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

담당 교수 : 김영재 교수

조 / 조원 : 20171682 임정호

개발 기간 : 2021.09.12 ~ 2021.09.28

1. **개발 목표**

프로젝트 1번의 개발 목표는 system call, system call handler, argument passing 그리고 user stack을 구현하여 user와 kernel을 연결해주는 것이다. pintos가 동작하는 방식을 이해하고, 필요한 이론을 적용할 수 있다.

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

user level에서 프로그램을 실행할 때 사용하는 argument에 대한 정보를 kernel level까지 전달받아야 한다. 이 때 esp를 stack처럼 사용하여 argument의 정보를 전달할 수 있다.

1. User Memory Access

user program을 실행하면서 사용되는 프로세스들이 각 프로세스마다 할당받은 메모리 영역에만 접근할 수 있도록 제어할 수 있다. 할당받지 않은 메모리에 접근하는 경우 프로세스를 종료한다.

1. System Calls

kernel에는 이미 여러 유용한 API가 구현되어 있다. syscall number로 구분하여 여러 system call 함수를 직접 구현하여 kernel에 있는 API와 연결할 수 있다.

* 1. **개발 내용**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명

user prgram을 실행할 때 넘겨 받은 arguments를 공백을 기준으로 파싱한다. 파싱한 arguments를 esp 스택의 top부터 넣어주면서 내려온다. arguments 값을 다 넣어주면, 이후 빠른 데이터 접근을 위해 word-alignment를 처리한다. 그리고 나서 argument가 저장된 곳의 주소값을 넣어줘야 하는데, 이 때 주소값과 데이터 값을 구분하기 위해 4바이트만큼 esp에 0을 넣는다. esp에 arguments의 주소값을 넣어주면서, argumnet 개수와 return address를 저장한다.

* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명
  + Invalid memory access를 어떻게 막을 것인지 설명

pintos가 사용하는 메모리는 크게 kernel영역과 user영역으로 나눌 수 있다. 이 때 메모리는 모두 4096바이트 크기의 virtual memory 영역(page)으로 mapping되어 사용된다. 할당받은 메모리들은 독립적으로 존재하지 않기 때문에, user process에서 본인에게 할당된 메모리 영역이 아닌 영역을 접근하면 안 된다. Pintos 상에서의 invalid memory access는 위 처럼 할당받지 못 한 메모리에 접근하는 것을 의미한다. 이러한 invalid memory access를 막기 위해 vaddr.h에 구현되어 있는 is\_kernel\_vaddr()과 접근하려는 메모리 공간이 mapping되어 있는지 확인하는 pagedir\_get\_page()를 이용하였다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명
  + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

system call은 user와 kernel을 연결하는 매개체 역할을 한다. user program은 system call을 통해 kernel api의 기능을 사용할 수 있다. 프로젝트 1에서 구현한 system call은 아래와 같다.

1. halt()

halt는 devices/shutdown.c에 구현되어 있는 shutdowon\_power\_off()를 호출하여 핀토스를 종료시킨다.

1. exit()

threads/thread.c에 구현되어 있는 thread\_exit()를 호출하여, 현재 실행되고 있는 thread를 종료시킨다.

1. exec()

process\_execute()를 호출하여, 매개변수로 받은 filename의 프로그램을 실행시킨다.

1. wait()

process\_wait()를 호출하여 매개변수로 받은 pid의 child process를 기다린 후, 사용한 resource를 회수하며, exit\_status를 받아서 리턴한다.

1. read()

매개변수로 받은 fd에서 size 바이트만큼 buffer에 입력을 받는다. 프로젝트 1에서는 fd가 STDIN인 경우만 구현하였다.

1. write()

매개변수로 받은 fd에 size 바이트만큼 buffer에 있는 값을 출력한다. 프로젝트 1에서는 fd가 STDOUT인 경우만 구현하였다.

