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

담당 교수 : 박성용

학번 / 이름 : 20151149 문상호

개발 기간 : 2021/10/13 ~ 2021/11/2

1. **개발 목표**

- 현재 주어진 pintos OS 코드에는 커널의 system call을 통해 제공되는 I/O functionality가 구현되어 있지 않다. 또한 kernel로 arguments들을 전달하고 저장하기 위한 user stack 역시 구현되어 있지 않다.

- 따라서 이 개발 프로젝트는 pintos OS 환경에서 user program이 작동할 수 있도록 하기 위해 1. 간단한 argument passing 구현, 2. user memory access 처리, 3. 기본적인 system call 구현 및 추가 system call을 구현하는 것이 목표다.

1. **개발 범위 및 내용**
   1. **개발 범위**

|  |  |
| --- | --- |
| **/userprog** | **syscall.c syscall.h process.c process.h exception.c** |
| **/examples** | **additional.c Makefile** |
| **/threads** | **thread.c thread.h** |
| **/lib** | **syscall-nr.h** |
| **/lib/user** | **syscall.c syscall.h** |

1. Argument Passing

- arguments들을 띄어쓰기를 기준으로 파싱한 후, 각각의 argument들을 80x86 calling convention에 따라 downward 방식으로 stack에 push 한다. esp라는 stack pointer를 이용하여 stack에 접근할 수 있다.

1. User Memory Access

- user가 잘못된 메모리에 접근하였을 경우, 예외처리를 통해 프로세스를 종료시켜주어야 한다.

1. System Calls

- system call handler 와 각각의 기본적인 system call을 구현한다. system call handler에서는 intr\_frame 구조체의 esp(stack pointer) 멤버를 이용하여 stack에 쌓여있는 argument에 접근할 수 있다.

* 1. **개발 내용**
* Argument Passing

- 입력된 arguments 들을 tokenizing 한다. 이후 각 토큰(argument)들을 user stack에 저장해야 하는데, 이 때 80x86 calling convention에 맞춰 downard 방식으로 stack에 쌓는다. esp 스택 포인터를 이용하여 stack 주소를 변경해가며 arguments들을 쌓은 뒤, 차례로 word\_align, null sentinel, arguments 주소, argument 개수(argc), return address를 stack에 push한다.

* User Memory Access

- 유저 메모리 영역이 아닌, 커널 메모리에 접근하는 경우, 즉 esp > PHYS\_BASE

인 경우에 잘못된 메모리 접근이라고 할 수 있다. 이를 막기 위해서는 threads/vaddr.h 에 들어있는 is\_kernel\_vaddr( ) 또는 is\_user\_vaddr( ) 함수를 호출하여 유효한 메모리 접근인지 확인할 수가 있다.

* System Calls

유저 프로그램은 유저레벨의 함수들 만으로는 많은 기능을 구현하기 힘들기 때문에, kernel의 도움이 필요하다.(파일 I/O 등) 이러한 작업은 유저모드에서는 수행할 수 없고, 반드시 커널모드로 전환한 후에야 해당 작업을 수행할 권한이 생긴다. 커널 모드를 통한 작업은 반드시 시스템 콜을 통해 수행하도록 설계되어 있다. 권한이 필요한 이유는 악의적인 이유로 시스템 콜을 호출했을 경우에 시스템 전체를 망가뜨릴 수 있기 때문이고, 따라서 이러한 명령어들은 특별하게 커널 모드에서만 실행할 수 있도록 설계되어 있다.

다음은 구현할 system call 들에 대한 간략한 설명이다.

- halt : pintos를 종료하는 함수이다. shutdown\_power\_off 를 호출한다.

- exit: 현재 구동중인 thread를 종료하고 status를 리턴하는 함수이다. wait하고 있는 parent thread에게 status를 알려준다. 정상적으로 종료되지 않았다면 -1 을 리턴한다.

- exec: 인자로 받은 command를 수행하는 함수이다. process\_execute를 호출하여 child process를 실행시키고, 새로 실행된 process의 id를 리턴한다.

- wait: child thread를 기다리는 함수이다. process\_wait를 호출한다.

- read: input\_getc 를 반복적으로 호출하여 buffer에 한 글자씩 추가한다. size만큼 읽고, size보다 더 적은 횟수에 읽기가 끝난 경우, 읽은 횟수 만큼의 값을 리턴한다.

- write: put\_buf 를 호출하여 size만큼 buffer로부터 불러온다.

- fibonacci: 인자 n을 받아 n번째에 해당하는 피보나치 수열의 값을 리턴한다.

- max\_of\_four\_int: 4개의 INT 타입 인자를 받아 가장 큰 값을 리턴한다.

