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

담당 교수 : 김영재

조 / 조원 : 최재원

개발 기간 : 2021/9/10 – 2021/10/2

1. **개발 목표**

Pintos 위에서 user program이 실행될 수 있도록 초기 상태의 pintos 코드를 수정한다. User virtual memory상에 Calling Convetion에 따라 명령어에 대한 정보를 적재(Load)하는것을 시작으로, System call handler를 작성하여 명령어에 따라 적절한 System call이 수행될 수 있도록 한다. 이 때 project 1에서는 halt, exit, exec, wait ,read(Stdin), write(Stdout) 에 대한 system call들의 기능과 추가로 additional(Fibonacci, Max\_of\_four\_int)까지 실행 가능하도록 한다.

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

Running program의 최초 argument들을 user stack에 pintos calling convention에 따라 절절히 적재한다. 이 때 명령어와 argument들을 parsing하여 intr\_frame 구조체의 esp를 stack pointer로 하여 저장하게된다.

pintos --filesys-size=2 -p ../examples/echo -a echo -- -f -q run 'echo x'

그림 1> pintos상에서 ‘echo x’를 실행

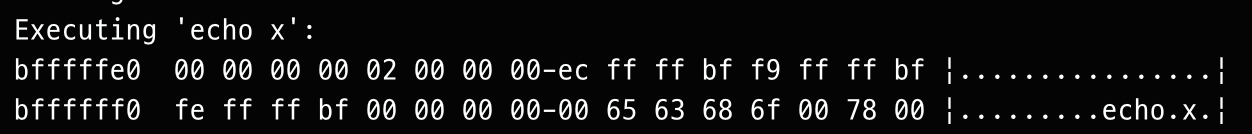
위와 같은 명령어를 수행하여 hex\_dump()라는 pintos에서 제공하는 debug용 함수의 실행 결과를 보면 user stack의 주소와 값을 16진수 형태로 확인할 수 있고, 따라서 argument passing이 올바르게 수행됐는지 알 수 있다.

그림 2> Argument passing 구현 후 Hex\_dump() 실행 결과

위 ‘그림 2’는 hex\_dump()의 출력 화면이다. 우선, user stack의 끝부분에 x\0 echo\0순으로 들어간 것으로 보아 argument의 맨 뒤부터 차례대로 들어간 것을 확인할 수 있다. 그리고 NULL까지 포함하여 총 7byte이기 때문에 4byte 단위로 word alignment를 하기 위해서 1byte 0값을 넣어주었다. 이 후 NULL pointer sentinel이 들어가고, 차례대로 echo와 x에 대한 주소값이 들어갔으며, 마지막으로 argc 값과 return address가 들어가 esp의 현재 주소 위치는 bffffe0임을 확인할 수 있고, argument passing이 정상적으로 수행되었음을 확인할 수 있다.

1. User Memory Access

User program이 open(NULL)과 같이 NULL 포인터를 argument로 넘기는 상황, Unmapped virtual memory에 접근하려는 상황, 다른 Running process의 virtual memory에 접근하려는 상황, 그리고 kernel address space에 접근하려는 상황 등에 대해서 적당한 조치가 필요하다. 특히 kernel address space에 접근하려는 오류는 매우 치명적이기 때문에 예외 처리가 필수적이다. 결과적으로 page fault 및 kernel 오류 없이 코드가 작동한다. ‘make check’로 확인해본 결과, sc-bad-sp, sc-bad-arg등에서 pass 됐음을 확인하였다.

1. System Calls

User program이 kernel의 API를 쓸 수 있도록 앞서 argument passing를 구현한 user stack을 사용하여 중간 매개체 역할인 system call handler와 Halt, exit, exec, wait, read(Stdin), write(Stdout) 그리고 추가적으로 fibonacci, max\_of\_four\_int에 대한 system call 함수를 작성하였다. 이 때 intr\_frame 구조체의 eax와 esp를 활용한다. 최종적으로, ‘그림 1’의 결과가 아래와 같이 정상적으로 나옴을 확인할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

