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

담당 교수 : 박성용 교수님

학번 / 이름 : 20190258/김혜린

개발 기간 : 2021.10.06 ~ 2021.10.31

1. **개발 목표**

이번 프로젝트에서는 User Program을 실행하기 위한 pintos 프로젝트 개발로 프로세스 실행 중에 system call interrupt가 발생했을 때 memory를 보호하면서 정상적으로 system call을 실행하고자 한다. User Program이 원하는 system call을 실행하기 위해 stack에 넘어온 argument를 처리하고 프로세스 실행에 필요한 argument를 parsing하고 stack에 push하는 과정을 구현하고자 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

Argument Passing 구현 후, stack pointer esp를 움직이면서 넘어온 argument들의 값을 저장하고 해당 값들이 저장된 address와 argument 개수 argc를 다시 저장한다. 마지막으로 return address를 stack에 쌓게 된다.

1. User Memory Access

User Memory Access 파트에서는 User thread가 유효한 메모리 접근을 확인한다. User thread에서 kernel 영역에 접근하지 않는지 할당받은 memory 영역을 벗어나지 않았는지 확인한다. 유효하지 않은 메모리 접근이 발생했을 경우 -1 status와 함께 프로세스를 종료시킨다.

1. System Calls

System Calls 구현 시, User Program에서 system call interrupt가 들어왔을 때 User Program이 원하는 적절한 system call을 수행하게 된다. 이번 프로젝트에서는 wait, exec, halt, exit, write에 대한 처리를 구현하며 추가로 fibonacci와 max\_for\_four\_int에 대한 system call을 구현할 예정이다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing

pintos에서는 80x86 Calling Convention 메모리 구조를 사용한다. 따라서 process를 시작하면서 thread를 생성할 때, 인자로 넘어온 filename에 대해 space를 기준으로 parsing을 진행한다. parsing은 strtok\_r을 이용해 진행한다. strtok\_r를 통해 넘어온 argument들을 마지막 argument부터 차례로 stack에 쌓고 4byte 단위로 메모리 주소가 끊어지게 하기 위해 word align을 진행한다. 그 후, 각 argument가 저장된 주소를 stack에 저장하고 argument address가 저장된 주소를 저장해 esp를 이용해 저장된 argument에 접근할 수 있도록 한다. 그 이후 argument의 개수를 쌓고 return address로 0을 넣으면 argument passing이 종료된다.

* User Memory Access

pintos OS에서는 User Program을 Virtual Memory를 이용해 관리하는데 이 메모리에는 kernel 과 user 영역이 있다. kernel 영역에서 user thread들을 관리하는 역할을 한다. 따라서 kernel 영역의 메모리를 보호해야하고 user thread들이 서로의 영역을 침범하지 않도록 해야한다. User thread는 paging을 이용해 memory에 올라가는데 올라간 page를 벗어나서는 안된다. Pintos 상에서의 invalid memory access는 이 할당된 영역을 벗어나 메모리에 접근하려는 것을 나타내며 이를 막기 위해서는 User memory가 어디까지 할당되었는지 user-provided pointer를 통해 확인한 후 벗어나지 않았는지 확인하고 PHYS\_BASE보다 아래로 넘어가 kernel 영역을 침범하지 않았는지

* 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

pintos OS에서는 User thread가 kernel 영역에 접근하지 못하도록 막기 때문에 kernel에서 제공하는 다양한 기능을 사용할 수 없다. 따라서 이를 이용할 수 있도록 하기 위해 system call이 필요하다.

1. Halt

thread의 활동을 중지시키는 system call로 shutdown\_power\_off()를 이용해 구현한다.

1. Execute

User process에서 child process를 만들고자 할 때 구현되는 system call로 thread 내에서 parent-child에 해당하는 계층구조가 생성되고 이에 대한 컨트롤을 semaphore을 이용해 구현한다.

1. Wait

process의 state를 wait으로 바꾸는 system call로 주로 parent process가 child process가 종료되기를 기다릴 때 사용한다.

1. Write & Read

stdin과 stdout 방식의 read & write system call을 구현하며 Buffer와 length를 받아 input\_get\_c와 putbuf를 이용해 stdin과 stdout을 구현한다.

1. Fibonacci

추가적으로 구현되는 system call로 Fibonacci는 말 그대로 Fibonacci 연산해주는 system call이다. 인자로 넘어온 n만큼 fibonacci를 수행해 결과를 반환한다.

