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

담당 교수 : 문의현 교수님

학번 / 이름 : 20191112 / 심은서

개발 기간 : 2022.10.05 ~ 2022.10.31

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

기존의 pintos에서는 user program을 실행할 수 없는 상태로, echo x를 입력했을 때 아무런 처리가 되지 않았다. 따라서 user program을 실행하기 위한 기본적인 구조를 만드는 것이 이번 프로젝트의 목표이다.

크게 네가지 내용으로 나눌 수 있다. 첫째로, user program의 system call을 처리하기 위하여 입력된 명령어를 공백단위로 파싱한 후 이를 convention에 맞게 올바르게 stack에 저장한다. 둘째로, system call을 올바르게 처리하기 위한 system call handler를 구현하여 명령에 맞는 함수를 연결한다. 셋째로, user가 잘못된 영역에 memory access를 하지 않도록 검사한다. 넷째로, fibonacci와 max\_of\_four\_int라는 새로운 system call을 직접 구축하여 사용할 수 있도록 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

사용자가 입력한 argument를 공백 단위로 분리하여 stack에 분리된 argument들과 argument의 갯수, argument의 주소 등 필요한 정보들을 올바르게 저장한다.

1. User Memory Access

stack pointer가 null pointer이거나 kernel virtual memory, 또는 맵핑되지 않은 virtual memory에 접근하는 경우를 막고 잘못된 접근이 발생할 경우 exit(-1)을 호출한다.

1. System Calls

user program에서 호출한 system call을 system call handler를 통해서, argument passing단계에서 stack에 저장된 값들을 이용해 적절한 함수를 호출함으로써 사용자가 원하는 작업을 처리한다.

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

먼저 사용자가 입력한 명령어를 파싱한다. 즉, echo x를 입력했을 때 이 command는 ‘echo x’라는 전체 덩어리로 들어오지만 이 명령을 처리하기 위해서는 ‘echo’와 ‘x’로 분리하는 작업이 필요하다. command가 공백 단위로 분리되어 저장되었다면, 이를 아래와 같은 convention에 맞게 stack에 쌓아주어야 한다. Table

Description automatically generated

따라서 분리된 argument를 역순으로 stack에 넣은 뒤 word-align 즉 4byte 단위로 데이터가 저장될 수 되도록 하는 작업을 수행한다. 따라서 현재까지 저장된 데이터의 크기가 4의 배수가 아니라면 4의 배수가 될 수 있을만큼 dummy 값을 넣는다. 이후엔 argument의 주소값을 역순으로 넣고, argument들의 개수(argc)를 넣은 뒤 마지막으로 return address를 넣어준다.

* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명
  + Invalid memory access를 어떻게 막을 것인지 설명

user program에서 부적절한 pointer를 반환하여 잘못된 주소에 접근하지 않도록 예외처리를 해주어야 문제가 발생하지 않는다. 부적절한 pointer를 반환하는 경우는 크게 세가지 경우로 나눌 수 있다. 첫째로 null pointer일 경우인데, 이는 null여부를 확인하여 막을 수 있다. 둘째로 kernel virtual memory에 접근했을 경우인데, 이는 pintos에서 제공하는 is\_user\_vaddr()와 is\_kernel\_vaddr()을 사용하여 검사할 수 있다. 셋째로 맵핑되지 않은 virtual memory에 접근했을 경우인데, 이는 pagedir\_get\_page()를 사용하여 검사할 수 있다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

pintos에서는 memory를 user virtual memory와 kernel virtual memory로 나누어 관리함으로써 각 프로세스와 커널 코드를 보호한다. 따라서 안정성 있는 시스템이 구축될 수 있으나, 이로인해 user program이 kernel memory에 접근하지 못하게 되고 kernel에 구현되어있는 다양한 기능들을 사용할 수 없게 된다. 이때 system call이라는 매개체를 통해 user가 kernel에 접근할 수 있도록 하는 것이다.

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

이번 프로젝트에서 개발한 system call은 추가 구현하는 fibonacci와 max\_of\_four\_int를 포함하여 총 8가지이다.

