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

담당 교수 : 박성용 교수님

조 / 조원 : 20181703 황동준

개발 기간 : 2020-10-03 ~ 2020-11-01

1. **개발 목표**

이번 프로젝트의 개발 목표는 pintos 내에 있는 구성 요소들을 만드는 것인데, 실행될 부분의 argument를 자르고 dump에 그 argument들을 올리는 방식의 코드를 작성하고, 이후에 system call을 구현하는 것이다. Pintos를 본격적으로 시작하기 전에 구성요소를 만들어 놓는 것이 목표이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

아래의 순서대로 개발을 실행할 예정이며, Additional Implement는 system call에서 추가적으로 구현할 사항이다.

1. Argument Passing
2. User Memory Access
3. System Call Handler
4. System Call Implementation
5. Additional Implementation
6. Argument Passing

Argument Passing을 구현 했을 때, esp를 움직이면서 받은 Argument들을 값, 주소, return address 순서로 쌓게 된다. Stack에 쌓이게 되면 user 모드에서 이를 이용할 수 있고, 받은 Argument들은 아래에서 언급할 System Call에서 그 역할을 수행하게 된다. Memory에 Allocation 할 때는 저장하는 환경이 pintos이므로 80x86 Calling Convention을 이용할 것이다.

1. User Memory Access

이 부분에서는 User thread안에서 유효하게 메모리 값에 접근을 하고 있는지 파악할 것이다. User Thread에서 Kernel 영역에 접근하고 있지는 않는 지, 할당된 메모리 이외의 부분에 접근을 하고 있지는 않는지 등등의 memory validity를 확인하여 만약 다른 영역에 침범했을 경우 오류를 띄워주는 역할을 할 것이다.

1. System Calls

System call 부분에서는 Thread가 System call로 kernel 영역과 소통할 때 어떠한 system call이 들어와서 어떤 역할을 해내는 지를 구현할 것이다. 또한 thread가 어떻게 부모, 자식 프로세스를 갖고, 실행순서를 맞추는지 알아볼 것이다. 우리가 짤 코드는 wait, exec, halt, exit, write 그리고 additional implementation등이 있다. 아직 open close와 같은 부분은 작성하지 않을 예정이다.

* 1. **개발 내용**
* Argument Passing

Pintos에서는 80x86 Calling Convention이라는 메모리 구조를 가지고 있다. 따라서 process 영역에서 Argument를 받으면, 이를 strtok\_r로 자른 후에 자른 것들 각각에 NULL 값을 붙여 argument를 memory 에 esp(stack pointer)를 이용하여 저장해야 한다.



먼저, space bar를 기준으로 strtok\_r 한 argument들을 마지막 부분부터 쌓게 된다. 즉 받은 argument의 가장 마지막부터 esp를 옮겨 가면서 stack에 쌓고 그 다음 memory가 16진수 형태로 저장이 되어야 하므로, 남는 공간 즉, 메모리를 쌓다가 4 \* 16 bit으로 맞춰지지 않은 것들은 모두 0으로 초기화 시키고, 이제 각 argument들의 주소를 쌓는다. 그래야 하기 때문에 미리 위에서 argument를 저장해 놓을 때, 각각의 address 를 저장해 놓아야 할 것이다. Address 또한 마지막 argument의 값부터 쌓고 마지막으로 저장된 주소의 주소 값을 저장한다. 그 이후 받은 argument의 개수를 넣고 return address(0)을 넣은 후 argument passing을 종료한다. 위의 그림을 참고하면 좋다.

* User Memory Access

Pintos OS는 두 개의 thread 로 크게 나눌 수 있는데, kernel 영역과 user 영역으로 나눌 수 있다. kernel영역은 많은 user thread들을 관리하는 역할을 하며, 또한 user thread의 공유 영역이므로 이의 메모리를 침범해서는 안된다. 따라서 user 영역에서는 kernel 영역으로의 접근을 막기 위해서 접근하려는 Memory가 Kernel영역이라면 그러한 행동을 차단하여야 한다. 또한, User thread에는 각각 할당된page, 영역이 있는데, 그 영역을 벗어나게 되면, 각각의 User영역의 Memory가 독립적이지 않기 때문에 할당된 메모리 이외의 공간을 쓰면 안된다. 따라서 invalid memory acccess는 이러한 할당되지 않은 공간에 접근하는 것을 의미하며, 이러한 Invalid memory access를 막으려면 두 가지 방법이 있다. User Memory가 어디까지 할당되었는지 User-provided pointer를 통해 알아보고 invalid 한 메모리에 접근하지는 않았는지 확인하는 방법과, user pointer가 PHYS\_BASE보다 아래의 영역에 있는지 (kernel 영역) 확인한 후 dereference 하는 것이다. 그리고 우리는 이것 이외에 User 영역의 Virtual Memory가 mapping이 안된 부분에 접근하면 안되기 때문에 이 또한 주어진 헤더의 함수를 통해서 찾아낼 것이다.

* 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

시스템 콜은 User 영역과 Kernal 영역이 syscall\_handler를 통해서 해당 일을 수행할 수 있도록 하는 역할을 한다. User가 kernel 의 기능을 사용할 수 있도록 하는 것이다. Operating System과 User Program이 서로 소통할 수 있도록 해주는 역할을 System Call이 하게 된다.  
이번 프로젝트에서 만들 system call을 아래에 나열할 것이다.

1. Halt

Halt는 말 그대로 해당 Thread의 활동을 중지시키는 것을 말한다. Shutdown\_power\_off()라는 함수를 이용하여 User System을 Halt할 것이다.

1. Execute

이는 해당 User Process에서 다른 Child Process를 만드는 것을 의미한다. Thread 내에서 thread를 만드는 것이므로 계층구조가 생성될 것이고, child와 parent간의 semphore도 이용될 것이다.

1. Wait

Process\_exec 에 이용되는 함수이며, 계층 구조에서 다른 Child Process가 실행되고 있을 때 parent 가 wait 하는 경우에 주로 이용된다.