유저 레벨에서 시스템 콜 api를 호출하면, 어셈블리어로 구현되어 있는 syscall 매크로에 의해 system call number 와 arguments 들이 user stack에 push된다. 이후interrupt handler가 호출되고, interrupt handler는 system call handler를 호출하는데, system call handler는 intr\_frame 구조체의 esp 멤버 변수를 이용하여 스택에 접근할 수 있게 되므로 system call number와 arguments들을 가지고 각 번호에 해당하는 system call을 호출할 수 있게 된다. 또한 intr\_frame 구조체의 eax 변수를 이용하여 system call들의 리턴값을 저장할 수 있고, 작업이 끝난 후에는 다시 유저 레벨로 돌아오게 된다. 유저 레벨에서 커널 레벨, 커널 레벨에서 다시 유저 모드로의 전환을 위해서 os에서 mode\_bit을 변경해준다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |
| --- | --- |
| 10/13 ~ 10/19 | proj1.pdf 메뉴얼 학습 및 코드 분석 |
| 10/19 ~ 10/23 | argument passing 구현, thread 구조체 변수 추가,  exit, wait, exec system call 구현 |
| 10/24 ~ 10/30 | read, write system call 구현, system call handler 구현, additional system call(fibonacci, max\_of\_four\_int) 작성 |
| 10/31 ~ 11/2 | 디버깅, 보고서 작성 |

* 1. **개발 방법**

- 먼저 argument passing을 구현하기 위해서, process.c에 추가로 코드를 작성해야 한다. process\_execute 에서는 file\_name에서 명령어를 파싱하여야 하고, load에서는 인자로 전달된 각 arguments들을 token화하여 가지고 있어야 한다. 해당 방법을 구현하기 위해서는 strtok\_r, strlcpy 함수를 사용하여 tokenizing 할 수 있다. 이후 토큰화된 각 argument들을 동적할당한 argv 배열에 저장해두고, 이후 esp stack pointer를 사용하여 스택에 push한다. 이후 word align 처리, null sentinel 처리, 각 argv 주소, argument 개수, return address 들을 순차적으로 downward 방식으로 쌓아 스택 구성을 완료한다.

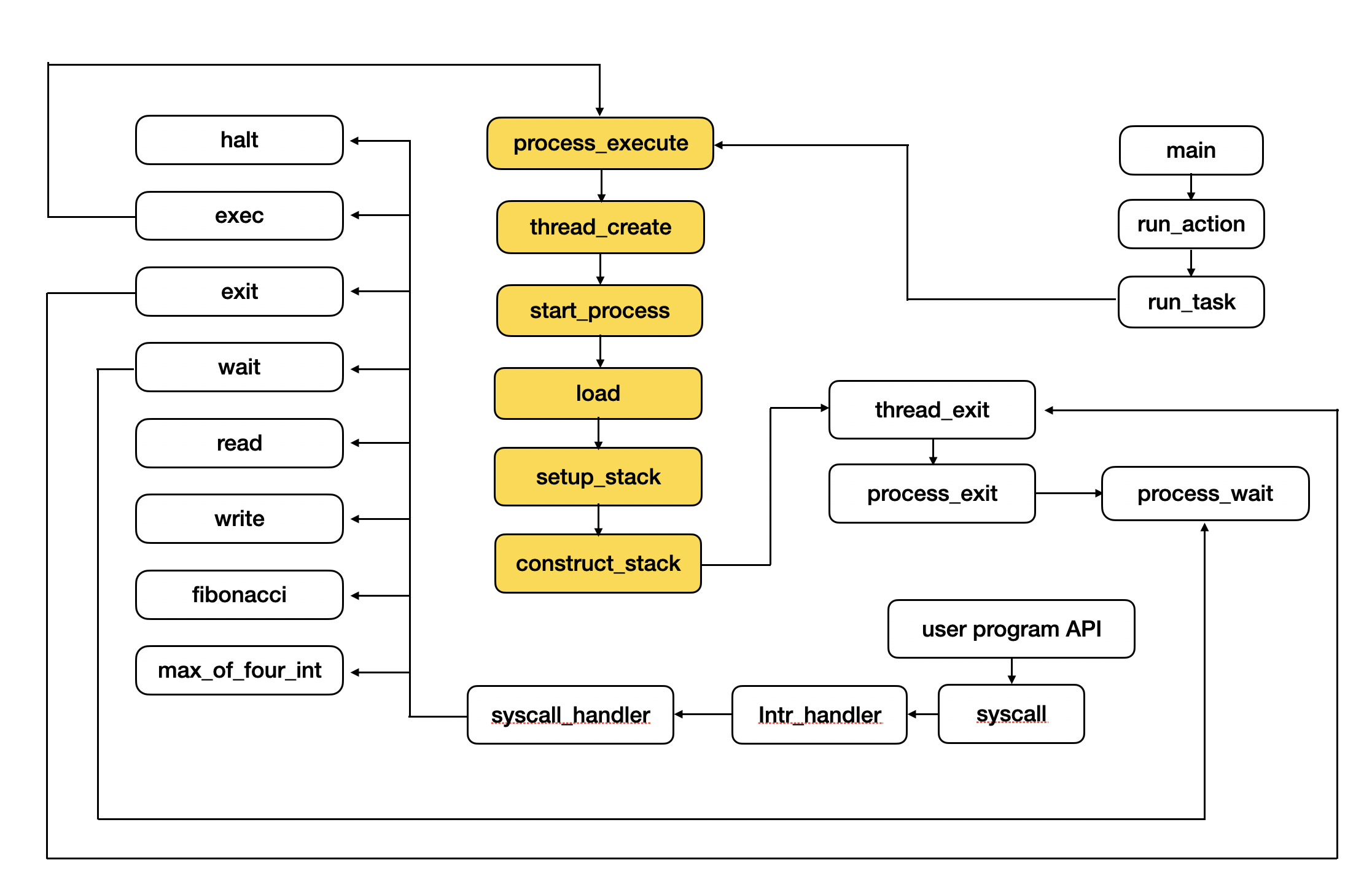
- process\_execute를 통해 새로 생성된 스레드들이 주어진 명령어를 수행할 때, 생성된 스레드의 부모 스레드는 자식 스레드가 종료될 때까지 먼저 종료되지 않고 기다려줘야 한다. 따라서 해당 부분을 구현하기 위해 busy waiting 또는 semaphore를 구현할 수 있는데, 본 프로젝트에서는 semaphore를 구현하기로 하였다. 먼저 thread.h에 해당 스레드의 semaphore 멤버 변수, status 멤버 변수, 부모의 tid 값을 가지는 멤버 변수를 추가한다. thread.c 파일에서는 init\_thread에서 해당 스레드의 semaphore를 initialized 시키는 부분을 추가 작성하고, tid 값을 인자로 받아 thread list에서 부모 thread를 리턴하는 함수를 구현한다. 이후 process.c 에서 process\_wait과 process\_exit 에서 각각 sema\_down과 sema\_up을 호출하여 스레드의 종료 시점을 파악할 수 있도록 한다.