1. Max\_of\_four\_int

4개의 int를 받아서 제일 큰 값을 반환해주는 system call로 추가적으로 구현되는 system call이다.

User Level에서 System Call API를 호출하면 API는 User Stack에 System Call Number와 해당 System Call에 필요한 argument들을 push해 stack에 쌓아준다. System Call interrupt는 0x30 번지로 해당 interrupt vector table로 이동해 system call handler 함수를 실행한다. 해당 함수에는 이번 프로젝트에서 구현한 system call들이 구현되어 있으며 interrupt frame을 통해 넘어온 esp 값을 이용해 argument를 pop해 system call number에 맞는 system call을 실행한 뒤 system call이 종료되면 다시 user level로 넘어간다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

2021.10.06 – 2021.10.13 load 함수에서 argument를 parsing하고 stack에 push하는 파트를 구현

2021.10.14 - 2021.10.28 User Memory Access 와 System Call 파트 구현

2021.10.29 - 2021.10.31 fibonacci와 max\_of\_four\_int에 해당하는 추가적인 system call 구현 및 보고서 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

process.c의 load함수를 수정해 argument passing을 구현할 예정이며 argument parsing과 passing 순서로 구현할 예정이다. 먼저 filesys\_open을 통해 file을 open하기 전에 parsing을 진행해야 한다. 왜냐하면 인자로 넘어온 filename에 대해 첫번째 argument로 file open을 진행해야하기 때문이다. strtok\_r을 이용해 parsing한 내용을 parse 배열에 저장한 뒤, setup\_stack을 통해 esp를 setup한 뒷 부분에 argument passing을 구현할 예정이다. parsing 과정에서 argument 개수를 counting 해 변수에 저장하고 이를 이용해 먼저 argument를 stack에 push한 뒤 , 해당 주소값을 저장해 word align 이후 argument address를 저장하는 데에 이용한다. 후에 argument 개수와 return address를 push한다. 개발 과정에서 argument passing이 잘 이루어졌는지는 hex\_dump를 이용해 확인한다.

* + User Memory Access

User Memory Access부분은 User Memory에 접근했을 때 이루어져야 하며 이는 system call 부분에서 이루어져야 한다. 각 system call number에 대해 is\_user\_vaddr 함수를 호출해 PHYS\_BASE보다 아래 영역의 memory에 접근하지 않았는지 확인하고 pagedir\_get\_page 함수를 이용해 mapping 되지 않은 메모리에 접근하지 않았는지 확인한다. 또한 esp에 저장된 각 argument의 주소값들도 유효성을 검사해주어야 한다.

* + System Call
    1. halt

shutdown\_power\_off() 함수를 이용해 thread halt를 시도한다.

* + 1. exec

인자로 넘어온 file로 child process를 생성하며 생성 후 반환되는 child process의 id를 user level에 다시 돌려준다. 이는 interrupt frame의 eax에 넣어져 반환된다.

* + 1. exit

process를 종료시키는 system call로 thread structure에 종료 코드를 저장하는 멤버 변수 exit\_status를 추가해 process의 exit\_status를 넘겨 받아 해당 process의 exit code를 출력하고 에러가 발생했을 경우 -1을 넘겨준다.

* + 1. wait

syscall\_handler에서는 인자로 넘어온 tid에 대해 process\_wait 함수만 실행해준다. process\_wait과 process\_exit에서 추가적인 구현이 들어가는데 pid로 넘어온 process가 wait 상태에 들어가면 움직일 수 없으므로 semaphore를 이용해 lock을 걸어준다. 이는 thread structure의 parent\_lock을 이용해 구현하며 이를 통해 각 thread마다 semaphore를 가지게 된다. 또한 children list를 이용해 기다리는 thread가 list 형태로 저장될 수 있도록 한다. 추가로 thread structure에 self\_lock을 추가하여 child가 죽고 난 뒤의 status를 받을 수 있도록 한다. process\_wait에서는 children list를 받아와 children이 존재할 경우 sema\_down을 이용해 thread를 잠재우고 child가 종료되면 process\_exit 아래에서 sema\_up을 이용해 원래 thread로 돌아가게 한다. 그렇게 되면 child process를 list에서 제거한 뒤 원래 있던 self\_lock을 풀어준다.

