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

담당 교수 : 김영재

조 / 조원 : 김효림

개발 기간 : 7일

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

Pintos 프로젝트 내에서 OS가 사용자의 명령을 잘 수행할 수 코드를 수정한다. Project 1은 구체적으로 5개의 단계를 거쳐 완성할 수 있다.

1. Argument passing

Input을 blank space를 기준으로 argument를 파싱한다. 그 후 스택에 각각의 인자를 쌓아준다. (userprog/process.c)

1. User Memory Access

접근하는 주소가 유효한 주소인지를 판단한다. 이때 threads/vaddr.h에 있는 함수를 이용한다.

1. System Call Handler

시스템 콜 요청에 대한 올바른 행동을 수행하기 위해, system call number에 따라 행동 양식을 코드로 작성한다.

1. System Call Implementation

Synchronization을 활용해 exec(), exit(), write(), read() 시스템 콜을 구현한다.

1. Additional Implementation

N 번 째 피보나치 수를 반환하는 Fibonacci 시스템 콜, 인자로 받은 네 개의 값 중 가장 큰 값을 반환하는 max\_of\_four\_int 시스템 콜을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

인자들을 공백을 기준으로 argv[i]에 담는다. 그 후 stack에 규약에 따라 인자들을 넣어준다. 구현한 코드가 해당 사항을 만족하는지는 hex\_dump() 함수를 사용하여 결과를 확인할 수 있다.

1. User Memory Access

각 시스템 콜 구현에 앞서, 각 명령들이 접근하는 주소들에 대하여 그 주소들이 유효한지를 확인할 수 있다. Vaddr.h에 있는 함수를 이용해 page directory에서 물리적인 주소와 잘 mapping 되었는지 확인한다.

1. System Calls

유저가 요청한 system call에 따라 handler가 각각의 기능을 잘 수행하게 된다. 이때 write와 read는 STDOUT, STDIN을 수행하고, halt(), exit(), exec(), wait()은 각각 중단, 프로세스 종료, 실행, thread wait을 구현하게 된다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명

텍스트, 스크린샷, 폰트, 타이포그래피이(가) 표시된 사진

자동 생성된 설명

0xc0000000 을 기준으로 그 아래에 User Memory가, 그 위로 Kernel Memory가 저장된다. Stack은 아래로 값이 쌓이기 때문에, argument passing에서 slicing하여 얻은 argv를 규약에 맞게 넣어줄 필요가 있다. 즉 입력한 문자열이 나눠진 후, ‘\0’가 붙어 스택에 쌓이게 된다.위의 사진은 echo x를 넣었을 때 hex\_dump를 실행한 결과이다. .

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

pintos에서 invalid memory access는 두 가지가 있다. 첫 번째는 커널 영역에 접근하는 것이고, 두 번째는 Mapping되지 않은 page directory에 접근하는 경우이다. 커널 영역에 접근하는 것은 is\_kernel\_vaddr을 보면 확인할 수 있듯이, PHYS\_BASE를 넘어서는 주소를 의미한다.

* + Invalid memory access를 어떻게 막을 것인지 설명

threads/vaddr.h에 정의되어 있는 is\_kernel\_vaddr()함수를 이용하여 접근하고자 하는 주소가 kernel 영역에 존재하는지를 판단한다. 또한 매핑되지 않는 페이지 디렉토리에 접근하는 것을 막기위해 NULL인지 판단한다. 이를 확인 후 invalid memory가 맞다면 exit을 이용해 프로그램을 종료한다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

I/O request나 CPU 메모리를 사용하는 경우, User mode에서는 권한이 제한적이기 때문에 하드웨어 자원에 직접 접근하기 어렵다. 시스템 콜을 사용하면 Kernel을 이용해 프로그램을 훼손하지 않고 자원에 접근할 수 있게 된다.

