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

담당 교수 : 박성용 교수님

학번 / 이름 : 20181746 / 박준성

개발 기간 : 10월 24일 ~ 11월 1일

1. **개발 목표**

주어진 pintos는 Argument Passing, User Memory Access, System Call Handler가 구현되어있지 않은 상태이다. 따라서 주어진 argument를 parsing, passing하는 과정을 구현하고 system call 중 exec(), wait(), write(), read(), halt(), exit()를 구현하고 이를 잘 동작할 수 있게 만들어주는 system call handler 또한 구현해주어 user program이 정상적으로 작동할 수 있도록 한다.

추가적으로 n번째 Fibonacci 수를 구해주는 system call 과 네 정수 중 최대값을 찾아주는 system call을 구현한다.

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

Argument를 입력 받아 이를 tokenizing하여 80x86 Calling Convention에 맞게 esp를 이용해 stack에 push한다. 이 과정을 process.c 파일에서 구현하였다.

1. User Memory Access

User가 memory access를 하는 과정에서 잘못된 메모리에 접근하는 경우가 있다. 이러한 부분을 방지하고자 잘못된 접근을 할 경우 프로그램을 종료하도록 설계하였다. 이 과정을 exception.c 파일에서 구현하였다.

1. System Calls

Argument passing을 통해 stack에 저장되어 있는 argument들을 이용할 수 있는 system call들을 구현하였다. 구현한 system call들은 exec(), wait(), write(), read(), halt(), exit(), Fibonacci(), max\_of\_four\_int() 이다. 이를 syscall.c에 구현하였고 세부적으로 동작하는 과정에서 thread.c 에 있는 부분을 이용하거나 process.c에 있는 함수 등을 이용하여 구현하였다.

* 1. **개발 내용**
* Argument Passing

입력으로 들어온 argument를 tokenizing을 한다. 이후tokenized된 결과를 stack에 차례대로 저장하는데 이 과정에서 stack pointer esp를 사용한다. Esp의 위치를 옮겨가면서 argv의 내용들, word align, argv에 저장된 string들의 pointer들, argv의 포인터, argc, return address 순으로 저장한다.

* User Memory Access

Pintos 상에서 memory에 access할 때 invalid 할 경우는 다음과 같다.

* + 1. Return address보다 위의 메모리에 접근 할 때
    2. 실제 가능한 메모리인 0x0804800보다 아래에 있는 메모리에 접근할 때
    3. User Virtual Memory가 아닌 다른 메모리에 접근 할 때

이와 같이 정상적이지 않은 메모리 접근이 있을 경우 invalid memory access라고 한다.

이러한 부분들을 막기 위해 exception.c에 구현되어 있는 page\_fualt 함수를 일부 수정하여 잘못된 메모리에 접근하는 것을 막았다.

* System Calls

System call 은 유저 모드에서 커널 모드로 진입하는 동작으로 시스템 안정성과 보안을 지킬 수 있고, 시스템 콜 구현을 통해 user application의 호환성과 이식성을 보장할 수 있다. 또한 system call을 구현함으로써 유저 입장에서는 파일 시스템과 프로세스 생성과 같은 시스템 내부의 동작을 신경 쓰지 않아도 된다.

1. Halt

Pintos를 종료시킨다.

1. Exit

현재 실행중인 process의 정보를 출력하고 종료한다.

1. Wait

Process\_wait를 호출해 입력받은 thread의 tid를 가진 child thread가 존재할 경우 해당 child thread가 끝날 때 까지 기다려준다.

1. Exec

Process\_execute를 호출해 process를 실행시킨다.

1. Read

정해진 size만큼을 입력 받아 buffer에 저장한다.

1. Write

정해진 size만큼 buffer에 출력한다.

1. Fibonacci

n번째 피보나치 수를 찾는다.

1. Max\_of\_four\_int

입력 받은 4개의 수 중 최대값을 찾는다.