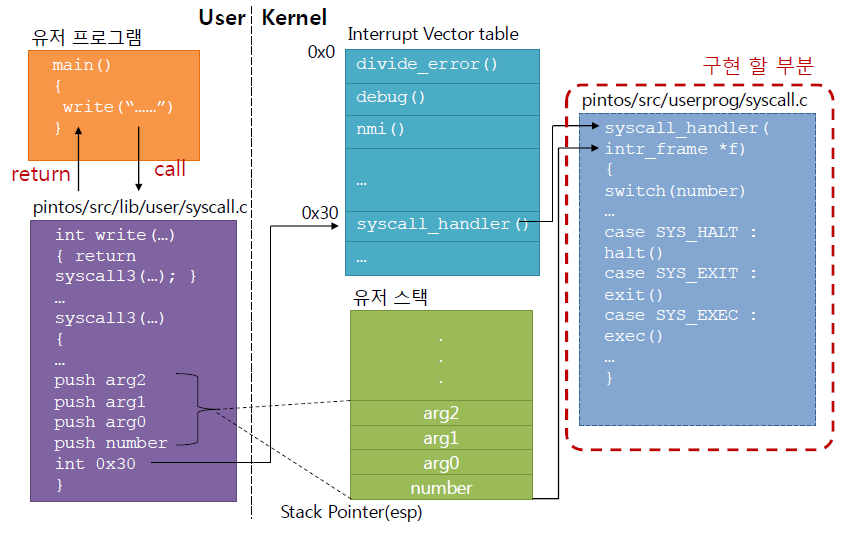
1. Write & Read

Buffer 과 size를 받고 Std\_In인지 Out인지 판별하여 Buffer에 size 만큼의 값을 pufbuf, input\_get\_c 해주는 역할을 한다. 이는 이번 System call에서는 자세히 구현하지 않을 것이다.

1. Fibonacci & Max\_of\_four\_int

Fibonacci는 Fibonacci 한 number을 return 해주는 역할을 하는 함수이며, Max\_of\_four\_int 는 4개의 수를 받아서 가장 큰 값을 찾아내는 함수로 System call에 추가 구현할 사항이다.

System Call에서는 먼저 User Program에서 System Call API를 호출하게 된다. 그 이후에, System Call API에서 User Stack에 Push를 하여 받은 각각의 Argument들을 모두 쌓아준다음 int 0x30 을 통해 Interrupt Vector Table의 system call handler 주소로 이동한 후, system call handler가 우리가 만든 System Call 에 접근하여 해당 입력받은 System Call을 수행하게 된다. 이때 우리는 esp 라는 User Stack Pointer을 받아와서 API에서 넘겨받은 Argument들을 이용할 수 있게 된다.



1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* 2020-10-03 ~ 2020-10-21 OS와 Pintos에 관련된 내용을 이해하고 구현할 내용의 구조 작성
* 2020-10-22 ~ 2020-10-28 Argument Passing & User Memory Access 부분 코드를 작성
* 2020-10-28 ~ 2020-10-30 System Call 부분 코드 작성
* 2020-10-30 ~ 2020-11-01 보고서 작성
  1. **개발 방법**
* Argument Passing

먼저, 받은 argument를 자르는 역할을 하는 것은 load에 자르지 않은 argument를 넘겨준 이후에 load안에서 file을 open할 때 할 것이다. 왜냐하면 file을 open 할 때는 첫번째 argument를 open 해야 하기 때문이다. 그 이후 자른 argument를 Stack에 쌓는 구조는 Stack setup 이 끝난 후 하게 될 것이다. 먼저 쌓은 Stack의 address 를 저장하는 memory를 하나 만들어 줄 것이다. 그 이후 초기화한 esp를 이동하면서 위에서 말했던 방식으로 data를 하나씩 쌓는다. Address 를 넣을 때는 char pointer로 선언하였으므로 해당 형식으로 넣어주고, argument count를 넣을 때는 int 형으로 변환하여 pointer에 넣어준다. 또한 이러한 data가 stack에 잘 들어갔는지를 확인하기 위해서 hex\_dump를 이용해 값을 수시로 찍어준다.

* User Memory Access

이를 구현할 때는 User Memory에 들어갔을 때 처리해 주어야 한다. 따라서 System Call에서 구현을 하게 된다. Vaddr, exception 등에 있는 is\_kernel\_vaddr 과 pagedir\_get\_page() 를 이용해서 kernel에 있는 virtual memory address인지 확인하고 pagedir\_get\_page를 통해서 mapping 되지 않은 memory에 접근하는 지 확인한다. 이때는 system call에 들어가고 만약 User Stack에 접근을 할 때 접근 하기 전에 이 함수를 이용해서 먼저 esp를 체크해줘야 한다. 또한 각각의 system call에서 program 이외의 다른 argument에 접근할 때 또한 그 접근 값이 valid 한지를 따져야 하기 때문에 위의 방법을 이용한 코드를 추가해 주어야 한다.

* System Call

먼저 선언되지 않은 library 를 system call header에 추가하고 나머지 system call을 구현한다.

1. Halt

Halt 에는 shutdown\_power\_off() 함수를 이용해서 halt를 시도한다.

1. Exec

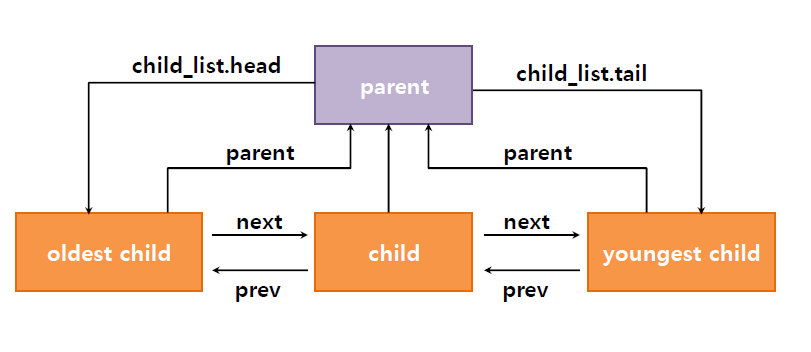
Process\_execute를 실행하여 또다른 child를 만든다. 여기에서는 wait를 이용하여 execute를 하게 되고 semphore를 이용하게 되는데 이는 wait에서 추가로 설명할 것이다. Exec에서는 system call 에서 execute 함수를 다시 선언하게 된다.

1. Exit

Exit 는 함수를 따로 구현해주어 status를 int 값으로 넘겨준다. 이때 마지막에 끝날 때 exit을 부르는 thread의 status를 함수에 넘겨준다. 만약 메모리에 잘 못 접근했거나 오류가 발생하면 -1을 return 하는 코드 또한 추가하게 된다.