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

1. Halt() : shutdown\_power\_off() 함수가 호출되며 프로그램을 종료시킨다.
2. Exit() : exit() 요청이 들어오면 인자로 전달받은 exit status를 커널에 반환한 후 프로그램 이름과 exit status를 출력한 후 프로그램을 종료시킨다.
3. Exec() : 인자로 넘어온 이름을 가진 자식 프로세스를 실행한다. 이때 process\_execute 함수를 사용하며, 리턴 값으로 해당 프로세스의 pid 값을 반환한다.
4. Wait() : 인자로 들어온 pid 값을 가진 자식 프로세스가 종료될 때까지 기다린 후에, 자식 프로세스의 exit status 값을 반환하게 된다. 이때 synchronization 기법이 활용된다.
5. Read() : fd에서 size만큼의 byte를 buffer에 읽힌다.
6. Write() : buffer에서 size만큼의 byte를 fd에 쓴다.
   * 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

User level에서 시스템 콜 API를 호출하게 되면, system call을 통해 kernel에 요청을 보내게 된다. System call이 호출되면 user mode에서 kernel 모드로 전환이 된다. 앞서 구현했던 System call 중 halt를 예로 들면, halt() 함수에서 syscall0를 호출하게 된다. 이때 SYS\_HALT라는 시스템 콜 넘버가 넘어간다. Halt의 경우 인자의 개수가 0개이기 때문에 syscall0를 호출하며, 인자의 개수가 n개라면 syscalln이 호출된다. 호출된 syscall0에서는 system call number를 스택에 push한 후에, interrupt를 발생시킨다. 이때 user mode에서 kernel 모드로 전환이 되는 것이다. 이후 intr\_handler 함수를 불러, Interrupt Descriptor Table에서 인터럽트에 맞는 handler 함수의 주소를 찾는다. 그 후 Syscall\_handler 함수에 구현된 각 system call number에 해당하는 시스템 콜이 수행된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**

Server 다운 문제로 10/4 이전까지는 매뉴얼 및 pdf를 통해 각 단계의 flow chart를 구상하기만 하였다. 이후 10/5~10/6 동안 구현을 진행하였다. 10/5에는 Argument passing과 user memory access, halt(), exit(), read(), write(), exec(),wait() 시스템 콜 핸들러만 구현하였다. 이때 뼈대만 구현했기 때문에, process\_execute함수나 process\_wait 및 process\_exit 함수는 수정하지 않았다. 이후 10/6에 synchronization 기법을 활용하여 process\_wait, process\_exit, process\_execute 함수를 수정하여 wait() exec() exit() 세 개의 시스템 콜 handler 가 알맞은 동작을 수행할 수 있도록 하였다.

추가적으로 구현하는 Fibonacci, max\_of\_four\_int 시스템 콜은 10/7에 구현하였다.

* 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)**
  + 1. Argument passing

Userprog/process.c 내부에 존재하는 load 함수에서 argument passing을 수행한다. 이때 void stack\_esp(const char\* file\_name, void\*\* esp) 함수를 선언하고, 해당 함수를 load 함수 내에서 호출하였다. stack\_esp 함수의 동작은 크게 두 부분으로 나뉜다.

첫 번째는 공백을 기준으로 argument를 **파싱**하는 것이다. 이를 위해 strtok\_r 함수를 사용하여 공백을 기준으로 입력 문자열을 **토큰화**한 후, 각 argument를 argv 배열에 저장한다. 이때 각 토큰은 null 문자(\0)로 종료된다.

두 번째 단계는 파싱된 argument들을 **스택에 역순으로 저장**하는 것이다. 각 argument의 크기를 계산하여 esp를 적절히 조정한 후, strlcpy를 사용해 그 위치에 문자열을 복사한다. 각 argument가 스택에 복사된 후, 그 **주소**는 addr 배열에 저장된다. 이 주소 배열은 나중에 스택에 다시 저장된다.

스택에 데이터를 저장할 때, **word alignment**를 맞추기 위해 4바이트 단위로 스택을 정렬하는 과정이 필요하다. 이는 스택이 4바이트 경계에서 시작되도록 하여, CPU가 메모리에 접근할 때 성능을 최적화하기 위함이다.

마지막으로, argv 배열의 주소들(각 argument의 시작 주소)을 스택에 저장한 후, **argc** 값을 스택에 추가한다. 그런 다음, 프로그램이 시작할 때 사용할 **초기 스택 포인터** 값을 설정하여, 이 함수에서 수행할 작업을 완료한다.