* + 1. write

write system call의 경우 3개의 인자 fd, buffer, length에 대해 fd가 0인지 1인지 확인하고 각 인자의 주소의 유효성을 확인한 뒤, putbuf를 이용해 buffer에 write해준다.

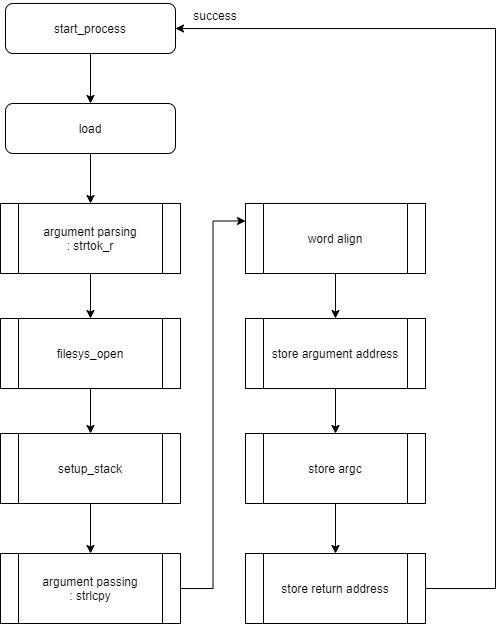
* + 1. Fibonacci

해당 시스템 콜의 경우 argument가 1개 넘어오는 system call이기 때문에 기존의 system call을 사용하면 되며, Fibonacci 함수에 대한 syscall을 lib/user/syscall.c와 lib/user/syscall.h에 구현한다. 또한 lib/syscall-nr.h에 해당 system call에 대한 system call number를 추가하여 syscall\_handler에서 이를 적절히 처리할 수 있도록 한다.

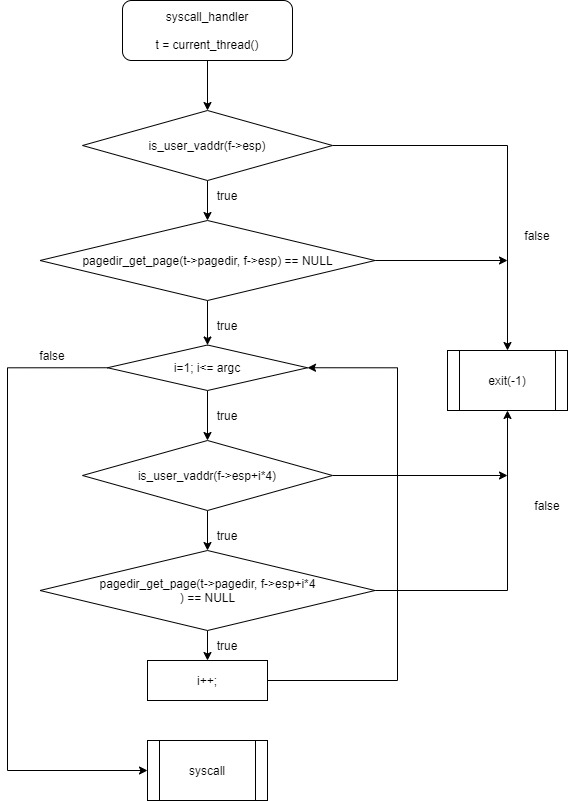
* + 1. max\_of\_four\_int

해당 시스템 콜의 경우 argument가 4개가 필요한 system call로 기존의 pintos상에 구현되지 않은 system call이기 때문에 4개의 argument를 처리하는 함수를 추가적으로 lib/user/syscall.c에 구현해주어야 한다. 또한 lib/user/syscall.c와 lib/user/syscall.h에 해당 함수에 대한 syscall을 구현하고 lib/syscall-nr.h에 해당 system call에 대한 system call number를 추가한다. userprog/syscall.c에도 해당 system call에 대한 처리를 추가한다.