4. Wait

Wait 에서는 바꿀 만한 부분이 많다. 먼저 wait 할 동안 해당 thread는 움직일 수 없으므로 semphore을 선언해주어야 하는데, 먼저 thread.h 파일 각각에 semaphore을 선언해 주어 thread 마다 하나의 semaphore을 가질 수 있도록 한다. 또한 기다리는 thread가 list 형태로 저장될 수 있도록 list를 선언해 준다. 그렇게 되면 계층 구조가 생기는 thread들은 다음과 같은 구조를 가지게 된다.



그런다음 memory를 lock해줄 semaphore을 구현하여 child가 죽었을 때의 status를 제대로 받을 수 있도록 한다. Init\_thread 에서 semaphore 들과 list를 초기화 해주는 코드를 작성하고, wait에서 list를 받아와서 만약 list에 child가 있을 경우 sema\_down 이용해서 thread를 잠재우고 child가 다 끝났으면 process\_exit 아래에서 sema\_up을 이용하여 thread를 다시 돌아가게 한다. 그렇게 되면, child process를 remove하고 원래 있던 thread에 exit에서 lock을 걸어줬던 memory를 다시 풀어주게 된다. 그래서 process의 wait와 exit의 코드를 수정해야 한다.

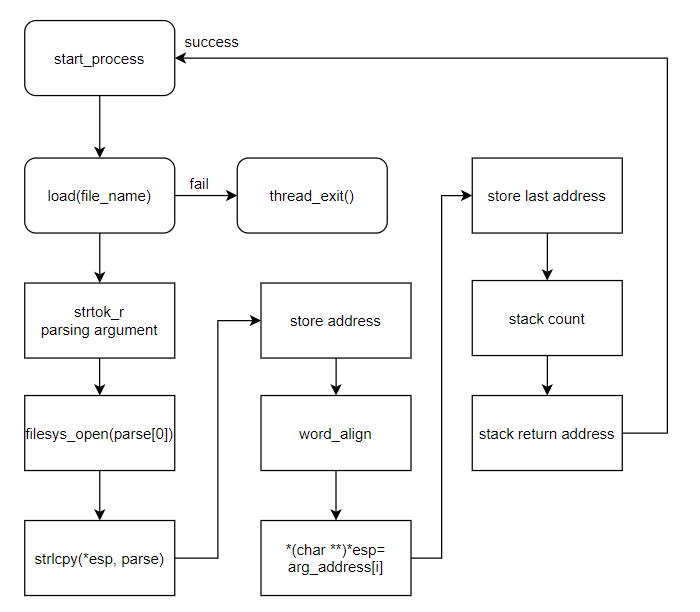
5. Write

Write는 먼저 접근하는 3개의 argument에 대해서 (fd, buffer, size) validity 를 check하는 코드를 쓰고, putbuf 함수를 이용해서 Buffer에 Write를 해주게 된다.

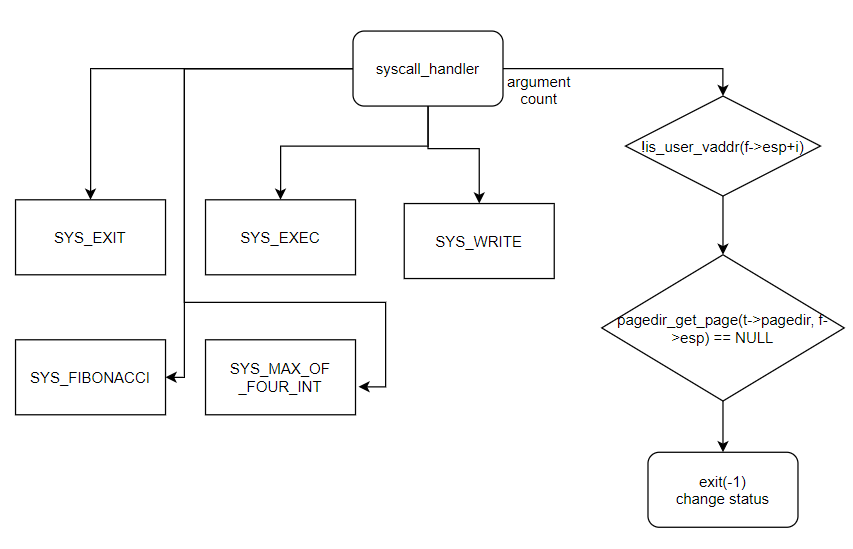
1. Fibonacci & Max\_of\_four\_int

이 또한 각각 1개, 4개의 argument의 validity를 check한다. 이때 각각에 맞는 함수를 만들어 주고 그 결과 값을 return 한 후 eax에 저장하는 코드를 작성해야 한다. 또한 아직 example에 additional.c (user program)을 만들어 testcase의 출력을 할 수 있도록 한다. 또한 system call의 API에 4개 argument를 받는 API를 추가하고 이를 선언하여 Max of four int에 그것을 쓸 수 있도록 한다.

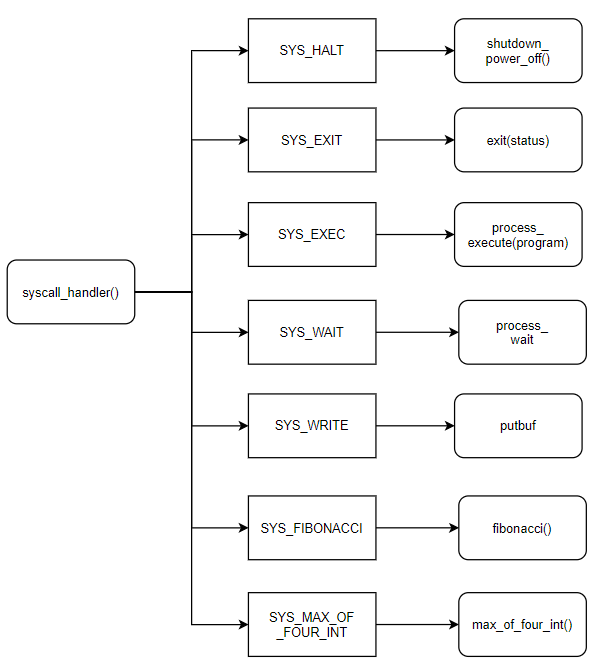
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing

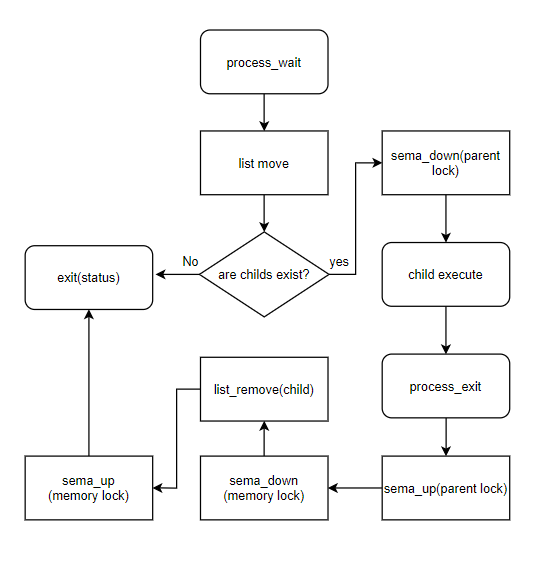


1. User Memory Access



1. System Calls





* 1. **제작 내용**

1. Argument Passing

|  |
| --- |
| bool load (const char \*file\_name, void (\*\*eip) (void), void \*\*esp) |
| /\* Argument passing (user) \*/  //esp is stack pointer of user stack(have value already)  //we must stack the top of pointer reverse  char \*\*arg\_address;  int word\_align\_num=0;  char \*word\_align;  arg\_address = (char \*\*)malloc(sizeof(char \*) \* cnt\_tok);  for(i=cnt\_tok-1;i>=0;i--){  \*esp -= (strlen(parse[i])+1); // 1 take NULL pointer  strlcpy(\*esp,parse[i],strlen(parse[i])+1);  arg\_address[i] = \*esp;  word\_align\_num += (strlen(parse[i])+1);  }  //word align part  word\_align\_num = 4 - word\_align\_num % 4;  \*esp -= word\_align\_num;  word\_align = (char \*)calloc(word\_align\_num, sizeof(char));  strlcpy(\*esp, word\_align, word\_align\_num);  //stack address  \*esp -= 4; \*(char\*\*)\*esp = 0;  for(i = cnt\_tok-1; i>=0; i--){  \*esp -= 4;  \*(char\*\*)\*esp = arg\_address[i];  }  //stack last address  \*esp -= 4;  \*(char \*\*\*)\*esp = (char \*\*)((\*esp)+4);    //stack cnt\_tok  \*esp -= 4;  \*(int \*)\*esp = cnt\_tok;  //stack return address  \*esp -= 4;  \*(char \*\*)\*esp = (char \*) 0; |
| 위 부분의 parsing 하는 부분은 코드를 생략하였다. 이는 strtok\_r(because of security)를 이용하여 할당한 메모리에 이를 붙여넣고 array에 집어넣는 역할을 한다.  위에서 설명했듯이, esp라는 stack pointer를 사용하여 움직인 후 이를 차곡차곡 쌓아서 data stack에 쌓게 된다. 따라서 setup된 \*\*esp의 주소인 \*esp를 움직이면서 argument를 넣을 때는 strlcpy 를 이용해서 argument의 길이만큼 복사해주고 복사한 이후에는 argument의 address 값 전체를 할당한 다른 메모리에 저장해 놓게 된다. Argument를 모두 넣은 이후에는 word\_align으로 4개 단위로 끊는 16진수의 남은 부분에 0을 채워주게 된다. 이후에 arg\_address에 저장해 놓았던 address를 char \* 형태로 esp에 할당해 주게 된다. 이후에 주소를 쌓은 stack의 마지막 address를 쌓고 int \*형으로 argument count를 넣어준 후 0이라는 return address를 쌓는다. |

Error Solution

|  |
| --- |
| char file\_name\_sub[100];  strlcpy(file\_name\_sub, file\_name, strlen(file\_name)+1);  char \*parse[100];  char \*token, \*ptr\_tok;  int cnt\_tok = 0;  if((token = strtok\_r(file\_name\_sub," ", &ptr\_tok))!=NULL){  parse[cnt\_tok++] = token;  while((token = strtok\_r(NULL, " ", &ptr\_tok))!=NULL)  parse[cnt\_tok++] = token;  }  /\* Open executable file. \*/  file = filesys\_open (parse[0]); |
| 위에서 설명하였을 때 parsing 부분이 위의 코드이다. 여기서 원래 처음 example 파일인 echo가 open 되지 않는다고 오류가 뜨게 되었다. 그래서 parsing 한 부분을 다시 보니 filesys\_open에 (file\_name)이 그대로 들어가게 된 것이다. File\_name은 parsing 하지 않고 echo x를 전부 넣은 것이다. 따라서 parsing 한 parse[0]을 넣어주어 진짜 program name을 넣어주어야 한다. |

|  |
| --- |
| for(i = cnt\_tok-1; i>=0; i--){  \*esp -= 4;  \*(char\*\*)\*esp = arg\_address[i];  } |
| 처음에 address를 어디에 넣어야 할 지 몰라서 strlcpy를 이용해 그대로 붙여넣어 보았지만 안되길래 arg\_address에 주소 값을 넣고 이를 그대로 붙여넣어서 입력을 한 결과 성공했다. |

추가로 구현한 함수나 이용한 라이브러리는 없다.

