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

담당 김영재 교수 :

조 / 조원 :

개발 기간 :

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**
* Argument Passing : 스페이스를 기준으로 Arguments를 파싱하고, 80x86 Calling Convention 에 의거하여 스택에 각각의 인자들을 쌓아준다.
* User Memory Access : 접근하고자 하는 주소가 접근이 제한된 영역의 주소인지 확인하는 함수 구현
* System Call Handler : 시스템 콜 요청이 들어왔을 때, 해당 시스템 콜 넘버에 맞는 작업을 수행하도록 구현
* System Call Implementation : write(), read(), halt(), exit(), exec(), wait() 시스템 콜을 구현
* Additional Implementation : fibonacci(), max\_of\_four\_int() 시스템 콜을 추가적으로 구현

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. **Argument Passing**

구현이 잘 되었다면,

* + 1. 공백을 기준으로 인자들이 잘 파싱되었다.
    2. 프로세스가 시작되기 전에 스택에 80x86 Calling Convention에 맞게 인자들이 잘 들어가있다.

이 두 가지가 잘 되어야 한다. 이를 확인하기 위해 hex\_dump 라는 디버깅 툴로 원하는 주소에 인자들이 파싱되어 잘 들어 갔는지 출력해보면 된다.

1. **User Memory Access**

User Memory Access 의 구현이 잘 되었다면, 유저가 PHYS\_BASE 이상의 주소에 접근하려고 하는지 예외 처리해주어야 하고, 만약 유저 가상 주소에 접근한다고 하더라도 page directory에서 물리적인 주소와 잘 매핑이 되었는지 확인해주어야 한다.

궁극적으로 접근하려는 주소의 validity를 잘 체크하고 예외처리하는 것이 User Memory Access를 구현하였을 때의 결과이다.

1. **System Calls**

System Calls 의 구현 결과는 유저에서 요청한 시스템 콜 요청을 커널 단에서 잘 처리하는 것이다. 예를 들어, write() 라는 시스템 콜이 요청으로 들어왔을 때 fd 값이 1 이라면, stdout에 정확히 출력해주어야 한다. 이번 과제에서 구현해야 할 halt(), exit(), exec(), read(), write(), wait() 시스템 콜들은 서로 유기적으로 섞여 사용될 수 있기 때문에 올바르게 구현이 되었다면 각각의 기능이 문제없이 처리되어야 한다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* **Argument Passing**
  + **커널 내 스택에 argument를 쌓는 과정 설명**

PHYS\_BASE 의 주소 0xc0000000 을 기준으로 그 미만이 유저 메모리 영역, 그 이상이 커널 메모리 영역으로 나뉘게 되는데, 스택은 아래로 쌓이는 구조이다.

텍스트, 폰트, 화이트, 영수증이(가) 표시된 사진

자동 생성된 설명

핀토스 매뉴얼 문서에서 가져온 사진이다. 이는 /bin/ls -l foo bar 를 파싱한 결과 스택의 모습을 hex\_dump를 통해 찍은 모습이다.

텍스트, 스크린샷, 폰트, 블랙이(가) 표시된 사진

자동 생성된 설명

echo x를 인자로 넘겨주었을 때, 스택에 들어가는 순서는 ‘x\0’ , ‘echo\0’, word align, null sentinel, ‘x\0’ 의 주소, ‘echo\0’ 의 주소, argv 주소, argc 값, fake return address 0 이다.

위의 echo x의 hex\_dump 결과는 80x86 Calling Convention 에 맞게 스택에 잘 들어가있는 상태이다.

* **User Memory Access**
  + **Pintos 상에서의 invalid memory access 개념을 간략히 설명**

Pintos 상에서의 invalid memory access 는 두 가지로 볼 수 있다. 첫째는, PHYS\_BASE 이상의 주소, 즉 커널 영역에 접근하고자 할 때이다. 둘째는, NULL 포인터와 같이 매핑이 되지 않은 가상 메모리(page directory 에)에 접근하고자 할 때이다.