그림 3> run ‘echo x’ 실행 결과

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

Pintos에서 user program 실행시 대략적인 과정은 가장 먼저 system call이 initialize되고, thread가 create되며 그 다음 load 함수에서 user stack에 명령어 등을 적재시키는 수행을 하는 것이다. User stack을 구성하는 함수를 load 함수에서 따로 호출해주며, 함수 이름은 makeup\_stack(esp ,file\_name)이고, esp를 stack pointer로 사용하여 file\_name string을 적절히 Parsing하여 저장한다. setup\_stack 함수가 먼저 호출된 후 makeup\_stack함수가 호출되면, 그 안에서 먼저 ‘ ‘을 단위로 parsing이 이루어지고 가장 마지막에 parsing된 string, 즉 가장 마지막 매개변수부터 역순으로 stack에 저장된다. 명령어들의 총 byte수가 4byte단위로 word alignment되지 않았을시에 이를 위해 추가적으로 stack에 부족한 byte수만큼 적재한다. 다음으로 calling convention에 따라서 NULL pointer sentinel이 저장되고 앞서 저장된 명령어들의 주소값이 차례대로 저장된다. 마지막으로 저장된 주소값들의 시작 offset이 저장되고 parsing된 명령어 및 arguments의 개수가 저장되며, return address가 저장된다. 이렇게 구성된 user stack은 이 후 user process에서 systemcall을 하고 kernel가 이에 대한 수행을 하는 상황 등에 사용된다. 저장된 형태는 ‘그림 2’와 같다.

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

Pintos 상에서는 process끼리 서로 demage를 주거나(서로의 memory 영역 침범), process가 kernel code의 영역을 reference하는 문제를 막기위해 VM(virtual memory) system을 사용하여 각 process마다 VM을 할당한다. VM은 또다시 user memory 영역과 kernel memory 영역으로 나뉘는데(e.g. 0~3GB : user / 3GB ~ 4GB : kernel), 이 때 user virtual memory가 아닌 kernel virtual memory를 접근하는 것을 invalid memory access라고 한다.

* + Invalid memory access를 어떻게 막을 것인지 설명

Pintos 내에서 threads 디렉토리의 loader.h에는 PHYS\_BASE가 3GB로 정의되어 있고, 이는 user virtual memory의 size이므로 bound로써 사용된다. Vaddr.h 파일에는 is\_user\_vaddr()함수가 정의되어 있으므로 이 함수를 사용하여 user process가 invaild memory를 access를 하려고 할 때 process를 강제 종료 시킨다. 이를 system call handler에서 user stack에 저장되어 있는 값을 토대로 kernel 영역을 침범하는지를 판단하여 프로그램을 종료시켜주는 방식으로 구현한다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

Pintos의 kernel에는 유용한 API들이 많은데, user process는 privileged mode가 아니기 때문에 사용하지 못한다. User process상에서 kerenl의 다양한 API들을 활용하기 위해서 system call을 사용한다.

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

Pintos를 종료시키는 system call이며, deadlock situation등에서 정보를 보존하기 위해 가끔 사용된다.

* • exit()

현재 User program을 종료시킨다. 이 때 exit status을 kernel에게 전달하며, 0은 성공적으로 종료되었음을 의미하고, 0이 아닌값은 error를 나타낸다.

* • exec()

Cmd로 입력된 명령어와 arguments에 따라서 실행하며, child process가 정상적으로 수행을 완료했을시 해당 tid를 return한다.

* • wait()

모든 child process들이 정상종료 될 때까지 parent process가 wait을 수행한다. Child process의 exit signal에 의해서 wait 수행이 완료되고, 이를 위해 적절한 synchronization이 필요하다.

* • read()

이번 projcet에서는 stdin에 대한 read만 수행하며, 이 때 input\_getc() 함수를 사용한다. 읽은 bytes수를 return 하고, 읽을 수 없는 file일시에는 -1을 return한다.

* • write()

Fd Stdout만을 구현하며, 쓴 bytes수를 return한다. 즉, Console에 write를 수행하고, 이를 구현할 때 putbuf() 함수를 사용한다.

* • fibonacci()

Additional system call 구현중 하나로써, additional 명령어에 대해서 가장 첫번째 argument인 n에 대해서 피보나치 수열의 결과값을 return한다.

* • max\_of\_four\_int()

Additional system call 중 하나로써, additional 명령어시 입력되는 4개의 argument 중 가장 큰 값을 return 한다.

* + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

User program에서 system call을 하면 interrupt가 발생하고, cpu 제어권을 kernel에게 넘기며, os가 mode bit를 kernel mode로 바꾼다. Kernel은 interrupt handler에서 system call handler(0x30)을 호출한다. System call handler에서는 usre stack에 있는 정보를 바탕으로 system call function을 수행하며 그 결과를 eax에 저장한다. 이렇게 저장된 정보는 mode bit가 다시 user mode로 바뀌고 cpu 제어권이 user program으로 돌아가면 해당 process에서 사용하게 된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |  |
| --- | --- | --- |
| Date | Works | Details |
| 9.10-9.17 | Pintos code preveiw | Pintos manual& ppt 활용 |
| 9.18-9.23 | Argument passing | Make up User stack 구현 |
| 9.23-9.24 | Filename parsing 구현 |
| 9.25-9.27 | System call & User memory access & Additional system calls | Wait 제외 나머지 구현 |
| 9.27-9.28 | User memory access 구현 & wait & additional 구현 |
| 9.29-10.2 | report | Reporter 작성 |

* 1. **개발 방법**
* Argument Passing

텍스트이(가) 표시된 사진

자동 생성된 설명 User stack을 구현하기 위해 makeup\_stack()이라는 함수를 선언하고, process.c 파일의 load() 함수에서 setup\_stack() 함수 이후에 호출한다. 해당 함수에서 가장 먼저 file\_name에 저장되어 있는 명령어와 arguments들을 pasrsing한다. 이 때 length of arguments는 최대 4KB이고, 이번 project에서는 128 bytes라고 명시되어 있지만,

그림 4> process.c psuedo code parsing한 명령어들을 저장하기 위한 배열256bytes로 여유롭게 동적할당 하였다. 그렇게 parsing한 string들을 threads directory의 interrupt.h에 선언되어있는 intr\_frame 구조체의 esp를 stack pointer로 사용하여 저장한다. 앞서 설명한대로 calling convention에 따라서 arguments는 역순으로 뒤에서부터 저장하고, 4bytes word alignment가 되지않으면, 부족한 bytes만큼 stack에 저장한다. 이후에 앞선 명령어들의 주소값을 저장하고, 그 시작 주소를 저장하며, 명령어와 arguments의 수, return address를 저장한다. 한편 process\_execute함수와 load함수에서 filename을 먼저 따로 parsing 해주고, filesys\_open 함수를 사용하여 명령어가 NULL인 경우에 대한 예외처리를 해준다.

* User Memory Access:

||.B에서 설명한대로 user program이 kernel virtual memory에 access하는 것을 막기위해 syscall handler에서 접근하려는 주소에 대해 threads directory의 vaddr.h에 정의되어 있는 is\_user\_vaddr()함수를 사용한다. 해당 함수는 loader.h에 정의되어있는 PHYS\_BASE(3GB로 정의 되어 있음)를 사용하며, 이 때 잘못된 주소를 접근하는 경우에 대해 error처리로써 exit(-1)을 해줘야 하므로, 따로 protect\_user\_memory()라는 함수를 선언하여 사용한다. 따라서 interrupt.h에 정의되어 있는 intr\_frame 구조체 f의 esp를 사용하여 Stack에 저장된 주소값들에 대해서 protec\_user\_memory() 함수를 호출해준다. 한편, 각 thread는 memory에 대한 page를 지니고 있고, page fault에 대한 처리를 해줄때도, 이에 대한 invalid memory access도 일괄적으로 검사해주기 위해 userprog directory의 exception.c에 선언되어 있는 page\_fault() 함수에서 fault\_addr이 kernel scope에 해당하거나, 해당 함수에 선언된 user 변수가 False일 때 exit(-1)를 해준다.

* System Calls

Project1에서 구현해야 하는 system call은 ||.B에서 설명하였다. 이를 구현하기 위해서 우선 userprog의 syscall.h와 syscall.c를 수정해야 한다. Syscall\_handler함수에서 f->esp에 저장된 값(system call number)에 따라서 switch문을 활용하여 각 system call function을 호출하고, 이 때 사용하는 function들을 syscall.h에 추가한다.

