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

담당 교수 : 김영재

개발 기간 : 9/18 ~ 9/22

학번 : 20210428

이름 : 정현정

1. **개발 목표**

이번 Pintos 첫 번째 프로젝트는 user program을 실행시키기 위한 argument passing과 system call 구현이 목표이다. 기본적인 system call(halt, exit, exec, wait, read, write)와 더불어 fibonacci와 max\_of\_four\_int를 구현하고 additional.c를 통해 결과를 확인해야 한다.

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

User program을 실행시키려면 기본적으로 command로 입력을 받는 것들을 filename과 argument로 나누어 parsing하고 이들을 스택에 저장해야 한다. Argument Passing을 구현하면, 구분된 filename과 argument들을 80x86 convention에 맞게 스택에 쌓을 수 있다.

1. **User Memory Access**

User program을 실행시키는 과정에서 잘못된 주소에 접근할 수 있다. 커널 공간에 접근하려 하는 경우, NULL 포인터인 경우, 실제 메모리와 연결되지 않은 가상 주소가 그 예이다. 따라서 kernel panic을 방지하려면 주소에 대한 유효성 검사가 필요하다. User Memory Access를 통해 검사가 가능하다.

1. **System Calls**

User program 실행을 위해 halt, exit, exec, wait, read, write와 같은 기본적인 system call들을 구현한다. read와 write의 경우 STDIN과 STDOUT에 대해서만 구현한다.

* 1. **개발 내용**
* **Argument Passing**

우선 load 함수 내에서 filesys\_open으로 file을 열기 전에 입력으로 들어온 command를 파싱하는 작업을 수행한다. strtok\_r을 사용하여 공백을 기준으로 command를 분리한 후, argv 배열에 저장하고 argc 변수에는 총 인자의 개수를 저장한다. 이때 모든 파싱이 종료된 후 argv의 마지막에 NULL을 넣어 파싱이 종료되었음을 표시한다. 이후 argv 배열, argc, 스택 포인터를 갖고 80x86 convention에 맞게 스택에 쌓아준다. word alignment를 맞추기 위해서 주소값을 4의 배수에 맞춰주고 이후에도 계속 stack에 쌓으면서 스택 포인터를 4의 배수에 맞게 조절한다.

* **User Memory Access**

Pintos 상에서 invalid memory access는 커널 공간에 접근하려 하는 경우, NULL 포인터인 경우, 실제 메모리와 연결되지 않은 가상 주소에 접근하려는 경우를 말한다. Invalid memory access가 발생하면 page fault와 kernel panic이 발생할 수 있다. 이를 해결하기 위해 접근하려는 주소의 유효성 검사를 하는 함수를 만들고, 임의의 주소에 접근하려고 할 때마다 검사를 진행한다. 만약, invalid memory access를 시도한다면 exit(-1)을 호출한다.

* **System Calls**

시스템 콜은 프로그램 실행 도중 user mode에서 실행할 수 없는 동작을 수행하기 위해 kernel에 권한을 넘겨주겠다는 일종의 신호이다. 시스템 콜이 호출 되면 kernel mode로 바뀌고 system call 종류마다 정해진 특정 작업을 수행한다. 이번 프로젝트에서 구현해야 하는 system call들의 종류는 다음과 같다.

* + halt : shutdown\_power\_off을 통해 pintos를 종료시킨다.
  + wait : 자식 프로세스가 종료될 때까지 부모 프로세스의 작업을 중단시킨다.
  + exit : 현재 실행 중인 프로그램을 중단하고 종료 시 status code를 반환한다. 이를 통해 정상 종료 또는 비정상 종료를 구분할 수 있다.
  + exec :
  + read : STDIN 파일 디스크립터를 통해서 내용을 읽어들인다.
  + write : STDOUT 파일 디스크립터를 통해서 버퍼에 저장되어 있던 내용을 작성한다.
  + fibonnaci : 입력 받은 n값에 대해, n번째 피보나치 수를 반환한다.
  + max\_of\_four\_int : 입력 받은 4개의 정수 중 가장 큰 수를 반환한다.