1. User Memory Access

|  |
| --- |
| static void syscall\_handler (struct intr\_frame \*f UNUSED) |
| if(!is\_user\_vaddr(f->esp)||pagedir\_get\_page(t->pagedir, f->esp) == NULL)  exit(-1);  //system call  //1 : halt, exit, exec, wait, read, write  int system\_number = \*(int \*)(f->esp);  switch (system\_number){  case SYS\_EXIT:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  case SYS\_EXEC:  if(!is\_user\_vaddr(f->esp) || pagedir\_get\_page(t->pagedir, f->esp) == NULL)  exit(-1);  if(!is\_user\_vaddr(\*(char\*\*)(f->esp+4)) || pagedir\_get\_page(t->pagedir, \*(char\*\*)(f->esp+4)) == NULL)  exit(-1);  case SYS\_WAIT:  if(!is\_user\_vaddr(f->esp) || pagedir\_get\_page(t->pagedir, f->esp) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  case SYS\_WRITE:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+8) || pagedir\_get\_page(t->pagedir, f->esp+8) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+12) || pagedir\_get\_page(t->pagedir, f->esp+12) == NULL)  exit(-1);  if(!is\_user\_vaddr((void \*)\*(uint32\_t \*)(f->esp+8) || pagedir\_get\_page(t->pagedir, (void \*)\*(uint32\_t \*)(f->esp+8)) == NULL))  exit(-1);  if((int)\*(uint32\_t \*)(f->esp+4) == 1){  }  break;  case SYS\_READ:  break;  case SYS\_FIBONACCI:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  break;  case SYS\_MAX\_OF\_FOUR\_INT:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+8) || pagedir\_get\_page(t->pagedir, f->esp+8) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+12) || pagedir\_get\_page(t->pagedir, f->esp+12) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+16) || pagedir\_get\_page(t->pagedir, f->esp+16) == NULL)  exit(-1);  break;  }  } |
| User Access memory는 수시로 esp가 올바른 argument에 접근하는지 판단하여야 한다. 왜냐하면 esp가 만약 다른 곳으로 넘어간다면, kernel PANIC이 일어나기 쉽기 때문이다. 따라서 if(!is\_user\_vaddr(f->esp)||pagedir\_get\_page(t->pagedir, f->esp) == NULL) exit(-1); 과 같은 코드를 선언하여 page와 user virtual address에 접근하고 있는지 확인해준다. 이를 system call의 모든 부분에서 syscall API 가 몇 개의 Argument를 받고 있는지 확인하여 esp의 어느 부분까지 확인을 해줘야 하는지 파악하고 이를 그대로 확인하면 된다. 이를 따로 함수로 구현하지 않은 이유는 함수로 넘겼을 때 주소 값을 잘못 넘겨 받아 낼 수도 있기 때문이다. |

이용한 라이브러리 함수

|  |
| --- |
| vaddr.h |
| static inline bool  is\_user\_vaddr (const void \*vaddr)  {  return vaddr < PHYS\_BASE;  } |
| 이 함수는 PHYS\_BASE 아래에 User thread의 address가 존재하는지를 파악하는 코드이다. 따라서 esp를 넘겨 받고 이 포인터가 가리키는 주소 값이 kernel 영역의 것인지 user 영역의 것인지 알게 해준다. |

|  |
| --- |
| pagedir.c |
| void \*  pagedir\_get\_page (uint32\_t \*pd, const void \*uaddr)  {  uint32\_t \*pte;  ASSERT (is\_user\_vaddr (uaddr));    pte = lookup\_page (pd, uaddr, false);  if (pte != NULL && (\*pte & PTE\_P) != 0)  return pte\_get\_page (\*pte) + pg\_ofs (uaddr);  else  return NULL;  } |
| 이는 esp 와 thread의 pagedir 를 받아와서 할당받은 페이지가 자신에게 할당된 page 범위 내인지 파악하게 해준다. |

Error Solution

|  |
| --- |
| syscall.c |
| void exit(int status){  struct thread \*t = thread\_current();  t->exit\_status = status;  printf("%s: exit(%d)\n", t->name, status);  thread\_exit();  } |
| 이는 System call에서도 해당하는 내용이다. 그러나 User Memory Access와 그 의미가 일치하여 일단 추가를 하였다. 처음에 thread의 status가 정상 종료됐음에도 불구하고 0 status가 나오지 않아서 thread\_exit()의 순서가 바뀌었다는 것을 알고 thread\_exit을 가장 마지막에 선언함으로서 정상적으로 status를 띄울 수 있었다. |

1. System Calls

|  |
| --- |
| syscall.c |
| int system\_number = \*(int \*)(f->esp); |
| 다음과 같은 형식으로 system number을 가져왔다. 이는 당연하게도 User Stack에 오면서 쌓인 첫번째 program을 interrupt table에서 handler로 넘어가면서 할당된 System call number을 esp에 저장하였으므로, 이를 system\_number 에 넣겠다는 의미이다. |

1. HALT

|  |
| --- |
| syscall.c |
| case SYS\_HALT: shutdown\_power\_off(); break; |
| shutdown\_power\_off를 이용하여 Halt를 한다. 이용한 라이브러리는 다음과 같다. |

|  |
| --- |
| devices/shutdown.c |
| void  shutdown\_power\_off (void)  {  const char s[] = "Shutdown";  const char \*p;  #ifdef FILESYS  filesys\_done ();  #endif  print\_stats ();  printf ("Powering off...\n");  serial\_flush ();  /\* ACPI power-off \*/  outw (0xB004, 0x2000);  /\* This is a special power-off sequence supported by Bochs and  QEMU, but not by physical hardware. \*/  for (p = s; \*p != '\0'; p++)  outb (0x8900, \*p);  /\* For newer versions of qemu, you must run with -device  \* isa-debug-exit, which exits on any write to an IO port (by  \* default 0x501). Qemu's exit code is double the value plus one,  \* so there is no way to exit cleanly. We use 0x31 which should  \* result in a qemu exit code of 0x63. \*/  outb (0x501, 0x31);  /\* This will power off a VMware VM if "gui.exitOnCLIHLT = TRUE"  is set in its configuration file. (The "pintos" script does  that automatically.) \*/  asm volatile ("cli; hlt" : : : "memory");  /\* None of those worked. \*/  printf ("still running...\n");  for (;;);  } |
| 프로그램을 종료해주는 역할을 하게 된다. |

Error Solution

|  |
| --- |
| syscall.c |
| //thread\_exit(); |
| 처음에 HALT 마저 make check에 안 넘어 갔던 시기가 있었다. Argument passing은 dump를 찍어보았을 때 매우 잘 넘어간 것을 알 수 있었다. Syscall\_handler에서 dump를 찍고 보니 system\_number 가 9 한번만 넘어가고 안 넘어간다는 것을 알 수 있었다. 처음의 number 9는 begin 때문에 넘어간 것이고 그 이후에 thread가 그냥 종료 되버린 것이다. 이는 기존 코드에 있던 thread\_exit()을 주석처리한 후에 없어졌다. 그래서 thread\_exit은 오직 exit 함수 안에서만 쓰는 것으로 정했다. |