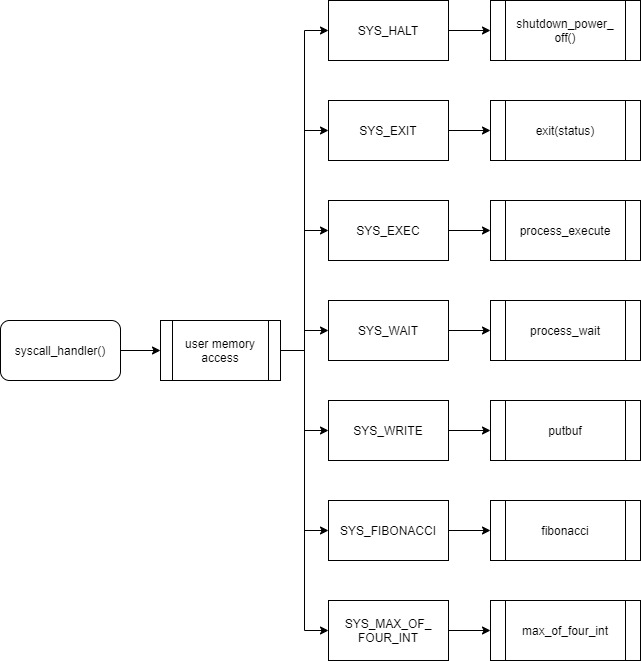
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing

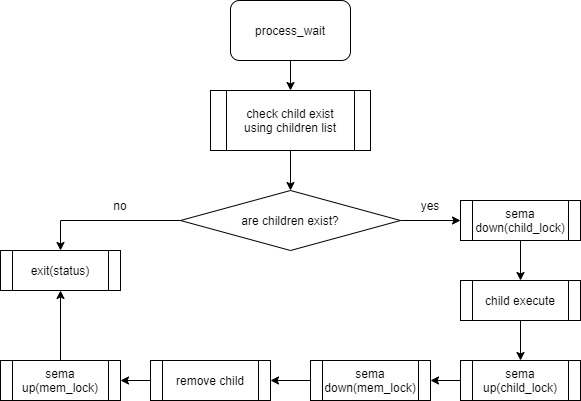


1. User Memory Access



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
2. argument parsing

|  |
| --- |
| bool load (const char \*file\_name, void (\*\*eip) (void), void \*\*esp) |
| char file\_buf[MAX\_BUF];    strlcpy(file\_buf, file\_name, strlen(file\_name)+1);    char \*argv[MAX\_BUF];    int argc = 0;    char \*token, \*last;    if((token = strtok\_r(file\_buf, " ", &last)) != NULL){      argv[argc++] = token;      while((token = strtok\_r(NULL, " ", &last)) != NULL){        argv[argc++] = token;      }    } |

MAX\_BUF를 정의하여 parsing 하고 난 token을 받을 변수를 선언한 뒤, strtok\_r 함수를 이용해 해당함 수가 NULL을 반환할 때까지 argument를 받아온다.

1. argument passing

|  |
| --- |
| bool load (const char \*file\_name, void (\*\*eip) (void), void \*\*esp) |
| /\* Argument Passing - 20190258 \*/    // argv[i] passing    char \*\*arg\_addr = (char\*\*)malloc(sizeof(char\*)\*argc);    int argv\_bytes = 0;    for(i=argc-1; i>=0; i--){      \*esp -= (strlen(argv[i]) + 1);      strlcpy(\*esp, argv[i], (strlen(argv[i]) + 1));      arg\_addr[i] = (char\*)\*esp;      argv\_bytes += (strlen(argv[i]) + 1);    }    // word align    //int word\_align\_num = ROUND\_UP(argv\_bytes, 4) % 4;    int word\_align\_num = 4 - argv\_bytes % 4;    char \*word\_align = (char\*)calloc(word\_align\_num, sizeof(char));    \*esp -= word\_align\_num;    strlcpy(\*esp, word\_align, word\_align\_num);    // arg[i] address passing    \*esp -= 4;    \*(char\*\*)\*esp = 0;    for(i=argc-1; i>=0; i--){      \*esp -= 4;      \*(char\*\*)\*esp = arg\_addr[i];    }    // argv address passing    \*esp -= 4;    \*(char\*\*\*)\*esp = (char\*\*)((\*esp) + 4);    // arg passing    \*esp -=4;    \*(int\*)\*esp = argc;    // return address passing : need additional study    \*esp -= 4;    \*(char\*\*)\*esp = (char\*) 0;    /\* Start address. \*/    \*eip = (void (\*) (void)) ehdr.e\_entry; |

esp stack pointer를 먼저 argument 개수만큼 argument 크기만큼 push하며 stack에 쌓은 후 word align을 진행해 4byte 단위로 끊어지게 만든 뒤, arg\_addr에 저장해 둔 argument address를 0으로 하나 채운 뒤, 차례로 채우고 argc와 argument address의 address를 넣고 return address로 0을 쌓는다. 각 요소들은 4bytes 단위로 움직인다.

