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

담당 교수 : 박성용 교수님

조 / 조원 : 20191559 강상원

개발 기간 : 2023. 9. 18. ~ 2023. 9. 30.

1. **개발 목표**

* PintOS에 기본적으로 제공되어 있는 코드에 더해 구현되어 있지 않은 Argument passing, user memory access, system call 구현을 하며 halt, exit, … wait system call 이외에 추가적인 fibonacci, max\_of\_four\_int 구현을 한다. Argument passing을 할 때 argv, argc로 적절히 파싱하여 Stack 자료구조에 80x86 calling 컨벤션에 맞게 쌓아주었다.

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

입력받은 명령어의 argument들을 tokenize하여 각 argv로 나누고 argc를 카운트한다. 이를 Stack 자료구조에 80x86 calling convention에 따라 하나씩 쌓는다. File name을 찾아 Pintos에서 어떤 프로그램을 실행할 지 알 수 있게 한다.

1. User Memory Access

비정상적인 주소가 넘어갔을 때 user program 단에서 이를 검증하지 못해 Kernel Panic 등의 오류가 발생하는 문제를 해결하였다. 해당 부분에 적절한 예외처리를 통해 Pintos를 적절히 종료할 수 있게 하였다.

1. System Calls

Pintos manual 3.3.4에 의거 구현되어 있지 않았던 system call handler를 구현하여 Halt, exit, exec, wait, read, write의 system call을 구현하였다. 추가적으로 fibonacci, max\_of\_four\_int 두 개의 새로운 system call도 구현한다.

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

먼저 process\_execute 함수에서 첫 번째 공백 위치를 찾아 file name을 찾아 준다. load 함수에서 strtok\_r 함수를 이용하여 나머지 부분: argument들을 나눠주고 argc를 세어 준다. 이후 make\_stack 함수를 호출하여 80x86 convention에 맞게 stack에 쌓아 준다. esp에 memcpy를 통해 넘겨주고 추가 변수 alignment를 통해 4 word alignment를 맞춰 준다. 차례로 argument들의 주소, argc, return address를 Stack에 쌓는다.

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

PintOS 상에서 정의된 valid한 메모리 범위(PHYS\_BASE~0xc0000000)를 넘어선 주소 값에 접근을 시도 할 때 invalid memory access가 나오게 된다. User program이 범위를 넘어서서 kernel memory 영역에 접근을 시도하거나 하면 Kernel Panic과 같은 오류가 발생한다.

* + Invalid memory access를 어떻게 막을 것인지 설명

threads/vaddr에 미리 정의되어 있는 is\_user\_vaddr과 is\_kernel\_vaddr 함수를 이용하여 주어진 메모리 주소가 알맞은 범위 안에 있는지 확인하며 비정상적인 접근 시도라면 -1 exit하여 미연에 오류를 방지한다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

운영체제는 상호불가침의 영역, User Program과 Kernel으로 나뉜다. File read, write와 같이 user program 단독으로 실행할 수 없는 기능을 user program에서 수행하기 위해 kernel 영역의 기능을 요청해야 하는데, 이를 System Call을 이용해 호출하게 된다.

* + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명
    1. halt: shutdown\_power\_off 함수를 통해 PintOS를 종료시키는 요청을 보낸다.
    2. exit: 실행중인 User Program을 종료, 해당 status를 커널에 보고한다. 0이 아닌 반환값으로 오류를 나타낸다.
    3. exec: 인자를 통해 넘어온 이름에 해당하는 프로세스를 자식 프로세스를 fork하여 실행시킨다.
    4. wait: 해당 Pid의 자식 프로세스 종료를 기다리고 Exit status 값을 확인한다. 기다리는 프로세스가 wait 전에 끝나지 않도록 전처리한다.
    5. read: Open file descriptor (해당 프로젝트에선 STDIN)의 입력을 받아 buffer에 작성한다.
    6. write: Open file desciptor (해당 프로젝트에선 STDOUT)에 출력한다. 반환값으로 byte 수를 내보낸다.
    7. FIBONACCI: 입력받은 정수 n에 해당하는 피보나치 수열 값을 반환한다.
    8. MAX\_OF\_FOUR\_INT: 입력받은 4개의 정수값 중 가장 큰 값을 반환한다.
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