- 유효하지 않은 메모리 접근을 막기 위해서는 threads/vaddr.h 파일에 있는 함수들을 이용할 수 있다. syscall.c, exception.c 에 해당 부분들을 구현한다. syscall.c 에서는 switch 문을 통해 분기되는 각각의 system call 호출 부분에 앞서 is\_user\_vaddr 함수를 호출하여 유저 메모리 영역에 접근하는 것이 맞는지, 유효한 접근인지 확인하고, 유효하지 않다면 exit 처리한다. exception.c에서는 page fault 함수 내에 추가로 동일한 부분을 작성하여 메모리 엑세스 검사를 시행한다.

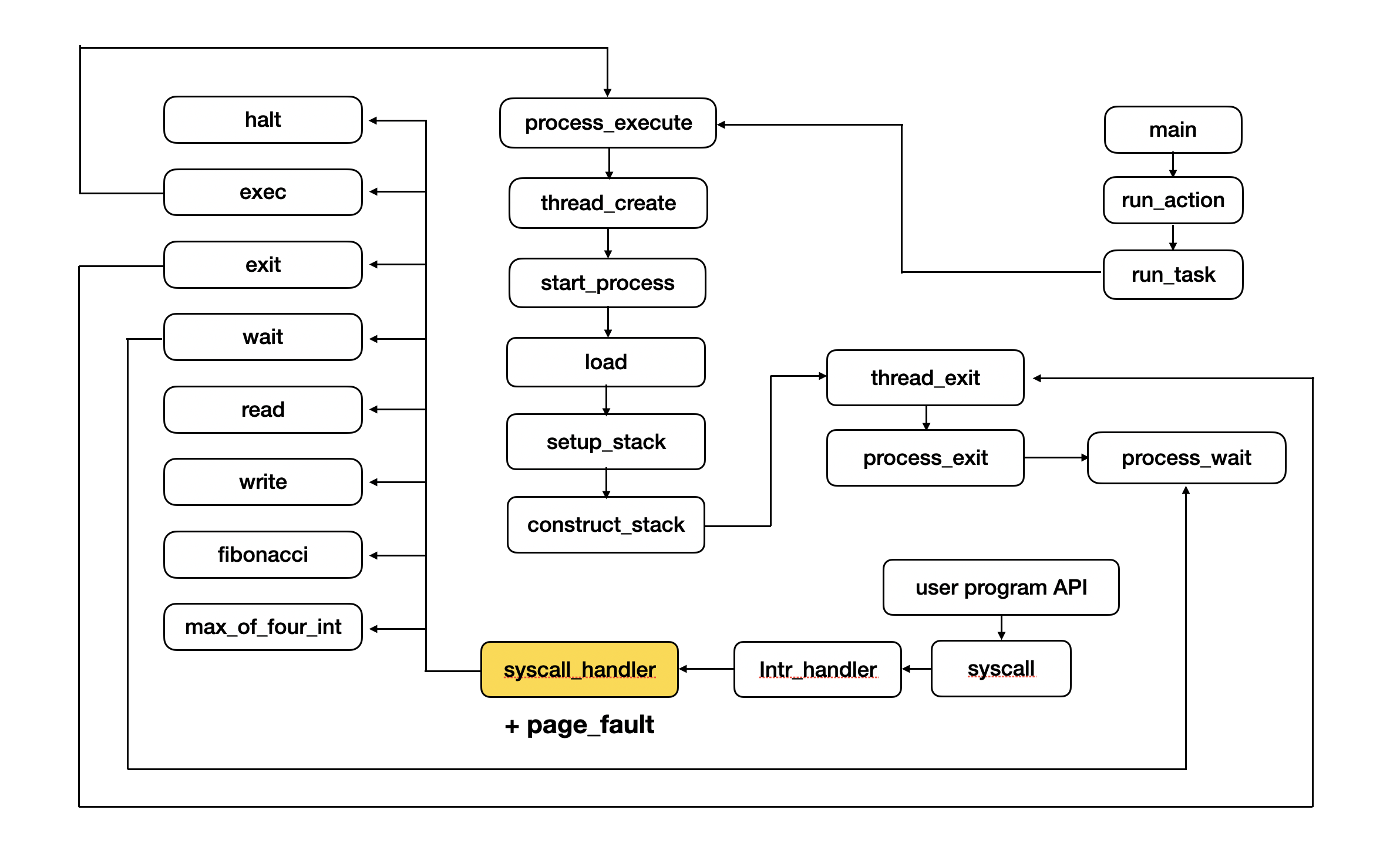
추가로 process.c의 process\_execute 함수 내에서는 파싱한 명령어 값을 filesys\_open 함수의 인자로 전달하여 결과값이 NULL인 경우를 검사한다.