* + 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

1. **추진 일정 및 개발 방법**
   1. **추진 일정**
   * 10월 24일 ~ 28일 : 주어진 매뉴얼과 짜여져 있는 코드를 살펴보며 구현 방법을 구상
   * 10월 29일 : argument passing과 system call 부분 구현 및 디버깅
   * 10월 30일 : process\_wait을 구현 및 디버깅
   * 10월 31일 : 전반적인 오류 및 코드 수정
   * 11월 1일 : 전체적인 오류 수정 및 디버깅을 위한 부분 정리, 보고서 작성
   1. **개발 방법**
2. **Argument passing**
   * Process.c 에 command의 이름을 return 해주는 함수 get\_name\_of\_system\_call 함수를 작성
   * 입력을 pasing 해준 후 이 값들을 stack에 삽입해주는 construct\_esp 함수 작성
   * Get\_name\_of\_system\_call을 start\_process에 사용해 command를 찾아주고 construct\_esp 함수를 load 함수에서 불러 stack에 데이터를 삽입
   * 이때 데이터는 stack에서 아래로 커지는 형태로 구현
3. **User Memory Access**
   * Exception.c의 page\_fault 함수 내에서 비정상적인 메모리 접근이 발생하였을 때 exit를 사용해 종료하는 부분을 추가
4. **System Call**
   * Syscall.c에 syscall\_handler를 구현
   * Syscall.c에서 memory access vaildalbe을 검사해준 후 요청된 system call을 호출
   * 각 system call을 syscall.c에 구현.
     1. Halt

Shutdown\_power\_off 함수를 호출해 pintos 종료

* + 1. Exit

Status를 받아 현재 thread의 exit\_status를 set하고 이를 확인 할 수 있게 출력

* + 1. Exec

Process.c의 process\_execute 함수를 호출

process\_execute 함수에서 get\_name\_of\_system\_call을 호출해 system call의 이름을 추출

file이 없는 경우 -1을 return하도록 반환하는 부분도 추가

* + 1. Wait

Process.c의 process\_wait 함수를 호출

Process\_wait 함수는 tid를 받아 tid 값이 일치하는 자식을 기다려준다.

* + 1. Write

Standard ouput (fd == 1) 인 경우만 구현하며 putbuf 함수를 이용해 buffer에 씀

Size를 반환

* + 1. Read

Standard in (fd == 0) 인 경우만 구현하며 input\_getc를 size만큼 호출하며 buffer에 입력을 받는다 이후 size를 반환

* + 1. Fibonacci

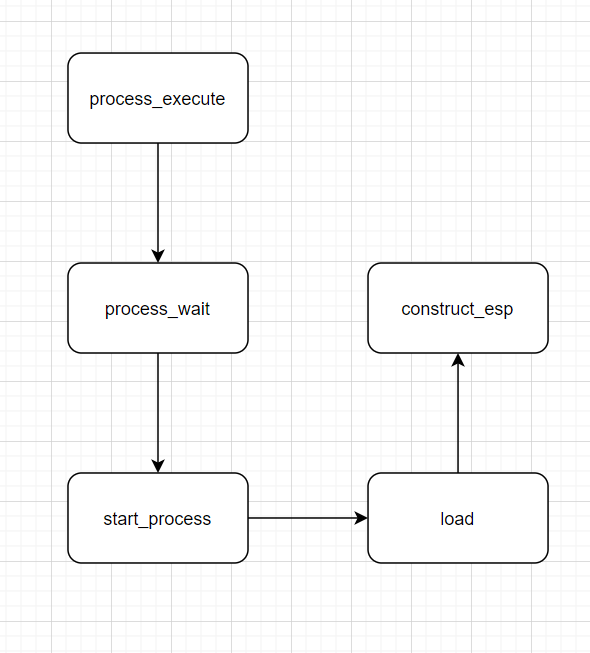
n번째 피보나치 수를 반환

* + 1. Max\_of\_four\_int

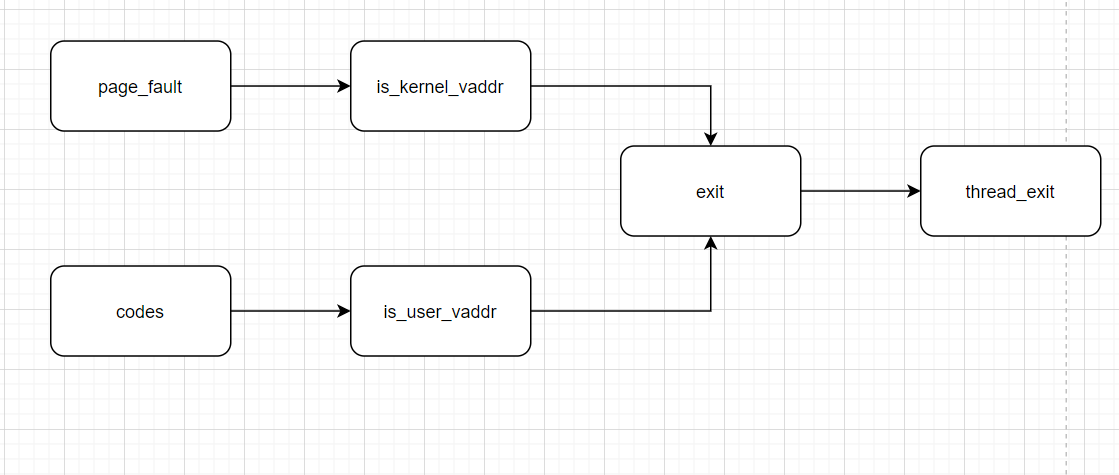
Int형 자료 2개 중 큰 수를 반환해주는 max2(int, int) 함수를 추가 구현

