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

1. **개발 목표**

프로젝트 0에서 설치한 pintos는 argument passing과 system call이 구현되지 않은 상태이다. 프로그램을 실행하여도 아무런 실행이 되지 않고 있기 때문에 원하는 결과 값을 볼 수 없다. 본 프로젝트의 목표는 핀토스 운영체제에서 user program이 실행되는 원리와 과정을 이해하고 user program이 정상적으로 실행될 수 있도록 pintos에 argument passing과 system call을 구현하는 것이다.

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

userprog/process.c의 setup\_stack() 함수에서 스택을 할당받은 후 사용자로부터 받은 arguments들을 parsing하여 user stack에 80x86 calling convention방식으로 저장한다. 유저 프로그램의 argument가 caller stack에 정상적으로 저장되어 이후 프로그램을 실행할 때 stack에 접근하여 실행하게 된다.

1. User Memory Access

system call이 호출되고 user 프로그램이 컴퓨터의 메모리 공간에 접근할 경우 kernel 메모리 공간에 침범하지 않도록 확인하는 과정이 필요하다. system call handler에서 접근하는 메모리 주소가 유효하지 않거나 kernel 메모리를 침범한다면 프로그램을 종료하게 한다. user program에서 잘못된 메모리에 접근하여 프로그램 또는 운영체제 자체에서 문제가 생기는 것을 방지해준다.

1. System Calls

system call이 호출되면 system call number에 따라서 각각 알맞는 시스템 콜을 호출해주어야 한다. 이를 위해서 System call handler를 구현하고 halt, exec, exit, wait write, read 시스템 콜을 구현해서 프로그램이 요구하는 알맞은 동작을 하도록 한다. 시스템 콜이 구현되면 프로그램이 정상적으로 실행된다.

* 1. **개발 내용**
* Argument Passing

user program은 보통 여러개의 arguments를 가지고 있다. 이를 적절히 parsing하여 caller stack에 저장한 다음 이후 system call에서 stack에 접근하여 이를 사용할 수 있도록 한다. 핀토스에서는 80x86 calling convention방식으로 arguments를 stack에 저장한다. user stack은 user virtual memory의 가상 메모리의 가장 위인 PHYS\_BASE에서 시작해서 아래로 내려가며 저장된다.

예시로 ‘/bin/ls -l foo bar’이라는 프로그램을 생각해보자. 이 프로그램은 총 4개의 arguments를 가지고 있다. stack에 이를 저장하기 위해서는 먼저 공백(“ “)을 기준으로 이를 parsing해준 후 argv에 저장한다. argc에는 argument의 개수를 저장한다. 아래는 parsing 후의 결과이다.

|  |  |
| --- | --- |
| argv[0] = “/bin/ls\0”  argv[1] = “-l\0”  argv[2] = “foo\0” | argv[3] = “bar\0”  argv[4] = NULL  argc = 4 |

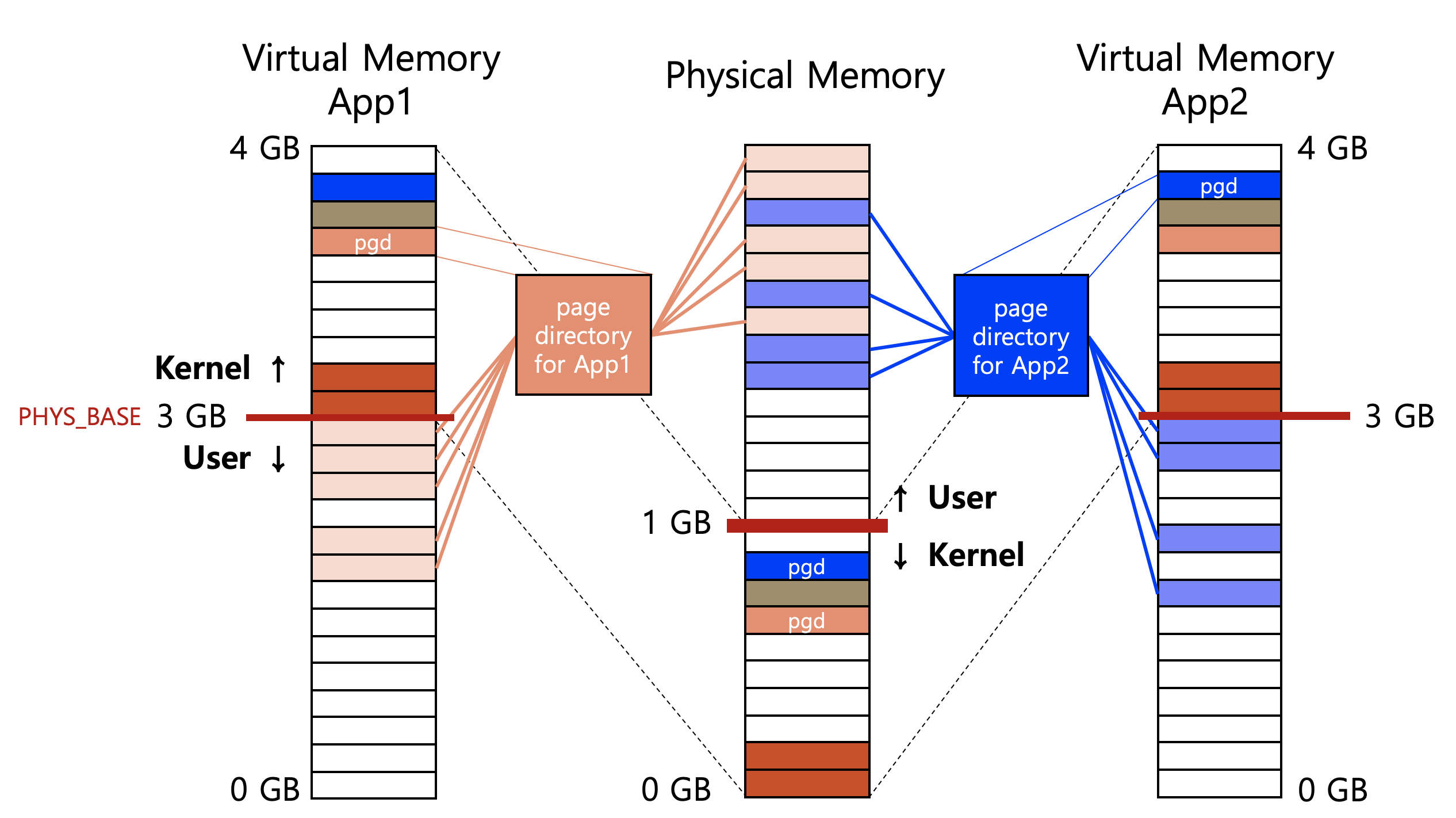
parsing한 이후에는 스택을 가르키는 포인터 esp를 사용해 이 정보들을 메모리에 저장한다. esp는 PHYSE\_BASE부터 시작하기 때문에 값을 감소시키면서 스택에 저장한다. 먼저 argv[0] ~ argv[argc-1]을 스택에 쌓은 후 esp의 값이 4의 배수가 되도록 word-align을 추가한다. 이후 앞서 삽입한 문자열들의 주소를 스택에 저장한다. 이 때는 반드시 역순으로 주소값을 저장한다. 이후 argv, argc와 프로세스가 종료 된 후 부모 프로세스로 돌아가기 위한 return address까지 저장한다.



