온 명령어들을 토큰화한다. 이를 높은 주소에서부터 저장시키고, 4 \* 16bit의 블럭 단위로 맞춰지도록 0을 추가한다. 해당 argument가 저장된 주소값을 따라 저장하고 argument의 개수를 저장한 뒤 return address 0을 추가하여 종료한다.

* User Memory Access

현재 메모리의 접근이 적절한지 체크하기 위해서 syscall.c 파일에 check\_address() 함수를 추가했다. is\_user\_vaddr()함수를 이용해 사용자가 적절한 주소에 접근하려는지 검사하고, 또는 유저 프로세스가 접근할 수 있는 최대 영역인 PHYS\_BASE를 넘어서는지 확인하고, pagedir.c에 존재하는 pagedir\_get\_page() 함수를 이용하여 메모리 영역에 대한 page가 해당 프로세스에 적절히 할당되었는지 확인한다.

* System Calls

시스템 콜이 불릴때 system\_handler를 호출한다. 이 함수 안에서 각각의 시스템 콜을 다루며, 이때 system call number와 argument는 struct intr\_fram \*f를 통해 알 수 있다. f->esp에 해당 정보들이 스택 구조로 저장되어 있으며, 이를 이용해 system call number를 기준으로 switch문을 작성하여 syscall.c에 구현하였다. 각 시스템 콜을 실행하기 전, 정상적인 메모리 접근인지 확인하는 작업을 매번 수행하였다.

(1) Halt

Shutdown\_power\_off() 함수를 호출하여 간단히 구현하였다.

(2) Exit

현재 프로세스에 대한 exit status를 파라미터로 넘겨주어 exit에 대한 정보를 출력한 뒤 thread\_exit()을 호출하여 종료시킨다.

(3) Exec

자식 프로세스가 완료될 때 까지 부모 프로세스는 기다리기 위해 wait 기능을 사용하고, process\_execute()를 호출한다.

(4) Wait

우선 process.c의 process\_wait()함수에 아무 것도 작성되어있지 않았으므로 해당 부분을 작성하였다. wait 상태를 유지하기 위해서 semaphore를 사용하였고, 이를 위해 thread.h 파일을 수정하여 thread의 구조체에 semaphore 변수를 추가하였다. 그리고 waiting list를 위한 list를 추가하였고 init\_thread()에 추가한 변수들을 초기화하였다. 한 프로세스가 wait을 호출하여 자식 프로세스가 종료되기까지 기다리는 동안, list에 노드가 있을 경우 유지하고 비게 되면 sema\_up을 이용해 깨운다. list에 있었던 child node는 삭제한 뒤 lock을 해제시킨다..

(5) Read

syscall.c에서만 수정하였으며, input\_getc()를 이용해 입력받은 명령어를 버퍼에 저장했다.

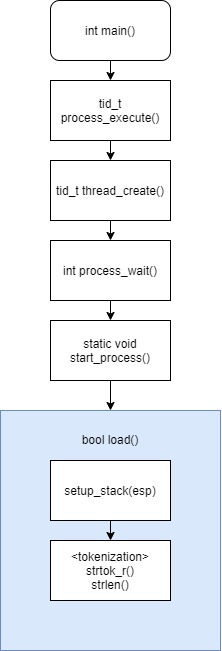
(6) Write

read와 마찬가지로 syscall.c에서만 수정하였으며 putbuf()함수를 사용하여 구현하였다.

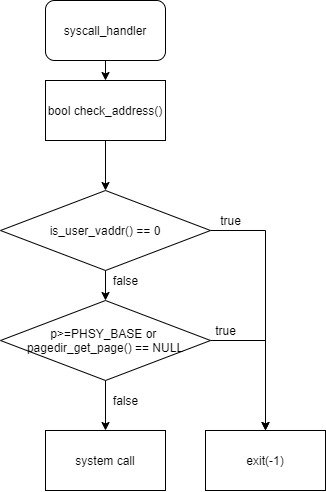
(7) addritional

syscall-nr.h에 SYS\_FIBO, SYS\_MOFI를 추가하여 넘버를 할당하고 4개의 argument를 처리하는 API를 추가한 뒤 syscall.c에 구현한 fibonacci(), max\_of\_four\_int()함수를 호출하여 해당 시스템 콜이 수행되도록 처리하였다.