User level에서 System Call API를 호출하면 mode bit을 통해 이를 표시하여 kernel 모드로 진입한다. User stack에 System call number와 주어진 인자들이 쌓이게 되어 정보가 저장된다. system call table에서 해당 system call number를 확인하고 이에 해당하는 system call 함수의 주소를 찾아 상응하는 동작을 수행한다. 이 작업이 끝나면 다시 user level로 돌아오게 되고 mode bit도 표시한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

총 개발 기간: 2023. 9. 18. ~ 2023. 9. 30

* 9/18 ~ 9/21: Argument Passing 구현
* 9/22 ~ 9/25: User Memory Access 구현
* 9/26 ~ 9/27: System Calls 구현
* 9/27 ~ 9/28: Additional System Call (FIBONACCI, MAX\_OF\_FOUR\_INT) 구현
* 9/29 ~ 9/30: 보고서 작성
  1. **개발 방법**
     1. Argument Passing

[userprog/process.c]에서 첫 번째 공백 위치를 찾아 cmd\_name를 찾는다. load 함수에서 strtok\_r 함수를 이용하여 나머지 부분: argument들을 나눠주고 argc를 세어 준다. 이후 make\_stack 함수에서 80x86 convention에 맞게 stack에 쌓아 준다. esp에 memcpy를 통해 넘겨주고 추가 변수 alignment를 통해 4 word alignment를 맞춰 준다. 차례로 argument들의 주소, argc, return address를 Stack에 쌓는다.

* + 1. User Memory Access

[userprog/exception.c]에서 is\_kernel\_vaddr 함수를 이용하여 user program이 kernel 메모리 영역을 침범하는지 확인하여 kernel panic을 미연에 예방한다.

[userprog/process.c]의 load에서는 filesys\_open의 반환값이 NULL이라면 (파일 읽기에 실패했다면) 에러를 반환하게 하여 예외 처리한다.

Invalid memory access를 막기 위해 [threads/vaddr.c]에 미리 정의되어 있는 is\_user\_vaddr과 is\_kernel\_vaddr 함수를 이용하여 주어진 메모리 주소가 알맞은 범위 안에 있는지 확인하며 비정상적인 접근 시도라면 -1 exit하여 미연에 오류를 방지한다.

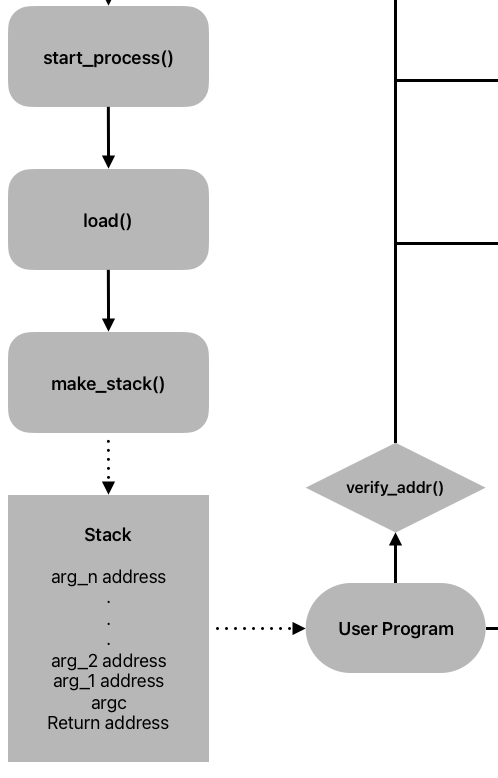
* + 1. System Calls

[userprog/syscall.c]에서 syscall\_handler 함수 내 switch문으로 각 system call number마다 수행할 작업을 정의한다. 이 때 각 동작이 수행되기 전 미리 정의한 verify\_addr 함수로 user adress 범위 내에 있는지 확인하여 오류를 방지한다. 각 함수의 반환값은 eax register에 저장한다. 추가적으로 구현한 FIBONACCI, MAX\_OF\_FOUR\_INT system call에 해당하는 system call number는 [lib/syscall-nr.h]에 새로 정의한다.