결과적으로 위 그림처럼 각 arguments들이 스택에 저장되게 되고 esp는 스택의 주소를 가리키고 있게 된다.

* User Memory Access

운영체제에서는 메모리를 user과 kernel메모리로 나눠서 관리를 하게 된다. 운영체제에서 invalid memory access란 프로세스 도중 user program이 kernel 메모리 공간을 침범하는 것이다. 이 경우 운영체제에 문제가 생겨 큰 오류가 발생할 수 있다. 이러한 문제들을 방지하기 위해서 운영체제는 가상 메모리 시스템을 지원한다. 가상메모리는 실제 물리적으로 존재하는 메모리는 아니지만 각 프로세스가 자신의 메모리처럼 사용할 수 있는 공간이다. 가상 메모리 또한 user와 kernel메모리 영역으로 나눠진다. 각 프로세스마다 4GB의 가상 메모리를 할당받게 되고 PHYS\_BASE (3GB) ~ 4GB가 kernel 메모리, 그 이외의 3GB는 user 메모리 공간이다. 운영체제는 이 메모리 공간을 page단위로 나눠서 page directory를 사용하여 가상 메모리와 실제 메모리를 mapping한다.

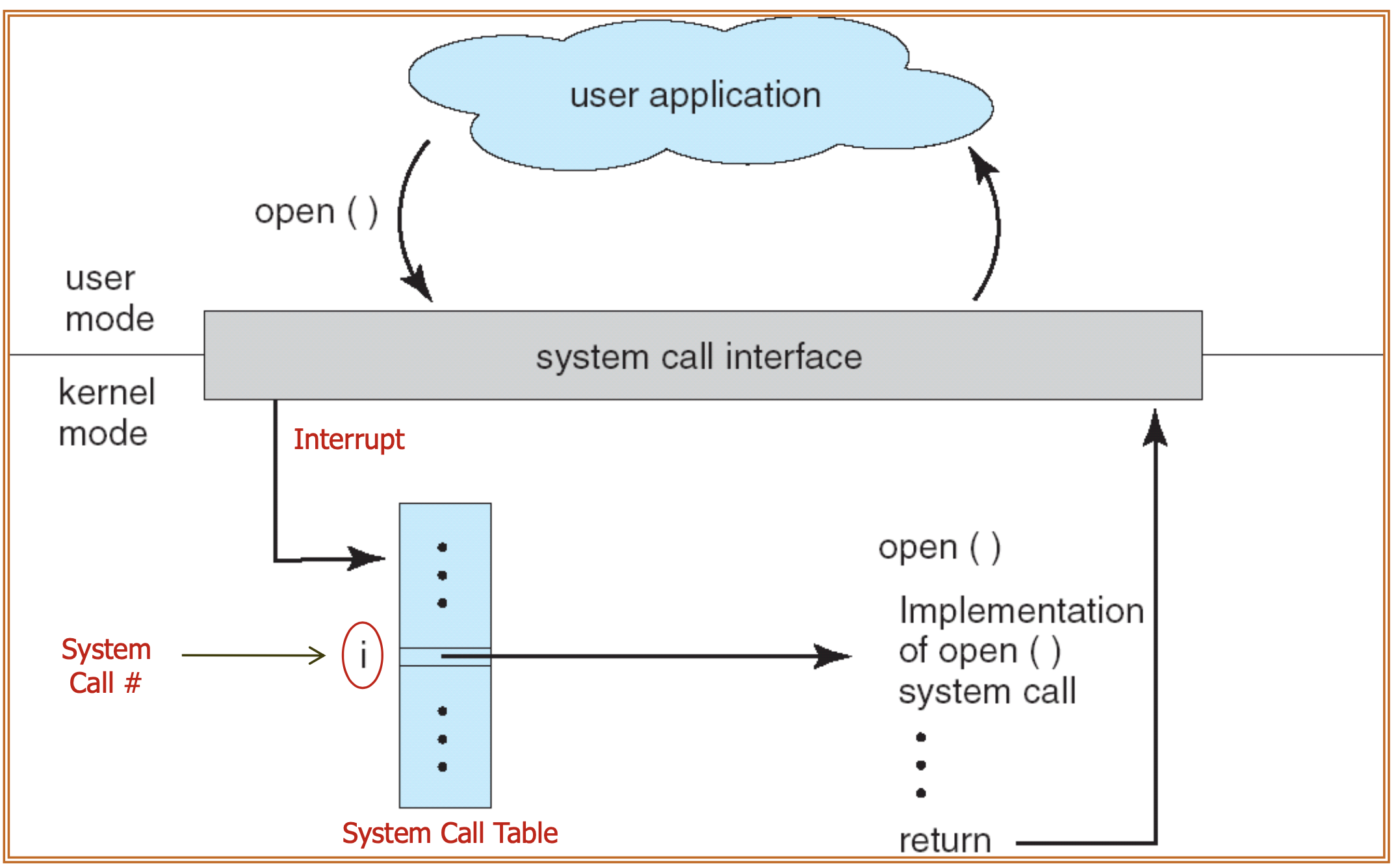


invalid memory access 문제를 방지하기 위해서는 접근하는 메모리 공간이 user 메모리 공간인지를 확인하는 과정이 필요하다. 이를 위해서는 system call handler에서 메모리 공간을 확인하는 함수를 구현하여 invalid memory access가 일어난 경우 프로그램을 종료하도록 한다. kernel memory에 접근하는 경우, 메모리가 유효하지 않거나 mapping된 page가 없는 경우에 프로그램을 종료하여 invalid memory access로 인한 프로그램 에러를 방지한다.