Exit 함수에서는 thread\_exit함수를 호출하고, 이 때 thread.h의 thread 구조체 안에 exit\_number 변수를 선언해주고, 정상적으로 종료되었는지 아니면 error가 발생하여 종료되었는지의 여부를 판별하기 위해서 해당 exit 함수에서 current thread의 exit\_number에 exit num을 저장해준다. Halt 함수에서는 manual에 적힌대로 devieces 디렉토리의 shutdown.h에 정의되어 있는 shutdown\_power\_off() 함수를 호출한다.

Exec함수에서는 process\_execute() 함수를 호출하며, Read 함수와 write 함수는 각각 File descriptor가 stdin과 stdout에 대해서만 구현하고, ppt에 명시된대로 input\_getc()와 putbuf()함수를 사용한다. Additional 함수들은 각각 기능에 맞추어 구현하고, 결과값을 return한다.

Wait 함수에서는 process\_wait을 호출하여 실질적으로 process.c에 있는 해당 함수에서 wait을 수행한다. 이 때 synchronization을 사용하고 이를 위해서 threads directory의 thread.h에 정의되어 있는 thread 구조체에 synchronization을 위한 semaphore 구조체 변수와 list 구조체 변수를 선언해준다. 그리고 process\_wait 함수에서 wait 함수로부터 넘겨받은 child\_tid 값에 대한 parent process가 child porcess가 끝날때까지 blocked상태로 대기한다. 이 때 child process가 먼저 끝나서 parent process에서 메모리를 삭제하지 못하는 경우가 발생할 수 있으므로, process\_exit에서 parent process를 깨워주면서 동시에 child process를 잠시 blocked상태로 대기시킨다. 이 후 parent process에서 child process의 메모리를 remove하고나서 child process를 kill 시킨다.

마지막으로 additional system call 중 sum\_of\_four\_int() 함수를 위해서 syscall4를 lib 디렉토리의 user 디렉토리 안에 syscall.c 파일에 선언해주고, 해당 함수와 fibonacci 함수를 각 syscall#을 호출하도록 선언해준다. 물론 system call handler에서 syscall# 매크로를 통해서 해당 system call을 넘겨받을 수 있도록 lib 디렉토리의 syscall-nr.h 파일에서 enum 으로 선언된 system call number들에 SYS\_FIBO와 SYS\_MAXFOUR를 추가해준다. 이 후 userprog 디렉토리의 syscall.c 파일에 해당 함수들을 구현하고, 마지막으로 구현한 시스템콜을 확인하기 위해서 examples 디렉토리에 additional.c 파일을 작성하고 makefile을 수정한다.