- 각각의 system call 구현을 위해서는 syscall.c 파일 내에 각 system call 함수들을 작성한다. syscall.h에 해당 함수들의 프로토타입을 선언한다. 추가로 작성해야하는 시스템 콜(fibonacci, max\_of\_four\_int)의 경우 먼저 lib/syscall-nr.h 파일에 해당 함수들의 syscall\_num 값을 enum으로 정해주고, lib/user/syscall.c 에서는 max\_of\_four\_int system call을 위해 argument를 4개 받을 수 있는 syscall4 매크로를 구현한다. 이후 해당 syscall 매크로들을 호출하는 각 함수를 작성하고, syscall.h에 작성한 함수의 프로토타입들을 선언한다. 마지막으로 additional.c 파일에 의도하는 시스템 콜을 호출하는 함수들을 호출하고 출력하는 코드를 작성하고 MAKEFILE에 additional.c 부분을 추가한다.

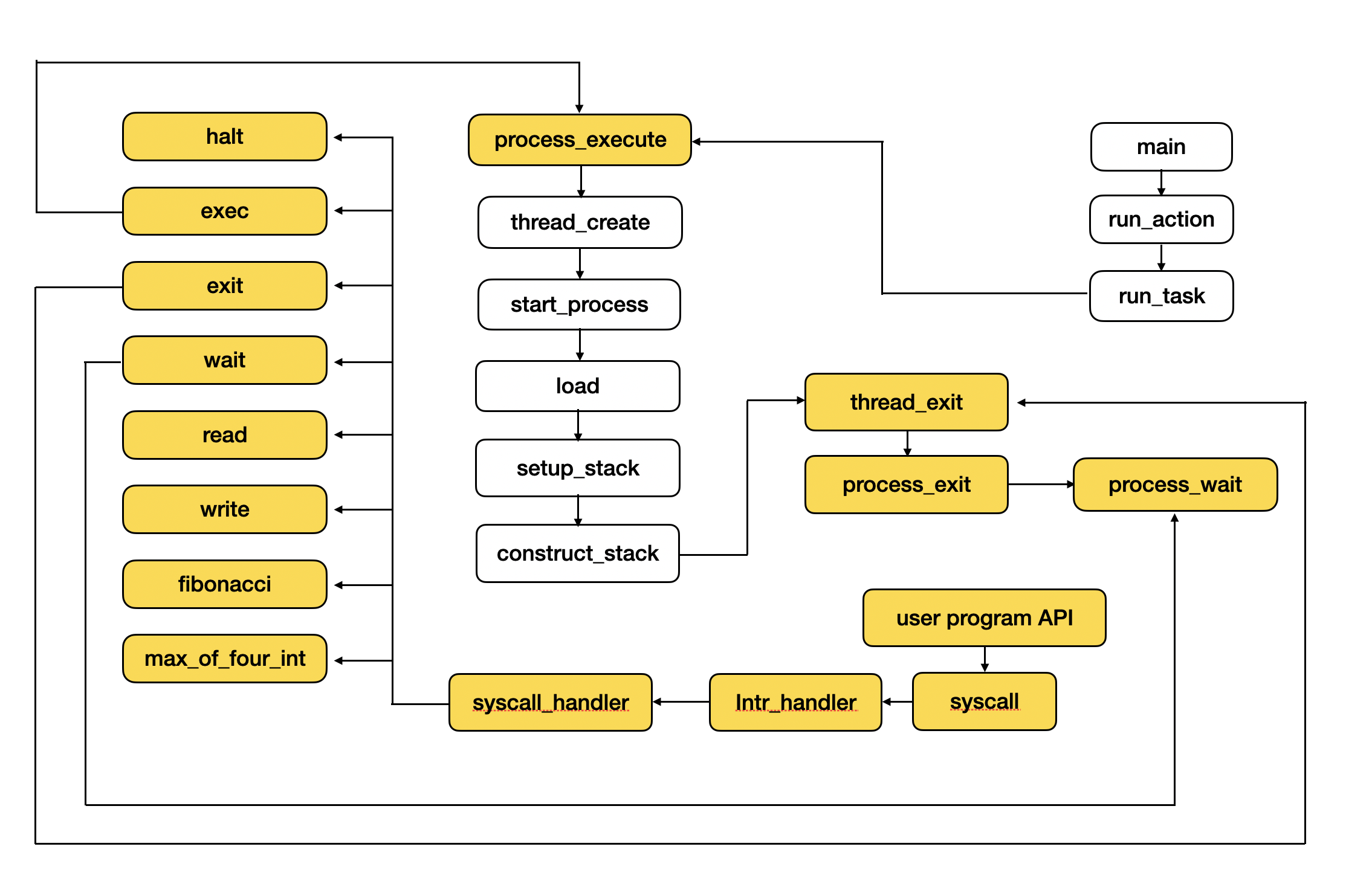
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing



1. User Memory Access



1. System Calls



* 1. **제작 내용**

1. Argument Passing

**1-1. /userprog/process.c**

**tid\_t process\_execute (const char \* file\_name)**

- 인자로 들어온 file\_name 값을 이름으로 가지는 child thread를 생성하는 함수이다. 입력받은 argument 중 맨 앞의 명령어를 파싱한다. 이후 해당 명령어를 thread\_create에 전달하고, 리턴된 id값을 가지고 get\_thread\_by\_tid 함수를 호출하여 새로 생성된 쓰레드를 가져오고, 이의 parent\_tid 멤버 변수에(직접 구조체에 추가 구현) 현재 쓰레드의 tid 값을 할당한다.

|  |
| --- |
| tid\_t  process\_execute (const char \*file\_name)  {  char \*fn\_copy;  tid\_t tid;  /\* CSE4070 implementation \*/  **int i;**  **char cmd[256];**  **char \*save\_ptr;**  /\* 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);  // parsing command  **strlcpy(cmd, file\_name, strlen(file\_name) + 1);**  **while (1) {**  **if(cmd[i] == '\0' || cmd[i] == ' ') {**  **break;**  **}**  **i++;**  **}**  **cmd[i] = '\0';**  **if (filesys\_open(cmd)==NULL) {**  **return -1;**  **}**  /\* Create a new thread to execute FILE\_NAME. \*/  tid = thread\_create (cmd, PRI\_DEFAULT, start\_process, fn\_copy);  if (tid == TID\_ERROR)  palloc\_free\_page (fn\_copy);  **get\_thread\_by\_tid(tid)->parent\_tid = thread\_current()->tid;**  return tid;  } |

**void construct\_stack (char \*file\_name, void \*\*esp)**

- 80X86 calling convention에 따라 arguments들을 stack에 쌓는 함수이다. file\_name의 값을 tokenizing하여 인자 개수를 받고, 각 인자 값을 담을 argv 배열을 동적할당받는다. 이후 차례로 마지막 argument 부터 esp 값을 감소시켜가며 stack에 저장한다. 인자를 다 쌓은 후에는 word align, null sentinel, argument address를 차례로 더 쌓고, 최종적으로 마지막 argument address, argument count(argc), return address를 stack에 저장한다.