프로그램에서 system call api가 호출되면 어떤 종류의 system call인지를 구분할 수 있는 system call 번호와 argument들이 스택에 쌓인다. 그 후 int $0x30 명령어에 의해 interrupt handler가 interrupt vector의 30번 주소에 위치해 있는 system call handler를 실행시킨다. System call handler는 스택에 접근하여 system call 번호를 체크하고 각각의 번호에 맞는 system call을 호출한다. System call이 종료되면 다시 user mode로 복귀한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

9/18 : 프로젝트 명세서 분석

9/19 : argument passing 구현

9/20 ~ 9/21 : user memory access, system calls 구현

9/22 : additional 구현 및 보고서 작성

* 1. **개발 방법**
* Argument passing

pintos의 기본 코드에 따르면 userprog/process.c의 load함수에서 내부적으로 file\_name인자를 갖고 file을 open한다. 이 부분을 수정하여 file\_name에 command의 다른 argument를 제외하고 정확한 파일명만 들어갈 수 있도록 한다. command의 파일명을 제외한 다른 argument들은 argv 배열에 저장한다. 이후 push\_stack이라는 함수를 호출하여 80x86 calling convention에 맞게 stack에 쌓고 word alignment를 수행하여 적정하게 stack pointer인 esp를 조정한다.

* User Memory Access

userprog/syscall.c에 check\_valid\_addr 함수를 추가하여서 주소에 접근하려고 할 때마다 주소의 유효성 검사를 수행한다. 접근하려는 주소가 NULL 포인터인지, 커널의 공간인지, 실제 physical memory와 pagedir에서 매핑되지 않은 포인터인지를 순서대로 검사한다. 커널 공간으로의 접근은 is\_kernel\_vaddr 함수를, pagedir으로의 접근은 pagedir\_get\_page 함수를 사용한다.

* System calls

userprog/syscall.c에서 명세서에 설명되어 있는 system call 함수들을 구현한다. 이들은 system call handler에서 호출되며 system call handler는 각각의 system call number에 맞는 system call 함수를 호출하는 역할을 한다.

system call handler는 system call 함수를 호출하기 전에 먼저 stack pointer부터 system call number마다 필요한 인자 개수만큼 for loop을 돌며 접근하려는 stack 주소의 유효성 검사를 수행한다. (check\_valid\_addr 함수 사용) 이후 if-else문을 거쳐 system call number마다 적절한 system call 함수를 호출한다. 반환값을 갖는 system call의 경우에는 f->eax에 그 값을 저장한다.

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

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

1. User Memory Access

도표, 스케치, 그림, 텍스트이(가) 표시된 사진

자동 생성된 설명

1. System Calls

텍스트, 스크린샷, 도표이(가) 표시된 사진

자동 생성된 설명

* 1. **제작 내용**

1. Argument Passing
   1. load 함수 수정

bool

load (const char \*file\_name, void (\*\*eip) (void), void \*\*esp)

{

  //생략

  //file\_name 파싱

  int argc = 0;

  char\* argv[64];

  char\* next\_ptr;

  char tmp[128];

  strlcpy(tmp, file\_name, strlen(file\_name) + 1);

  char\* ret\_ptr = strtok\_r(tmp, " ", &next\_ptr);

  while(ret\_ptr){

    argv[argc++] = ret\_ptr;

    ret\_ptr = strtok\_r(NULL, " ", &next\_ptr);

  }

  argv[argc] = NULL;

  file\_name = argv[0];

  /\* Open executable file. \*/

  file = filesys\_open (file\_name);

  if (file == NULL)

    {

      printf ("load: %s: open failed\n", file\_name);

      goto done;

    }

  /\* Set up stack. \*/

  //4KB의 stack page 할당하는 함수

  //void \*\*esp

  //\*esp = PHYS\_BASE;

  if (!setup\_stack (esp))

    goto done;

  //스택 쌓기

  push\_stack(argv, argc, esp);

  //생략

}

load 함수에서는 command 파싱과 stack에 argument 쌓기로 크게 두 가지 작업을 추가하였다.