* System Calls

앞서 언급했듯 운영체제에는 kernel과 user mode가 존재한다. 운영체제에서는 user mode에서 kernel 메모리에 접근하는 instructions를 하지 못하도록 막는다. 하지만 user program에서는 kernel 메모리에 접근하여 처리해야하는 명령들이 있을 것이다. user program에서 kernel의 기능을 이용하기 위해서는 user mode에서 직접적으로 하지 못하기 때문에 system call을 호출하여 kernel에게 이를 요청한다. 즉 system call은 user mode에서 priviledged instructions(kernel mode에서만 수행할 수 있는 작업들)를 처리하기 위해 꼭 필요한 것이 system call이다. 운영체제에서는 이 system call을 라이브러리인 API로 지원한다. 즉 system call은 user program에서 API 함수를 호출하는 거라고 볼 수 있다.

user progam이 system call을 하면 API의 함수가 호출되게 되고 system call number과 arguments가 stack에 저장된다. System call이 kernel mode로 넘어가기 위해서는 interrupt가 일어나는데system call의 고유 interrupt number인 %0x30 interrupt가 일어난다. kernel mode에서는 interrupt를 위한 stack을 생성하고 interrupt handler를 호출한다. 이 interrupt handler는 system call handler를 호출하고 system call handler는 stack에 저장된 system call number에 접근해 알맞는 system call 함수를 호출해준다. system call이 실행된 후 반환되면 다시 user mode로 이동해 process를 계속 진행한다.



본 프로젝트에서 구현해야할 system call은 총 6가지 이다.

|  |  |
| --- | --- |
| system call | 설명 |
| halt | shutdown\_power\_of()를 호출하여 핀토스를 종료한다. |
| exit | 현재 프로세스를 종료하고 exit status를 반환한다. exit status가 0이라면 정상인 종료이고 0이외의 값이라면 비정상적인 종료이다. |
| exec | child process를 생성하고 프로그램을 실행한다. |
| wait | child process가 유효한지 확인하고 실행이 종료될 때까지 기다린다. child thread의 exit status를 반환한다. |
| write | putbuf()를 이용해서 STDOUT에 출력한다. |
| read | input\_getc()를 이용해서 STDIN에서 읽는다. |

이외에도 본 프로젝트에서는 추가적으로 2개의 새로운 시스템 콜을 구현한다.

-fibonacci(int n) : n번째 피보나치 수열을 반환한다

-int max\_of\_four\_int(int a, int b, int c, int d) : a, b, c, d 중 가장 큰 숫자를 반환한다

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |
| --- | --- |
| 날짜 | 수행 내용 |
| 10.05 ~ 10.10 | 이론 이해 및 프로젝트 계획 |
| 10.10 ~ 10.17 | Argument Passing 구현 |
| 10.10 ~ 10.17 | System Call Implementation |
| 10.17 ~ 10.24 | System Call Handler |
| 10.17 ~ 10.24 | User Memory Access |
| 10.24 ~ 11.01 | Additional Implementation, evaluation |
| 10.05 ~ 11.01 | Documentation |

* 1. **개발 방법**

<Argument Passing>

- userprog/process.c 에 argument를 parsing하여 stack에 쌓아주는 make\_stack()이라는 함수를 작성한다.

- userprog/process.c load()에서 file\_name에서 실제 프로그램 이름만을 추출하여 filesys\_open()의 인자로 넣어주어 프로그램이 load되도록 해준다.

- userprog/process.c load()에서 setup\_stack()이후 make\_stack()을 호출해 stack에 arguments를 저장한다.

<System Calls>

-userprog/syscall.c에 system call handler()를 구현하여 시스템 콜 번호에 따라 적절한 system call을 수행할 수 있도록 해준다. stack에 접근할 경우에는 항상 invalid memory access check를 한다.

-userprog/syscall.c에 system call 함수 halt, exit, exec, wait, read, write를 작성한다.

-threads/thread.h에 thread 구조체에 child process를 관리하는 list와 list 원소를 추가한다. wait system call을 구현하기 위해서 2개의 semaphore도 구현한다. 또한 thread가 생성될 때 이 변수들을 초기화할 수 있도록 init\_thread()에 초기화 코드를 작성한다.

-userprog/process.c의 process\_wait()에서는 child thread를 저장하는 list를 iterate해서 기다리고 있는 child thread를 찾는다. 이후 sema\_down(a)을 이용해 기다린다. child process에서 종료가 되면서 sema\_up(b)을 하면 list에서 종료된 child를 remove해주고 sema\_up(b)을 해 child process 완전히 종료시킨다.

-userprog/process.c의 process\_exit()에서는 thread가 종료된다는 것을 알리기 위해 sema\_up(a)을 하고 parent가 child thread를 list에서 제거하기 까지 기다리기 위해 sema\_down(b)을 한다. parent가 remove를 하고 sema\_up(b)을 하면 child thread를 완전히 종료한다.

<Additional System Calls>

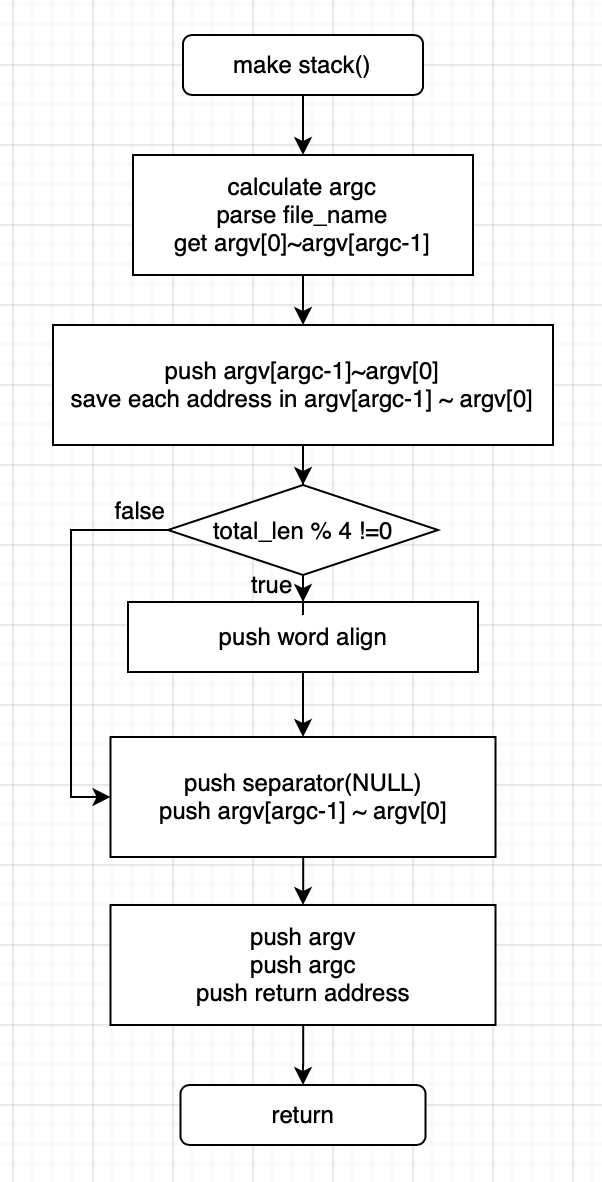
-lib/syscall-nr.h 에 system call number SYS\_FIBONACCI, SYS\_MAX\_OF\_FOUR\_INT 를 추가한다.