|  |
| --- |
| **void construct\_stack(char\* file\_name, void \*\*esp){**  **int i = 0;**  **char\*\* argv;**  **int argc = 0;**  **int argv\_len = strlen(file\_name) + 1;**  **int total\_len = 0;**  **char temp\_cmd[256];**  **char\* token;**  **char\* save\_ptr;**    **strlcpy(temp\_cmd, file\_name, argv\_len);**    **token = strtok\_r(file\_name, " ",&save\_ptr);**  **// check argument count**  **while (token != NULL) {**  **argc++;**  **token = strtok\_r(NULL, " ", &save\_ptr);**  **}**  **// memory dynamic allocation**  **argv = (char\*\*)malloc(sizeof(char\*) \* argc);**  **// tokenizing**  **token = strtok\_r(temp\_cmd, " ", &save\_ptr);**  **for (i = 0 ; i < argc; i++) {**  **argv[i] = token;**  **token = strtok\_r(NULL, " ", &save\_ptr);**  **}**  **// save token(each argv)**  **for(i = argc - 1; i >= 0; i--){**  **argv\_len = strlen(argv[i]) + 1;**  **\*esp -= argv\_len;**  **total\_len += argv\_len;**  **strlcpy(\*esp, argv[i], argv\_len);**  **argv[i] = \*esp;**  **}**    **// process word align**  **if (total\_len % 4 != 0) {**  **int word\_align\_cnt = 4 - (total\_len % 4);**  **for(i = 0 ; i < word\_align\_cnt; i++){**  **\*esp -= 1;**  **\*\*(uint8\_t\*\*)esp = 0;**  **}**  **}**  **// process null sentinel**  **\*esp -= 4;**  **\*\*(uint32\_t\*\*)esp = 0;**  **// save argv address**  **for(i = argc - 1 ; i >= 0; i--){**  **\*esp -= 4;**  **\*\*(uint32\_t\*\*)esp = argv[i];**  **}**  **\*esp -= 4;**  **\*\*(uint32\_t\*\*)esp = \*esp+4;**  **// argc**  **\*esp -= 4;**  **\*\*(uint32\_t\*\*)esp = argc;**  **// return address**  **\*esp -= 4;**  **\*\*(uint32\_t\*\*)esp = 0;**  **// memory free**  **free(argv);**  **}** |

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

- child thread가 종료될 때까지 parent thread가 기다려주게끔 도와주는 함수이다. busy waiting 또는 semaphore 방법을 택할 수 있는데, semaphore로 구현하였다. 먼저 인자로 들어온 child\_tid값으로 child\_thread를 찾는다. child\_thread가 없다면 -1 을 리턴하고 있다면 sema\_down 함수를 호출하여 child thread가 종료되기까지 기다린다. 이후 process\_exit 함수에서 sema\_up(child->child\_sema)가 호출되면 lock이 풀리고, exit 쪽에서 리턴한 exit\_status 값을 받아 리턴한다.

|  |
| --- |
| **int**  **process\_wait (tid\_t child\_tid UNUSED)**  **{**  **// for hex\_dump test**  **// while (1) { }**  **struct thread \* child\_thread = NULL;**  **int exit\_status = -1;**  **child\_thread = get\_thread\_by\_tid(child\_tid);**    **if (child\_thread == NULL) {**  **return exit\_status;**  **}**    **sema\_down(&(child\_thread->child\_sema));**  **exit\_status = thread\_current()->exit\_status;**  **return exit\_status;**  **}** |

**void process\_exit (void)**

- process를 종료하는 함수이다. 현재 프로세스가 종료됨을 알리기 위해 sema\_up 호출부를 추가로 작성하였다.

|  |
| --- |
| 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)  {  cur->pagedir = NULL;  pagedir\_activate (NULL);  pagedir\_destroy (pd);  }    **sema\_up(&(cur->child\_sema));**  } |

**1-2. /threads/thread.c**

**static void init\_thread( )**

- 멤버 변수로 추가한 child\_sema 값을 initializing한다.

|  |
| --- |
| static void  init\_thread (struct thread \*t, const char \*name, int priority)  {  ...  /\* CSE4070 implementation \*/  sema\_init(&(t->child\_sema), 0);  ...  } |

**struct thread \* get\_thread\_by\_tid (tid\_t tid)**

- tid 값을 인자로 받아 해당 tid를 가지는 thread를 리턴하는 함수이다. 글로벌 변수 all\_list에 담긴 thread들을 iterate 하게 탐색하고, 해당 list\_element의 tid값이 전달된 인자값과 같다면 해당 list\_element(thread)를 리턴한다. 없다면 NULL값을 리턴한다.

|  |
| --- |
| struct thread \* get\_thread\_by\_tid(tid\_t tid){  struct thread \* child\_thread;  struct list\_elem \* elem;  for(elem = list\_begin(&all\_list); elem != list\_end(&all\_list) ; elem = list\_next(elem)){  child\_thread = list\_entry(elem, struct thread, allelem);  if(child\_thread->tid == tid){  return child\_thread;  }  }  return NULL;  } |

**1-3. /threads/thread.h**

- 새로운 멤버 변수들을 추가하였다. thread 종료 시 상태값을 담을 **exit\_status**, 부모 thread의 tid값을 담을 **parent\_tid**, tparent thread가 child thread의 종료를 기다리게끔 처리하기 위해서 semaphore 구조체인 **child\_sema** 멤버 변수들을 추가로 선언하였다.