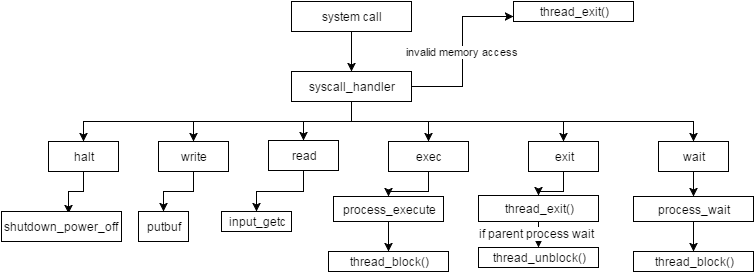
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing

****

1. User Memory Access



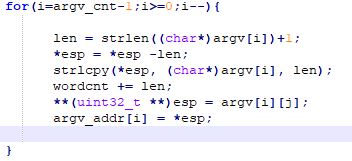
1. System Calls



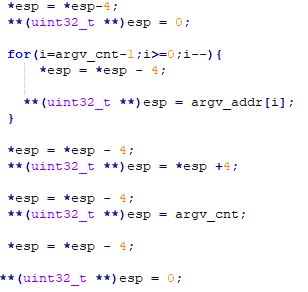
* 1. **제작 내용**

1. Argument Passing

들어온 명령어를 토큰화 하는 것은 strtok\_r()을 사용하였다. 우선 esp에 스택을 셋업하고, 각각의 argument의 길이만큼 esp를 이동시켜 저장한다. 각 argument의 address또한 여기서 함께 저장해 둔다.

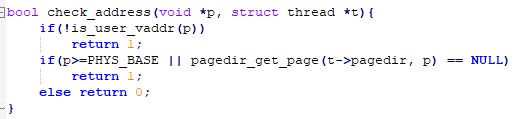


이후 4 \* 16 bit 블럭의 형태가 되도록 0을 적절히 추가하고, argument 쌓는 과정에서 저장한 주소값도 함께 스택에 쌓는다. argument의 갯수를 저장한 뒤 return address 0을 쌓아 종료한다.



1. User Memory Access

접근하려는 메모리가 적절한지 체크하는 함수를 추가하였다. void \*p에는 esp로 접근한 주소값이 들어오며, vaddr.h의 in\_user\_vadd()r, pagedir.c의 pagedir\_get\_page() 함수 및 PHYS\_BASE 값을 통해 적절하지 않은 주소값일 경우 1을 반환하여 exit하도록 구현하였다.



1. System Calls

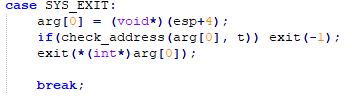
(1) Halt



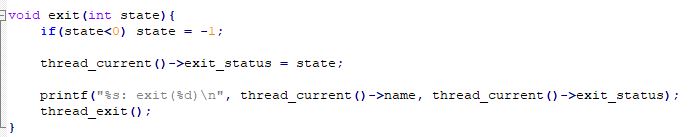
shutdown.c의 shutdown\_power\_off()를 사용하였다. 현재 프로세스를 종료시켜주는 함수이다.

[ error ] 기존에 주어진 syscall.c 코드에는 코드 마지막 부분에 일괄적으로 thread\_exit()을 하는 부분이 있었는데, 이 때문에 정상적으로 프로세스가 종료되지 않았었다. exit 처리는 해당 쓰레드의 기능이 모두 끝난 뒤에 수행해야 한다는 것을 알게 되었다.

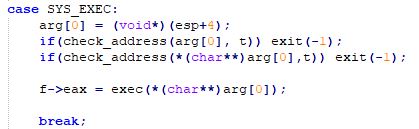
(2) Exit



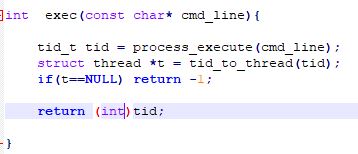
정상적인 메모리 접근으로 exit이 이루어질경우, 해당 arg의 값을 state 파라미터로 넘긴다. thread 구조체에 추가한 exit\_status에 state를 저장하고, exit 결과를 출력한 후 thread\_exit()을 호출하여 종료한다. 비정상적인 접근으로 인한 종료일 경우 state를 -1로 바꾸어 저장한뒤 종료한다.