-lib/user/syscall.h fibonacci()와 max\_of\_four\_int()의 prototype을 작성한다.

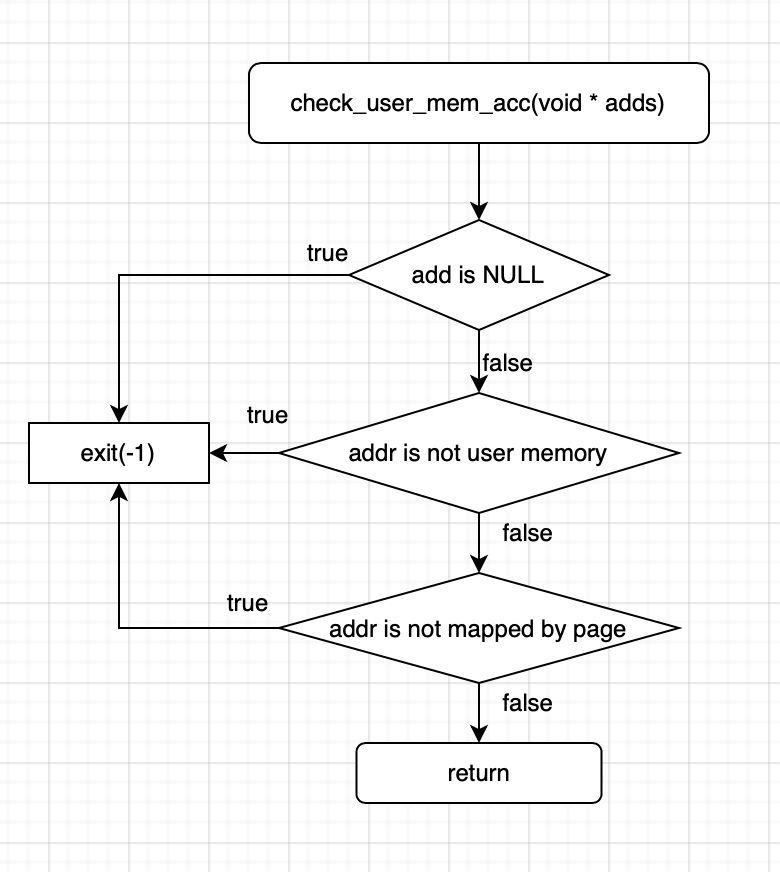
-lib/user/syscall.c 인자가 4개인 시스템 콜 매크로 syscall4를 구현한 이후 fibonacci(int n) 에서는 syscall1(SYS\_FIBONACCI, n)을 max\_of\_four\_int(int a, int b, int c, int d)에서는 syscall4(SYS\_MAX\_OF\_FOUR\_INT, a, b, c, d)를 호출한다.

-userprog/syscall.c 의 system call handler()에서 새로운 시스템 콜을 처리하게 해준다. 또한 실제 시스템 콜인 fibonacci()와 max\_of\_four\_int()를 작성한다.