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

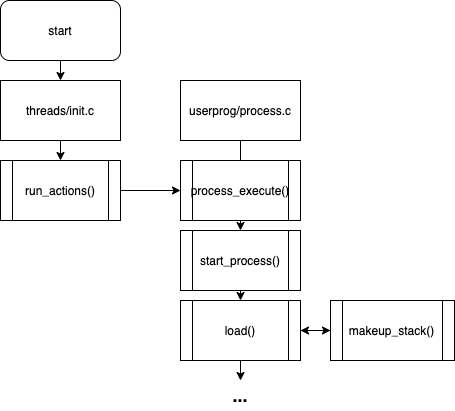


그림 5> flow chart – Argument passing

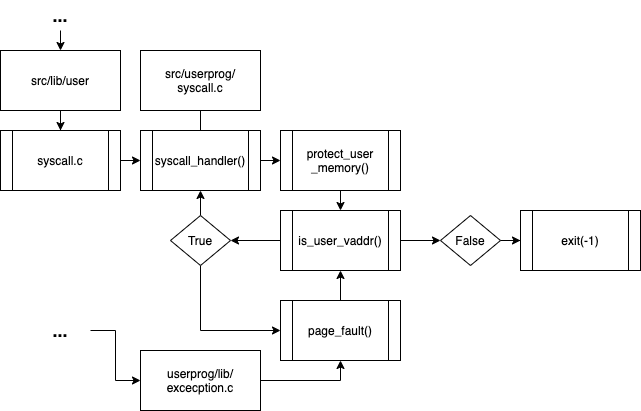
1. User Memory Access

그림 6> flow chart – User memory access

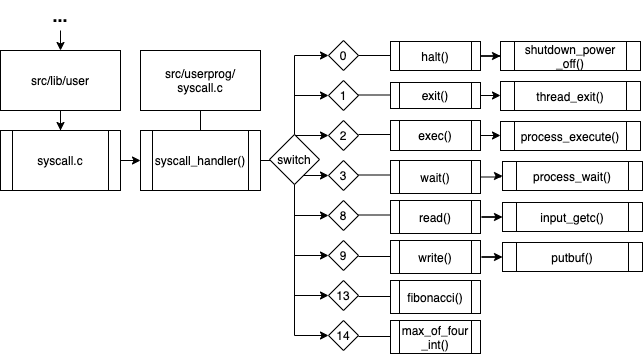
1. System Calls

그림 7> flow chart – System calls

* 1. **제작 내용**

1. 텍스트이(가) 표시된 사진

   자동 생성된 설명Argument Passing

Load 함수에서 setup\_stack을 호출 한 다음에 왼쪽에 보이는 makeup\_stack 함수를 호출한다. Char \*\* Argv와 char\* buf를 동적할당 받아 사용한다. argv에는 parsing된 string이 ‘\0’을 포함하여 저장되며, buf는 argument로 받은 file\_name을 parsing 할 때 character를 하나씩 살펴볼 포인터로써 사용된다. Char\*delim은 parsing의 기준인 ‘ ‘(space)를 buf가 만났을 때 해당 주소를 가르켜 ‘\0’를 삽입하는 역할로 사용된다. Parsing을 마치고, esp에 argv에 저장된 string들을 argc-1부터 역순으로 저장한다. 이 때 32byte – cpu 기준으로 4byte word alignment로 생각하므로, esp를 -4단위로 subtract해가며 저장한다. 이 때 esp는 threads dir의 interrupt.h에 정의되어 있는 intr\_frame 구조체의 변수이다. 이 후 int word\_aligne 변수에 앞서 저장한 argv의 elements의 총 bytes에서 4bytes로의 word alignment에 부족한 bytes 수만큼을 저장하여 해당 byte 수만큼 uint8\_t 자료형(unsigned char)을 사용하여 esp stack에 byte수를 채워준다. 다음으로 NULL pointer 텍스트이(가) 표시된 사진

자동 생성된 설명sentinal과 argv elements의 주소값을 저장하며, 마지막으로 argv[0]의 주소값을 저장한 주소값과 argc, return address를 차례대로 esp에 저장하고 해당 함수를 마친다. 마지막에 동적으로 할당한 배열을 메모리 해제해 주었다.

그림 8> process.c – makeup\_stack()

추가적으로, process\_execute 함수에서 thread\_create를 하기 전에 file\_name으로부터 맨 앞의 명령어를 따로 parsing하여 filesys\_open() 함수를 호출함으로써 NULL일 경우 -1을 return 해주었고, parsing한 명령어를 argument로 thread\_create 해주었다. 마찬가지로 load에서 filesys\_open을 하기 전에, file\_name으로부터 맨 앞의 명령어를 따로 parsing하여 filesys\_open() 함수를 해당 argument로 하여 호출해주었다.

1. User Memory Access

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명Invalid User memory access를 방지하기 위해서 앞 서 설명한 바와 같이 threads 디렉토리의 vaddr.h에 정의되어 있는 is\_user\_vaddr 함수를 사용한다. Userprog dir의 syscall.c에서 syscall\_handler가 호출되면 해당 함수에서 다시 protect\_user\_memory 함수를 호출하며, 해당 함수에서 is\_user\_vaddr 함수가 사용된다. Is\_user\_vaddr 함수에서는 threads dir의 loader.h의 PHYS\_BASE (=3GB)가 조건식에 사용되므로, user virtual memory bouund가 아닌 접근에 대해서는 exit(-1)을 해주어 error인 경우로 처리해주었다.

그림 9> syscall.c – protect\_usr\_memory()

텍스트이(가) 표시된 사진

자동 생성된 설명

그림 10> exception.c – page\_fault()

위의 그림은 exception.c의 page\_fault 함수 처리과정에서 fault\_addr이 user 영역이 아니거나, 해당 함수에 기존에 있던 user 변수의 값이 False일시에 exit(-1)을 함으로써 앞에서와 마찬가지로 invalid memory access에 대한 예외처리를 해주었다.