* + 1. User memory access

Threads/vaddr.h에 정의되어 있는 is\_kernel\_vaddr() 함수를 활용하여 커널 영역에 접근하는지를 판단하고, 외에도 NULL인 경우(mapping되지 않은 경우)에도 비정상적인 접근으로 판단할 수 있도록 한다.

* + 1. System call

모든 system call들은 syscall.c의 syscall\_handler 함수 내에서 각각의 API에 따라 정의된 행동을 수행한다.

Halt() : shutdown\_power\_off() 함수를 호출한다.

Exit() : 파일 이름과 exit status를 프린트한 후, 현재 thread의 exit status를 업데이트한다. 그 후 thread\_exit()을 통해 프로그램을 종료시킨다.

Exec() : process\_execute() 함수를 호출한다.

Wait() : process\_wait() 함수를 호출한다. 이 함수는 process.c 내부에 정의되어 있으며, 주어진 코드를 수정하여 정상적으로 작동할 수 있도록 해 주었다.

Read() : input\_getc() 함수를 호출하여 STDIN을 수행한다.

Write() : putbuf() 함수를 호출하여 STDOUT을 수행한다.

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

* threadsthreads/init.c

main()

run\_actions()

run\_task()

process\_execute()

start\_process()

load()

Argument\_parsing

Construct\_stack

1. User Memory Access

Valid\_addr()

Syscall\_handler()

1. System Calls

텍스트, 폰트, 친필, 라인이(가) 표시된 사진

자동 생성된 설명

* 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

* Argument parsing

strlcpy(fn\_copy, file\_name, strlen(file\_name)+1);

  argc=0;

  token=strtok\_r(fn\_copy, " ", &next);

  argv[argc++] = token;

  while(token = strtok\_r(NULL, " ", &next)){

    argv[argc++] = token;

  }

Strtok\_r을 이용해 공백을 기준으로 file\_name을 token화한 후, argv에 인자들을 저장한다. 최종적으로 argc는 전체 argument의 개수를 저장하게 된다. 해당 함수의 첫 번째 인자는 분리할 문자열이고, 두 번째 인자는 구분자가 된다. pintos에서 argument parsing을 위해서는 공백을 기준으로 분리하기 때문에 “ “가 두 번째 인자로 전달되었다. 마지막으로 세 번째 인자는 분리된 문자열의 상태를 저장하며, 호출하며 계속해서 상태를 이어받는다.

* Construct stack

char\*\* addr;

  addr = (char \*\*) malloc(sizeof(char \*) \* argc);

  for(i = argc-1; i >= 0; i--){

    len = strlen(argv[i]) + 1;

    \*esp -= len;

    word\_align += len;

    strlcpy(\*esp, argv[i], len);

    addr[i] = \*esp; //주소 할당

  }

  /\* word alignment \*/

 if(word\_align %4 != 0){

    word\_align = 4 - (word\_align % 4);

  }else{

    word\_align = 0;

  }

  \*esp -= word\_align;

  \*esp-=4;

  \*\*(uint32\_t\*\*) esp = 0;

  for(i = argc-1;i >= 0; i--){

    \*esp -= 4;

    \*\*(uint32\_t\*\*) esp = addr[i];

  }

  \*esp -= 4;

  \*\*(uint32\_t\*\*)esp = \*esp + 4;

  \*esp -= 4;

  \*\*(uint32\_t\*\*)esp = argc;

  \*esp -= 4;

  \*\*(uint32\_t\*\*)esp = 0;

  free(addr);

argv 배열에 저장된 argument들을 stack에 역순으로 저장한다. 높은 주소에서 낮은 주소로 데이터가 쌓이는 stack의 성질을 고려하여, 마지막 argument가 가장 높은 위치에 저장되도록 하는 것이다. 우선 argument의 크기를 계산한 후 esp를 감소시켜 해당 위치에 strlcpy 함수를 이용해 문자열을 복사한다.

Strlcpy 함수는 버퍼 오버플로우를 방지하면서 안전하게 문자열을 복사할 수 있게 해 준다. Strlcpy에 넘어가는 첫 번째 인자는 문자열이 복사될 곳을 의미하며, 두 번째 인자는 원본 문자열이다. 마지막으로 세 번째 인자는 대상 버퍼의 크기를 제공하여 안전하게 값을 복사한다.