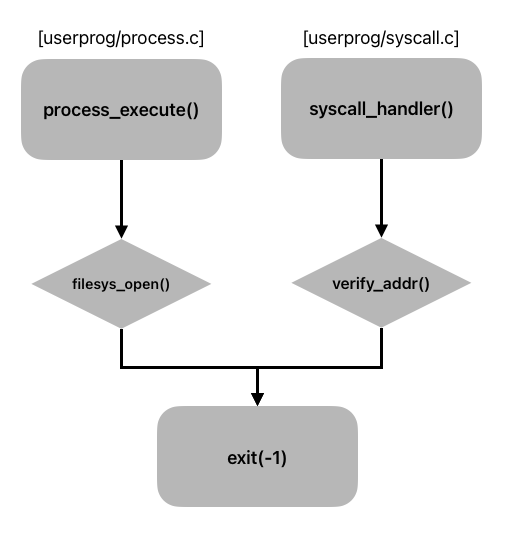
[lib/user/syscall.c]에서 syscall4 함수를 새로 정의하여 MAX\_OF\_FOUR\_INT에서 사용할 수 있게끔 한다. [examples/additional.c]를 작성하여 두 system call에 해당하는 함수를 작성하고 [examples/Makefile]을 수정하여 make 시 추가로 작성한 additional.c도 함께 컴파일되도록 하였다.

[userprog/process.c]에서 [threads/thread.h]에 추가된 semaphore인 mem\_lock, child\_lock을 이용하여 synchronization을 해 준다.

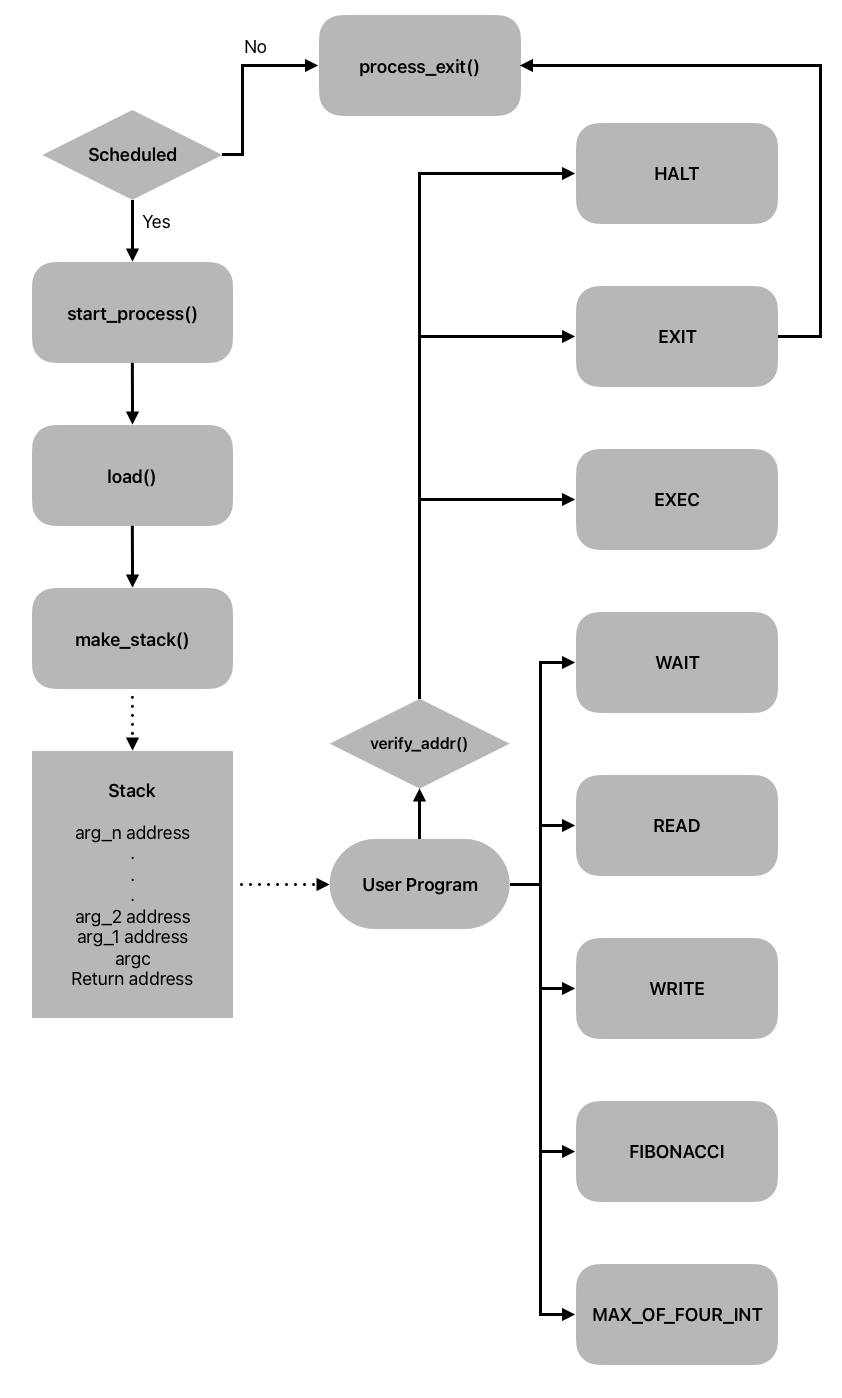
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing



1. User Memory Access



1. System Calls



* 1. **제작 내용**

1. Argument Passing

* [userprog/process.c]

**tid\_t**  
process\_execute (**const** **char** \*file\_name)   
{  
 **char** \*fn\_copy;  
 **tid\_t** tid;  
  
 /\* Make a copy of FILE\_NAME.  
 Otherwise there's a race between the caller and load(). \*/  
 fn\_copy = palloc\_get\_page (0);  
 **if** (fn\_copy == NULL)  
 **return** TID\_ERROR;  
 strlcpy (fn\_copy, file\_name, PGSIZE);  
  
 // Find command name by finding first space  
 **char** cmd\_name[128];  
 strlcpy(cmd\_name, file\_name, 1+strlen(file\_name));  
  
 FOR(i, strlen(cmd\_name)){   
 **if**(cmd\_name[i] == ' '){  
 cmd\_name[i] = '\0';  
 **break**;  
 }  
 }  
 **if**(filesys\_open(cmd\_name) == NULL) **return** TID\_ERROR;

process\_execute()에서 첫 번째 공백을 찾아 cmd\_name을 찾는다. 초기에는 cmd\_name을 parsing하고 나서 마지막 '\0'을 추가하지 않아서 차후 문자열 처리에서 커맨드를 제대로 못 다루는 문제가 있었다. 이후 cms[i]=’\0’ 코드를 추가하여 문제를 해결하였다.

**bool**  
**load** (**const** **char** \*file\_name, **void** (\*\*eip) (**void**), **void** \*\*esp)   
{  
 **struct** **thread** \***t** = **thread\_current** ();  
 **struct** **Elf32\_Ehdr** **ehdr**;  
 **struct** **file** \***file** = **NULL**;  
 **off\_t** file\_ofs;  
 **bool** success = false;  
 **int** i;  
  
 /\* Allocate and activate page directory. \*/  
 t->pagedir = pagedir\_create ();  
 **if** (t->pagedir == NULL)   
 **goto** done;  
 process\_activate ();  
   