이 함수를 이용해 4개의 정수 중 최대값을 반환

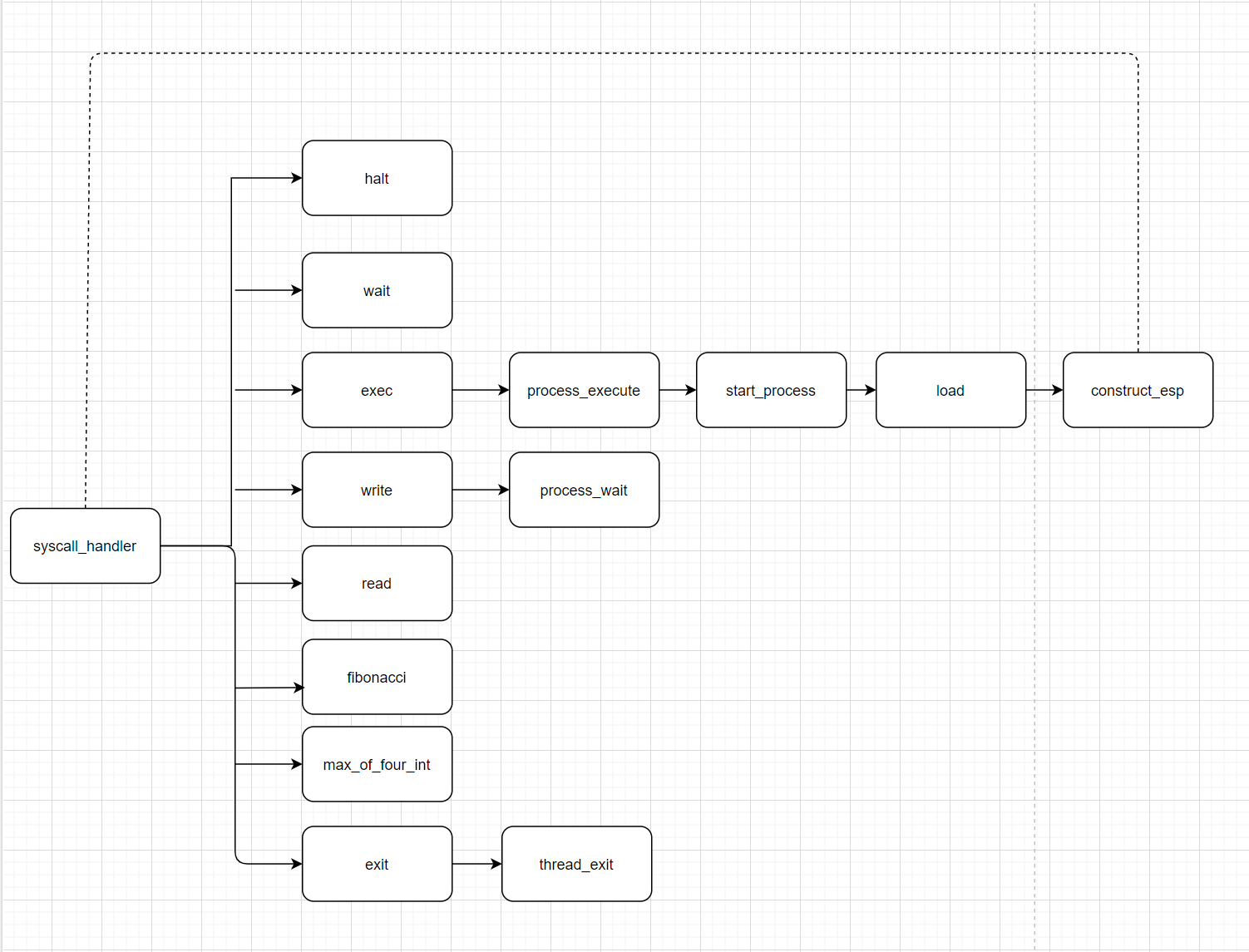
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing



1. User Memory Access



1. System Calls



* 1. **제작 내용**

1. Argument Passing

src/userprog/process.c

|  |
| --- |
| void get\_name\_of\_system\_call(char \*src, char \*dest) {  int i;  strlcpy(dest, src, strlen(src) + 1);  for (i = 0; dest[i] != '\0' && dest[i] != ' '; i++);  dest[i] = '\0';  } |

Src의 값을 strlcpy를 이용해 copy 해준 후 white space를 만나면 null문자를 입력해주는 방식으로 구현하였다.

src/userprog/process.c

|  |
| --- |
| void construct\_esp(char \*file\_name, void \*\*esp) {  char \*\*argv;  int argc = 0;  int total\_len = 0;  char file\_name\_array[256];  char \*token, \*last;  int i;  int len;  strlcpy(file\_name\_array, file\_name, strlen(file\_name) + 1);  token = strtok\_r(file\_name\_array, " ", &last);  while (token != NULL) {  ++argc;  token = strtok\_r(NULL, " ", &last);  }  argv = (char \*\*)malloc(sizeof(char \*) \* argc);  strlcpy(file\_name\_array, file\_name, strlen(file\_name) + 1);  for (i = 0, token = strtok\_r(file\_name\_array, " ", &last); i < argc; i++, token = strtok\_r(NULL, " ", &last)) {  len = strlen(token);  argv[i] = token;  }  for (i = argc - 1; 0 <= i; i--) {  len = strlen(argv[i]) + 1;  \*esp = \*esp - len;  total\_len = total\_len + len;  strlcpy(\*esp, argv[i], len);  argv[i] = \*esp;  }  if (total\_len % 4 != 0)  \*esp = \*esp - (4 - total\_len % 4);    \*esp = \*esp - 4;  \*\*(uint32\_t \*\*)esp = 0;  for (i = argc - 1; 0 <= i; i--) {  \*esp = \*esp - 4;  \*\*(uint32\_t \*\*)esp = argv[i];  }  \*esp = \*esp - 4;  \*\*(uint32\_t \*\*)esp = \*esp + 4;  \*esp = \*esp - 4;  \*\*(uint32\_t \*\*)esp = argc;  \*esp = \*esp - 4;  \*\*(uint32\_t \*\*)esp = 0;  free(argv);  } |

File\_name의 사본을 file\_name\_array에 만든 후 이를 strtok\_r 함수를 이용해 토큰화 시키면서 argc 의 개수를 구해준 후 argv를 argc의 개수에 맞게 malloc한다. 이후 token 값들을 하나씩 argv에 저장한 후 이 값들을 esp를 이용해 stack에 저장해준다. 이후 argv의 주소와 argc 값, return address를 차례대로 stack에 push 한 후 argv를 free 시켜준다.

1. User Memory Access

|  |
| --- |
| static void  page\_fault (struct intr\_frame \*f)  {  …  if (!(user && is\_kernel\_vaddr(fault\_addr)))  exit(-1);  } |

src/userprog/exception.c

page\_fault에 is\_kernel\_vaddr로 검사하는 부분을 추가해 memory access invalid 한 상황을 막았다.

src/userprog/syscall.c

|  |
| --- |
| static void  syscall\_handler (struct intr\_frame \*f UNUSED)  {  /\* My Additional Code (Project 1) : system call handler \*/  switch (\*(uint32\_t\*)(f->esp))  {  …  case SYS\_EXIT:  if (!is\_user\_vaddr(f->esp + 4)) exit(-1);  exit(\*(uint32\_t \*)(f->esp + 4));  break;  case SYS\_EXEC:  if (!is\_user\_vaddr(f->esp + 4)) exit(-1);  f->eax = exec((const char \*)\*(uint32\_t \*)(f->esp + 4));  break;  …  } |

메모리에 접근할 때 is\_user\_vaddr 함수를 이용해 메모리 접근에 문제가 있는 경우를 차단하였다.

Is\_user\_vaddr 함수와 is\_kernel\_vaddr 함수는 모두 threads/vaddr.h에 정의되어있다.