* + **Invalid memory access를 어떻게 막을 것인지 설명**

위 두 가지의 invalid memory access를 막기 위해서는 threads/vaddr.h 에 정의되어 있는 is\_kernel\_vaddr() 함수와 userprog/pagedir.c 에 정의되어 있는 pagedir\_get\_page() 함수를 활용하면 된다. 전자가 커널 영역에 접근하는지를 판단하는 함수이고, 후자가 NULL 포인터와 같이 매핑되지 않은 주소를 처리해주는 함수이다.

이 두 함수를 이용해서 해당 주소가 커널 영역에 접근하는지, 매핑이 되지 않은 주소인지를 판단해서, 맞다면 exit 을 해주는 방식으로 함수를 구현하면 될 것이다.

* **System Calls**
  + **시스템 콜의 필요성에 대한 간략한 설명**

유저가 I/O request 나 CPU나 메모리와 같은 시스템 자원을 사용하는 등 restricted operation 을 하고 싶을 때, CPU의 사용권을 커널에게 넘겨주어야 하고 이를 위해서 System Call 이라는 일종의 API가 필요하다. 결국 유저 모드는 제한된 권한으로 인해 직접적으로 권한이 제한된 작업을 할 수 없고, 커널이 이를 대신해주는데 이러한 transfer를 용이하게 하기 위해서 유저와 커널 사이에 System Call이 존재하는 것이다.

* + **이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)**

1. halt() : halt() 시스템 콜 요청이 들어오게 된다면, shutdown\_power\_off() 함수 호출을 통해 핀토스를 종료시킨다.
2. exit() : exit() 시스템 콜 요청이 들어오게 된다면, 인자로 들어온 상태 코드(exit code)를 커널에 반환하며 현재 유저 프로그램을 종료시켜야 한다. Invalid Memory Access를 하는 경우에도 이 exit() 시스템 콜과 유사하게 프로그램을 종료시킨다.
3. exec() : exec() 시스템 콜 요청이 들어오게 되면, 인자로 넘어온 이름을 가진 자식 프로세스를 실행하며, 리턴 값으로 해당 프로세스의 pid 값을 반환한다. 프로세스를 생성할 수 없는 경우나 비정상적으로 실행될 경우에는 -1을 반환한다.
4. wait() : wait() 시스템 콜 요청이 들어오게 되면, 인자로 들어온 pid 값을 가진 자식 프로세스가 종료되기까지 기다려야 하고, 자식 프로세스의 exit status의 값을 반환해야 한다. 부모 프로세스가 자식 프로세스의 종료를 기다려야 하므로, synchronization 기법이 필요하다.
5. read() : read() 시스템 콜 요청이 들어오게 되면, 인자로 들어온 int fd, void \* buffer, unsigned size 를 고려해서, fd에서 size bytes만큼 buffer에 읽어 들인다.
6. write() : write() 시스템 콜 요청이 들어오게 되면, 인자로 들어온 int fd, void \* buffer, unsigned size 를 고려해서, buffer에서 size bytes만큼 fd에 쓴다.
   * **유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명**
7. 유저 레벨에서 halt() 함수 호출.

**텍스트, 폰트, 친필, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

1. halt() 함수에서 syscall0 호출 / 이 때, halt() 시스템 콜은 인자로 넘겨주는 것이 없기 때문에, syscall0을 부르고, SYS\_HALT라는 시스템 콜 넘버만 넘겨준다.

**텍스트, 폰트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명**

1. syscall0()

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

Caller 스택에 인자로 넘어온 system call number를 push 해주고, 시스템 콜에 대한 interrupt를 발생시킨다. (trap instruction) 여기서 커널 모드로 들어가게 되고, 권한 레벨이 올라가 시스템 콜 요청을 수행할 수 있게 된다. 만약 인자들이 있다면, 인자들을 마지막부터 순서대로 넣어준 후, 시스템 콜을 넣어야 한다.