1. fibonacci()

examples/additional.c에 이번 프로젝트에서 추가로 구현한 함수로, additional을 실행할 때 argument로 정수 4개를 받아 첫 번째 정수의 fibonacci()값을 리턴한다.

1. max\_of\_four\_int()

위 함수와 마찬가지로 additional을 실행할 때 정수 4개를 argument를 받아, 가장 큰 값을 리턴한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

09.12 – argument parsing 구현

09.22~23 – esp에 대한 이해 및 user memory access 코드 작성

09.24~25 – sycall.c 구현

09.26~09.27 – synchronization 구현

09.28 – 주석 작성

* 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)**
* Argument Passing

우선 매개변수로 받아온 arguments를 우선 공백 기준으로 파싱을 해야 한다. process.c에 있는 load() 내부에서 process\_activate() 호출 이후와 filesys\_open()호출 전 부분에 parsing하는 부분을 넘겨준다. “echo x”의 예시로 echo와 x로 분리하고, filesys\_open()은 디렉토리에서 매개변수로 넘기는 파일이 존재하는지 확인할 수 있는데, 이 때 echo라는 실행파일을 찾아야 하기 때문이다. 그 후 esp 스택이 setup이 완료되면, esp stack을 쌓는데, 반복문을 이용해 앞에서 파싱한 문자열을 넣어주고, 그 주소 값을 이후에 넣어야 하므로 주소값을 저장할 변수를 선언하여 할당한다. word-alignment와 argument의 주소값을 넣어준다. 그리고 argc와 return address를 넣어주면 Argument Passing은 완성된다.

* User Memory Access

우선 syscall.c와 exception.c에서 구현해야 한다. syscall.c에는 현재 접근하려는 주소가 kernel 영역의 주소인지 확인할 함수를 구현한다. system call을 하기 전에 미리 검사를 해준다. 이 함수는 vaddr.h에 친절하게 이미 구현되어 있다. is\_kernel\_vaddr()을 사용하면 된다. page\_fault 에러가 발생했을 때, 프로세스는 -1을 status 값으로 exit해야 한다. 이를 위해 exception.c에서 잘못된 mapping되지 않은 메모리에 접근할 때나 invalid memory access를 if문으로 검사하여 예외처리 해준다. 이 때는 pagedir\_get\_page()를 사용하면 된다.

* System Calls

우선 syscall number을 알아야 하는데, 이 enum형 변수는 lib/syscall-nr.h에 이미 선언되어 있다. 이제 syscall number을 switch()문으로 각 syscall 마다 나눠주면 된다. 프로젝트 1에서 구현해야할 syscall은 halt, exit, exec, wait, read, write, fibonacci, max\_of\_four\_int이다.

1. halt()

halt는 devices/shutdown.c에 구현되어 있는 shutdowon\_power\_off()를 호출하면 된다.

1. exit()

threads/thread.c에 구현되어 있는 thread\_exit()를 호출하면 되는데, 어떤 쓰레드가 어떤 exit status를 가지며 exit하는지 출력하는 부분을 추가하고, 이후에 process\_wait()에서 사용될 현재 쓰레드의 exit\_status에 매개변수로 받은 status값을 저장한다. 더 자세한 내용은 4번 wait()에서 설명한다.

1. exec()

process\_execute()에 실행시킬 file을 매개변수로 넘겨주고, 생성된 쓰레드의 pid를 리턴한다. 쓰레드를 생성할 때, 쓰레드의 이름이 argument까지 포함하지 않게 하기 위해 여기서도 첫 번째 argument만 parsing을 하여 thread\_create()에 넘겨준다.