1. User Memory Access

|  |
| --- |
| static void memory\_access\_check(void \*esp, struct thread \*t,int c); |
| memory\_access\_check(void \*esp, struct thread \*t, int c){    if(c == 0){      if(!is\_user\_vaddr(esp)||pagedir\_get\_page(t->pagedir, esp) == NULL)        exit(-1);    }    else {      for(int i=1; i<=c; i++){        if(!is\_user\_vaddr((void\*)(esp+4\*i))||pagedir\_get\_page(t->pagedir, (void\*)(esp+4\*i)) == NULL)          exit(-1);      }    }  } |

user memory access를 구현하기 위해 memory\_access\_check함수를 구현하였다. c함수에서는 현재 thread와 인자로 넘어온 esp주소값을 이용해 해당 메모리가 유효한 메모리 접근인지 확인하고 추가로 argc에 해당하는 argument 개수를 인자로 받아 인자 개수만큼 4bytes씩 움직이며 해당 esp값에 대해 유효한 memory access인지 확인하였다. 유효하지 않을 경우 status -1로 exit 함수를 호출하여 프로세스를 종료하였다.

memory\_access\_check 함수에서는 vaddr.h에 정의된 is\_user\_vaddr 함수와 pagedir.c에 정의된 pagedir\_get\_page 함수를 이용해 유효한 메모리 접근인지 확인하였다. is\_user\_vaddr 함수는 PHY\_BASE 아래에 유저 메모리가 있는지 확인하는 코드이고 pagedir\_get\_page함수의 경우 esp와 thread의 page dir를 받아와서 현재 가리키는 영역이 자신에게 할당된 page 범위 내인지 파악할 수 있게 해준다.

1. System Calls

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

|  |
| --- |
| static void syscall\_handler (struct intr\_frame \*f UNUSED) |
| {    struct thread \*t = thread\_current();    memory\_access\_check(f->esp, t, 0);    int sys\_num = \*(int \*)(f->esp);    // 1: halt, exit, exec, wait, read, write    switch(sys\_num){      case SYS\_HALT:        halt();        break;      case SYS\_EXIT:        memory\_access\_check(f->esp, t, 1);        exit(\*(int\*)(f->esp+4));        break;      case SYS\_EXEC:        memory\_access\_check(f->esp, t, 1);        f->eax = exec(\*(char\*\*)(f->esp+4));        break;      case SYS\_WAIT:        memory\_access\_check(f->esp, t, 1);        f->eax = wait(\*(pid\_t\*)(f->esp+4));        break;      case SYS\_READ:        memory\_access\_check(f->esp, t, 3);        read(\*(int\*)(f->esp+4), \*(void\*\*)(f->esp+8), \*(unsigned\*)(f->esp+12));        break;      case SYS\_WRITE:        memory\_access\_check(f->esp, t, 3);        write(\*(int\*)(f->esp+4), \*(void\*\*)(f->esp+8), \*(unsigned\*)(f->esp+12));        break;      case SYS\_FIBONACCI:        memory\_access\_check(f->esp, t, 1);        f->eax = fibonacci(\*(int\*)(f->esp+4));        break;      case SYS\_MAX\_OF\_FOUR\_INT:        memory\_access\_check(f->esp, t, 4);        f->eax = max\_of\_four\_int(\*(int\*)(f->esp+4), \*(int\*)(f->esp+8), \*(int\*)(f->esp+16), \*(int\*)(f->esp+20));        break;    default:      thread\_exit();    }  } |

먼저 syscall\_handler가 수행되면 현재 thread를 불러오고 memory\_access\_check 를 통해 interrupt frame에 저장된 esp가 유효한 메모리를 접근하고 있는지 확인한다. esp에서는 system call number가 저장되어 있으므로 esp에 저장된 내용을 int type으로 casting해 system number를 받아온다 그리고 switch-case문을 이용해 일치하는 system number에 대한 system call을 수행한다.

SYS\_HALT의 경우 device/shutdown.c에 정의된 shutdown\_power\_off 함수를 call함으로써 수행하였으며 해당 함수는 프로그램을 종료해주는 역할을 하게 된다.