(3) Exec



비정상적 접근에 대한 결과를 처리한 뒤, exec()함수를 호출하여 return value를 eax에 저장한다.



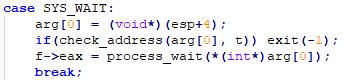
사용자 정의 함수로, process\_execute()의 실행시켜 쓰레드의 id를 반환받아 해당 쓰레드에 접근한다. 자식 프로세스의 실행이 끝날 동안 부모프로세스는 wait 상태가 된다.

[ error ] exec-missing을 해결하던 도중 찾은 부분이다. exec을 실행하면 thread create 이후 부모 프로세스는 wait상태가 되어있어야 한다. 그런데 자식 쓰레드가 끝나기 전에 부모 프로세스가 process execute()내에서 계속해서 실행되는 경우가 생겼었다. 때문에 해당 함수에

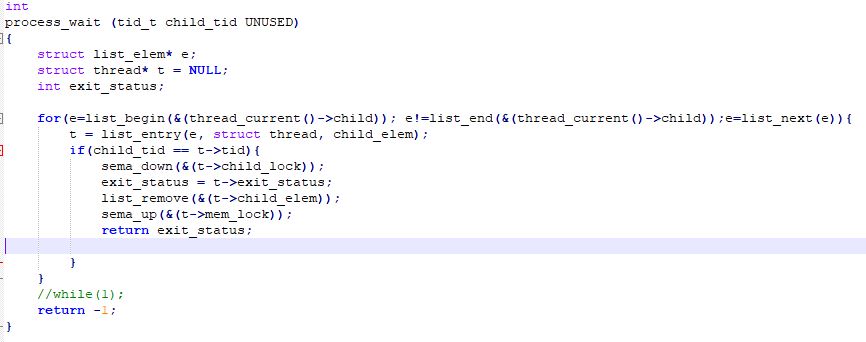
if(filesys\_open(fname) == NULL) return -1;

을 추가하여 실행시 오류의 케이스를 처리하였다.

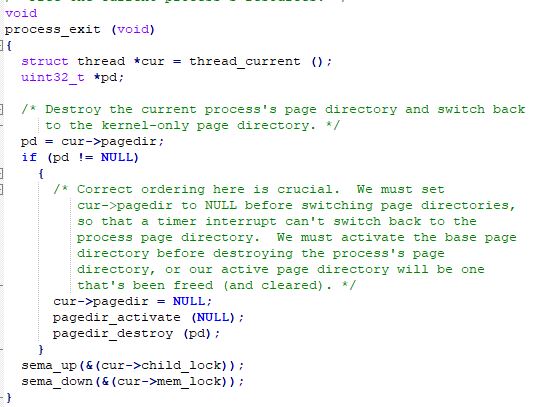
(4) Wait



비정상적 접근에 대한 결과를 처리한 뒤, process\_wait()함수를 호출한다.

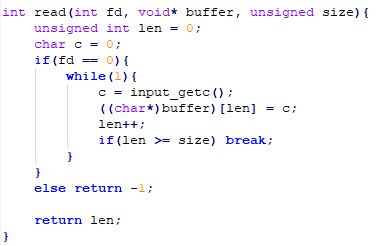
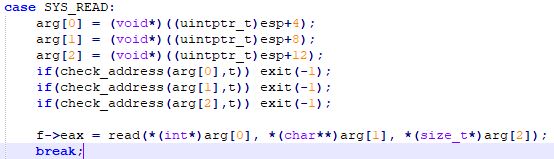


wait함수의 구현은 위와 같다. 우선 자식 프로세스에 대한 쓰레드 id를 파라미터로 받아 오고, 현재 쓰레드의 list에 하나씩 접근하여 해당 쓰레드의 id와 파라미터로 받은 tid를 비교한다. 같을경우, sema\_down을 통해 lock을 건다. status를 업데이트 한 후 해당 자식 프로세스를 지우는데, 이때 list\_remove가 수행 될 때 까지 자식프로세스에 관한 메모리는 사용가능해야한다. 그 이후 lock을 해제시켜 함수가 종료된다.