1. call interrupt handler

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

intr\_handler 함수를 부른다. Intr\_handler 안에서는 IDT, 즉 Interrupt Descriptor Table 에서 각 인터럽트에 맞는 핸들러 함수의 주소를 찾아준다.

1. call system call handler

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Intr\_handler 함수 안에서 intr\_handler 테이블에서 0x30 번지에 가서 시스템 콜 핸들러 함수를 호출.

1. System call handling

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

현재 intr\_frame 안에 있는 esp가 가리키고 있는 주소에는 전에 push 해주었던 시스템 콜 넘버 값이 저장되어 있다. switch 문을 통해서 해당 시스템 콜 넘버에 맞는 시스템 콜을 수행하고, return 해주면 된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**

9월 21일 – Argument Passing 구현 ( Arguments 를 파싱하고, 파싱된 결과물을 80x86 Calling Convention에 맞게 커널 내 스택에 알맞게 적재), hex\_dump를 활용해 결과물 확인 후 제대로 값이 들어갔고, 올바른 주소에 들어갖는지 직접 확인.

9월 28일 – User Memory Access 구현 및 halt() 와 exit() 시스템 콜 구현 / 유저가 접근하는 주소가 Valid 한 주소인지 체크하는 함수를 구현하고, 예외 처리가 잘 되는지 확인한다. 잘못된 접근에 대해서는 exit()을 해주어야 하므로, exit() 시스템 콜까지 구현하여 확인한다.

9월 29일 – wait()를 제외한 나머지 시스템 콜 exec(), read(), write() 구현 / write() 구현을 마치고, echo x가 잘 출력되는지 확인한다. Echo는 stdout에 인자를 그대로 출력하는 역할을 하기 때문에 write() 구현이 정상적으로 되었다면, 화면에서 결과를 확인할 수 있을 것이다.

9월 30일 – wait() 시스템 콜 구현 및 fibonacci, max\_of\_four\_int 시스템 콜 구현 / wait() 구현을 위해서 synchronization 개념 공부 (semaphore, lock 등)

* 1. **개발 방법**
* **II. B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

1. **Argument Passing**

**Arguments Parsing**

process.c 의 load 함수 내에서 Arguments Parsing 구현하게 되는데, 이를 위해서 미리 내장되어 있는 string 라이브러리의 strtok\_r 함수를 쓸 것이다. 추가적으로 원본 문자열을 복사해서 안전하게 사용하기 위해서 strlcpy 함수를 사용할 것이다.

**Kernel stack setup**

process.c 의 load 함수 내에서 커널 스택에 파싱된 인자들을 적재할 것이다. 이를 위해서 memcpy와 memset 내장 함수를 적절히 활용할 것이다. Memcpy와 strlcpy는 비슷한 역할을 수행하는데, strlcpy는 문자열 복사를 위해서 만들어진 문자열 내장 함수이고, memcpy는 메모리 자체에 대한 복사를 위해 만들어졌다. memset은 특정 값으로 해당 주소를 초기화하기 위해서 사용했다.

1. **User Memory Access**

threads/vaddr.h 에 정의되어 있는 is\_kernel\_vaddr() 함수와 userprog/pagedir.c 에 정의되어 있는 pagedir\_get\_page() 함수를 활용하여 유저 메모리 접근 예외 처리를 하는 user\_valid\_addr 함수를 만들 것이다. 전자가 커널 영역에 접근하는지를 판단하는 함수이고, 후자가 NULL 포인터와 같이 매핑되지 않은 주소를 처리해주는 함수이다. user\_valid\_addr 함수는 src/userprog/syscall.c 소스파일에 정의하여 사용할 것이다.