 **int** argc=0;  
 **char**\* argv[128];  
 **char** cp\_file\_name[128];  
 **char** \*token\_nxt;  
  
 snprintf(cp\_file\_name, **sizeof**(cp\_file\_name), "%s", file\_name);  
   
 **for**(**char**\* token = strtok\_r(cp\_file\_name, " ", &token\_nxt);   
 token != NULL; token = strtok\_r(NULL, " ", &token\_nxt)) {  
 argv[argc++] = token;  
 }  
  
 **char**\* file\_name\_ptr = argv[0];

load()에서는 이후 argment들과 argc를 센다. strtok\_r으로 공백 기준으로 인자들을 나누며, 파일 이름 포인터에 첫 번째 인자를 넣는다.

void make\_stack(char\* argv[], void \*\*esp, int argc) {  
 int ptr\_argv[128];  
 int alignment= 0;  
 int i = argc-1;  
  
 **while**(i >= 0) {  
 int argvl = 1 + strlen(argv[i]);  
 \*esp -= argvl;  
 memcpy(\*esp, argv[i], argvl);  
 alignment += argvl;  
 ptr\_argv[i--] = \*esp;  
 }  
 alignment = (alignment % 4 == 0) ? 0 : 4 - alignment % 4;  
 \*esp -= alignment;  
 memset(\*esp, 0, alignment); // NULL padding  
  
 \*esp -= 4;  
 memset(\*esp, 0, 4); // argv[argc] = NULL  
  
 i = argc-1;  
 **while**(i >= 0) { // argv  
 \*esp -= 4;  
 memcpy(\*esp, &ptr\_argv[i--], 4);  
 }  
  
 int argv\_addr\_ptr = \*esp; // argv  
 \*esp -= 4;  
 memcpy(\*esp, &argv\_addr\_ptr, 4); // argv  
  
 \*esp -= 4;  
 memcpy(\*esp, &argc, 4); // argc  
  
 \*esp -= 4;  
 memset(\*esp, 0, 4); // fake return address  
  
 // hex\_dump(\*esp, \*esp, 100, true); // FOR DEBUGGING  
  
}

이후 make\_stack()에서 아까 load()에서 알맞게 넣은 argv, argc를 이용하여 80x86 convention에 맞게 Stack에 쌓는다. 4 word alignment를 위한 추가 변수를 만들고, 최소 padding을 추가한다. 차례로 argument들의 주소, argc, return address를 Stack에 쌓는다.

1. User Memory Access

* [userprog/process.c]

**if**(filesys\_open(cmd\_name) == NULL) **return** TID\_ERROR;

load()에서는 filesys\_open의 반환값이 NULL이라면 (파일 읽기에 실패했다면) 에러를 반환하게 하여 예외 처리한다.

* [userprog/syscall.c]

void  
verify\_addr(void \*addr) {  
 **if** (false == is\_user\_vaddr(addr)) EXIT(-1);  
}

syscall.c에 추가한 verify\_addr 함수로 user adress 범위 내에 있는지 확인하여 오류를 방지한다.

* [userprog/exception.c]

/\* Added to check iff User Program access Kernel Address. \*/  
 if (is\_kernel\_vaddr(fault\_addr) || !user) {  
 EXIT(-1);   
 }

exception.c에 추가한 is\_kernel\_vaddr() 함수로 kernel 메모리 영역을 침범하지 않았는지, user 영역에 있는지 쉽게 체크할 수 있도록 한다.