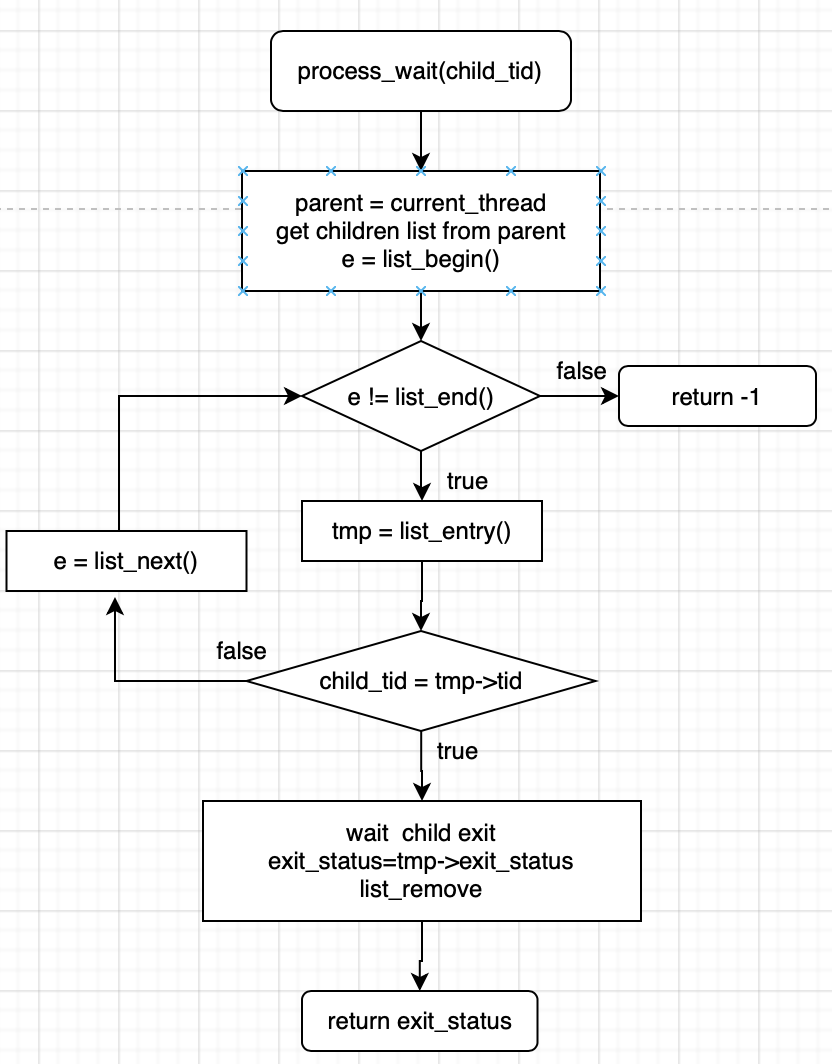
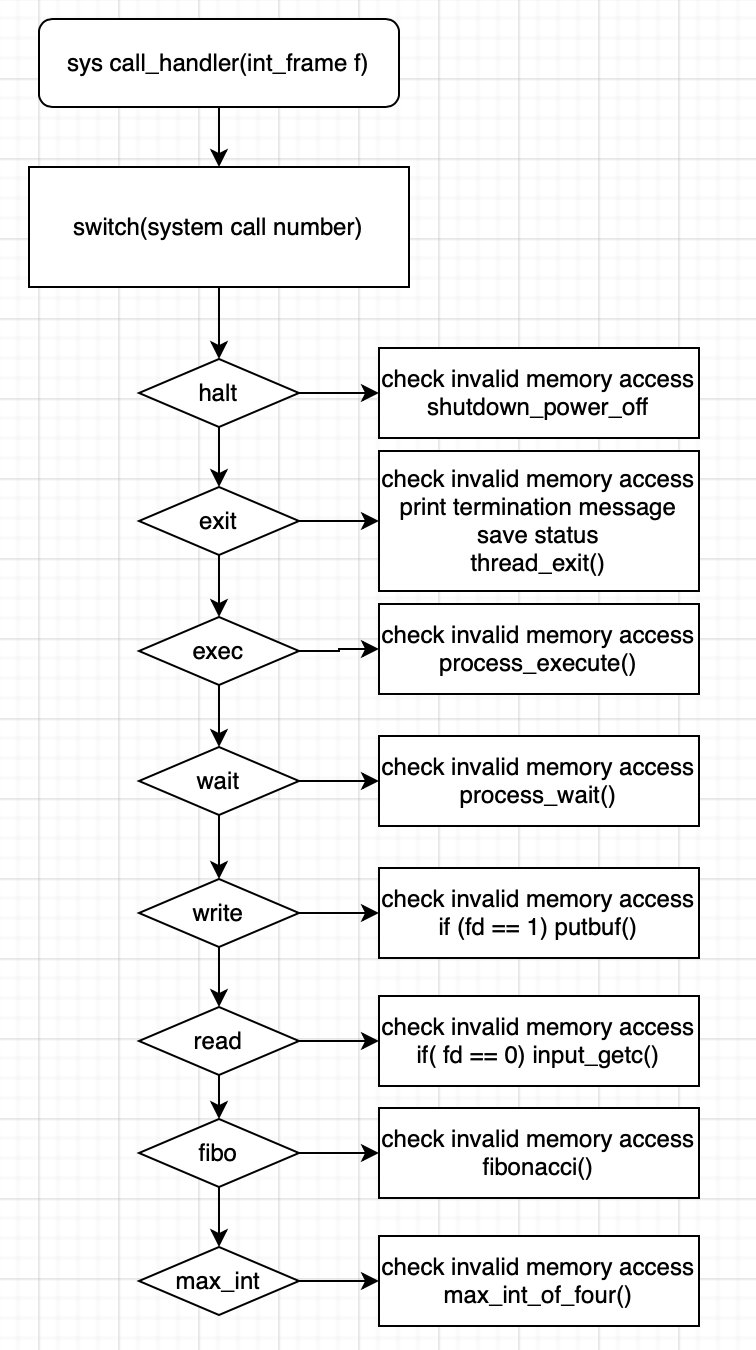
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing



1. User Memory Access



1. System Calls



* 1. **제작 내용**

1. Argument Passing

|  |  |
| --- | --- |
| userprog/process.c | 설명 |
| void make\_stack(char \*file\_name, void \*\*esp) {  char \*\* argv;  int argc, total;  char copied\_file\_name[256];  char \*token, \*save;  int I, len;    strlcpy(copied\_file\_name, file\_name, strlen(file\_name) + 1);  token = strtok\_r(copied\_file\_name, " ", &save);  argc = 0;    //get argc  while (token != NULL) {  argc++;  token = strtok\_r(NULL, " ", &save);  }  argv = (char \*\*)malloc(sizeof(char \*) \* argc);  //get argv  strlcpy(copied\_file\_name,file\_name, strlen(file\_name) + 1);  token = strtok\_r(copied\_file\_name, " ", &save);  for(i = 0; i<argc; i++){  argv[i] = token;  token = strtok\_r(NULL, " ", &save);  }  //push strings in argv  total = 0;  for (i = argc - 1; 0 <= i; i --) {  len = strlen(argv[i])+ 1;  \*esp -= len;  total += len ;  strlcpy(\*esp, argv[i], len);  argv[i] = \*esp;  }  //push word alignment  if( total % 4 !=0){  \*esp -=4 - (total % 4);  }  //push seperator  \*esp -= 4;  \*\*(uint32\_t \*\*)esp = 0;  //push argument address  for (i = argc - 1; 0 <= i; i--) {  \*esp -= 4;  \*\*(uint32\_t \*\*)esp = argv[i];  }    //push argv  \*esp -= 4;  \*\*(uint32\_t \*\*)esp = \*esp + 4;  //push argc  \*esp -= 4;  \*\*(uint32\_t \*\*)esp = argc;  //push return adress  \*esp -= 4;  \*\*(uint32\_t \*\*)esp = 0;  free(argv);  //debug  //hex\_dump( \*esp, \*esp, 100, 1);  } | -file\_name을 parsing하여 스택 주소 esp에 순차적으로 저장한다.  -file\_name을 보존하기 위해서 copied\_file\_name에 복사한다.  -argument의 개수를 구하기 위해서 copied\_file\_name에 “ “를 delimeter로 strtok\_r()를 실행하여 arg 값을 count up 해준다.  -argc를 계산했으면 각 argument를저장할 2차원 문자열 배열을 그 크기만큼 argv에 동적 할당 해준다.  -다시 copied\_file\_name을 초기화한 후 strtock\_r을 사용해 이번엔 argv에 각 argument를 저장한다.  -esp의 주소를 argument 문자열 길이만큼 뺀 후에 esp에 argv내용을 복사한다. argv[argc-1]부터 argv[0]까지 저장한다.  -총 저장한 argument의 길이 정보를 가지는 total변수를 두고 argument를 저장할 때마다 이를 갱신했다.  -total이 4의 배수가 아니라면 esp의 주소 또한 4로 나눠 떨어지기 않기 때문에 필요한 만큼 1-bit word-align을 추가했다.  -주소값을 저장하기 전에 null을 저장해 구분해준다.  -앞서 argv[i]를 저장한 후 argv[i]가 저장된 주소값 esp를 저장했다.  -이 또한 역순으로 stack에 저장하였다. 포인터 변수의 크기는 4이기 때문에 저장 전 esp에서 4를 빼준다.  -argv를 스택에 저장한다. 즉 argv[0]이 저장된 주소를 스택에 쌓는다.  -arguments의 개수인 argc를 스택에 저장한다.  -마지막으로 return address까지 스택에 저장한다.  -동적 할당한 argv를 메모리에서 해제한다.  -스택에 argument가 잘 저장되어있는지 확인하기 위해 디버깅으로 hex\_dump를 이용했다. |
| char \*token, \*save;  char copy\_file\_name[256];  strlcpy(copy\_file\_name,file\_name, strlen(file\_name) + 1);  token = strtok\_r(copy\_file\_name, " " , &save);  /\* Open executable file. \*/  file = filesys\_open (token);  if (file == NULL)  {  printf ("load: %s: open failed\n", file\_name);  goto done;  } | -file\_name을 보존하기 위해서 copy\_file\_name에 file\_name을 복사한다.  -strtok\_r을 이용해서 공백을 기준으로 가장 앞에 잇는 argument를 parse했다.  -프로그램의 첫번째 인자는 항상 프로그램 이름이다. 이 프로그램 이름을 filesys\_open에 전해줘야 하기 때문에 parse한 문자열을 인자로 넣어주었다. |