1. **System Calls**

halt() : src/userprog/syscall.c 의 syscall\_handler 함수 내 switch 문안 SYS\_HALT case에서 직접 devices/shutdown.h 안의 shutdown\_power\_off() 함수를 호출한다.

exit() : src/userprog/syscall.c 의 syscall\_handler 함수 내 switch 문안 SYS\_EXIT case에서 직접 src/userprog/process.c 의 process\_exit() 함수를 호출한다.

exec() : src/userprog/syscall.c 의 syscall\_handler 함수 내 switch 문안 SYS\_EXEC case에서 직접 src/userprog/process.c 의 process\_execute() 함수를 호출한다.

wait() : src/userprog/syscall.c 의 syscall\_handler 함수 내 switch 문안 SYS\_WAIT case에서 직접 src/userprog/process.c 의 process\_wait() 함수를 호출한다. process\_wait() 함수의 구현은 threads/sync.c 의 sema\_down 및 sema\_up 함수를 이용해서 구현할 것이다.

read() : src/userprog/syscall.c 의 syscall\_handler 함수 내 switch 문안 SYS\_READ case에서 직접 src/devices/input.c 의 input\_getc() 함수를 호출해서 입력을 받아온다.

write() : src/userprog/syscall.c 의 syscall\_handler 함수 내 switch 문안 SYS\_WRITE case에서 직접 src/lib/kernel/console.c 의 putbuf() 함수를 호출해서 버퍼에 쓴다.

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

텍스트, 친필, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

1. User Memory Access

텍스트, 친필, 폰트, 서예이(가) 표시된 사진

자동 생성된 설명

1. System Calls

텍스트, 스케치, 친필, 도표이(가) 표시된 사진

자동 생성된 설명

* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

1. Argument Passing

Arguments Parsing

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

파싱을 하기 위해서 string.h 에 정의되어 있는 strtok\_r 함수를 이용했다.

file\_name 변수는 인자로 넘어온 아직 파싱이 되지 않은 문자열이고, 원본을 직접 수정하는 것은 자칫 위험할 수 있기에 fn\_copy 변수에 strlcpy 함수를 이용하여 그대로 복제한다.

strtok\_r 함수는 delimiter 를 기준으로 문자열을 자르게 되는데, 우리는 공백을 기준으로 나누기 때문에 두번째 인자로 “ “를 넘겨준다. 처음 호출할 때는 파싱하고자 하는 문자열 전체를 첫번째 인자로 넘겨주고, 세번째 인자로는 다음 파싱할 때 위치를 저장하기 위해서 포인터의 주소를 넘겨준다. 처음 호출은 strtok\_r(“echo x”, “ “, &next) 이런 형태로 호출된다고 보면 된다. 그에 대한 반환값은 “echo\0” 이다. 우리는 echo 라는 프로세스를 띄울 것이므로 이를 new\_fn에 넣어준다. 그리고 나머지 인자들은 while 문을 돌면서 strtok\_r을 호출하여 각각 argv[]에 하나씩 저장한다. 이 때, 후에 스택에 쌓을 때 사용하기 위해 argc 변수를 증가시킨다. 두번째 strtok\_r 부터는 첫번째 인자로 NULL을 넘겨주는데, 이 때 strtok\_r 내부적으로 첫번째인자로 NULL이 넘어오게 되면, next에 저장되어 있던 주소로 처음 파싱을 시작할 위치를 옮겨준다. 처음 호출 이후 next에 저장된 위치는 “echo x”에서 x의 주소이다. 따라서, 두번째 호출부터 x를 파싱하게 되는 것이다.

1. User Memory Access

src/userprog/syscall.c

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

먼저, is\_kernel\_vaddr를 통해서 현재 접근하려는 주소가 커널 영역의 가상 메모리 주소인지 판단하고, 맞으면 thread\_exit()을 호출시킨다. 이 때의 exit\_code는 -1이다.