1. wait()

process\_wait()를 호출하여 매개변수로 받은 pid의 child process를 기다린 후, 사용한 resource를 회수하며, exit\_status를 받아서 리턴한다. 이 때, process\_wait()에서 부모 프로세스가 자식 프로세스보다 먼저 죽는 걸 막기 위해 semaphore 개념을 활용한다. thread.h에서 thread의 구조체 변수로 struct semaphore 변수 2개를 선언한다. 하나는 부모가 먼저 죽지 않게 기다리는 용도이고, 나머지 하나는 부모가 자식의 exit\_status 값을 저장할 때까지 자식이 죽지 않게 기다리게 하기 위함이다. 쓰레드 구조체 변수에 exit\_status를 선언하여 자식이 exit()에서 부모에게 자신의 exit status를 전달할 수 있게 한다. 또한 부모 쓰레드는 자식 쓰레드를 관리하고 있어야 한다. 이를 위해 init thread가 사용하는 struct list\_elem변수와 struct list 변수를 선언한다. struct list children은 부모 쓰레드가 관리하는 자식 쓰레드를 관리하는 리스트 변수이다. struct list 구조체는 /lib/kernel/list.h에 선언되어 있다. pintos는 double linked-list로 관리하는데, 일반 double linked-list와 다르게 data값을 바로 접근 못 하기 때문에 까다롭다. 그렇기에 매크로 함수로 정의되어 있는 list\_entry()를 이용해 list\_elem에 해당하는 데이터를 접근할 수 있다. **init\_thread()**에서 생성할 쓰레드의 자식 쓰레드를 관리할 리스트를 생성한다. 그리고 나서 list\_push()를 이용하여 현재 쓰레드, 즉 부모 쓰레드의 자식 리스트에 지금 생성하고 있는 자식 쓰레드를 리스트에 list\_push\_back()으로 넣어준다. 위에서 언급했던 semaphore 변수 2개를 0으로 초기화한다. **process.c**에서 반복문으로 현재 쓰레드 (부모 쓰레드)의 children을 탐색하면서 wait를 해줘야 할 자식 쓰레드의 pid를 찾는다. 부모 쓰레드는 자식 쓰레드가 process\_exit()에서 첫 번째 세마포의 값을 올릴 때까지 기다리다가, 자식 쓰레드가 sema\_up()으로 부모에게 알려주면, 부모 쓰레드는 관리하고 있던 리스트에서 현재 죽기 직전의 자식 쓰레드를 list\_remove()를 통해 부모 쓰레드는 자식의 exit\_status를 저장한다. 저장하기 전에 자식 쓰레드가 죽는 것을 막기 위해 나머지 세마포 변수로 자식 쓰레드는 sema\_down()을 시도하면서 기다린다. 초기 세마포 변수의 값은 0이었으므로 부모가 값을 다 저장하고 나면 sema\_up()을 해주고, 자식에게 이제 죽어도 됨을 알린다. 초기에 자식은 **syscall.c**에서 thread\_exit()을 호출하고, thread\_exit()에서 proess\_exit()를 호출하는 형식이기 때문에 자식 쓰레드는 process\_exit()를 종료되면 thread\_exit()로 돌아가 리소스를 반납하고 죽게 된다. 부모 쓰레드가 저장한 exit\_status를 리턴하며 종료된다.

1. read()

매개변수로 받은 fd에서 size 바이트만큼 buffer에 입력을 받는다. 프로젝트 1에서는 fd가 STDIN인 경우만 구현하였다. pintos manual에 있는 input\_getc()를 이용한다.

1. write()

매개변수로 받은 fd에 size 바이트만큼 buffer에 있는 값을 출력한다. 프로젝트 1에서는 fd가 STDOUT인 경우만 구현하였다. pintos manual에 있는 putbuf()를 이용한다.