먼저 process\_execute()부터 함수 호출을 따라가다 보면 load 함수에 인자로 전달된 file\_name에는 command line 전체가 들어가 있다. 따라서 이를 실제 file\_name만 전달할 수 있도록 공백을 기준으로 파싱을 수행한다. 파싱한 인자들은 argv 배열에 저장하고 argc에는 총 argument의 개수를 넣는다. 또 argv 배열의 마지막에는 파싱이 종료되었다는 의미의 NULL을 넣는다. 그러면 argv[0], 즉 file\_name만 갖고 filesys\_open을 수행할 수 있다.

* 1. push\_stack() 함수 구현

void push\_stack(char\*\* argv, int argc, void\*\* esp){

  //\*esp = 스택 포인터 = 주소값을 넣어야함

  int len = 0;

  for(int i = argc - 1; i >= 0; i--){

    (\*esp) -= strlen(argv[i]) + 1;

    len += strlen(argv[i]) + 1;

    strlcpy(\*esp, argv[i], strlen(argv[i]) + 1);

    argv[i] = \*esp;

  }

  //4byte alignment 필요

  if(len % 4 != 0){

    int alignment = 4 - len % 4;

    for(int i = 0; i < alignment; i++){

      (\*esp)--;

      \*\*(uint8\_t\*\*)esp = 0;

    }

  }

  for(int i = argc; i >= 0; i--){

    (\*esp) -= 4;

    \*\*(uint32\_t\*\*)esp = argv[i];

  }

  \*esp -= 4;

  \*\*(char \*\*\*)esp = \*esp + 4;

  (\*esp) -= 4;

  \*\*(uint32\_t\*\*)esp = argc;

  (\*esp) -= 4;

  \*\*(void \*\*\*)esp = 0;

}

둘째로 push\_stack() 함수를 통해 argv 배열에 저장한 argument들을 80x86 calling convention에 맞게 stack에 쌓는다. 4의 배수로 word alignment도 맞춰주는 과정도 push\_stack() 함수에서 수행한다.

1. User Memory Access
   1. check\_valid\_addr() 함수 구현

void check\_valid\_addr(uint32\_t\* esp, int cnt){

  for(int i = 0; i <= cnt; i++){

    //커널 공간인지, null 포인터인지

    if((esp + 4 \* i) == NULL || is\_kernel\_vaddr(esp + 4 \* i)){

      exit(-1);

    }

    //a pointer to unmapped virtual memory

    if(pagedir\_get\_page(thread\_current()->pagedir, esp + 4 \* i) == NULL){

      exit(-1);

    }

  }

}

check\_valid\_addr() 함수는 esp와 argument의 개수를 인자로 받는다. 그리고 esp를 통해 argument의 개수만큼 접근하려는 주소의 유효성을 검사한다. 조건문을 통해 NULL 포인터인지, 커널의 공간인지(is\_kernel\_vaddr() 사용), unmapped virtual memory인지(pagedir\_get\_page() 사용)인지를 체크한다. 만약 잘못된 접근이라면 exit system call을 호출하여 비정상 종료임을 표시한다.

* 1. page\_fault() 함수 수정

static void

page\_fault (struct intr\_frame \*f)

{

  bool not\_present;  /\* True: not-present page, false: writing r/o page. \*/

  bool write;        /\* True: access was write, false: access was read. \*/

  bool user;         /\* True: access by user, false: access by kernel. \*/

  void \*fault\_addr;  /\* Fault address. \*/

  //생략

   if(!user || is\_kernel\_vaddr(fault\_addr)) exit(-1);

  /\* To implement virtual memory, delete the rest of the function

     body, and replace it with code that brings in the page to

     which fault\_addr refers. \*/

  printf ("Page fault at %p: %s error %s page in %s context.\n",

          fault\_addr,

          not\_present ? "not present" : "rights violation",

          write ? "writing" : "reading",

          user ? "user" : "kernel");

  kill (f);

}

테스트를 돌리던 도중 지속적으로 kernel panic이 발생하는 것을 보았고, kernel panic이 page\_fault로 인해 발생하는 것을 확인하였다. 이를 해결하기 위해 page\_fault() 함수 안에 if문으로 kernel 공간에 접근하려는 경우와 kernel access인 경우 exit system call을 호출하여 -1을 종료 코드로 전달하였다.