SYS\_EXIT의 경우 exit 함수를 따로 구현하여 수행하였으며 해당 함수의 코드는 아래와 같다.

|  |
| --- |
| void exit(int status){      struct thread \*t = thread\_current();      t->exit\_status = status;      printf("%s: exit(%d)\n", t->name, status);      thread\_exit();  } |

현재 thread를 받아와 exit status와 함께 출력해준 뒤, thread\_exit 함수를 불러와 thread를 종료시킨다. thread\_exit함수는 process\_exit함수를 call해 process를 종료하는데 process\_exit함수에는 parent-child 계층구조가 생성됨에 따른 추가적인 코드가 구현되어 있다.

|  |
| --- |
| void process\_exit (void) |
| sema\_up(&(cur->parent\_lock));  sema\_down(&(cur->self\_lock)); |

해당 함수에는 parent\_lock과 self\_lock에 대한 구현이 추가되어 있는데 process가 종료되면 parent\_lock을 풀어주고 self\_lock을 걸어 process\_wait 함수에서 child process를 리스트에서 제거할 수 있도록 하였다.  
SYS\_EXEC의 경우, 첫번째 argument로 저장된 file에 대해 process\_execute함수를 실행해 child process를 생성하였으며 process\_execute 함수에서는 strtok\_r을 이용해 첫번째 argument를 parsing해 filesys\_open을 실행하였다.

|  |
| --- |
| tid\_t process\_execute (const char \*file\_name) |
| /\* user \*/    char \*file\_name\_ptr = NULL, \*last;    file\_name\_ptr = palloc\_get\_page (0);    strlcpy(file\_name\_ptr, file\_name, PGSIZE);    strtok\_r(file\_name\_ptr, " ", &last);    if(filesys\_open(file\_name\_ptr) == NULL) return -1; |

SYS\_WAIT의 경우 argument 하나를 받아와 process\_wait 함수를 실행한다. process\_wait 함수에서는 parent-child 계층 구조 생성에 따른 semaphore 컨트롤이 추가적으로 구현되어 있다.

|  |
| --- |
| int process\_wait (tid\_t child\_tid) |
| struct list\_elem \*e;    struct thread \*t = NULL;    int exit\_status;    for(e = list\_begin(&(thread\_current()->children)); e != list\_end(&(thread\_current()->children));e = list\_next(e)){      t = list\_entry(e, struct thread, child\_elem);      if(child\_tid == t->tid){        sema\_down(&(t->parent\_lock));        exit\_status = t->exit\_status;        list\_remove(&(t->child\_elem));        sema\_up(&(t->self\_lock));        return exit\_status;      }    }    return -1; |

해당 syscall을 호출한 thread는 말 그대로 wait 상태로 넘어가며 인자로 넘어온 child\_tid가 종료될 때까지 해당 상태를 유지한다. 따라서 child 프로세스가 실행되는 동안 parent\_lock을 걸어서 parent process인 현재 thread가 더 이상 실행될 수 없게 하며 child가 종료 되고 난 뒤, 해당 process의 status를 받고 child를 제거한 뒤, self\_lock을 해제해 해당 thread를 실행한다.

SYS\_WRITE의 경우, 3개의 첫번째 argument를 받아와 fd가 0인지 1인지에 따라 STD\_IN과 STD\_OUT을 결정하고 1일 경우 2번째, 3번째 argument에 대해 putbuf함수를 실행해 STD\_OUT을 진행한다. esp로부터 4, 8, 12byte씩 이동하여 각 argument를 가져와 추가로 정의한 write 함수에 인자로 넘겨준다. write 함수의 코드는 아래에 첨부되어 있다. putbuf함수는 lib/kernel/console.c 에 정의되어 있으며 buffer에 저장된 n개만큼의 내용을 console창에 출력한다.

|  |
| --- |
| static int write (int fd, const void \*buffer, unsigned size) |
| {    struct thread \*t = thread\_current();    if(!is\_user\_vaddr(buffer || pagedir\_get\_page(t->pagedir, buffer) == NULL))        exit(-1);    if (fd == 1)    {      putbuf(buffer, size);    }    else exit(-1);    return size;  } |