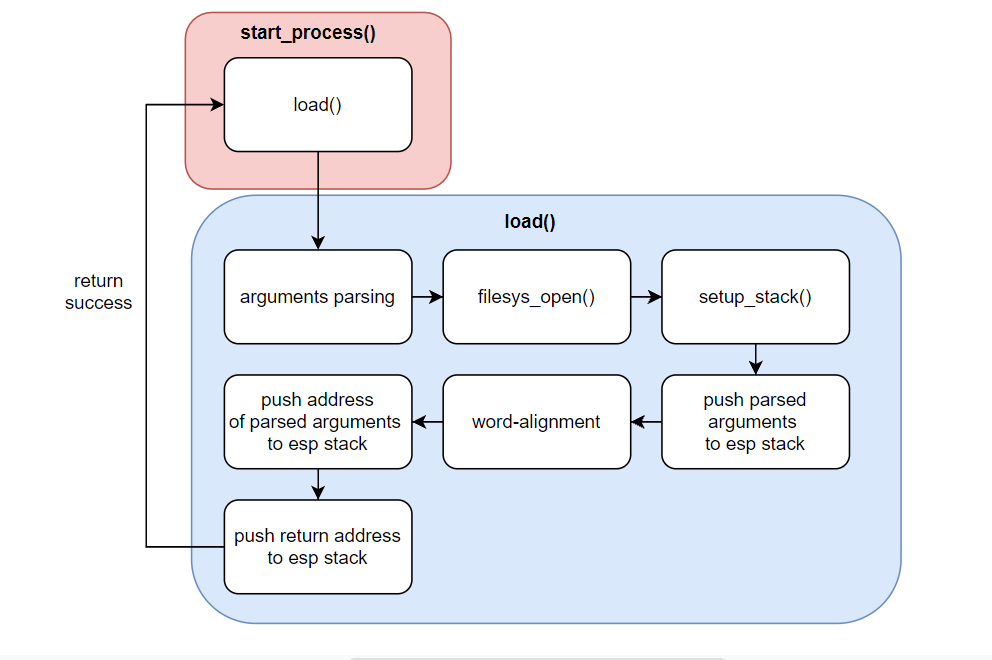
1. fibonacci()

examples/additional.c에 이번 프로젝트에서 추가로 구현한 함수로, additional을 실행할 때 argument로 정수 4개를 받아 첫 번째 정수의 fibonacci()값을 리턴한다. 재귀함수로 구현하였다.

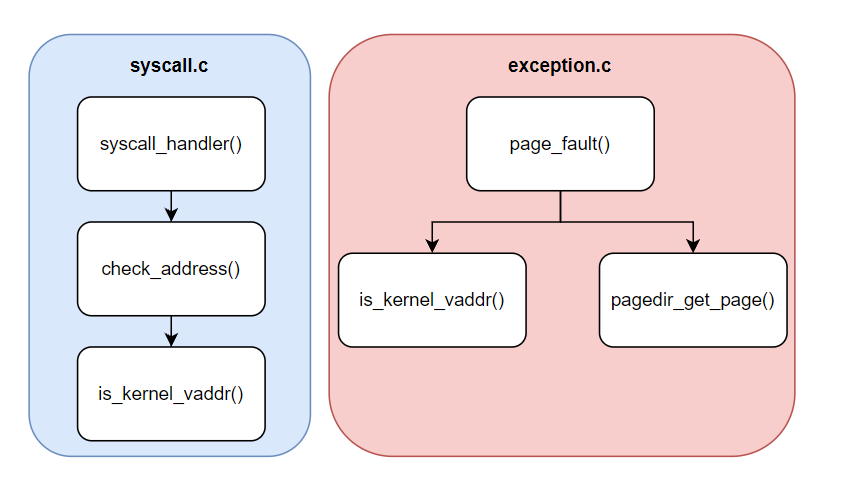
1. max\_of\_four\_int()

위 함수와 마찬가지로 additional을 실행할 때 정수 4개를 argument를 받아, 가장 큰 값을 리턴한다. 삼항 연산자를 3번 사용하여 구한다.