1. EXEC

|  |
| --- |
| syscall.c |
| case SYS\_EXEC:  if(!is\_user\_vaddr(f->esp) || pagedir\_get\_page(t->pagedir, f->esp) == NULL)  exit(-1);  if(!is\_user\_vaddr(\*(char\*\*)(f->esp+4)) || pagedir\_get\_page(t->pagedir, \*(char\*\*)(f->esp+4)) == NULL)  exit(-1);  f->eax = process\_execute(\*(char \*\*)(f->esp +4)); break; |
| 이는 f->esp 에 해당 thread 의 status가 저장되어 있어 이를 process\_execute를 해주게 된다. Process\_execute를 통해 실행하면서 child가 생성되고 child가 생성될 때까지 wait를 실행하게 되는데 이에 대해서는 wait에서 설명을 할 것이다. |

이용한 라이브러리 함수

|  |
| --- |
| process\_execute (const char \*file\_name) |
| /\* user \*/  char \*file\_name\_ptr = NULL, \*ptr\_tok;  file\_name\_ptr = palloc\_get\_page (0);  strlcpy(file\_name\_ptr, file\_name, PGSIZE);  strtok\_r(file\_name\_ptr, " ", &ptr\_tok);  if(filesys\_open(file\_name\_ptr) == NULL) return -1; |
| 이 함수를 이용할 때는 우리가 입력받은 파일 이름을 open 하고 실행하는 역할을 한다. 위에는 길어서 안나와 있지만 page 할당, Error check들의 역할을 한다. |

Error Solution

|  |
| --- |
| process\_execute (const char \*file\_name) |
| if(filesys\_open(file\_name\_ptr) == NULL) return -1; |
| 이는 exec\_missing 부분의 파트를 해결할 수 있었다. 이는 execution 하는 파일의 이름이 만약 invalid한 이름이었고 실제로 존재하지 않는 파일 명이라면 에러를 return 하여 thread execution을 멈춰 버리는 것이다. 이것을 실행하지 않아 처음에 open 되지 않은 파일도 thread status 값을 가져 해결하기 상당히 힘들었다. |

|  |
| --- |
| process\_execute (const char \*file\_name) |
| char \*file\_name\_ptr = NULL, \*ptr\_tok;  file\_name\_ptr = palloc\_get\_page (0);  strlcpy(file\_name\_ptr, file\_name, PGSIZE);  strtok\_r(file\_name\_ptr, " ", &ptr\_tok); |
| 처음에 thread를 create 할 때 앞에서 file\_name을 parsing 해주지 않고 execution으로 들어오게 되어 thread->name에 parsing 되지 않은 argument 값이 들어있게 되었다. 따라서 create 앞에 프로그램 이름만 parsing하여 그 부분을 해결하였다. |

1. EXIT

|  |
| --- |
| syscall.c |
| case SYS\_EXIT:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  exit(\*(uint32\_t \*)(f->esp+4)); break; |
| 해당 thread에서 나가는 부분을 맡고 있다 이때 exit의 안에 들어가는 값은 나가려는 thread의 status이다. 이를 넘겨서 프로그램이 정상종료되었는지 아닌지를 파악한다. |

이용한 라이브러리 함수

|  |
| --- |
| syscall.c |
| void exit(int status){  struct thread \*t = thread\_current();  t->exit\_status = status;  printf("%s: exit(%d)\n", t->name, status);  thread\_exit();  } |
| 이 함수를 이용해서 먼저 thread를 구하고 exit\_status라는 int 형 자료를 thread에 추가하여 status를 저장할 수 있게 끔 만들었다. 그 이유는 child 의 exit\_status를 parent가 볼 수 있기 때문이기도 하며, print한 status와 비교해보기 위함도 있다. 그리고 마지막에 이러한 status를 저장하고 thread\_exit(). |

Error Solution

User Memory Access 부분에서 언급하였다.

1. WAIT

|  |
| --- |
| syscall.c |
| case SYS\_WAIT:  if(!is\_user\_vaddr(f->esp) || pagedir\_get\_page(t->pagedir, f->esp) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  f->eax = process\_wait(\*(tid\_t \*)(f->esp+4)); break; |
| process wait는 말 그대로 해당 thread를 wait하게 만든다. wait에서 넘겨주는 tid는 f->esp+4인데 위에서 언급했던 execution에서의 설명과 일치한다. Child의 값을 넘겨주어 child가 실행중일 때는 parent를 wait하게 한다는 것을 의미한다. execution에서 실행을 하였으면, wait에서 넘어와 child의 tid를 받고 이를 list에 담겨져 있는 것들과 비교하여 child process가 존재한다면 parent를 semaphore을 이용하여 lock 시키는 것이다. |

이용한 라이브러리 함수

|  |
| --- |
| process.c |
| int  process\_wait (tid\_t child\_tid)  {  //int i;  //for(i=0;i<1000000000;i++);  struct list\_elem \*e;  struct thread \*t = NULL;  int exit\_status;  for(e = list\_begin(&(thread\_current()->child)); e != list\_end(&(thread\_current()->child));e = list\_next(e)){  t = list\_entry(e, struct thread, child\_elem);  if(child\_tid == t->tid){  sema\_down(&(t->child\_lock));  exit\_status = t->exit\_status;  list\_remove(&(t->child\_elem));  sema\_up(&(t->mem\_lock));  return exit\_status;  }  }  return -1;  } |
| process\_wait라는 함수를 SYS\_WAIT에서 선언하게 된다. 이는 child\_tid가 들어왔을 때 이 child\_tid가 entry list에 들어있는지 확인하고 만약 들어있다면, sema\_down을 이용해서 parent에 lock을 걸어준다. 이때 list\_entry는 list에 linked 되어있는 thread를 가져오는 역할을 하게 된다. 그리고 그 thread의 tid와 받아온 tid를 비교하는 것이다. Exit\_status는 위에서 언급한 대로 thread의 구조체에 추가한 변수이다. 이를 이용해서 thread가 exit 될 때 어떤 status을 가지고 나오는 지 알 수 있다. Child가 끝났을 경우 아래의 exit으로 넘어가게 된다. |