그 다음에 커널 영역의 주소는 아니라면, pagedir\_get\_page 함수를 호출하여 반환 값이 NULL인지 확인한다. 이 때, NULL이면 실제 물리적인 주소에 매핑되지 않은 주소라는 뜻이므로 똑같이 exit code -1을 가지고, thread\_exit()를 시킨다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

is\_kernel\_vaddr 함수 내부의 동작 방식은 간단하다. 주어진 주소가 PHYS\_BASE 이상이면 true를 아니면 false를 반환한다. 가상 메모리 공간은 PHYS\_BASE 이상은 커널 영역, 미만은 유저 영역으로 나뉜다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

pagedir\_get\_page는 pagedir 의 주소를 첫번째 인자로 받고, 접근하려는 주소를 두번째 인자로 받는다. 현재 쓰레드의 pagedir와 주어진 주소를 가지고, lookup\_page 함수를 통해서 page table entry 주소를 가져온다. 이렇게 가져온 entry 포인터를 통해서 pagedir 에 매핑되지 않은 가상 주소는 NULL을 반환하도록 한다. 따라서, 이를 통해 예외 처리를 해주면 접근하려는 영역이 valid한지 확인할 수 있게 된다.

1. System Calls

* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**

syscall\_handler 함수

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**

syscall\_handler 함수에서 각 시스템 콜 넘버에 대해서 시스템 콜 처리를 해준다.

먼저 user\_valid\_addr(f->esp); 를 통해서 현재 intr\_frame의 esp 값이 valid 한 주소를 가리키고 있는지 확인한다.

현재 f->esp 포인터는 시스템 콜에 해당 하는 시스템 콜 넘버를 가리키고 있다.

텍스트, 폰트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명

switch 문을 사용해서 각 시스템 콜 넘버에 맞게 case 를 나누었다.

halt() 시스템 콜의 경우에는 앞서도 언급하였던 shutdown\_power\_off(); 함수를 호출함으로써 핀토스를 종료시킨다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

exit() 시스템 콜의 경우에는 먼저 첫번째 인자로 status 가 들어오는데, 이 값은 현재 f->esp 주소부터 4바이트 더한 주소에 위치하고 있다. 따라서, 이를 ptr 포인터에 가져와서 valid 한 주소인지 체크하고, 매뉴얼의 exit 출력 포맷과 동일하게, 종료하려는 thread\_name()과 exit status 값을 출력해준 후에 thread\_current()->exit\_code 변수에 해당 status 값을 넣어준다. 그리고 thread\_exit() 함수를 호출하여 프로세스를 종료시킨다. thread\_current() 함수는 현재 실행되고 있는 쓰레드의 포인터, 즉 struct thread \* 를 반환한다. thread 구조체를 보자.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

이 안에는 많은 필드들이 선언되어 있지만, 현재 exit에서 사용되는 필드는 int exit\_code 이다. 현재 쓰레드의 종료 코드를 담고 있는 필드이다.

thread\_exit() 함수는 현재 쓰레드를 현재 모든 쓰레드들이 들어가있는 리스트에서 제거하고, 쓰레드의 상태를 THREAD\_DYING으로 바꿔준 후에 schedule() 함수를 호출해 다음 쓰레드로 리스케쥴링해준다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

exec 시스템 콜의 경우에는 첫번째 인자로 실행할 프로세스의 이름이 char \* 형태로 받아온다. 이 경우도 마찬가지로 주소 validity를 체크한 후에 process\_execute() 함수를 해당 인자를 넘겨주며 실행시킨다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

process\_execute() 함수는 file\_name char 포인터를 변수로 받아와서, 해당 파일 이름을 가진 프로세스를 실행시켜야 한다. 하지만, 프로세스 이름은 예를 들어, ‘echo x’ 라는 file\_name이 들어왔을 때, echo 여야 한다. 즉, process\_execute() 함수 내부에서도 해당 문자열을 파싱해야 한다. 이를 위해서 file\_name을 복사하고, Argument Passing 파트에서 파싱을 진행했던 것처럼 처음 문자열만 가져와서 thread\_create()를 할 때 인자로 넘겨준다. 프로세스가 정상적으로 만들어졌다면, tid 값을 반환하는데, 이 값을 f->eax 에 저장한다. 그 이유는 eax가 함수의 return 값을 저장하는 용도로 설계된 범용 레지스터이기 때문이다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