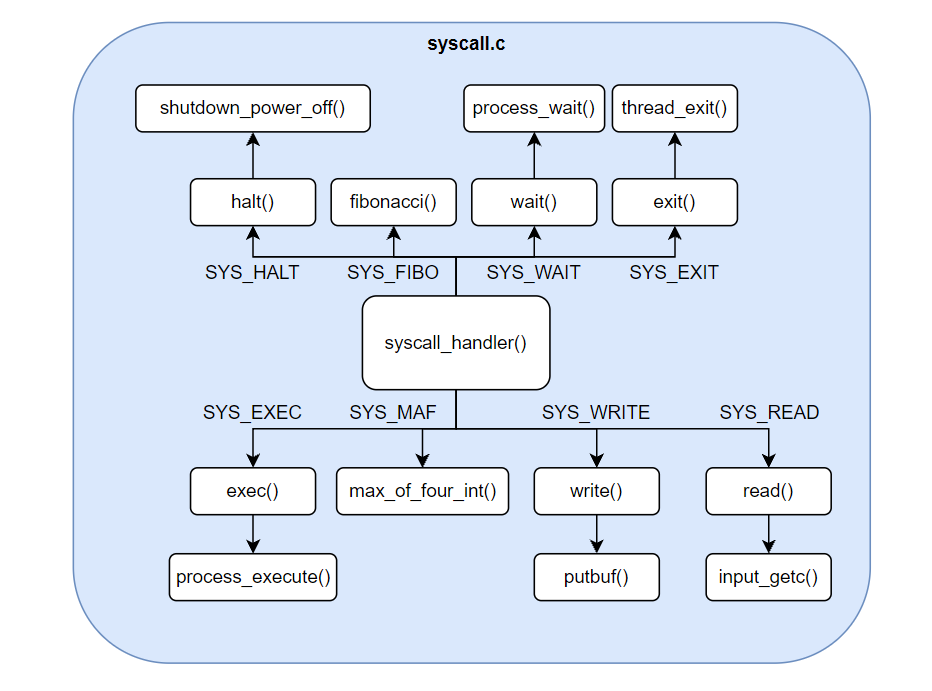
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing

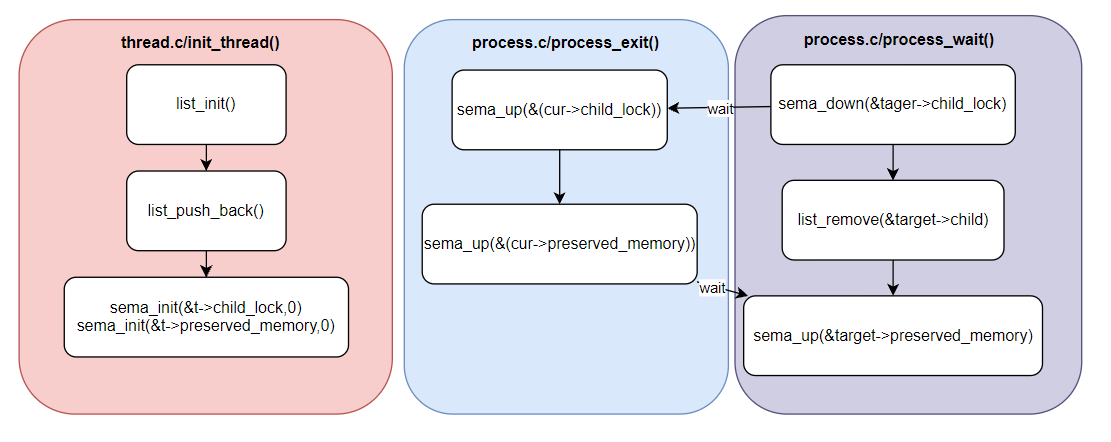


1. User Memory Access



1. System Calls





* 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

load()에서 const char\*의 형태로 file\_name을 받아오는데, 이 file\_name을 파싱해야 한다. char\* copy\_file\_name을 선언하여 const\_char\*의 형태의 file\_name을 복사한다. 이 때, strcpy()을 사용하면 에러가 발생하여, strlcpy()로 하였다. null값 까지 복사하기 위해 strlen(file\_name)에 +1만큼 동적할당하였다. 이후 strtok\_r()을 이용하여 공백을 기준으로 파싱을 시작한다. 이 때 strtok()를 초기에 사용했는데, 에러가 발생하여 strtok\_r()로 수정하였다. 이제 디렉토리에서 실행시킬 파일을 filesys\_open()에 매개변수로 넘긴는데, 파싱한 문자열의 첫번째 값을 넘겨주었다.