|  |
| --- |
| process.c |
| void process\_exit (void)  {  struct thread \*cur = thread\_current ();  uint32\_t \*pd;  /\* Destroy the current process's page directory and switch back  to the kernel-only page directory. \*/  pd = cur->pagedir;  if (pd != NULL)  {  cur->pagedir = NULL;  pagedir\_activate (NULL);  pagedir\_destroy (pd);  }  sema\_up(&(cur->child\_lock));  sema\_down(&(cur->mem\_lock));  } |
| 이를 통해서 만약 child가 끝났을 경우 그 thread의 sema를 다시 풀어 parent가 다시 작동하게 만든다. 이때 parent는 wait에서 list\_remove를 수행하려고 할 것이다. 이때 memory sema를 이용해서 remove 시키려는 list를 제거 할 수 있도록 하고 그 이후에 다시 memory sema를 up 시켜서 메모리 또한 풀어줄 수 있도록 하는 것이다. |

|  |
| --- |
| thread.c |
| #ifdef USERPROG  sema\_init(&(t->child\_lock), 0);  sema\_init(&(t->mem\_lock), 0);  list\_init(&(t->child));  list\_push\_back(&(running\_thread()->child), &(t->child\_elem));  #endif |
| 여기에서는 만들었던 구조체를 초기화 시키는 역할을 한다. Sema\_init는 이미 semaphore.c 에서 정의 되어있는 함수이며, 이를 이용하면 sema의 값을 0으로 초기화 시켜주게 된다. 또한 list\_init도 라이브러리에서 이미 만들어져 있다. 따라서 이런 식으로 해당 thread에 연결되어있는 child thread를 만들기 위한 list를 초기화 할 수 있다. |

|  |
| --- |
| struct thread |
| #ifdef USERPROG  /\* Owned by userprog/process.c. \*/  uint32\_t \*pagedir;  struct semaphore child\_lock;  struct semaphore mem\_lock;  struct list child;  struct list\_elem child\_elem;  int exit\_status;  /\* Page directory. \*/  #endif |
| 여기에서 또한 알 수 있듯이 semaphore 을 따로 선언하여 그에 대한 lock을 만들어 줄 수 있다. List child는 child가 포함 되어있는 list\_element들을 연결해 놓은 것을 말하고, list\_elem는 child의 정보를 가지고 있는 element이다. |

Error Solution

|  |
| --- |
| struct thread |
| #ifdef USERPROG  /\* Owned by userprog/process.c. \*/  uint32\_t \*pagedir;  struct semaphore child\_lock;  struct semaphore mem\_lock;  struct list child;  struct list\_elem child\_elem;  int exit\_status;  /\* Page directory. \*/  #endif |
| 위에서 이미 언급하였던 부분인데, 원래는 이 부분이 선언이 안되어 있다가 추가한 경우라, struct semaphore child\_lock 과 mem\_lock을 선언할 때, definition 되지 않은 type이기 때문에 thread 에서 init을 할 수 없다고 에러가 발생을 했었다. 그러나 이는 thread.h에 #include "threads/synch.h"  를 선언 함으로서 해결 할 수 있었다. 이 include 가 되지 않아서 발생한 오류였다. |

|  |
| --- |
| process\_execute (const char \*file\_name) |
| return tid; |
| 이 부분에서 원래는 return process\_wait(tid)를 통해서 execution이 끝난 이후에 child에서도 또 다른 child가 생성될 수 있기 때문에 wait를 return으로 넘겨주었었는데, 원래는 그럴 필요가 없이 execution이 되면 알아서 wait system call을 불러 주기 때문에 그냥 tid를 불러주면 되는 문제였다. 이는 return 값을 바꾸면서 해결할 수 있었던 문제였다. |

1. WRITE

|  |
| --- |
| syscall.c |
| case SYS\_WRITE:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+8) || pagedir\_get\_page(t->pagedir, f->esp+8) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+12) || pagedir\_get\_page(t->pagedir, f->esp+12) == NULL)  exit(-1);  if(!is\_user\_vaddr((void \*)\*(uint32\_t \*)(f->esp+8) || pagedir\_get\_page(t->pagedir, (void \*)\*(uint32\_t \*)(f->esp+8)) == NULL))  exit(-1)  if((int)\*(uint32\_t \*)(f->esp+4) == 1){  putbuf((void \*)\*(uint32\_t \*)(f->esp+8), \*(size\_t \*)(f->esp+12));  } |
| write는 총 3개의 argument를 받게 되는데 먼저 STD\_IN, STD\_OUT을 정하는 fd, 쓰거나 읽어야 하는 memory인 buffer, 쓰거나 읽어야 하는 memory의 크기인 size이다. 총 3개의 argument를 check 해야함으로 User Access Memory에서 말했다시피 3개의 변수 모두 validity를 check해주었다.  또한 fd의 값은 f->esp+4에 저장이 되게 되는데, 이 값이 1이면 STD\_IN을 의미하므로, Write를 하게 된다. 이때 Write에는 putbuf라는 라이브러리 함수를 이용하게 된다. |

이용한 라이브러리 함수

|  |
| --- |
| lib/kernel/console.c |
| void  putbuf (const char \*buffer, size\_t n)  {  acquire\_console ();  while (n-- > 0)  putchar\_have\_lock (\*buffer++);  release\_console ();  } |
| 이 함수를 이용해서 해당 buffer에 데이터를 저장할 수 있게 된다. 해당 함수는 c언어에서 fgets와 같은 역할을 담당하게 되는 것이다. |

Error Solution

Write를 짤 때 추가적인 Error는 발생하지 않았다.