1. System Calls

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

Systemcall\_handler 함수에서 thread의 interrupt.h의 intr\_frame 구조체 f->esp에 저장되어 있는 system call number에 대해서 switch문으로 case를 나누어 각 system call에 대한 function을 호출하였다. Halt의 경우를 제외한 나머지 모든 경우에서 우선적으로 접근하려는 address에 대해서 protec\_user\_memory를 호출해주었다. 해당 system call에 대한 구현을한 syscall.c는 kernel code에 해당되며, 각 system call에 대한 function의 결과값은 f->eax에 저장하여 user program에서 결과값을 사용할수 있도록 하였다.

1. Halt

Pintos manual에 명시되어 있는대로 halt() 함수에서는 shutdown\_power\_off 함수를 호출하였다. 이 함수는 devices dir의 shutdown.h에 정의되어있으며, deadlocak 상황등에서 해당 system call이 호출된다.

1. Exit

Exit 함수에서는 현재 thread의 이름, 즉 명령어의 string과 exit status number를 출력한다. Exit 함수의 argument로 넘어온 f->esp+4에 해당 status number가 저장되어 있으며, 앞서 개발 방법에서 명시한대로 threads dir의 thread.h의 thread 구조체에 exit\_number라는 변수를 선언하고, system call 함수로 Exit이 호출되면, f->esp+4에 있는 값을 thread\_current()의 exit\_number에 저장한다. 그리고 마지막으로 thread.c에 정의된 함수인 thread\_exit()를 호출하고 해당 함수에서는 process\_exit()을 호출하여 exit을 수행한다.

1. Exec

Exec 함수에서는 process\_execute() 함수를 호출한다. 해당 함수는 process.c에 정의되어있으며, argument에는 실행할 명령어와 인자들이 string 형태로 pass된다. Process\_execute에서는 앞서 argument passing을 구현하며 file\_name을 parsing하여 filesys\_open을 통한 예외처리를 한 이후에 thread\_create를 하고, 해당 tid가 TID\_ERROR인 경우에 free\_page를 해준다.

1. Wait

텍스트이(가) 표시된 사진

자동 생성된 설명 Wait 함수에서는 process\_wait() 함수를 호출하고, 이는 process.c에 구현하였다.

그림 11> process.c – process\_wait()

개발 방법에서 설명한대로 threads dir의 thread.h에 정의되어있는 thread 구조체에 sync.h에 정의되어 있는 semaphore 구조체와 lib/kernel dir의 list.c에 정의되어있는 list 구조체 및 list\_elem 구조체를 추가해주었다. List 구조체변수로 선언한 list\_child는 doubly linked list로 구현된 자료구조로써 해당 process의 child들을 저장한다. 그리고 list\_elem 구조체 변수로 선언한 element에는 현재 process의 메모리 주소가 저장되어 있다. 새로 선언한 이 변수들을 thread.c의 init\_thread 함수에서 USERPROG에대한 전처리기 안에서 그 값을 초기화해주었다. Semaphore 구조체 변수를 pblock과 defer\_block 두개를 선언해 주었는데, 이 이유는 child의 메모리 해제와 연관이 있다. 이를 설명하기에 앞서 먼저 process\_wait 함수로 돌아가면, 위 그림에서 볼 수 있듯이 parent process의 child들이 저장되어 있는 doubly linked list의 시작점과 끝점을 각각 cur, end가 처음에 가르키고 있으며, 조건문을 통해 wait할 child가 없거나, child의 pid가 TID\_ERROR일시에 return -1을 해주었다. 조건문에 걸리지 않고, 정상적으로 child process를 wait해야할 경우에는 while문에서 wait해야할 pid의 child를 찾아 해당 child process를 기다린다. 이 때 sema\_down이 사용되는데, 이는 해당 프로세스를 blocked상태로 만든다. Pblock에 의해 parent process가 child process를 기다리고 있는 상황에서 child process가 모든 작업을 마치고 정상적으로 종료될 때, process\_exit()함수에서 pblock을 sema\_up시켜주어 다시 parent process를 unblock 시켜준다. Sema\_up에서는 thread\_unblock 함수를 호출하는데, 해당 함수에서는 process의 상태를 ready 상태로 바꾸어준다. 다시 돌아와서, child process가 정상적으로 종료되면 해당 메모리를 해제해야하는데, child process가 parent process가 다시 running되기전에 먼저 죽어버리면 메모리를 remove할 수 없게된다. 이것이 thread 구조체에 semaphore 구조체 변수를 2개 선언한 이유이다. Defer\_block 변수는 child process가 process\_exit() 함수에서 pblock을 다시 sema up시켜준 다음 sema down되어, child process가 blocked되는 것을 지연시킨다. 이 후 다시 parent process의 코드가 실행되어, list\_remove를 통해 child process의 메모리 해제를 완료하면, 그 다음에야 child process를 defer\_block sema up을 통해 완전히 종료시켜준다.