setup\_stack()으로 esp를 세팅하였으면, 이제 파싱한 문자열들을 esp 스택에 쌓는다. for loop을 이용하여 입력할 때, 끝에 있던 argument부터 넣는다. \*esp를 내리면서 넣는데, word-alignment를 위해 byte\_cnt로 사용한 바이트수를 카운트한다. 4바이트 단위로 align 되기 때문에 사용한 바이트 수가 4로 나눴을 때 나머지만큼 \*esp값을 빼준다. esp는 intr\_frame 구조체 변수로 void\* esp로 선언되어 있다. start\_process()에서intr\_frame->esp의 주소값을 넘기기 때문에 load()에서 더블 포인터의 형태로 넘어온다. void\*\* 이기 때문에, \*esp는 esp가 가리키는 곳의 주소값이고, \*\*esp는 esp가 가리키는 곳의 값이다. esp 스택에 값을 넣어줘야 하기 때문에, 넣을 값의 type에 맞게 casting을 해준 뒤, \*\*(casted esp)에 값을 대입하였다. 잘 들어갔는지 확인하기 위해, hex\_dump()로 값을 확인해보려 했는데, 출력되는게 없었다. 찾아보니 부모프로세스가 자식프로세스가 죽기 전까지 기다려야 하는데, 부모 프로세스가 먼저 죽어버렸기에 생긴 문제였다. 그래서 임시 방편으로 process\_wait()에 for문을 10억번 돌려서 결과를 확인했다.

1. User Memory Access

pintos manual 3.1.5를 읽어보니 vaddr.h가 언급되어 있어서 vaddr.h에 가서 찾아보았다. threads/vaddr.h에 is\_kernel\_vaddr()이라는 함수가 구현되어 있었다. 어떤 식으로 구현되어 있는지 확인해보니, 확인할 주소값이 PHYS\_BASE보다 위에 있는지 아래에 있는지로 kernel 영역의 메모리인지, user 영역의 메모리인지 구분하는 함수였다. 그래서 초기에는 syscall\_handler()에서 대부분의 syscall()들에게 f->esp값을 각 type에 맞게 캐스팅해서 넘겨주기 전에 검사해주는 식으로 하였다. 이렇게 했더니, sc-bad-arg는 pass가 뜨지만, sc-bad-sp는 fail이 떴다. 그래서 pintos manual에 적혀있던 exception.c에 있는 page\_fault()로 가서 주석을 읽어보니 “Some solutions to project 2 may also require modifying this code”라는 문구를 발견했다. 다시 매뉴얼을 얽어보니 “a pointer to unmapped virtual memory”를 봤고, 주소값이 할당받지 않은 page를 가리켰을 때도 처리해야함을 알았고, page\_fault()의 printf() 이전에 if문으로 is\_kernel\_vaddr()와 pagedir\_get\_page()를 호출하여 해당되면 -1로 exit하게끔 수정하였더니, sc-bad-sp또한 해결되었다. 이게 정답인지는 확실하지 않으나, pass가 떴기에 넘어갔다.