Argument에 주소를 할당한 후에는, 4 byte 단위로 스택을 정렬하기 위해 word alignment가 필요하다. 4바이트 경계에 맞지 않는 경우(모든 argument 길이가) 스택 포인터를 조정해 준다. 그후 각 argument의 주소를 stack에 저장하는데, 이때 stack pointer를 4바이트씩 줄이면서 주소를 저장해 준다.

1. User Memory Access

void valid\_addr(const void \*vaddr){

    if (vaddr == NULL || !is\_user\_vaddr(vaddr)){

        exit(-1);

    }

}

Valid\_addr 함수에서 인자로 받는 vaddr은 접근하고자 하는 주소를 의미한다. is\_kernel\_vaddr함수를 호출하여 현재 접근하려는 주소가 커널 영역의 주소인지 판단하고, 맞으면 exit(-1)을 통해 프로그램을 종료시킨다. 외에도 page directory 영역이 아니라면 (mapping되지 않았다면-> NULL) 프로그램을 종료시킨다.

텍스트, 폰트, 스크린샷, 친필이(가) 표시된 사진

자동 생성된 설명

Is\_kernel\_vaddr은 인자로 받은 주소가 PHYS\_BASE 이상일 경우, 즉 커널 영역일 경우 true를 반환한다.

1. System Calls

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

static void

syscall\_handler (struct intr\_frame \*f UNUSED)

{

    valid\_addr(f->esp);

    switch(\*(uint32\_t\*)(f->esp)){

        case SYS\_HALT:

          halt();

          break;

        case SYS\_EXIT:

          valid\_addr(f->esp + 4);

          exit(\*(uint32\_t\*)(f->esp + 4));

          break;

        case SYS\_EXEC:

          valid\_addr(f->esp + 4);

          f->eax = exec(\*(uint32\_t\*)(f->esp + 4));

          break;

        case SYS\_WAIT:

          valid\_addr(f->esp + 4);

          f->eax = wait( \*(uint32\_t\*)(f->esp+4));

          break;

        case SYS\_CREATE:

        break;

        case SYS\_REMOVE:

        break;

        case SYS\_OPEN:

        break;

        case SYS\_FILESIZE:

        break;

        case SYS\_READ:

          valid\_addr(f->esp+4);

          valid\_addr(f->esp+8);

          valid\_addr(f->esp+12);

          f->eax = read((int) \*(uint32\_t\*)(f->esp+4),(void\*)\*(uint32\_t\*)(f->esp +8),(unsigned) \* (uint32\_t\*)(f->esp+12));

          break;

        case SYS\_WRITE:

          valid\_addr(f->esp+4);

          valid\_addr(f->esp+8);

          valid\_addr(f->esp+12);

          f->eax = write((int) \*(uint32\_t\*)(f->esp+4),(const void\*)\*(uint32\_t\*)(f->esp +8),(unsigned) \* (uint32\_t\*)(f->esp+12));

          break;

        case SYS\_SEEK:

        break;

        case SYS\_TELL:

        break;

        case SYS\_CLOSE:

        break;

        case SYS\_FIBONACCI:

        break;

        case SYS\_MAX\_OF\_FOUR\_INT:

        break;

    }

}

Syscall\_handler 함수에서 각 시스템 콜 넘버에 대해 시스템 콜 처리를 해 준다.

시스템 콜 호출에 있어서 여러 주소에 접근해야 하는 경우, valid\_addr 함수를 호출하여 유효한 주소인지를 판단한다. f->esp는 시스템 콜에 해당하는 시스템 콜 넘버를 가리킨다.

텍스트, 폰트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명

Halt()의 경우, shutdown\_power\_off 함수를 호출하며 핀토스를 종료시킨다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Exit()의 경우 인자로 들어오는 status는 f->esp에서 4바이트를 더한 값이 저장되어 있다. 이 주소가 유효한지를 판단한 후에, exit 출력 포맷에 따라 파일이름: exit(status)를 출력한다. 이후 thread\_current에서 exit status를 업데이트하고, thread\_exit()으로 프로그램을 종료한다. Thread\_exit() 함수는 현재 thread를 모든 thread들이 들어가 있는 리스트에서 제거하고, thread의 상태를 THREAD\_DYING으로 업데이트한다. 그 후 schedule() 함수를 호출하여 다음 thread로 rescheduling한다.