1. System Calls

* [userprog/syscall.c]

**static** void  
syscall\_handler (struct intr\_frame \*f UNUSED)   
{  
 // printf ("system call!\n");  
 // hex\_dump(f->**esp**, f->**esp**, 100, 1); // Added to print stack  
 switch (\*(uint32\_t \*)(f->**esp**)) {  
 case SYS\_HALT:  
 HALT();  
 break;  
 case SYS\_EXIT:  
 verify\_addr(f->**esp** + 4);  
 EXIT(\*(uint32\_t \*)(f->**esp** + 4)); //  
 break;  
 case SYS\_EXEC:  
 verify\_addr(f->**esp** + 4);  
 f->**eax** = EXEC(\*(uint32\_t \*)(f->**esp** + 4)); //  
 break;  
 case SYS\_WAIT:  
 verify\_addr(f->**esp** + 4);  
 f->**eax** = WAIT(\*(uint32\_t \*)(f->**esp** + 4)); //  
 break;  
 case SYS\_CREATE:  
 // f->**eax** = CREATE(\*(uint32\_t \*)(f->**esp** + 4), \*(uint32\_t \*)(f->**esp** + 8)); //  
 break;  
 case SYS\_REMOVE:  
 // f->**eax** = REMOVE(\*(uint32\_t \*)(f->**esp** + 4)); //  
 break;  
 case SYS\_OPEN:  
 // f->**eax** = OPEN(\*(uint32\_t \*)(f->**esp** + 4)); //  
 break;  
 case SYS\_FILESIZE:  
 // f->**eax** = FILESIZE(\*(uint32\_t \*)(f->**esp** + 4)); //  
 break;  
 case SYS\_READ:  
 verify\_addr(f->**esp** + 4);  
 f->**eax** = READ(\*(uint32\_t \*)(f->**esp** + 4), \*(uint32\_t \*)(f->**esp** + 8), \*(uint32\_t \*)(f->**esp** + 12)); //  
 break;  
 case SYS\_WRITE:  
 f->**eax** = WRITE(\*(uint32\_t \*)(f->**esp** + 4), \*(uint32\_t \*)(f->**esp** + 8), \*(uint32\_t \*)(f->**esp** + 12));  
 break;  
 case SYS\_SEEK:  
 // SEEK(\*(uint32\_t \*)(f->**esp** + 4), \*(uint32\_t \*)(f->**esp** + 8)); //  
 break;  
 case SYS\_TELL:  
 // f->**eax** = TELL(\*(uint32\_t \*)(f->**esp** + 4)); //  
 break;  
 case SYS\_CLOSE:  
 // CLOSE(\*(uint32\_t \*)(f->**esp** + 4)); //  
 break;  
 case SYS\_FIBONACCI:  
 f->**eax** = FIBONACCI(\*(uint32\_t \*)(f->**esp** + 4)); //  
 break;  
 case SYS\_MAX\_OF\_FOUR\_INT:  
 f->**eax** = MAX\_OF\_FOUR\_INT(\*(uint32\_t \*)(f->**esp** + 4), \*(uint32\_t \*)(f->**esp** + 8), \*(uint32\_t \*)(f->**esp** + 12), \*(uint32\_t \*)(f->esp + 16)); //  
 break;  
 }  
 // thread\_exit ();  
}

syscall.c에서 각 system call number에 따라 수행할 작업을 switch문으로 나눈다. 앞서 흐름도에서 설명했듯이 verify\_addr()로 잘못된 주소 접근을 방지하고, argument 필요 개수에 따라 알맞은 주소값을 eax 레지스터에 전달한다.

**void** **HALT** (**void**) {  
 shutdown\_power\_off();  
}

HALT system call에 대해서는 내장 함수인 shutdown\_power\_off()를 사용하여 현재 실행 중인 프로그램을 종료시키는 역할을 수행하도록 하였다.

void EXIT (int status) {  
 printf("%s: exit(%d)\n", thread\_current()->name, status);  
 **thread\_current**()->exit\_status = status; // Added to print like 'exit(81)'  
 thread\_exit();  
}

EXIT system call에 대해서는 thread\_exit()을 실행시켜 exit하기 전에 current thread에 있는 exit status 값을 알맞게 바꿔주고, exit 문구를 syntax에 맞게 출력한다.

**int** **EXEC** (**const** **char** \*cmd\_lime) {  
 **return** **process\_execute**(cmd\_lime);  
}

EXEC system call에 대해서는 process\_execute()로 thread를 생성하고 반환값을 전달한다.