1. System Calls

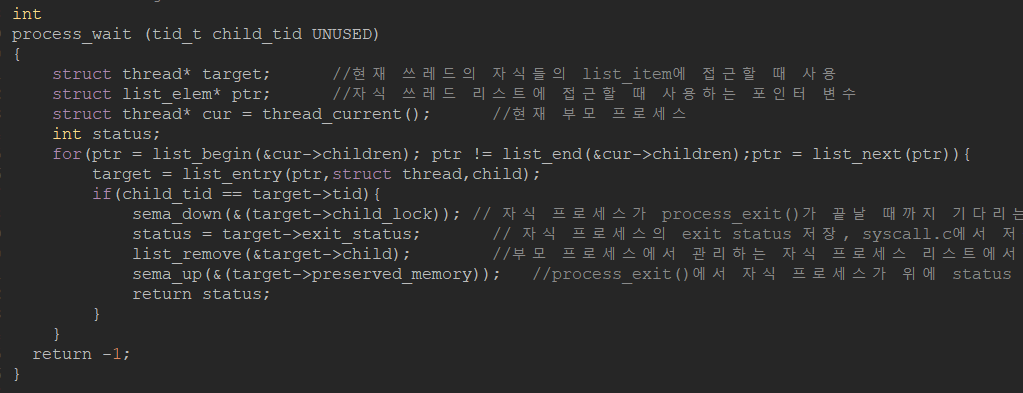
* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**

syscall.c의 syscall\_handler()에서 intr\_frame 변수를 매개변수로 받는다. proj1.ppt를 보니 syscall.c에서 esp는 syscall number를 가리키고 있어서 f->esp 값을 int로 casting해서 확인했더니 9가 나왔다. 명령어가 echo x였고, lib/syscall-nr.h에 선언되어 있는 enum 변수 중에 9에 해당하는 값을 찾았더니 SYS\_WRITE이었다. f->esp가 syscall number를 가리키는 것을 확인했다. 이 때 문제가 하나 있었는데, “echo x”였으면 “echo”, “x” “\n”, “exit” 총 4개의 syscall\_number가 출력되어야 할 것 같은데 1번밖에 나오지 않았다. thread\_exit()가 syscall\_handler() 밑 부분에 호출되고 있었음을 알았고, 이를 주석처리하니 9,9,9,1이 잘 출력되었다. syscall.c는 ppt에 적혀있는 대로 일단 따라가면서 구현하였다. 구현할 때, 넘기는 매개변수들은 lib/user/syscall.c에 구현되어 있는 syscall()를 참고하였다. 여기서 syscallN()을 호출할 때, 넘기는 argument에 힌트를 얻어 userprog/syscall.c에 적용하였다. f->esp부터 100바이트 정도씩 hex\_dump()로 값을 확인하면서 값이 제대로 넘어오는지 확인했고, 리턴값이 있는 함수이면, 그 리턴 값을 f->eax에 저장하였다. syscall()에 넘길 때, f->esp를 적절하게 type casting하여 보냈다. pintos manual과 ppt를 참고하면서 각 syscall()을 구현했고, make chek를 했을 때, fail이 생각보다 많이 떠서, build/tests/userprog/result.txt를 확인했더니, thread가 exit할 때 “echo: exit(0)” 이런 식으로 출력이 되어야 하는데, 나는 “echo x: exit(0)”으로 출력이 되고 있었다. 쓰레드의 이름 또한 파싱이 되어야 함을 알았고, process\_execute()에서 thread\_create()를 호출할 때, 첫 번째 매개변수로 쓰레드 이름이 될 문자열을 넘기는 것을 찾았다. 파싱한 문자열을 매개변수로 넘기게 수정했고, 맞게 출력이 되었다. 이제 for문을 10억번 돌리는 부분을 고치는 일이 남았다. ppt를 보니 synchronization을 해야했고, 이를 위해 지난 sp때 다뤘던 semaphore를 사용했다. 부모 프로세스는 자식 프로세스를 기다려야 하는 상황이고, 자식 프로세스들을 관리하고 있어야 한다. 이를 위해 thread.h에 정의되어 있는 thread 구조체 변수에 세마포 변수 2개와, 자식 프로세스를 관리할 리스트 struct list children, 그리고 부모의 children에 들어갈 struct list\_elem child를 추가하였다. 대략적인 설명은 3.B wait() 부분에 되어있고, 구현 측면에서 보면 다음과 같다.