1. User Memory Access

|  |  |
| --- | --- |
| userprog/syscall.c |  |
| void check\_user\_mem\_acc(void \*addr){  struct thread \*c\_thread = thread\_current();    //check if NULL pointer  if(addr == NULL) exit(-1);    //check if user virtual address  if(!is\_user\_vaddr(addr)) exit(-1);    //check if unmapped  if(!pagedir\_get\_page(c\_thread->pagedir, addr))  exit(-1);  } | -invalid memory access를 체크한다.  -메모리 주소가 유효하지 않은 값 NULL일 경우 프로그램을 종료한다.  -메모리 주소가 user 영역이 아니라면 프로그램을 종료한다.  -메모리 주소가 page에 mapping되지 않았다면 프로그램을 종료한다. |

1. System Calls

|  |  |
| --- | --- |
| userprog/syscall.c | 설명 |
| static void  syscall\_handler (struct intr\_frame \*f UNUSED)  {  //Check User Memory Access  void \*addr = (f->esp);  check\_user\_mem\_acc(addr);    switch(\*(uint32\_t \*)(f->esp)){  case SYS\_HALT :  halt();  break;  case SYS\_EXIT :  check\_user\_mem\_acc(f->esp + 4);  exit(\*(uint32\_t \*)(f->esp + 4));  break;    case SYS\_EXEC :  check\_user\_mem\_acc(f->esp + 4);  f->eax = exec((const char \*)\*(uint32\_t \*)(f->esp + 4));  break;    case SYS\_WAIT :  check\_user\_mem\_acc(f->esp + 4);  f-> eax = wait((pid\_t)\*(uint32\_t \*)(f->esp + 4));  break;    case SYS\_READ :  check\_user\_mem\_acc(f->esp + 20);  check\_user\_mem\_acc(f->esp + 24);  check\_user\_mem\_acc(f->esp + 28);  f->eax = read((int)\*(uint32\_t \*)(f->esp+20), (void \*)\*(uint32\_t \*)(f->esp + 24), (unsigned)\*((uint32\_t \*)(f->esp + 28)));  break;    case SYS\_WRITE :  check\_user\_mem\_acc(f->esp + 20);  check\_user\_mem\_acc(f->esp + 24);  check\_user\_mem\_acc(f->esp + 28);  f->eax = write((int)\*(uint32\_t \*)(f->esp+20), (void \*)\*(uint32\_t \*)(f->esp + 24), (unsigned)\*((uint32\_t \*)(f->esp + 28)));  break;  //thread\_exit ();  } | -system call 번호에 따라 적절한 system call 함수를 호출해주는 system call handler를 구현한다.  먼저 system call number가 담긴 주소가 valid한지 확인한다.  -switch문을 사용해 각각의 system call number에 따라 처리해준다.  -인자가 있는 경우에는 system call number 다음으로 stack에 차례대로 저장되어 있기 때문에 4를 더해서 접근한다.  -esp로 접근하는 모든 메모리는 check\_user\_mem\_acc()함수를 사용해서 invalid memory access를 검사한다.  -read, write같은 경우에는 인자들이 4칸씩 밀려 저장되었기 때문에 system call num에서 각각 20, 24, 28씩 더해 저장해준다. |
| void halt(void){  shutdown\_power\_off();  }  void exit(int status){  //process termination message  printf("%s: exit(%d)\n", thread\_name(), status);  thread\_current() -> exit\_status = status;  thread\_exit();  }  tid\_t exec( const char \*cmd\_line){  return process\_execute(cmd\_line);  }  int wait(tid\_t pid){  return process\_wait(pid);  }  int read(int fd, void \*buffer, unsigned size){  if (fd == 0){ //stdin  for(int i=0; i<size; i++)  ((char \*)buffer)[i]=input\_getc();  return size;  }  return -1;  }  int write(int fd, const void \*buffer, unsigned size){    if(fd ==1 ){  putbuf(buffer, size);  return size;  }  return -1;  } | -shutdown\_power\_off()를 호출해 pintos를 종료한다.  -현재 thread이름을 출력한 후 status를 변경해준다. 또한 thread\_exit()을 호출해 thread를 종료시킨다.  -process\_execute()함수를 호출한다.  -process\_wait()함수를 호출한다.  -fd가 0인 경우 input\_getc()를 이용해서 사이즈 만큼을 표준 입력에서 읽어온다. 실패하면 -1을 반환하고 성공하면 읽어온 데이터의 길이를 반환한다.  - fd가 1인 경우 putbuf()를 이용해서 사이즈 만큼을 표준 출력에서 출력한다. 실패하면 -1을 반환하고 성공하면 출력한 데이터의 길이를 반환한다. |