**int** **WAIT** (**int** pid) {  
 **return** **process\_wait**(pid);  
}

WAIT system call에 대해서는 process\_wait()의 반환값을 전달한다. 인자로 전달한 pid에 해당하는 child process를 설정한 semaphore를 통해 기다리고, semaphore가 up되면 종료 상태를 반환하는 함수이다.

int  
process\_wait (tid\_t child\_tid UNUSED)   
{  
 // TODO  
   
 **struct** thread \*cur = thread\_current();  
 **struct** thread \*child = NULL;  
 int exit\_status = -1; // Default: -1  
   
 for (**struct** list\_elem \*e = list\_begin(&(cur->child\_list));   
 e != list\_end(&(cur->child\_list));   
 e = list\_next(e)) {  
 child = list\_entry(e, struct thread, child\_elem);  
   
 **if** (child->tid == child\_tid) { // If found child  
 sema\_down(&(child->child\_lock));  
 exit\_status = child->exit\_status;  
 list\_remove(e);  
 sema\_up(&(child->mem\_lock));   
 break; // Get out of loop  
 }  
 }  
   
 return exit\_status;  
}

**int** READ (**int** fd, **void** \*buffer, **unsigned** size) {  
 if (!fd) {  
 for (**unsigned** i = 0; i < size; i++) {  
 \*((uint8\_t \*)buffer + i) = input\_getc();  
 }  
 return size;  
 }  
 return -1;  
}

READ system call에서는 file descriptor가 STDIN(0)인 경우 한 글자씩 입력을 받는다.

**int** WRITE (**int** fd, **const** **void** \*buffer, **unsigned** size) {  
  
 if (fd == 1) {  
 putbuf(buffer, size);  
 return size;  
 }  
 return -1;   
}

WRITE system call에서는 file descriptor가 STDOUT(1)인 경우 한 글자씩 putbuf()를 통해 출력한다.

* [threads/thread.h]

**struct** **thread**  
 {  
 ...

#**ifdef** USERPROG  
 /\* Owned by userprog/process.c. \*/  
 **uint32\_t** \*pagedir; /\* Page directory. \*/  
 /\* Added in #Proj 1 \*/  
 **int** exit\_status; /\* Exit status of process \*/  
 **struct** **list** **child\_list**; /\* List of child processes \*/  
 **struct** **list\_elem** **child\_elem**; /\* Element of child\_list \*/  
 // Semaphores  
 **struct** **semaphore** **mem\_lock**; /\* Lock of memory access \*/  
 **struct** **semaphore** **child\_lock**; /\* Lock of child process \*/  
#**endif**  
  
 /\* Owned by thread.c. \*/  
 **unsigned** magic; /\* Detects stack overflow. \*/  
 };

앞서 설명한 바와 같이 child process wait sync에 필요한 semaphore들을 process.h에 추가하였다.

* [threads/thread.c]

**static** void  
init\_thread (struct thread \*t, const char \***name**, int priority)  
{  
 ...  
  
 #ifdef USERPROG  
 sema\_init(&(t->**child\_lock**), 0);  
 sema\_init(&(t->**mem\_lock**), 0);  
  
  
 list\_init(&t->**child\_list**);  
 list\_push\_back(&(running\_thread()->**child\_list**), &(t->child\_elem));  
 #endif  
}

또한 이를 Initiation하기 위한 함수를 thread.c에 추가하였는데, C++ vector syntax와 같이 list\_push\_back 할 수 있게 만들었다.

앞서 보였던 process\_wait()에서 child\_lock semaphore를 down하고 mem\_lock을 up하였는데, 이에 상응하게 process\_exit()에서 반대로 child\_lock semaphore를 up하고 mem\_lock semaphore를 down한다.

1. Additional System calls

* [lib/syscall-nr.h]

syscall-nr.h에 추가 구현한 system call인 FIBONACCI와 MAX\_OF\_FOUR\_INT에 대한 system call number를 추가한다.