다음은 wait() 시스템 콜이다. 사실 이 wait() 구현에 가장 오랜 시간을 쏟았다. wait() 시스템콜은 첫번째 인자로 자식 프로세스의 pid를 가진다. 이는 자식 프로세스 중에 해당 id를 가진 프로세스가 종료될 때까지 부모 프로세스는 기다려야 한다는 뜻이다. 이를 구현하기 위해서는 원래 10억번 for loop을 돌도록 구현해놓은 process\_wait를 수정해야 할 필요가 있었다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

process\_wait 함수의 구현은 동기화 기법인 세마포어를 사용해서 구현했다. 앞서 struct thread 구조 스크린샷을 보게 되면 struct semaphore sema 변수와 struct semaphore mem 변수가 있다. 먼저 전자의 sema는 자식 프로세스가 모든 작업을 마치고 종료될 때까지 부모 프로세스에서 기다리는 처리를 하기 위한 세마포어이다. 두 번째, mem 세마포어는 부모 프로세스가 THREAD\_DYING 상태의 자식 프로세스를 리스트로부터 삭제해야 하기 때문에 이 용도로 사용하는 세마포어이다. 왜냐하면, 부모 프로세스가 자식 프로세스를 리스트로부터 삭제하기 전에 자식 프로세스가 종료되어 삭제할 수 없게 되는 사태를 방지하고자 하는 것이다.

이 함수의 구조를 보면 for loop을 통해서 각각 자식 쓰레드를 e에 가져오고, list\_entry라는 매크로 함수를 통해서 자식 쓰레드의 구조체를 받아온다. 이 때, 자식 쓰레드의 tid가 인자로 받아온 tid와 동일하다면 종료될 때까지 기다려야 한다. 이 때, 0으로 초기화된 sema 세마포어를 down 해준다. 그렇게 되면, 부모 프로세스는 멈추게 되고, 자식 프로세스가 모든 일을 마치고, process\_exit()이 될 때 다시 sema\_up이 되면서 부모 프로세스가 실행될 수 있게 된다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

이처럼 프로세스가 종료될 때, sema 세마포어는 up을 mem 세마포어는 역으로 down을 시켜준다. 그 이유는 자식 프로세스가 종료되면서 mem 세마포어를 Down하게 되면, 부모 프로세스가 다시 up을 하기 전까지는 종료하지 않고 기다린다. 즉, 부모 프로세스로 하여금 자신을 list\_remove 하고 나서 sema\_up을 하도록 해준다.

이렇게 동기화 기법을 적용해서 process\_wait 함수를 구현했고, return 값은 종료된 자식 프로세스의 tid가 되겠다. 비정상적인 종료의 경우 모두 -1 을 리턴해야된다.

이 리턴값을 f->eax에 받아오고, return 해준다.

텍스트, 전자제품, 스크린샷, 디스플레이이(가) 표시된 사진

자동 생성된 설명

read() 시스템 호출이다. 먼저, 이번 proj1 에서는 stdin만 고려한다. 이 시스템 콜은 인자를 세 개 받는데, 첫번째인자는 fd 정수 값, 두번째인자는 입력을 저장할 버퍼, 그리고 세번째 인자는 읽어들일 사이즈이다. 고로, len 정수에 세번째 인자값이 저장되어 있는 주소를 참조하여 가져오고, for 문을 돌면서 input\_getc() 함수를 통해서 입력을 받아온다. 그리고 memset을 통해 버퍼에 해당 char을 1바이트 읽어들인다.

리턴 값은 성공적으로 읽어들인 바이트 수이므로, f->eax 에 입력이 ‘\0’일 경우에 i 값을 넣어주고 return 하고 만약 fd가 0이 아닌 경우에는 -1을 넣어주고 리턴한다.