1. System Calls
   1. syscall\_handler() 함수 구현

static void

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

{

  //syscall number마다 필요한 argc 저장하는 배열

  int arg\_cnt[22];

  arg\_cnt[SYS\_HALT] = 0;

  arg\_cnt[SYS\_EXIT] = 1;

  arg\_cnt[SYS\_EXEC] = 1;

  arg\_cnt[SYS\_WAIT] = 1;

  arg\_cnt[SYS\_READ] = 3;

  arg\_cnt[SYS\_WRITE] = 3;

  arg\_cnt[MAX\_OF\_FOUR\_INT] = 4;

  arg\_cnt[FIBONACCI] = 1;

  uint32\_t\* esp = (uint32\_t\*)f->esp;

  check\_valid\_addr(esp, arg\_cnt[\*esp]);

  if (\*esp == SYS\_HALT) {

    halt();

  }

  else if (\*esp == SYS\_EXIT) {

    int status = (int)\*(esp + 1);

    exit(status);

  }

  else if (\*esp == SYS\_EXEC) {

    char \*cmd\_line = (char\*)\*(esp + 1);

    f->eax = exec(cmd\_line);

  }

  else if (\*esp == SYS\_WAIT) {

    tid\_t pid = (tid\_t)\*(esp + 1);

    f->eax = wait(pid);

  }

  else if (\*esp == SYS\_READ) {

    int fd = (int)\*(esp + 1);

    void\* buffer = (void\*)\*(esp + 2);

    unsigned size = (unsigned)\*(esp + 3);

    f->eax = read(fd, buffer, size);

  }

  else if (\*esp == SYS\_WRITE) {

    int fd = (int)\*(esp + 1);

    void\* buffer = (void\*)\*(esp + 2);

    unsigned size = (unsigned)\*(esp + 3);

    f->eax = write(fd, buffer, size);

  }

  else if(\*esp == MAX\_OF\_FOUR\_INT){

    int a = (int)\*(esp + 1);

    int b = (int)\*(esp + 2);

    int c = (int)\*(esp + 3);

    int d = (int)\*(esp + 4);

    f->eax = max\_of\_four\_int(a, b, c, d);

  }

  else if(\*esp == FIBONACCI){

    int n = (int)\*(esp + 1);

    f->eax = fibonacci(n);

  }

  else {

    exit(-1);

  }

}

syscall\_handler() 함수는 stack pointer인 f->esp에 접근하기 전에 먼저 각각의 system call 함수에 대한 기본 정보인 argument 개수를 arg\_cnt 배열에 저장한다. 이후 check\_valid\_addr() 함수에 stack pointer와 요청이 들어온 system call이 필요로 하는 인자의 개수를 넘겨주어 주소의 유효성 검사를 수행한다. 그 다음 esp에 담겨 있는 system call number에 대응되는 system call 함수들을 조건문을 통해 구분하여 실행시킨다. 이때 stack pointer를 조절하여 필요한 인자들을 스택에서 꺼내 변수에 저장한다. 그리고 저장한 값들을 system call 함수들의 파라미터로 전달한다.

* 1. system call 구현
     1. halt()

void halt(void){

  shutdown\_power\_off();

}

pintos의 manual을 참고하여 shutdown\_power\_off()를 사용한다. 그 이외의 작업은 수행하지 않는다.

* + 1. exit()

void exit(int status){

  struct thread\* current\_thread = thread\_current();

  char\* next\_ptr;

  char tmp[128];

  strlcpy(tmp, current\_thread->name, strlen(current\_thread-> name) + 1);

  char\* ret\_ptr = strtok\_r(tmp, " ", &next\_ptr);

  printf("%s: exit(%d)\n", ret\_ptr, status);

  current\_thread -> exit\_status = status;

  thread\_exit();