#ifndef \_\_LIB\_SYSCALL\_NR\_H  
#define \_\_LIB\_SYSCALL\_NR\_H  
  
/\* System call numbers. \*/  
enum   
 {  
 ...  
 // Additional system calls  
 SYS\_FIBONACCI, /\* Fibonacci \*/  
 SYS\_MAX\_OF\_FOUR\_INT, /\* Max of four integers \*/  
 ...  
  
 };  
  
#endif /\* lib/syscall-nr.h \*/

* [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; \  
 })

**...**

**int** **FIBONACCI**(**int** n) {  
 **return** syscall1(SYS\_FIBONACCI, n);  
}  
  
**int** **MAX\_OF\_FOUR\_INT**(**int** a, **int** b, **int** c, **int** d) {  
 **return** syscall4(SYS\_MAX\_OF\_FOUR\_INT, a, b, c, d);  
}

추가 구현한 system call 중 MAX\_OF\_FOUR\_INT는 인자가 네 개가 필요하므로 syscall.c에 구현되어 있는 syscall1,2,3 함수에 더해 syscall4()를 작성한다. 추가로 각 system call의 인자 개수에 맞게 syscall 함수들을 호출한다.

* [userprog/syscall.c]

int FIBONACCI(int n) {  
 int a = 0, b = 1, c = 0;  
 **if** (n == 0)   
 **return** a;  
  
 **for** (int i = 2; i <= n; i++) {  
 c = a + b;  
 a = b;  
 b = c;  
 }  
 **return** b;  
}  
  
int MAX\_OF\_FOUR\_INT(int a, int b, int c, int d) {  
 int max\_ab = a > b ? a : b;  
 int max\_cd = c > d ? c : d;  
 **return** max\_ab > max\_cd ? max\_ab : max\_cd;  
}

syscall.c에 각 함수의 동작을 구현하여 집어넣고, 마찬가지로 [userprog/syscall.h]에 각 함수의 프로토타입을 작성한다.

* [examples/additional.c]

/\* Added in #Proj 1. \*/  
  
#**include** <stdio.h>  
#**include** <syscall.h>  
#**include** <stdlib.h>  
  
  
**int** **main**(**int** argc, **char** \*argv[]) {  
 **if**(argc != 5) {  
 printf("Usage: ./additional [num 1] [num 2] [num 3] [num 4]\n");  
 **return** EXIT\_FAILURE;  
 }  
   
 printf("%d %d\n", FIBONACCI(atoi(argv[1])),   
 MAX\_OF\_FOUR\_INT(atoi(argv[1]), atoi(argv[2]), atoi(argv[3]), atoi(argv[4])));  
  
 **return** EXIT\_SUCCESS;  
}

추가 작성한 additional.c에서, 인자 개수가 1+4=5개가 맞는지를 확인하여, 에러 헨들링을 먼저 한다. 이후, FIBONACCI, MAX\_OF\_FOUR\_INT 함수를 호출하고 반환값을 출력한다.

* [examples/Makefile]

SRCDIR = ..  
  
# Test programs to compile, and a list of sources for each.  
# To add a new test, put its name on the PROGS list  
# and then add a name\_SRC line that lists its source files.  
PROGS = cat cmp cp echo halt hex-dump ls mcat mcp mkdir pwd rm shell \  
 bubsort lineup matmult recursor additional

...

additional\_SRC = additional.c  
  
...

**include** $(SRCDIR)/Make.config  
**include** $(SRCDIR)/Makefile.userprog

추가한 additional.c가 examples 안의 코드들과 같이 컴파일되게 하기 위해 Makefile을 수정하였다.

* 1. **시험 및 평가 내용**

**텍스트, 전자제품, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명**

추가 구현한 additional system call이 잘 작동되는 모습을 볼 수 있다.

추가적으로 다음은 make check 했을 때의 결과값이다. 프로젝트 명세서에 명시된 구현 사항을 모두 충족함을 볼 수 있다.

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

자동 생성된 설명