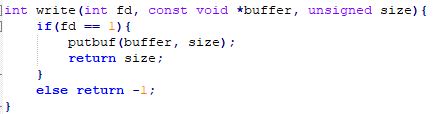
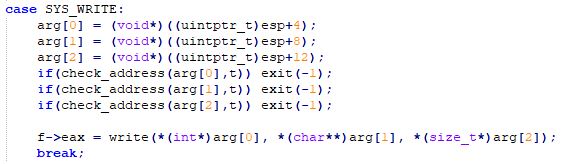
자식 프로세스가 종료될 때 위와 같이 자식에 대한 lock은 풀고, 메모리는 다시 잠금시킨다.

(5) Read



플래그 역할의 fd, 읽은 값이 저장될 buffer, 읽을 메모리의 size가 파라미터로 오는 read함수를 작성하였으며 input.c 파일의 input\_getc()함수를 사용하여 한글자씩 가져와 저장하였다.

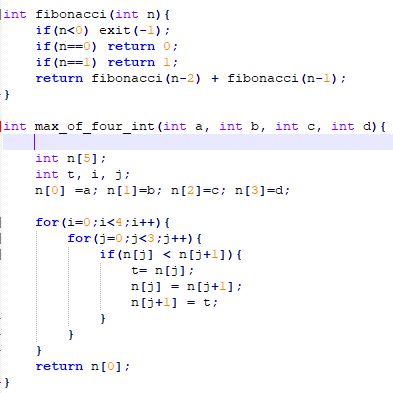
(6) Write



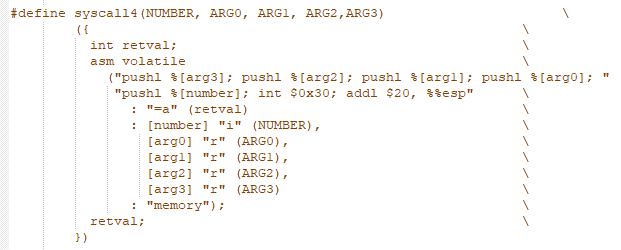
마찬가지로 플래그 역할의 fd와 읽은 값이 저장된 buffer, 읽은 메모리의 size를 파라미터로 가져오는 write함수를 작성하였고, consol.c 파일의 putbuf()함수를 사용하여 저장하였다.

1. Additional System calls

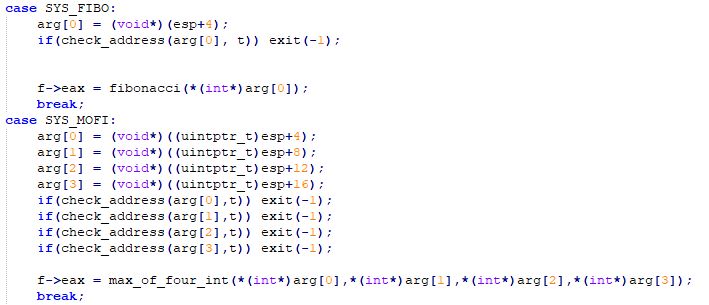
우선 syscall.c에 fibonacci()함수와 max\_of\_four\_int()함수를 작성하였다.



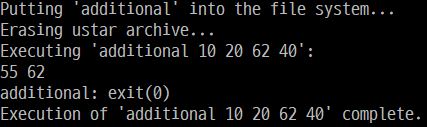
그리고 lib/user/syscall.h에 선언을 해주었고, lib/syscall-nr.h에는 SYS\_FIBO, SYS\_MOFI을 추가하여 이후 system call number에대한 숫자를 넘버링해주었다.

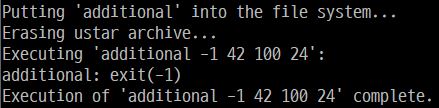


lib/user/syscall.c에는 4개의 argument를 받는 API를 작성하였으며, 이를 통해 max\_of\_four\_int()를 system call을 이용해 부를 수 있도록 해주었다.



* 1. **시험 및 평가 내용**

****

****

**음수가 피보나치 값에 들어올 경우 해당 프로세스를 종료시킨다.**