}

thread의 생성과 실행 과정을 따라오다 보면, thread\_current() 함수를 통해 현재 실행 중인 thread 구조체의 name에 command 전체를 16바이트 배열에 저장하고 있음을 확인할 수 있다. 따라서 terminate 문구의 형식에 맞게 현재 실행 중인 thread의 이름만을 받아오기 위해서는 strtok\_r을 사용한 토큰화가 필요하다. 토큰화를 수행한 후, thread 구조체에 인자로 받은 status를 종료 상태를 표시하는 필드에 저장한다. 따라서 thread 구조체에 종료 상태를 표시하는 새로운 필드가 필요하다. 새로운 필드를 생성하여 종료 상태를 저장하는 이유는 thread 구조체의 status 필드를 직접적으로 조작하면 안 되기 때문이다. *int exit\_status;* 필드를 구조체에 추가했다. terminate 문구를 출력한 후 thread\_exit() 함수를 호출한다. thread\_exit() 함수는 내부적으로 process\_exit() 함수를 호출한다.

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)

    {

      /\* Correct ordering here is crucial.  We must set

         cur->pagedir to NULL before switching page directories,

         so that a timer interrupt can't switch back to the

         process page directory.  We must activate the base page

         directory before destroying the process's page

         directory, or our active page directory will be one

         that's been freed (and cleared). \*/

      cur->pagedir = NULL;

      pagedir\_activate (NULL);

      pagedir\_destroy (pd);

    }

  sema\_up(&(cur->wait\_exit)); //up하는 순간 부모 프로세스에서 동작 시작

  sema\_down(&(cur->wait\_free)); //부모 프로세스에서 삭제한 뒤 메모리가 풀리도록 대기

}

process\_exit()에서는 세마포어를 사용하여서 process\_wait()에서 자식 프로세스의 종료를 기다리고 있던 부모 프로세스에서의 동작을 수행하고, 부모 프로세스에서 자식 프로세스가 삭제된 뒤 자식 프로세스의 메모리 자원이 해제되도록 한다. 따라서 총 2개의 세마포어가 필요하다. 만약 두번째 세마포어(cur->wait\_free)를 사용하지 않으면 자식 프로세스의 메모리가 해제되기 전에 부모 프로세스의 리스트에서 자식 프로세스가 삭제되는 일이 발생한다. 이를 위해 thread 구조체를 다음과 같이 수정해야 한다.

struct thread

  {

    /\* Owned by thread.c. \*/

    tid\_t tid;                          /\* Thread identifier. \*/

    enum thread\_status status;          /\* Thread state. \*/

    char name[16];                      /\* Name (for debugging purposes). \*/

    uint8\_t \*stack;                     /\* Saved stack pointer. \*/

    int priority;                       /\* Priority. \*/

    struct list\_elem allelem;           /\* List element for all threads list. \*/

    /\* Shared between thread.c and synch.c. \*/

    struct list\_elem elem;              /\* List element. \*/

#ifdef USERPROG

    /\* Owned by userprog/process.c. \*/

    uint32\_t \*pagedir;                  /\* Page directory. \*/

    struct semaphore wait\_exit; //자식 프로세스 종료 대기

    struct semaphore wait\_free; //메모리 해제 대기

    int exit\_status; //종료 상태

    struct list child\_threads; //자식 프로세스 모아두는 링크드리스트

    struct list\_elem child\_elem; //내가 자식 프로세스일 때 부모 프로세스의 child\_threads에서의 포인터 정보

#endif

    /\* Owned by thread.c. \*/

    unsigned magic;                     /\* Detects stack overflow. \*/

  };

* + 1. wait()

int wait(tid\_t pid){

  return process\_wait(pid);

}

system call 함수인 wait() 자체는 단순하게 process\_wait()함수를 호출하고 반환값을 return한다. 수정된 process\_wait() 함수는 다음과 같다.

int

process\_wait (tid\_t child\_tid)

{

  struct list\_elem\* elem;

  struct list\* child\_list = &(thread\_current()->child\_threads);

  int exit\_code = -1;

  for(elem = list\_begin(child\_list); elem != list\_end(child\_list); elem = list\_next(elem)){

    struct thread\* child = list\_entry(elem, struct thread, child\_elem);

    if(child->tid == child\_tid){

      sema\_down(&(child->wait\_exit));

      exit\_code = child->exit\_status;

      list\_remove(&(child->child\_elem));

      sema\_up(&(child->wait\_free));

    }

  }

  return exit\_code;

}

* + 1. exec()

int exec(char \*cmd\_line){

  return process\_execute(cmd\_line);

}

exec() 함수는 단순히 process\_execute() 함수의 반환값을 return한다.