* + - halt() : shutdown\_power\_off()함수를 호출하여 pintos를 terminate시킨다.
    - exit() : 현재 실행중인 user program을 종료하고 status를 반환한다.
    - exec() : child process를 생성하고 새로 생성된 process의 id를 반환한다.
    - wait() : child process가 끝나기 전에 parent process가 종료되지 않도록 기다리게 한다. child tread의 id의 유효성을 검사하고 child thread가 끝나면 exit status를 가져온다.
    - write() : stdout 기능을 수행하며 putbuf() 함수를 이용한다.
    - read() : stdin 기능을 수행하며 input\_getc() 함수를 이용한다.
    - fibonacci() : 인자 n에 대하여 피보나치 수열의 n번째 수를 반환한다.
    - max\_of\_four\_int() : 인자 a,b,c,d에 대하여 가장 큰 숫자를 반환한다.
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

system call API가 호출되면 user mode에서 kernel mode로 전환하여 interrupt handler를 통해 system call을 처리한다. 먼저 main()의 run\_actions(argv)가 호출되고 그 안에서 run\_task()가 process\_execute()를 실행하게 된다. 어떤 작업 요청했는지 확인하기 위하여 system call number를 가지고 system call table을 참조한다. 확인되었다면 start\_process를 통해 user program을 실행한 뒤 종료되면 exit이 호출된다. 부모 process는 child process가 종료될 때 까지 기다리고, 모든것이 완료되면 다시 kernel mode에서 user mode로 되돌아간다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

10.05-10.15 : 명세서 분석 및 개발 내용 이해

10.16-10.22 : argument passing 및 accessing user memory 구현

10.23-10.28 : system call, system call handler 구현

10.29-10.31 : 코드 리뷰 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

사용자의 명령어를 파싱하기 위해 userprog/process.c의 load()함수에서 전달받은 file\_name을 strtok\_r()을 이용하여 공백 단위로 파싱한다. 파싱한 명령어 조각의 배열을 이용하여 convention에 맞게 stack에 올바른 데이터를 쌓는데, \*esp를 4씩 조정하면서 포인터를 옮기며 올바른 위치에 데이터를 저장한다.

* user memory access

올바른 메모리 접근을 관리하기 위하여 userprog/exception.c에서 null pointer 여부 확인과 is\_kernel\_vaddr()를 이용해 주소값이 올바른지 확인하고 아니라면 exit(-1)을 호출함으로써 예외처리를 한다. 또한 userprog/syscall.c에서 각 system call별로 작업을 수행하기 전에 is\_user\_vaddr()를 이용해 현재 접근하고자 하는 메모리 주소가 유효한지 체크한다.

* system calls

lib/syscall\_nr.h에서 system call number를 확인하여, 해당하는 system call들을 userprog/syscall.c의 syscall\_handler 함수에서 switch문을 통해 구현한다. 각 switch문에서는 해당하는 system call에서 필요한 함수들을 호출하도록 하며, 이 함수들 또한 userprog/syscall.c에 구현한다. 새로 만든 함수들에 대한 프로토타입을 userprog/syscall.h에 정의해야한다. 각 system call은 f->\*esp와 같이 메모리에 접근하여 값을 참조함으로써 구현할 수 있다. 또한 반환값이 존재하는 경우에는 이를 f->\*eax에 저장한다. wait() 기능을 위해서는 userprog/process.c의 process\_wait()함수를 수정하여 사용하는데, semaphore를 이용하여 child thread가 죽을 때 까지 parent thread가 기다리도록 한다. 이를 위해서는 threads/synch.h의 struct semaphore를 활용한다.

추가적인 system call인 fibonacci와 max\_of\_four\_int함수를 위해 lib/syscall-nr.h에 두 함수에 대한 number를 추가해준다. 또한 max\_of\_four\_int함수에서 인자를 4개로 하는 system call이 발생하는데 이를 처리하기 위하여 lib/user/syscall.c에 syscall4()를 정의한다. 명세서에 맞게 fibonacci와 max\_of\_four\_int의 수행 결과를 출력하기 위하여 examples 디렉토리에 additional.c를 생성하여 두 함수를 올바르게 처리하여 결과를 출력하도록 하고, makefile에 additional을 추가하여 컴파일 될 수 있도록 한다.

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

Diagram

Description automatically generated

1. User Memory Access

Diagram

Description automatically generated

1. System Calls

Diagram

Description automatically generated

* 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

userprog/process.c의 load()에서 넘겨받은 file\_name을 parsing해야 한다.