텍스트이(가) 표시된 사진

자동 생성된 설명

t->children을 list로 생성해서, running\_thread()->children에 t->child를 push한다. 이 때, running\_thread()는 부모 프로세스를 의미하고, 부모프로세스가 관리하는 child에 들어가는 것이다. 세마포 변수 2개를 생성하여, process\_wait()과 process\_exit()간에 부모와 자식이 서로 기다릴 수 있게 한다.



for loop을 이용해 부모 프로세스가 관리하는 쓰레드 리스트에서 매개변수로 받은 tid를 가진 자식 프로세스를 찾아 기다려 준다. sema\_down(&target->child\_lock)으로 기다린다.process\_exit()에서 sema\_up()할 때까지 기다린다. process\_wait()는 자식 프로세스의 exit status를 리턴해야 하는데, 자식 쓰레드를 관리하는 리스트에서 현재 기다리는 쓰레드를 list\_remove()를 통해 제거하게 되면, exit\_status에 접근할 방법이 없다. 따라서 exit\_status를 따로 저장을 하는데, target->exit\_status의 exit status 값은 syscall.c/exit()에서 미리 저장해준다. 이 때 status값을 저장하기 전에 자식 프로세스가 thread\_exit()으로 돌아가 리소스를 반환하고 죽는 경우를 막아야 한다. 이 때 2번째 세마포 변수, preserved\_memory가 사용된다. 자식 프로세스가 proess\_exit()에서 child\_lock을 sema\_up()해서 부모에게 알려주고 나면, preserved\_memory를 sema\_down()을 하면서 기다린다. init\_thread()에서 0으로 초기화 되어있고, 이 세마포 변수는 부모 프로세스가 process\_wait()에서 exit status를 다 저장하기 전까지는 sema\_up() 되지 않는다. 부모가 다 저장하고 나면 자식 프로세스는 thread\_exit()으로 돌아가 최후를 맞이하게 된다.

1. Additional System calls

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

ppt를 참고하면서 구현하였다. lib/user/syscall.h에 fibonacci()와 int max\_of\_four\_int()를 선언하였고, lib/user/syscall.c에 max\_of\_four\_int()를 구현하기 위해 syscall4()를 구현하였다. syscall1부터 syscall3이 매크로 함수로 정의되어 있었는데, 규칙을 보면서 argument를 하나씩 추가하였다. 그리고 max\_of\_four\_int()와 fibonacci()를 정의했는데, 다른 함수들처럼 syscall4, syscall1에 매개변수로 syscall number와 각 함수의 매개변수들을 넘겨주었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

userprog/syscall.c에서 fibonacci()와 max\_of\_four\_int()를 구분하기 위해 syscall number을 생성해야 한다. 이는 lib/syscall-nr.h에 정의되어 있는 enum 변수들 뒤에다가 SYS\_FIB, SYS\_MAF로 추가해주었다. 그러고 나서 userprog/syscall.h에 fibonacci()와 max\_of\_four\_int()를 선언하였고, syscall.c에 그 기능을 구현하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

위 두 함수 역시 check\_address()를 이용해 kernel영역의 메모리를 가리키고 있는지 검사를 하였고, examples에서 additional.c를 생성하여 fibonacci()와 max\_of\_four\_int()를 호출하는 코드를 작성하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

양식은 examples에 있는 다른 함수들 참고했다. 그러고 나서 make를 하려 했더니, Makefile에 additional.c가 포함되어 있지 않았다. 그래서 Makefile또한 다른 함수들과 동일하게 넣어주었다.

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

* 1. **시험 및 평가 내용**
* 텍스트이(가) 표시된 사진

  자동 생성된 설명**fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**