1. Additional System calls

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

SYS\_FIBONACCI 와 SYS\_MAX\_OF\_FOUR\_INT에 대한 numbering을 lib/syscall-nr.h에 추가로 정의해준다.

|  |
| --- |
| /\* additional syscall \*/      SYS\_FIBONACCI,      SYS\_MAX\_OF\_FOUR\_INT |

또한 examples에 additional.c 파일을 추가해 additional을 실행해 두 syscall에 대해 시스템 콜을 수행할 수 있도록 한다.

|  |
| --- |
| 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[4])));    else      return EXIT\_FAILURE;    return EXIT\_SUCCESS;  } |

additional을 실행할 경우 additional과 max\_of\_four\_int 함수에 넘길 argument 4개까지 해서 총 5개의 argument가 넘어와야 한다. 따라서 argc가 5개인지 확인한 후, 맞다면 해당 함수들을 실행해 시스템 콜이 실행될 수 있도록하고 아닐 경우 EXIT\_FAILURE를 반환한다.

additional 실행파일을 만들기 위해서는 examples에 있는 Makefile도 수정해주어야 한다. 해당 file에 다음과 같이 additional.c를 컴파일 해줄 수 있도록 아래 코드를 추가한다.

|  |
| --- |
| additional\_SRC = additional.c |

시스템 콜에 대한 syscall\_handler에도 해당 system call을 처리할 수 있도록 함수를 추가해주어야 한다.

|  |
| --- |
| static void syscall\_handler (struct intr\_frame \*f UNUSED) |
| case SYS\_FIBONACCI:        memory\_access\_check(f->esp, t, 1);        f->eax = fibonacci(\*(int\*)(f->esp+4));        break;      case SYS\_MAX\_OF\_FOUR\_INT:        memory\_access\_check(f->esp, t, 4);        f->eax = max\_of\_four\_int(\*(int\*)(f->esp+4), \*(int\*)(f->esp+8), \*(int\*)(f->esp+16), \*(int\*)(f->esp+20));        break; |

다른 시스템 콜과 마찬가지로 check\_argv\_valid를 수행해준 뒤, fibonacci와 max\_of\_four\_int 함수에 argument를 4byte 간격으로 pop해 넘겨주어 함수의 반환값을 interrupt frame f의 eax에 저장하여 user level로 넘어간다. fibonacci와 max\_of\_four\_int 함수의 코드는 아래와 같다.

|  |
| --- |
| int fibonacci(int n){      int xn0 = 0;      int xn1 = 1;      if(n <= 0)          exit(-1);      if(n == 1)          return 0;      for(int i=2;i<=n;i++){          int xn2 = xn0 + xn1;          xn0 = xn1;          xn1 = xn2;      }      return xn1;  } |

fibonacci함수의 경우 인자로 넘어온 n에 대해 n번의 fibonacci 연산을 수행한다. 0번째와 1번째는 0과 1이고 따라서 넘어온 인자가 n<=1 일 경우 따로 연산을 수행하지 않고 해당 n에 따른 연산결과를 반환한다. 나머지 경우에 대해서는 2부터 n까지 iteration을 수행하여 바로 앞의 연속한 두 인자를 더하는 연산을 수행해 결과를 반환한다.

|  |
| --- |
| int max\_of\_four\_int(int a, int b, int c, int d){      int max1 = a > b ? a : b;      int max2 = c > d ? c : d;      return max1 > max2 ? max1 : max2;  } |

인자로 넘어온 네 개의 정수에 대한 대소 비교 결과 가장 큰 값을 반환하는 함수로 앞의 두 인자에 대한 대소 비교 결과를 max1에 저장하고 뒤의 두 인자에 대한 대소 비교 결과를 max2에 저장하여 최종적으로 두 변수의 대소 비교 결과 더 큰 값을 반환한다.

max\_of\_four\_int 시스템 콜의 경우 인자 4개가 들어오는 시스템 콜로 기존의 pintos 프로젝트에는 구현되어 있지 않기 때문에 해당 argument를 처리하는 systemcall에 대한 정의를 lib/user/sycall.c에 추가로 구현해 주었다.

|  |
| --- |
| #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;                                               \          }) |

해당 함수는 define 매크로를 써서 구현하였고 속도를 높이기 위해 인라인 어셈블리 코드를 사용하여 구현하였다.

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

텍스트이(가) 표시된 사진

자동 생성된 설명