- tid 값을 인자로 받아 thread를 리턴해주는 get\_thread\_by\_tid 함수의 prototype을 선언하였다.

|  |
| --- |
| **struct thread**  **{**  **...**  **int exit\_status;**  **tid\_t parent\_tid;**  **struct semaphore child\_sema;**  **...**  **};**  **struct thread \* get\_thread\_by\_tid(tid\_t tid);** |

1. User Memory Access

**2-1. /userprog/exception.c**

- page\_fault 함수 내에 유효하지 않은 메모리에 접근하는 것을 막는 분기문을 하나 추가하였다. user가 없거나, fault\_addr가 유저 메모리의 주소가 아니거나, 커널 메모리의 주소일 경우 exit system call을 호출하여 종료시킨다.

|  |
| --- |
| **static void**  **page\_fault (struct intr\_frame \*f)**  **{**  **...**  **/\* CSE4070 implementation \*/**  **if(!user || !is\_user\_vaddr(fault\_addr) || is\_kernel\_vaddr(fault\_addr)) {**  **exit(-1);**  **}**  **...**  **}** |

**2-2. /userprog/syscall.c**

- system call 호출 전에 유효한 메모리 접근인지 검사를 하는 분기문을 작성하였다. is\_user\_vadder 함수를 호출하여 유효하지 않다면 프로세스를 종료시킨다.

|  |
| --- |
| static void  syscall\_handler (struct intr\_frame \*f UNUSED)  {  switch (\*(uint32\_t\*)(f->esp)){  ...  case SYS\_EXIT: {  **if(!is\_user\_vaddr(f->esp+4)) {**  **exit(-1);**  **}**  exit(\*(uint32\_t \*)(f->esp + 4));  break;  }  ...  // 다른 SYSTEM CALL 분기문들에도 다 작성되어 있음  }  } |

1. System Calls

**3-1. /userprog/syscall.c**

**void halt (void)**

- shutdown\_power\_off 를 호출하여 pintos를 종료시키는 함수이다.

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

**void exit (int status)**

- 현재 실행 중인 thread를 종료시키는 함수이다. child thread가 종료되면 parent thread가 child의 exit status를 갖게 되므로 parent thread를 찾아 exit\_status 값을 할당해준다.

|  |
| --- |
| **void exit (int status) {**  **struct thread\* curr, \* parent\_thread;**  **int parent\_tid;**  **curr = thread\_current();**  **printf("%s: exit(%d)\n", curr->name, status);**  **parent\_tid = curr->parent\_tid;**  **parent\_thread = get\_thread\_by\_tid(parent\_tid);**  **parent\_thread->exit\_status = status;**    **thread\_exit();**  **}** |

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

- 인자로 받은 cmd\_line을 실행시켜주는 함수이다. process\_execute로 cmd\_line을 전달하고, 해당 함수를 통해 새로 생성된 thread의 tid값을 반환해준다.

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

**int wait (pid\_t pid)**

- parent thread가 child thread 종료되기 전에 종료되는 것을 막기 위해 기다리게 끔 해주는 함수이다. 인자로 받은 pid를 가지는 thread(child thread)가 끝날 때까지 기다리다가 종료되면 해당 thread의 status를 리턴한다.

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

**int read (int, void \*, unsigned)**

- input\_getc를 iterate하게 호출하여 buffer에 문자를 담는데, size만큼 읽어들인 후에 그 size를 리턴한다. size 보다 더 적은 순서에 읽기가 끝난 경우에는 그 때까지 읽은 횟수를 리턴한다. fd 값이 0이 아니라면 -1을 리턴한다.

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

**int write (int, const void \*, unsigned)**

- fd 값이 1인 경우 (STDOUT)에 putbut 함수를 호출하여 size만큼의 buffer를 출력하고 해당 사이즈를 리턴한다. fd값이 1이 아닌 경우는 -1을 리턴한다.

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

**int fibonacci (int n)**

- 인자로 들어온 n 번째 값에 해당하는 피보나치 수열의 값을 리턴한다.

|  |
| --- |
| **int fibonacci (int n)**  **{**  **if (n <= 0) {**  **return 0;**  **} else if (n == 1) {**  **return 1;**  **}**  **return fibonacci(n - 2) + fibonacci(n - 1);**  **}** |

**int max\_of\_four\_int (int n1, int n2, int n3, int n4)**

- 들어온 네 개의 인자 중 가장 큰 값을 리턴한다.

|  |
| --- |
| **int max\_of\_four\_int (int n1, int n2, int n3, int n4){**  **int temp1, temp2;**  **temp1 = n1 >= n2 ? n1 : n2;**  **temp2 = n3 >= n4 ? n3 : n4;**  **return temp1 >= temp2 ? temp1 : temp2;**  **}** |

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

- 인자로 들어온 interrupt frame의 esp (stack pointer)를 이용하여 system call number에 해당하는 system call을 호출시켜주는 함수이다. switch 문을 사용하여 각 system call 호출부를 분기하였고, 리턴값은 eax 멤버 변수에 할당한다.