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;

  if(file\_name == NULL || strlen(file\_name) == 0){

    return -1;

  }

  char\* next\_ptr;

  char tmp[128];

  strlcpy(tmp, file\_name, strlen(file\_name) + 1);

  char\* ret\_ptr = strtok\_r(tmp, " ", &next\_ptr);

  struct file\* file = filesys\_open(ret\_ptr);

  if(file == NULL){

    return -1;

  }

  strlcpy (fn\_copy, file\_name, PGSIZE);

  /\* Create a new thread to execute FILE\_NAME. \*/

  tid = thread\_create (ret\_ptr, PRI\_DEFAULT, start\_process, fn\_copy);

  if (tid == TID\_ERROR)

    palloc\_free\_page (fn\_copy);

  return tid;

}

process\_execute() 함수는 먼저 인자로 전달받은 file\_name이 NULL인지, 즉 길이가 0인 문자열인지를 확인한다. 만약 file\_name이 NULL 포인터라면 -1을 반환한다. 또 코드를 따라가다 보면 인자로 전달받은 file\_name이 command line 전체를 담고 있음을 알 수 있다. 따라서 정확하게 파일명만 filesys\_open으로 열 수 있도록 토큰화를 해주어야 한다. strtok\_r을 사용해서 공백을 기준으로 한 번 파싱을 수행하고 filesys\_open을 진행한다. 만약 filesys\_open의 반환값이 NULL인 경우, 즉 없는 파일을 실행시키려 하는 경우에도 -1을 반환하도록 함수를 수정했다.

* + 1. write()

int write(int fd, void\* buffer, unsigned size){

  putbuf(buffer, size);

  return size;

}

이번 프로젝트에서는 STDOUT에 대한 write만을 구현하는 것이므로 putbuf를 사용해서 buffer에 size만큼 쓰고 size를 반환하도록 구현하였다.

* + 1. read()

int read(int fd, void\* buffer, unsigned size){

  int i;

  uint8\_t temp;

  for(i = 0; (i < size) && (temp = input\_getc()); i++){

    \*(uint8\_t\*)(buffer + i) = temp;

  }

  return i;

}

이번 프로젝트에서 read는 STDIN에 대해서만 입력을 구현한다. input\_getc() 함수를 사용해서 입력을 받는다. 그리고 읽어들인 글자의 수만큼을 반환한다.

1. Additional System calls
   1. fibonacci()

int fibonacci(int n){

  int a = 1;

  int b = 1;

  int res = 0;

  for(int i = 2; i < n; i++){

    res = a + b;

    a = b;

    b = res;

  }

  return res;

}

fibonacci() 함수는 인자로 받은 n번째 피보나치 수를 반환하는 system call이다. for문을 활용하여 값을 계산한다.

* 1. max\_of\_four\_int()

int max\_of\_four\_int(int a, int b, int c, int d){

  int max1, max2;

  if(a >= b) max1 = a;

  else if(a < b) max1 = b;

  if(c >= d) max2 = c;

  else if(c < d) max2 = d;

  if(max1 >= max2) return max1;

  else if(max1 < max2) return max2;

}

인자로 a~d까지 총 4개의 정수를 전달받고 이들 중 가장 큰 정수를 반환한다.