  /\*TODO: parse file name\*/

*char* \*\* argv;

*int* argc = 0;

*char* \*token;

*char* \*ptr1;

*char* \*ptr2;

*char* \*fn\_copy;

  fn\_copy = palloc\_get\_page (0);

  if (fn\_copy == NULL)

    return TID\_ERROR;

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

  token = strtok\_r(fn\_copy, " ", &ptr1);

  while(token){

    argc ++;

    token = strtok\_r(NULL, " ", &ptr1);

  }

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

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

  token = strtok\_r(fn\_copy, " ", &ptr2);

  for (*int* i=0; i<argc; i++){

    argv[i]=token;

    token = strtok\_r(NULL, " ", &ptr2);

  }

file\_name은 const char\* 타입이므로 strlcpy를 통해 fn\_copy변수로 hard copy하여 처리를 시작한다. 먼저 공백 단위로 명령어를 나누었을 때 인자가 몇개인지를 확인하기 위하여 while문을 통해 명령어의 끝까지 탐색하며 공백단위로 token을 끊으면서 argc값을 늘린다. 이 작업을 마치고나면 argc변수에 명령어가 몇개의 인자로 이루어져있는지 알 수 있다. 따라서 이 argc 변수를 이용하여 다시 for문을 통해 각 인자를 argv배열에 저장한다. 이로써 argv배열에는 공백단위로 끊어진 인자가 하나씩 저장된다. 이렇게 argument parsing을 마쳤다면 이를 convention에 맞게 stack에 저장해주어야 한다.

*int* argv\_len = 0;

*char* \*\* argv\_addr = (*char* \*\*)malloc(sizeof(*char*\*)\*argc); //인자들의 주소 저장

  //push arguments

  for (*int* i=argc-1; i>=0; i--){

    \**esp* -= strlen(argv[i])+1;

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

    argv\_len += strlen(argv[i])+1;

    argv\_addr[i] = \**esp*;

  }

  //push word align

*int* word\_align = 0;

  if (argv\_len%4 != 0){ //전체 사이즈가 4의 배수가 아니면

    \**esp* -= 4 - argv\_len % 4; //4의 배수로 맞춰준다

  }

  //push NULL

  \**esp* -= 4;

  \*\*(uint32\_t \*\*)*esp* = 0;

  //push address of arguments

  for (*int* i=argc-1; i>=0; i--){

    \**esp* -= 4;

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

  }

  //push address of argv

  \**esp* -= 4;

  \*\*(uint32\_t \*\*)*esp* = \**esp* + 4;

  //push argc

  \**esp* -= 4;

  \*\*(uint32\_t \*\*)*esp* = argc;

  //push return address

  \**esp* -= 4;

  \*\*(uint32\_t \*\*)*esp* = 0;

  free(argv);

  free(argv\_addr);

\*esp는 stack pointer이므로 이를 4씩 조정해가며 올바른 위치에 데이터를 삽입한다.

stack에 첫번째로 넣어야 할 것은 parsing을 통해 나뉘어 argv배열에 순서대로 저장된 각 argument들인데, 역순으로 저장해야 하므로 int i=argv-1부터 시작하여 i를 1씩 줄이며 for문을 진행한다. 데이터의 크기만큼 \*esp를 조정하게 되는데, 문자열의 마지막에 ‘\0’가 들어가야하므로 strlen(argv[i])+1만큼씩 esp를 이동시킨다. 이 다음에는 word align작업을 통해 데이터에 4byte 단위로 접근할 수 있도록 해야한다. 따라서 argv\_len에 지금까지 쌓인 데이터의 크기를 저장해 두었고, 이 크기가 4의 배수가 아니라면 4의 배수가 될만큼 esp를 조정한다. 다음으로는 다시 esp를 4만큼 옮겨 NULL값을 넣어준 뒤 argument들의 주소를 다시 역순으로 저장한다. 그 후엔 argc, return address를 차례로 저장한 뒤 동적할당된 메모리인 argv와 argv\_addr을 해제한다. 이로써 사용자가 입력한 command가 argument단위로 파싱되어 올바르게 stack에 저장되었다.

1. User Memory Access

먼저 userprog/exception.c의 page\_fault()를 보면 page fault의 원인을 알려주는 flag가 있다. 이를 이용하여 아래와 같은 코드를 추가하였다.

  if(not\_present|| !user || is\_kernel\_vaddr(fault\_addr)) exit(-1);

이로써 올바르지 않은 메모리 접근시 exit(-1)을 호출하도록 하였다.