|  |
| --- |
| **static void**  **syscall\_handler (struct intr\_frame \*f UNUSED)**  **{**  **uint32\_t syscall\_number;**  **if (!is\_user\_vaddr(f->esp)) {**  **exit(-1);**  **}**  **syscall\_number = \*(uint32\_t\*)(f->esp);**  **switch (syscall\_number){**  **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(\*(char\*\*)(f->esp + 4));**  **break;**  **}**  **case SYS\_WAIT: {**  **if (!is\_user\_vaddr(f->esp + 4)) {**  **exit(-1);**  **}**  **f->eax = wait(\*(int\*)(f->esp + 4));**  **break;**  **}**  **case SYS\_READ: {**  **if (!is\_user\_vaddr(f->esp + 4) || !is\_user\_vaddr(f->esp + 8) || !is\_user\_vaddr(f->esp + 12)) {**  **exit(-1);**  **}**  **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: {**  **if (!is\_user\_vaddr(f->esp + 4) || !is\_user\_vaddr(f->esp + 8) || !is\_user\_vaddr(f->esp + 12)) {**  **exit(-1);**  **}**  **f->eax = write((int)\*(uint32\_t\*)(f->esp + 4),(void\*) \*(uint32\_t\*)(f->esp + 8),(unsigned)\*(uint32\_t\*)(f->esp + 12));**  **break;**  **}**  **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) || !is\_user\_vaddr(f->esp + 8) || !is\_user\_vaddr(f->esp + 12) || !is\_user\_vaddr(f->esp + 16)) {**  **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;**  **}**  **}**  **}** |

**3-2. /userprog/syscall.h**

- system call 함수들의 prototype을 선언하였다.

|  |
| --- |
| #ifndef USERPROG\_SYSCALL\_H  #define USERPROG\_SYSCALL\_H  #include "lib/user/syscall.h"  void syscall\_init (void);  /\* CSE4070 implementation \*/  **void halt (void);**  **void exit (int);**  **pid\_t exec (const char \*);**  **int wait (pid\_t);**  **int read (int, void \*, unsigned);**  **int write (int, const void \*, unsigned);**  **int fibonacci (int);**  **int max\_of\_four\_int (int, int, int, int);**  #endif /\* userprog/syscall.h \*/ |

1. Additional System calls

**4-0. /userprog/syscall.c**

**int fibonacci (int n)**

- 위 2-1에 기술.

**int max\_of\_four\_int (int n1, int n2, int n3, int n4)**

- 위 2-1에 기술

**4-1. /examples/additional.c**

- arguments 를 받아 fibonacci, max\_of\_four\_int 각각의 system call을 호출하여 값을 출력한다.

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

**4-2. /examples/Makefile**

- additional.c 파일이 링킹 및 컴파일될 수 있도록 Makefile을 수정한다.

|  |
| --- |
| ...  PROGS = cat cmp cp echo halt hex-dump ls mcat mcp mkdir pwd rm shell \  bubsort lineup matmult recursor **additional**  ...  **additional\_SRC = additional.c**  ... |

**4-3. /lib/user/syscall.c**

- 기존에는 3개까지의 인자를 받는 syscall3 매크로까지만 구현되어 있었으나, max\_of\_four\_int system call을 위해 argument를 4개 받을 수 있는 syscall4 매크로를 구현하였다.

|  |
| --- |
| **#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; \**  **})** |

- syscall 매크로를 호출하는 fibonacii, max\_of\_four\_int 함수를 아래에 정의하였다.

|  |
| --- |
| **int**  **fibonacci (int n)**  **{**  **return syscall1 (SYS\_FIBONACCI, n);**  **}**  **int**  **max\_of\_four\_int (int n1, int n2, int n3, int n4)**  **{**  **return syscall4 (SYS\_MAX\_OF\_FOUR\_INT, n1, n2, n3, n4);**  **}** |

**4-4. /lib/user/syscall.h**

- /lib/user/syscall.c에 정의한 함수를 헤더파일에 선언하였다.

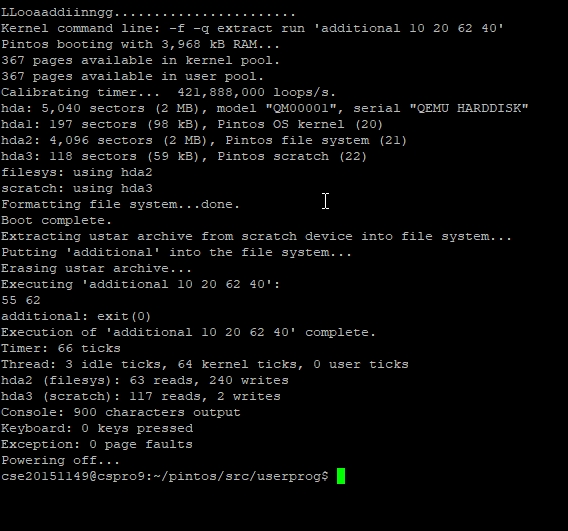
|  |
| --- |
| ...  **int fibonacci (int);**  **int max\_of\_four\_int (int, int, int, int);**  ... |

**4-5. /lib/syscall-nr.h**

- fibonacci, max\_of\_four\_int system call 호출을 위한 enum 값을 추가한다.

|  |
| --- |
| enum  {  ...  **SYS\_FIBONACCI,**  **SYS\_MAX\_OF\_FOUR\_INT,**  } |

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

****