1. Additional System calls

|  |
| --- |
| lib/user/syscall.h && lib/syscall-nr.h |
| /\* user \*/  int fibonacci(int n1);  int max\_of\_four\_int(int n1, int n2, int n3, int n4);  /\* user \*/  SYS\_FIBONACCI,  SYS\_MAX\_OF\_FOUR\_INT |
| 선언한 함수와 헤더들은 모두 additional의 함수를 구현하기 위한 것들이다. 먼저 SYS\_NUMBER들은 System call Handler으로 넘어갈 각각의 program에 대한 number을 의미한다.  또한 fibonacci에서 선언한 것들은 모두 system call API에서 System call handler로 넘어갈 것이다. Return 값 또한 int로 선언하였다. |

|  |
| --- |
| lib/user/syscall.c |
| #define syscall4(NUMBER, ARG0, ARG1, ARG2, ARG3) \  ({ \  int retval; \  asm volatile \  ("pushl %[arg3]; pushl %[arg2]; pushl %[arg1]; pushl %[arg0]; " \  "pushl %[number]; int $0x30; addl $20, %%esp" \  : "=a" (retval) \  : [number] "i" (NUMBER), \  [arg0] "r" (ARG0), \  [arg1] "r" (ARG1), \  [arg2] "r" (ARG2), \  [arg3] "r" (ARG3) \  : "memory"); \  retval; \  })  /\* user \*/  int  fibonacci (int n1)  {  return syscall1 (SYS\_FIBONACCI, n1);  }  int  max\_of\_four\_int (int n1, int n2, int n3, int n4)  {  return syscall4 (SYS\_MAX\_OF\_FOUR\_INT, n1, n2, n3, n4);  } |
| 먼저 system call API 중 하나는 Syscall4 라는 API이다. 이는 Argument를 총 4개 받기 위해서 만들어진 API로 원래 기존에 있던 API들은 0~3개 까지의 argument를 받을 수 있었는데, 이는 1개가 더 추가돼다. 따라서 memory에 쌓이는 영역도 0x20 으로 3개 보다 4만큼 커진 것을 알 수 있다. 이를 이용해서 argument를 4개 받는 max\_of\_four\_int에 해당 API를 설정하여 return 할 수 있다. 그래서 max\_of\_four\_int는 위와 같이 코드가 작성되는 것이다.  fibonacci는 argument를 하나만 받기 때문에 syscall1 을 이용하여 return 해주었다. |

|  |
| --- |
| examples/additional.c |
| /\* user \*/  #include <stdio.h>  #include <syscall.h>  #include <stdlib.h>  int  main (int argc, char \*argv[])  {  if(argc==5)  printf("%d %d\n", fibonacci(atoi(argv[1])), max\_of\_four\_int(atoi(argv[1]), atoi(argv[2]), atoi(argv[3]), atoi(argv[3])));  else  return EXIT\_FAILURE;  return EXIT\_SUCCESS;  } |
| User Program을 만들어 주어 example에서 출력을 할 수 있도록 만들었다. 이때 만약 argument가 4개 들어오지 않는다면 EXIT\_FAILURE를 해줄 수 있도록 처리 하였다. 여기서 argc는 argument가 들어오는 총 개수(system call number 까지 포함해서) 그리고, argv는 그 들어온 argument를 배열로 만든 것이다. 이 example 파일을 make해서 예시파일로 만들 수 있고, 이를 pintos 환경에서 실행하면 이 파일의 실행 결과를 콘솔창에서 알 수 있다. |

|  |
| --- |
| examples/Makefile |
| PROGS = cat cmp cp echo halt hex-dump ls mcat mcp mkdir pwd rm additional shell \  bubsort lineup matmult recursor  # user  additional\_SRC = additional.c |
| Makefile은 위와 같이 수정해서 다른 c파일과 똑같이 컴파일 할 수 있도록 만들었다. |

|  |
| --- |
| userprog/syscall.c |
| case SYS\_FIBONACCI:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  f->eax = fibonacci(\*(int \*)(f->esp+4));  break;  case SYS\_MAX\_OF\_FOUR\_INT:  if(!is\_user\_vaddr(f->esp+4) || pagedir\_get\_page(t->pagedir, f->esp+4) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+8) || pagedir\_get\_page(t->pagedir, f->esp+8) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+12) || pagedir\_get\_page(t->pagedir, f->esp+12) == NULL)  exit(-1);  if(!is\_user\_vaddr(f->esp+16) || pagedir\_get\_page(t->pagedir, f->esp+16) == NULL)  exit(-1);  f->eax = max\_of\_four\_int(\*(int \*)(f->esp+4), \*(int \*)(f->esp+8), \*(int \*)(f->esp+12), \*(int \*)(f->esp+16));  break; |
| 받는 argument에 맞게 esp에 저장된 argument들의 Validity를 체크해 주었다. 마지막에 결과 값을 f->eax에 저장하여 convention stores return value of system call in EAX 라고 나와 있으므로, 해당 함수의 결과 값을 return 하여 준다. |

|  |
| --- |
| userprog/syscall.c |
| int fibonacci(int n1){  int a[3]={0};  a[0] = 0; a[1] = 1;  if(n1 <= 0)  exit(-1);  if(n1 == 1)  return 0;  for(int i=0;i<n1-1;i++){  a[2] = a[0] + a[1];  a[0] = a[1]; a[1] = a[2];  }  return a[2];  }  int max\_of\_four\_int(int n1, int n2, int n3, int n4){  int max=0;  if(n1 > n2) max = n1;  else max = n2;  if(n3 > max) max = n3;  if(n4 > max) max = n4;  return max;  } |
| 위의 함수는 Fibonacci 수열의 결과 값을, max\_of\_four\_int는 받은 4개의 수중 최댓값을 찾는 것이다. 이는 system call handler에서 이 함수를 호출하여 실행한 후 그 값을 eax 결과에 입력시키는 역할을 하게 된다. 이때 fibonacci는 0보다 작거나 같은 때는 -1로 exit를 시켜 잘못된 입력 값이 들어왔다는 것을 알려주었고, 초기값 또한 지정해주어 여러가지 케이스 입력에 대비하였다. |

Error Solution

|  |
| --- |
| lib/user/syscall.c |
| /\* user \*/  #define syscall4(NUMBER, ARG0, ARG1, ARG2, ARG3) \  ({ \  int retval; \  asm volatile \  ("pushl %[arg3]; pushl %[arg2]; pushl %[arg1]; pushl %[arg0]; " \  "pushl %[number]; int $0x30; addl $20, %%esp" \  : "=a" (retval) \  : [number] "i" (NUMBER), \  [arg0] "r" (ARG0), \  [arg1] "r" (ARG1), \  [arg2] "r" (ARG2), \  [arg3] "r" (ARG3) \  : "memory"); \  retval; \  }) |
| system call의 API를 구현할 때 형식이 어떻게 되어있는지 몰라 \를 제외하고 코드만 따라서 쓴 후 코드를 실행하였더니 많은 에러가 발생했다. API를 선언할 때는 코드의 끝에 꼭 \를 붙여주어야 한다고 한다. |

* 1. **시험 및 평가 내용**