텍스트, 스크린샷, 디스플레이, 폰트이(가) 표시된 사진

자동 생성된 설명

마지막 write 함수이다. Write 시스템 콜도 인자로 세 개를 받아오는데, 첫번째 인자는 fd 값, 두번째 인자는 char \* 버퍼, 그리고 세번째는 역시 size이다. Putbuf 함수를 통해 해당 버퍼의 내용을 size만큼 읽어들여 stdout에 출력되도록 한다. f->eax 에는 역시 성공적으로 write한 바이트 수를 저장하고, fd가 1이 아닌 경우에는 0을 저장하고 return 한다.

1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**

수정한 소스 파일

* lib/user/syscall.h

텍스트, 폰트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명

시스템 콜을 하는 fibonacci 함수와 max\_of\_four\_int 함수를 헤더 파일에 선언해준다.

* lib/user/syscall.c

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

max\_of\_four\_int 함수는 정수 4개를 인자로 받는데, 인자 4개를 받는 시스템 콜이 없기 때문에 syscall4 함수를 syscall.c 에 새로 선언해준다. 구현은 syscall3 함수를 참고했고,

pushl %[arg3]; 과 밑에 [arg3] “r” (ARG3) 를 추가했고, addl $20으로 바꿔주었다. 인자 하나당 4바이트를 차지하기 때문에 시스템 콜 핸들이 끝난 다음에 esp를 다시 20바이트만큼 더해주어야 한다.

fibonacci 함수는 인자 하나를 받기 때문에, return syscall1(SYS\_FIBONACCI, n)로 인자 하나를 갖는 시스템 콜 함수를 불러준다.

max\_of\_four\_int는 인자4개를 받는데, 아까 미리 정의해둔 syscall4를 부른다. 이때, 첫번째 인자로는 시스템 콜 넘버인 SYS\_MAX\_FOUR를 넘겨준다.

* lib/syscall-nr.h

텍스트, 폰트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

앞서 fibonacci 및 max\_of\_four\_int 함수 각각이 첫번째 인자로 시스템 콜 넘버를 넘겨주었는데, 이를 lib/syscall-nr.h enum 안에 추가해준다.

* userprog/syscall.h

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

이제 userprog 내 syscall에서 사용할 fibonacci 및 max\_of\_four\_int 함수를 헤더에 선언해준다.

* userprog/syscall.c

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

이제 헤더에서 정의한 두 함수의 실제 구현을 userprog/syscall.c에서 해주었다. 피보나치 함수는 n번째 피보나치 수를 반환해야 한다. 만약 입력된 n이 1보다 작으면 exit code로 -1을 갖고, thread\_exit()를 해준다. 피보나치 첫번째수는 1, 두번째 수도 1이다. 이제 while 문을 돌면서 n이 3이상인 경우에 대해서 x 에는 y 값, y에는 x+y 값을 넣으면서 차례로 n번째 수까지 구한다. 이 때, y를 반환하면 n번째 피보나치 수가 반환된다.

Max\_of\_four\_int는 구현이 간단하다. 먼저, int 자료형의 가장 작은 값을 int Max 변수에 넣어준 후에, a,b,c,d 각각의 변수와 비교해서 Max를 갱신하면 된다. 그 값을 반환한다.

userprog/syscall.c

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

이제 syscall handler 함수 안에 새로 구현한 시스템 콜 넘버인 SYS\_FIBONACCI와 SYS\_MAX\_FOUR를 추가해준다. 그 후 다른 시스템 콜과 같이 해당 주소가 valid 한 주소인지 체크한 후에 앞서 선언한 fibonacci 및 max\_of\_four\_int 함수를 각각의 시스템 콜 넘버 case 안에서 호출해준다. 반환값을 f->eax에 넣어주고 return 해준다.

* 1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명**