1. System Calls
   * Syscall handler

Src/userprog/syscall.c

|  |
| --- |
| static void  syscall\_handler (struct intr\_frame \*f UNUSED)  {  /\* My Additional Code (Project 1) : system call handler \*/  switch (\*(uint32\_t\*)(f->esp))  {  case SYS\_HALT:  halt();  break;  case SYS\_EXIT:  if (!is\_user\_vaddr(f->esp + 4)) exit(-1);  exit(\*(uint32\_t \*)(f->esp + 4));  break;  case SYS\_EXEC:  if (!is\_user\_vaddr(f->esp + 4)) exit(-1);  f->eax = exec((const char \*)\*(uint32\_t \*)(f->esp + 4));  break;  case SYS\_WAIT:  if (!is\_user\_vaddr(f->esp + 4)) exit(-1);  f->eax = wait((tid\_t)\*(uint32\_t \*)(f->esp + 4));  break;  case SYS\_WRITE:  if (!is\_user\_vaddr(f->esp + 20)) exit(-1);  if (!is\_user\_vaddr(f->esp + 24)) exit(-1);  if (!is\_user\_vaddr(f->esp + 28)) exit(-1);  f->eax = write(\*(int\*)(f->esp + 20), \*(void\*\*)(f->esp + 24), \*(unsigned\*)(f->esp + 28));  break;  case SYS\_READ:  if (!is\_user\_vaddr(f->esp + 20)) exit(-1);  if (!is\_user\_vaddr(f->esp + 24)) exit(-1);  if (!is\_user\_vaddr(f->esp + 28)) exit(-1);  f->eax = read(\*(int\*)(f->esp + 20), \*(void\*\*)(f->esp + 24), \*(unsigned\*)(f->esp + 28));  break;  case SYS\_FIBONACCI:  f->eax = fibonacci(\*(int\*)(f->esp + 4));  break;  case SYS\_MAX\_OF\_FOUR\_INT:  f->eax = max\_of\_four\_int(\*(int\*)(f->esp + 4), \*(int\*)(f->esp + 8), \*(int\*)(f->esp + 12), \*(int\*)(f->esp + 16));  break;  default:  break;  }  } |

Syscall\_hander는 각 system call을 호출해주는 역할을 한다.

* + halt

Src/userprog/syscall.c

|  |
| --- |
| void halt() {  shutdown\_power\_off();  } |

Halt는 shutdown\_power\_off()를 호출해 pintos를 종료해준다.

* + exit

Src/userprog/syscall.c

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

Exit은 현재 thread의 status를 출력해주고 set해준다. Thread의 이름과 status를 출력해주고 thread\_exit를 이용해 스레드를 종료해준다.

Src/userprog/syscall.c

|  |
| --- |
| tid\_t exec(const char \*cmd\_line) {  return process\_execute(cmd\_line);  } |

Exec는 process\_execute를 호출해준다.

* + exec

Src/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);  char system\_call\_name[256];  get\_name\_of\_system\_call(file\_name, system\_call\_name);  if (filesys\_open(system\_call\_name) == NULL)  return -1;  /\* Create a new thread to execute FILE\_NAME. \*/  tid = thread\_create (system\_call\_name, PRI\_DEFAULT, start\_process, fn\_copy);  if (tid == TID\_ERROR)  palloc\_free\_page (fn\_copy);  return tid;  } |

Process\_execute에서는 file을 열어주고 이 값을 이용해 thread를 만들어준다. 이후 start\_process가 실행된다.

Src/userprog/process.c

|  |
| --- |
| static void  start\_process (void \*file\_name\_)  {  char \*file\_name = file\_name\_;  struct intr\_frame if\_;  bool success;  char system\_call\_name[256];  get\_name\_of\_system\_call(file\_name, system\_call\_name);  /\* Initialize interrupt frame and load executable. \*/  memset (&if\_, 0, sizeof if\_);  if\_.gs = if\_.fs = if\_.es = if\_.ds = if\_.ss = SEL\_UDSEG;  if\_.cs = SEL\_UCSEG;  if\_.eflags = FLAG\_IF | FLAG\_MBS;  success = load (system\_call\_name, &if\_.eip, &if\_.esp);    if (success)  construct\_esp(file\_name, &if\_.esp);  palloc\_free\_page(file\_name);  /\* If load failed, quit. \*/  if (!success)  thread\_exit();  /\* Start the user process by simulating a return from an  interrupt, implemented by intr\_exit (in  threads/intr-stubs.S). Because intr\_exit takes all of its  arguments on the stack in the form of a `struct intr\_frame',  we just point the stack pointer (%esp) to our stack frame  and jump to it. \*/  asm volatile ("movl %0, %%esp; jmp intr\_exit" : : "g" (&if\_) : "memory");  NOT\_REACHED (); |

Start\_process에서는 load 함수를 호출하고 이 시행이 성공적으로 이루어 졌다면 construct\_esp 함수를 호출해 값을 stack에 쌓아준다.