Thread 구조체는 다음과 같다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

제공된 코드에서 추가적으로 선언한 변수 중 exit systemcall과 관련있는 변수는 exit\_status인데, 이는 현재 thread의 종료 코드를 저장하는 변수이다.

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

Exec 함수의 경우 handler에서 주소의 유효성을 판단한 후에 process\_execute 함수를 호출한다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Process\_execute에서 수정된 부분은 다음과 같다. 현재 file\_name에는 파일 이름만 저장되어 있는 것이 아니라 전체 입력열이 들어와 있기 때문에 공백을 기준으로 문자열을 파싱해 준다. 처음 문자열을 가질 때까지만 반복한 후에, 해당 파일 이름을 f\_name 변수에 저장한다. 이후 thread\_create을 할 때도 파싱한 파일 이름을 인자로 넘겨준다. 정상적으로 해당 함수가 작동했다면, tid값을 반환하고 그 값은 f->eax에 저장한다.

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

Wait의 경우 마찬가지로 주소의 유효성을 판단한 후에 process\_wait 함수를 실행한다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Wait을 구현함에 있어서 하나의 thread가 동작할 때 cpu 제어권이 넘어가지 않도록 semaphore를 이용했다. Thread에 선언했던 wait\_sema와 exit\_sema는 각각 자식 프로세스가 모든 작업을 마치고 종료될 떄까지 부모 프로세스가 기다릴 수 있도록 처리하는 역할, 부모 프로세스가 THREAD\_DYING 상태의 자식 프로세스를 리스트로부터 삭제할 때 자식프로세스가 종료되지 않도록 하는 역할을 담당한다.

이 함수는 인자로 받은 tid가 동일한 child thread를 찾을 때까지 child\_thread\_list를 순회한다. 그 후 tid가 동일한 자식 thread를 찾으면 세마포어를 이용해 동작이 잘 수행될 수 있도록한다. 과제 수행 전에 그려 본 플로우 차트는 다음과 같았다.

텍스트, 친필, 라인, 폰트이(가) 표시된 사진

자동 생성된 설명

따라서 process\_exit 부분에 semaphore를 up, down 해주는 부분을 추가하였다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

자식 프로세스가 종료되며 이전에 sema\_down을 했던 wait\_sema는 다시 up을 해주고, sema\_up을 했던 exit\_sema는 다시 down을 해준다.

텍스트, 스크린샷, 폰트, 디스플레이이(가) 표시된 사진

자동 생성된 설명

Read와 write의 경우 STDIN, STDOUT만 가능하도록 구현을 하면 됐다. 이 두 콜의 핸들러는 모두 입력받는 인자가 세 개이기 때문에, 각 주소들의 유효성을 판단한 후 매뉴얼에서 명시한 함수를 통해 구현하였다. Read에 사용되는 input\_getc() 함수는 키보드로부터 한 글자를 입력받는 함수인데, for 문을 돌면서 size 만큼 입력을 읽어온다. Write에 사용하는 putbuf의 경우 출력 버퍼에 데이터를 저장하고 출력한다.

1. Additional System calls

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

우선 syscall-nr.h에 다음과 같이 API를 추가했다.

폰트, 스크린샷, 텍스트, 타이포그래피이(가) 표시된 사진

자동 생성된 설명

또한 max\_of\_four\_int의 경우 4개의 정수를 인자로 받기 때문에 syscall4를 lib/user/syscall.c 함수에 추가로 선언해 주었다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

그후 시스템 콜 핸들러에 두 경우를 추가한 후에 각 경우에 사용할 수 있는 함수를 작성하였다. 이는 userprog/syscall.c, userprog/syscall.h에 작성하였다. Lib/user/syscall.c에도 추가하였다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

피보나치 함수는 n 번째 피보나치 수를 반환할 수 있도록 작성하였다.

텍스트, 스크린샷, 소프트웨어, 디스플레이이(가) 표시된 사진

자동 생성된 설명

Max\_of\_four\_int 함수는 값을 비교하며 가장 큰 값을 return할 수 있도록 하였다.

* 1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**
* **텍스트, 스크린샷, 폰트이(가) 표시된 사진

  자동 생성된 설명**