또한 userprog/syscall.c의 syscall\_handler()의 switch문에서 매 syscall마다 다음의 코드를 추가함으로써, 현재 접근하려는 메모리가 유효한지 체크하고 유효하지 않다면 exit(-1)을 호출하도록 하였다.

if (!is\_user\_vaddr(*f*->esp+4)) exit(-1);

1. System Calls

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

모든 system call들은 userprog/syscall.c의 syscall\_handler()의 switch문을 통해 구현이 되었다.

*void*

halt (*void*)

{

  shutdown\_power\_off();

}

 case SYS\_HALT:

    {

      halt();

      break;

    }

halt()는 shutdown\_power\_off()을 호출함으로써 시스템을 끝낸다.

*void*

exit (*int* *status*)

{

*char*\* process\_name;

*char*\* ptr;

  process\_name = strtok\_r(thread\_name()," ",&ptr);

  printf("%s: exit(%d)\n",process\_name,*status*);

  thread\_current()->exit\_status = *status*;

  thread\_exit();

}

case SYS\_EXIT:

    {

      if (!is\_user\_vaddr(*f*->esp+4)) exit(-1);

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

      break;

    }

exit()에서는 process termination message를 출력해야하므로 현재 thread의 name을 가져오는 thread\_name()을 활용하여 exit status를 출력한다.

pid\_t

exec (const *char* \**cmd\_line*)

{

*char*\* token;

*char*\* ptr;

*char* \*fn\_copy;

  fn\_copy = palloc\_get\_page (0);

  if (fn\_copy == NULL)

    return TID\_ERROR;

  strlcpy(fn\_copy, *cmd\_line*, strlen(*cmd\_line*)+1);

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

*struct* file \*file = NULL;

  file = filesys\_open(token);

  if(file==NULL){

    return -1;

  }

  return ((pid\_t)process\_execute(*cmd\_line*));

}

case SYS\_EXEC:

    {

      if (!is\_user\_vaddr(*f*->esp+4)) exit(-1);

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

      break;

    }

exec()에서는 입력된 커맨드에서 실행될 파일이름을 추출한 뒤 open되었는지 확인하고 정상적으로 open되었다면 process\_execute()를 실행한다. process\_execute()에서는 전달받은 커맨드를 이용해 thread\_create()을 실행하고, 이를 통해 tid를 얻어 반환한다.

*int*

wait (pid\_t *pid*)

{

  return process\_wait((tid\_t)*pid*);

}

case SYS\_WAIT:

    {

      if (!is\_user\_vaddr(*f*->esp+4)) exit(-1);

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

      break;

    }

wait()은 process\_wait()을 통해 처리된다.

*int*

process\_wait (tid\_t *child\_tid*)

{

*struct* thread \*t = thread\_current();

*struct* thread \*t\_child;

*struct* list\_elem \*begin = list\_begin(&(t->child\_thread));

*struct* list\_elem \*end = list\_end(&(t->child\_thread));

*struct* list\_elem \*ptr;

*int* exit\_status = -1;

  for (ptr = begin; ptr!=end; ptr=list\_next(ptr)){

    t\_child = list\_entry(ptr, *struct* thread, child\_thread\_elem);

    if (*child\_tid* == t\_child->tid) {

      sema\_down(&(t\_child->child\_thread\_lock));

      exit\_status = t\_child->exit\_status;

      list\_remove(&(t\_child->child\_thread\_elem));

      sema\_up(&(t\_child->child\_thread\_memory\_lock));

    }

  }

  return exit\_status;

}

process\_wait()에서는 child process가 죽을 때 까지 parent process가 죽지 않고 기다리도록 해야하기 위하여 semaphore을 구현하였다. 따라서 현재 thread의 child thread를 검사하면서 인자로 받은 tid와 같은 tid를 가진 thread가 있는지 확인하고, 있다면 sema\_down을 통해 parent가 wait하도록 만든다. child가 죽었다면 exit status를 가져와 저장한 뒤 child tread element를 제거하고 다시 sema\_up을 한다. 이 작업을 위해 thread 구조체에 다음과 같은 멤버들이 새로 요구되었다.