* + wait

Src/userprog/syscall.c

|  |
| --- |
| int wait(tid\_t pid) {  return process\_wait(pid);  } |

Wait 는 process\_wait를 호출해준다

Src/userprog/process.c

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

Process\_wait는 child\_tid를 현재 실행중인 thread의 자식중에 tid가 child\_tid와 동일한 자식이 있다면 wait해준다 이를 semaphore를 이용하여 이를 구현해주었다. 이를 위해 threads/thread.h의 struct thread에 아래와 같은 것들을 추가하였다.

|  |
| --- |
| struct semaphore child\_lock; // child\_tid와 같은 child의 lock이다  struct semaphore cur\_lock; // 본인의 lock이다  struct list child; // child thread들의 list이다  struct list\_elem child\_elem; // child\_tid와 같은 child의 list\_elem이다  int exit\_status; // exit\_status를 저장하는 변수이다 |

* + write

Src/userprog/syscall.c

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

Write 함수는 buffer를 입력 받아 size만큼 buffer에 쓴다. Stdout (fd == 1)에 대해서만 구현하였고 stdout일 경우 size를 아닐 경우 -1를 반환하도록 하였다.

* + read

Src/userprog/syscall.c

|  |
| --- |
| int read(int fd, const void \*buffer, unsigned size) {  if (fd == 0) {  for (int i = 0; i < (int)size; i++)  buffer = input\_getc();  return (int)size;  }  return -1;  } |

read 함수는 buffer를 입력 받아 size만큼 입력을 받는다. Stdin (fd == 0)에 대해서만 구현하였고 stdin일 경우 size를 아닐 경우 -1를 반환하도록 하였다.

1. Additional System calls
   * Fibonacci

Src/userprog/syscall.c

|  |
| --- |
| int fibonacci(int n) {  int a = 0, b = 1;  int tmp;  for (int i = 0; i < n; i++) {  tmp = b;  b = a + b;  a = tmp;  }  return a;  } |

반복문을 이용해 n번째 Fibonacci 수를 구하도록 구현하였다.

* + Max\_of\_four\_int

Src/userprog/syscall.c

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

두 int 중 큰 수를 반환하는 max2 함수를 이용해 max\_of\_four\_int 를 구현하였다.

* + Additional

Src/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(const int a, const int b, const int c, const int d) {  return syscall4(SYS\_MAX\_OF\_FOUR\_INT, a, b, c, d);  } |

Syscall4에 대한 매크로를 구현하였다. 이를 바탕으로 max\_of\_four\_int를 구현하였다.

Src/lib/syscall-nr.c

|  |
| --- |
| enum  {  …  SYS\_FIBONACCI,  SYS\_MAX\_OF\_FOUR\_INT,  …  }; |

System call handler를 위해 enum을 추가하였다.

Src/examples/additional.c

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <syscall.h>  int main(int argc, char \*\*argv) {  int a, b, c, d, \_fibo, \_max;  if (argc != 5) {  printf("Usage : additional a b c d\n");  return EXIT\_FAILURE;  }  a = atoi(argv[1]);  b = atoi(argv[2]);  c = atoi(argv[3]);  d = atoi(argv[4]);  \_fibo = fibonacci(a);  \_max = max\_of\_four\_int(a, b, c, d);  printf("%d %d\n", \_fibo, \_max);  return EXIT\_SUCCESS;  } |

Additional 실행을 위한 example을 구현하였다. Argc가 5가 아닐 경우 사용법을 출력하고 아닐 EXIT\_FAILURE를 출력하게 하고 아니면 각 syscall을 호출한 후 결과를 출력하게 하였다.

대부분의 구현 과정에서 크고 작은 오류가 있었는데 이를 디버깅하는 과정이 쉽지 않았다. Printf로 값을 찍는 식으로 디버깅하였고 stack에 대한 정보가 필요한 경우에 hex\_dump를 이용하였다. 디버깅시 편의를 위해 #define DDEBUG를 정의하고

#ifdef DDEBUG

Code

#endif

형식을 사용하여 디버깅하였다.

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