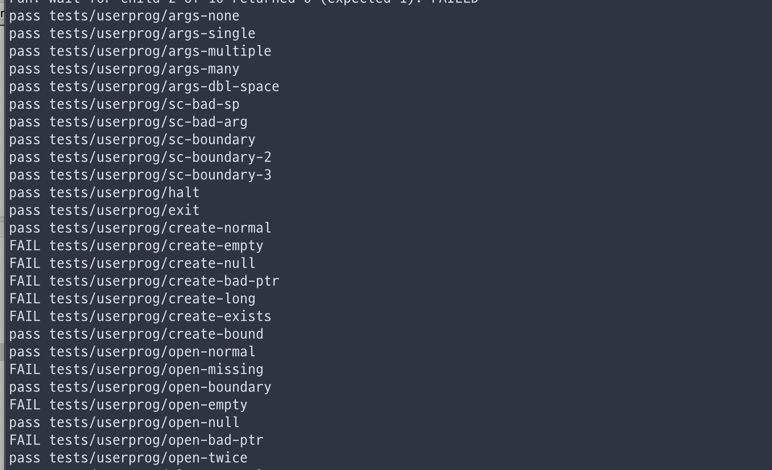
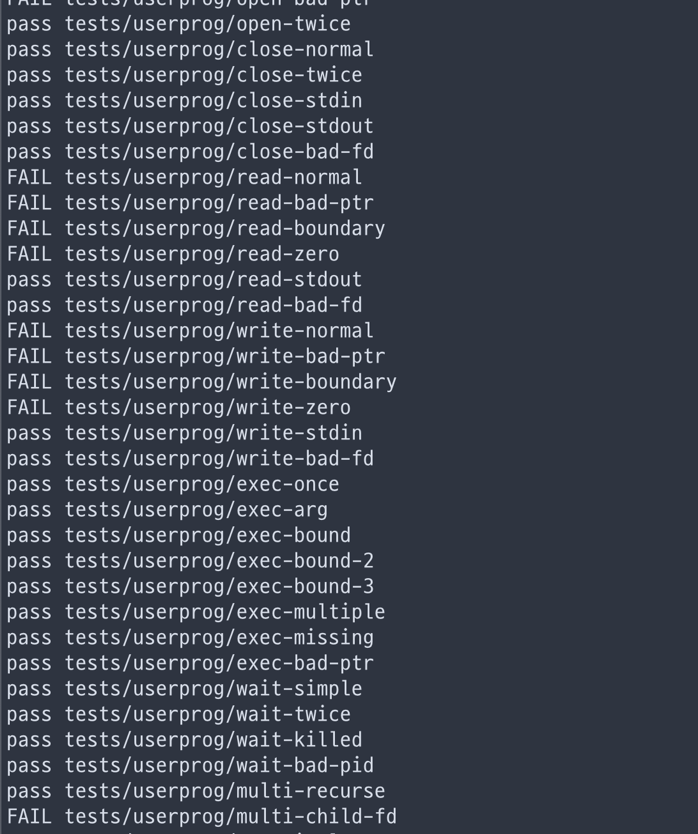
wait system call에서는 process\_wait()을 호출한다. process\_wait()에서는 child thread가 종료될 때까지 기다리고 exit\_status를 반환하는 기능을 구현해야 한다. 이를 위해 thread 구조체에 변수를 추가하고 semaphore를 사용해서 wait을 구현하였다.

|  |  |
| --- | --- |
| threads/threads.h |  |
| struct thread  {  …  #ifdef USERPROG  /\* Owned by userprog/process.c. \*/  uint32\_t \*pagedir;  struct list children;  struct list\_elem child\_elem;  struct semaphore c\_lock;  struct semaphore m\_lock;  int exit\_status;  #endif  } | -wait을 구현하기 위해 thread 구조체에 변수를 추가했다.  -child process를 저장하고 있는 list와 list의 원소를 추가했다.  -child process가 끝날 때까지 기다릴 수 있도록 semaphore를 추가했다.  -parent가 child process의 exit status를 받을 수 있도록 exit status를 추가했다. |
| threads/threads.c |  |
| static void  init\_thread (struct thread \*t, const char \*name, int priority)  {  …  #ifdef USERPROG  sema\_init(&(t->c\_lock), 0);  sema\_init(&(t->m\_lock), 0); /\* new \*/  list\_init(&(t->children));  list\_push\_back(&(running\_thread()->children), &(t->child\_elem));  #endif  } | -thread가 생성될 때 semaphore과 list를 초기화한다. |
| userprog/process.c |  |
| int  process\_wait (tid\_t child\_tid UNUSED)  {  struct thread \*parent = thread\_current();  struct list \*children = &(parent ->children);  struct list\_elem\* e;  struct thread\* tmp = NULL;  int exit\_status;  for (e = list\_begin(children); e != list\_end(children); e = list\_next(e)) {  tmp = list\_entry(e, struct thread, child\_elem);    if (child\_tid == tmp->tid) {  sema\_down(&(tmp->c\_lock));  exit\_status = tmp->exit\_status;  list\_remove(&(tmp->child\_elem));  sema\_up(&(tmp->m\_lock)); /\* new \*/  return exit\_status;  }  }  return -1;  } | -thread\_current()함수를 이용해 현재 thread인 parent thread를 가져온다.  -parent의 child가 저장되어 있는 리스트에 접근한다.  -차례대로 list를 iterate하면서 현재 기다리는 child의 tid와 같은 원소를 찾는다.  -기다리는 child thread를 찾았다면 child가 종료될 때까지 기다려주고 exit\_status를 저장한다. 또한 list에서 종료된 child를 제거하고 완전히 child thread를 종료시킨다. |
| void  process\_exit (void)  {  struct thread \*cur = thread\_current ();  …  sema\_up(&(cur->c\_lock));  sema\_down(&(cur->m\_lock));  } | -thread가 종료될 때는 먼저 c\_lock을 sema\_up해서 parent에게 이를 알린다.  -parent가 remove를 할 수 있게끔 기다리고 m\_lock을 sema up해서 thread를 완전히 종료시킨다. |