#ifdef USERPROG

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

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

*int* exit\_status;

*struct* semaphore child\_thread\_lock;

*struct* semaphore child\_thread\_memory\_lock;

*struct* list child\_thread;

*struct* list\_elem child\_thread\_elem;

#endif

*int*

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

{

*int* i=0;

  if (*fd* == 0){

    for (i=0; i<*size*; i++){

      if(input\_getc() == '\0') break;

    }

    return i;

  }

  return -1;

}

*int*

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

  if (*fd* == 1) {

    putbuf(*buffer*, *size*);

    return *size*;

  }

  return -1;

}

case SYS\_WRITE:

    {

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

      break;

    }

    case SYS\_READ:

    {

      if (!is\_user\_vaddr(*f*->esp+4)) exit(-1);

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

      break;

    }

read()와 write()은 각각 stdin과 stdout으로, input\_getc()와 putbuf()를 이용하여 간단히 구현할 수 있었다. 이때 file descriptor를 검사해주는데 stdin의 경우는 0, stdout의 경우는 1이므로 이 값이 맞지않으면 -1을 반환하였다.

1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**
* *int*
* fibonacci(*int* *n*){
* *int* fib1 = 0;
* *int* fib2 = 1;
* for (*int* i=0; i<*n*; i++){
* fib1 += fib2;
* fib2 = fib1-fib2;
* }
* return fib1;
* }
* *int*
* max\_of\_four\_int(*int* *a*, *int* *b*, *int* *c*, *int* *d*){
* *int* array[4] = {*a*,*b*,*c*,*d*};
* *int* max = *a*;
* for (*int* i=1; i<4; i++){
* if (array[i]>max) max = array[i];
* }
* return max;
* }
* case SYS\_FIBONACCI:
* {
* if (!is\_user\_vaddr(*f*->esp+4)) exit(-1);
* *f*->eax=fibonacci((*int*)\*(uint32\_t\*)(*f*->esp+4));
* break;
* }
* case SYS\_MAX\_OF\_FOUR\_INT:
* {
* if (!is\_user\_vaddr(*f*->esp+4)) exit(-1);
* *f*->eax=max\_of\_four\_int((*int*)\*(uint32\_t\*)(*f*->esp+4),(*int*)\*(uint32\_t\*)(*f*->esp+8),(*int*)\*(uint32\_t\*)(*f*->esp+12),(*int*)\*(uint32\_t\*)(*f*->esp+16));
* break;
* }

Fibonacci()와 max\_of\_four\_int() 함수 자체는 단순한 방식으로 구현할 수 있었으나 이를 새로운 system call로써 사용하기 위하여 로직을 구현하는 것이 헷갈리는 부분이었다. 다른 system call들과 마찬가지로 lib/user/syscall.h와 syscall.c에 두 함수의 동작을 정의해야 하는데, max\_of\_four\_int()의 경우 4개의 인자를 사용하는 함수이므로 syscall4()를 아래와 같이 새로 정의해주어야 했다.

* #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;                                               \
* })

또한 명세서에서 설명된대로 additional.c를 통해 두 함수가 작동하여 결과를 출력하도록 해야하므로, examples 디렉토리에 다음과같이 additional.c를 추가하였다.

* #include "userprog/syscall.h"
* #include <stdio.h>
* *int*
* main (*int* *argc*, *char* \**argv*[])
* {
* *char*\* a = *argv*[1];
* *char*\* b = *argv*[2];
* *char*\* c = *argv*[3];
* *char*\* d = *argv*[4];
* printf("%d %d\n",fibonacci(atoi(a)),max\_of\_four\_int(atoi(a),atoi(b),atoi(c),atoi(d)));
* return EXIT\_SUCCESS;
* }

처음에는 a,b,c,d를 integer로 변환하지 않고 그대로 넣어 자꾸만 잘못된 값이 출력되었는데, 디버깅을 통해 두 함수에 잘못된 인자가 전달되고 있는 것을 확인하였다. argv는 char형 array이므로 각 인자를 atoi()를 통해 정수로 바꾸어 전달해주니 문제가 해결되었다.

또한 additional.c의 컴파일을 위해 example 디렉토리의 makefile의 다음 부분을 수정하였다.

PROGS = additional cat cmp cp echo halt hex-dump ls mcat mcp mkdir pwd rm

additional\_SRC = additional.c

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

**Text

Description automatically generated**

pintos –filesys-size=2 -p ../examples/additional -a additional -- -f -q run ‘additional 10 20 62 40’ 을 실행한 결과로, 10번째 피보나치 수열의 값인 55와 10, 20, 62, 40중 최대값인 62가 올바르게 출력된 것을 확인할 수 있다.