* 1. syscall-nr.h 수정

#ifndef \_\_LIB\_SYSCALL\_NR\_H

#define \_\_LIB\_SYSCALL\_NR\_H

/\* System call numbers. \*/

enum

  {

    /\* Projects 2 and later. \*/

    SYS\_HALT,                   /\* Halt the operating system. \*/

    SYS\_EXIT,                   /\* Terminate this process. \*/

    SYS\_EXEC,                   /\* Start another process. \*/

    SYS\_WAIT,                   /\* Wait for a child process to die. \*/

    SYS\_CREATE,                 /\* Create a file. \*/

    SYS\_REMOVE,                 /\* Delete a file. \*/

    SYS\_OPEN,                   /\* Open a file. \*/

    SYS\_FILESIZE,               /\* Obtain a file's size. \*/

    SYS\_READ,                   /\* Read from a file. \*/

    SYS\_WRITE,                  /\* Write to a file. \*/

    SYS\_SEEK,                   /\* Change position in a file. \*/

    SYS\_TELL,                   /\* Report current position in a file. \*/

    SYS\_CLOSE,                  /\* Close a file. \*/

    /\* Project 3 and optionally project 4. \*/

    SYS\_MMAP,                   /\* Map a file into memory. \*/

    SYS\_MUNMAP,                 /\* Remove a memory mapping. \*/

    /\* Project 4 only. \*/

    SYS\_CHDIR,                  /\* Change the current directory. \*/

    SYS\_MKDIR,                  /\* Create a directory. \*/

    SYS\_READDIR,                /\* Reads a directory entry. \*/

    SYS\_ISDIR,                  /\* Tests if a fd represents a directory. \*/

    SYS\_INUMBER,                 /\* Returns the inode number for a fd. \*/

    MAX\_OF\_FOUR\_INT,

    FIBONACCI

  };

#endif /\* lib/syscall-nr.h \*/

syscall-nr.h에 fibonacci와 max\_of\_four\_int를 위한 system call number를 추가한다.

* 1. lib/user/syscall.c 수정

#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 max\_of\_four\_int(int a, int b, int c, int d){

  return syscall4(MAX\_OF\_FOUR\_INT, a, b, c, d);

}

int fibonacci(int n){

  return syscall1(FIBONACCI, n);

}

lib/user/syscall.c에 max\_of\_four\_int를 위한 syscall4를 새롭게 정의하였다. 그리고 fibonacci는 인자를 하나만 받기 때문에 syscall1을, max\_of\_four\_int는 인자를 4개 받기 때문에 syscall4를 호출한다.

* 1. additional.c 구현

#include <stdio.h>

#include <stdlib.h>

#include "../lib/user/syscall.h"

int main(int argc, char\* argv[]){

    int a = atoi(argv[1]);

    int b = atoi(argv[2]);

    int c = atoi(argv[3]);

    int d = atoi(argv[4]);

    printf("%d ", fibonacci(a));

    printf("%d\n", max\_of\_four\_int(a, b, c, d));

    return 0;

}

fibonacci와 max\_of\_four\_int를 테스트 하기 위한 additional.c를 구현했다. atoi() 함수를 사용하여서 argv에 저장된 argument들을 정수 자료형으로 바꿔주었다. 이들을 fibonacci와 max\_of\_four\_int() 함수의 인자로 전달해주었다. 그리고 반환값을 STDOUT으로 출력한다.

* 1. Makefile 수정

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

# Should work from project 2 onward.

cat\_SRC = cat.c

cmp\_SRC = cmp.c

cp\_SRC = cp.c

echo\_SRC = echo.c

halt\_SRC = halt.c

hex-dump\_SRC = hex-dump.c

lineup\_SRC = lineup.c

ls\_SRC = ls.c

recursor\_SRC = recursor.c

rm\_SRC = rm.c

additional\_SRC = additional.c

# Should work in project 3; also in project 4 if VM is included.

bubsort\_SRC = bubsort.c

matmult\_SRC = matmult.c

mcat\_SRC = mcat.c

mcp\_SRC = mcp.c

# Should work in project 4.

mkdir\_SRC = mkdir.c

pwd\_SRC = pwd.c

shell\_SRC = shell.c

include $(SRCDIR)/Make.config

include $(SRCDIR)/Makefile.userprog

examples/Makefile에서 make를 하였을 때 additional.c도 컴파일 될 수 있도록 Makefile을 수정했다.

**시험 및 평가 내용**

* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**

**텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

* **테스트 수행 결과**

**스크린샷, 패턴, 패브릭, 예술이(가) 표시된 사진

자동 생성된 설명**

**텍스트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명**