1. Read

Read 함수에서는 stdin만을 구현하였다. File descriptor 0에 해당하며, 따라서 argument로 넘어온 fd값이 0이 아닐시에는 -1을 return하였다. Argument로 넘어온 buffer와 size는 각각 내용을 저장할 버퍼와 크기이고, 이를 이용해 while문을 돌려 input\_getc()함수를 사용하여 read를 수행한다. 이 때 unit8\_t 자료형으로 casting함으로써 void형인 buffer에 character 단위로 저장하고, 총 읽어들인 byte수를 return한다.

1. Write

Write 함수에서는 stdout만을 구현하였고, 이는 file descriptor 1에 해당한다. Read에서 예외처리를 해준것 처럼 argument fd값이 1이 아닌 경우 return -1을 해주었다. Buffer는 write을 할 공간이고, size는 write할 size를 의미한다. Lib/kernel dir의 consol.c에 구현이 되어있고 stdio.h에 정의되어 있는 putbuf 함수를 사용하여 해당size만큼 buffer에 write를 수행하고, write한 size를 return 한다.

1. 텍스트이(가) 표시된 사진

   자동 생성된 설명Additional System calls

그림 12> lib/user – sycall.c – syscall4

왼쪽 그림은 max\_of\_four\_int 함수가 4개의 arguments를 사용하기 때문에 새로 정의한 매크로이다. NUMBER 파라미터는 해당 system call number를 의미하는데, 이는 lib dir의 syscall-nr.h에 정의된 enum type의 system call number이다. 이 파일에서 SYS\_FIBO 와 SYS\_MAXFOUR를 추가해주었다. Lib/user dir의 syscall.h에 해당 추가 시스템 함수를 정의해주었고, syscall.c 에서도 각각 fibonacci는 syscall1을 하고, max\_of\_four\_int는 syscall4를 해주었다. 이를 통해 userprog dir의 syscall.c가 호출되면 해당 함수에서 switch문에 의해서 각각 fibonacci 함수와 max\_of\_four\_int함수가 호출된다. Fibonacci 함수에서는 argument로 넘겨받은 n값에 대한 피보나치 수열값을 reuturnt 하며, max\_of\_four\_int 함수에서는 3번의 비교를 통해서 4개의 arguments 중 가장 큰 값을 return 한다. 해당 fibonacci 함수와 max\_of\_four\_int 함수는 명령어 additional에의해 실행되는데, 이에대한 test를 하기위해서 examples dir에 additional.c 파일을 만들었고, 해당 파일에서 fibonacci와 max\_of\_four\_int에대한 시스템 콜을 하였다. 이를 컴파일하기 위해 makefile도 바꿔주었다.

* 1. **시험 및 평가 내용**
* **텍스트이(가) 표시된 사진

  자동 생성된 설명fibonacci 및 max\_of\_four\_int**

그림 13> pintos --filesys-size=2 -p ../examples/additional -a additional -- -f -q run 'additional 10 20 62 40' 에 대한 실행 결과

위 결과를 보면 가장 첫번째 argument 10에 대한 fibonnaci 수열값 55와 4개의 arguments중 가장 큰 값 62가 정상적으로 출력된것을 확인할 수 있다.

* **Make check에 의한 results**

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

그림 14> make check 결과 build dir의 results

해당 프로젝트에서 pass되어야할 21개의 teset가 위 그림에서 볼 수 있듯 모두 pass됨을 확인하였다.