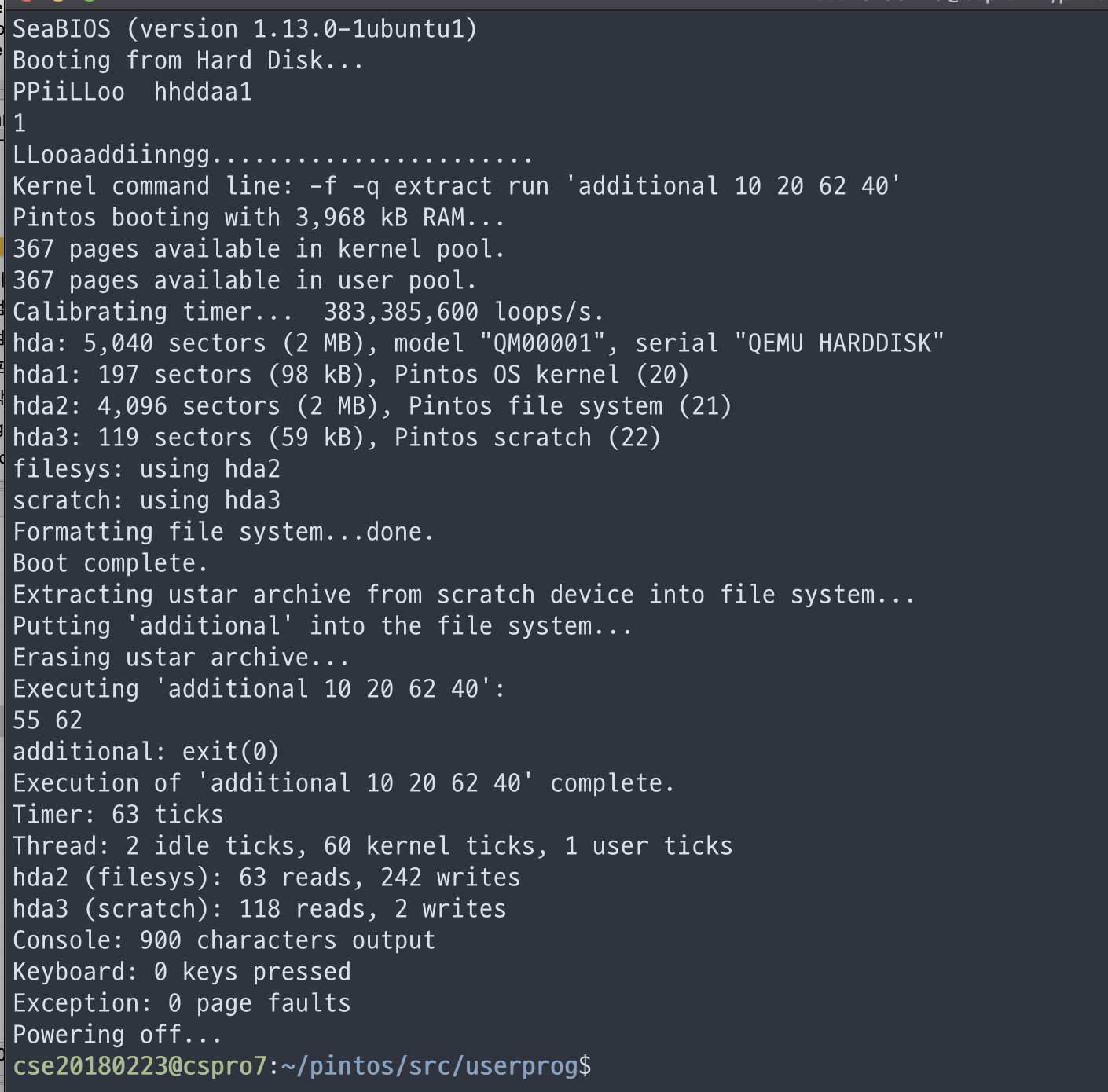
1. Additional System calls

|  |  |
| --- | --- |
| lib/syscall-nr.h |  |
| enum  {  …  SYS\_FIBONACCI,  SYS\_MAX\_OF\_FOUR\_INT  }; | -새로 구현할 system call number를 작성해주었다. |
| lib/user/syscall.h |  |
| #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] "g" (ARG0),  [arg1] "g" (ARG1),  [arg2] "g" (ARG2),  [arg3] "g" (ARG3)  : "memory");  retval;  })  int fibonacci(int n){  return syscall1(SYS\_FIBONACCI, n);  }  int max\_of\_four\_int(int a, int b, int c, int d){  return syscall4(SYS\_MAX\_OF\_FOUR\_INT, a, b, c,); | -4개의 인자를 처리하는 syscall4를 매크로로 정의해주었다.  -인자가 1개인 fibonacci()에서는 syscall1을 호출하도록 해주었다.  -인자가 4개인 max\_of\_four\_int()에서는 syscall4를 호출하도록 해주었다. |
| userprog/syscall.c |  |
| case SYS\_FIBONACCI :  check\_user\_mem\_acc(f->esp + 4);  f->eax = fibonacci((int)\*(uint32\_t \*)(f->esp + 4));  break;  case SYS\_MAX\_OF\_FOUR\_INT :  check\_user\_mem\_acc(f->esp + 24);  check\_user\_mem\_acc(f->esp + 28);  check\_user\_mem\_acc(f->esp + 32);  check\_user\_mem\_acc(f->esp + 36);  f->eax = max\_of\_four\_int((int)\*(uint32\_t \*)(f->esp + 24), (int)\*(uint32\_t \*)(f->esp + 28), (int)\*(uint32\_t \*)(f->esp + 32), (int)\*(uint32\_t \*)(f->esp + 36));  break; | -system call handler에서도 system call number가 SYS\_FIBONACCI 또는 SYS\_MAX\_OF\_FOUR\_INT인 경우를 처리해주었다.  -각 인자들의 invalid memory access를 검사한 이후에 함수를 호출한다. |
| int fibonacci(int n){  int n1 = 1, n2 = 1;  int i, temp;  if( n ==1 || n == 2 ){  return 1;  }  for( i = 3; i <= n; i++ ){  temp = n1;  n1 = n2;  n2 = n1 + temp;  }  return n2;  }  int max\_of\_four\_int(int a, int b, int c, int d){  int max=a;    if(max<b) max=b;  if(max<c) max=c;  if(max<d) max=d;    return max;  } | -n번째 피보나치 수열의 값을 반환하는 함수이다.  -n이 1 또는 2라면 1을 반환한다.  -n이 3이상의 수라면 for문을 사용해서 iterative하게 피보나치 수열을 계산한다.  -4개의 정수 중 가장 큰 정수를 반환하는 함수이다.  -max에 a를 할당하고 max가 b, c, d보다 작은 경우에 이를 갱신하였다. |

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

**  **

/userprog에서 make check를 한 결과 Proejct1에서 요구한 21개의 case를 모두 통과함을 알 수 있다.

****

examples에 additional.c를 추가하고 Makefile을 수정해 이를 컴파일할 수 있도록 하였다. ‘additional 10 20 62 40’을 실행하니 55와 62가 잘 출력됨을 확인할 수 있다.