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

담당 교수 : 박성용 교수님

조 / 조원 : 20150195 백인찬

개발 기간 : 2020.10.05 ~ 2020.11.01

1. **개발 목표**

* Pintos OS에서 user program을 작동하기 위해, Argument Passing을 가장 기본으로 구현하고, 간단한 System call인 halt, exit, read, exec, write, wait을 구현한다. 이를 위해 process와 thread의 개념, parent thread와 child thread, 그리고 thread간의 synchronization의 개념을 이해하고 구현할 수 있도록 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* Files to be modified and referenced

|  |  |
| --- | --- |
| **Path** | Files |
| **src/userprog** | process.h / process.c  syscall.h / syscall.c |
| **src/threads** | thread.h / thread.c  synch.h / synch.c  malloc.h |
| **src/devices** | shutdown.h / shutdown.c  input.h / input.c |
| **src/lib** | syscall-nr.h  user/syscall.h  user/syscall.c |
| **src/examples** | Makefile  additional.c |

1. Argument Passing

프로그램이 실행되는 Flow를 이해하고, 이를 바탕으로 입력 받은 인자를 공백(“ “)을 기준으로 tokenize한 후 순서에 맞게 Stack에 쌓는다. Stack은 위에서 아래로 쌓아 내려가는 방식으로 Stack Pointer인 ESP를 활용한다.

1. User Memory Access

프로그램이 잘못된 메모리에 접근하고 있을 경우에는 오류로 판단하여 프로그램을 종료해야 한다. 예를 들어, 사용자가 User Memory가 아닌 Kernel Memory에 접근하려 할 경우, 해당 프로세스에게 할당된 메모리 영역(Base ~ limit)이 아닌 다른 프로세스의 메모리를 참조하려 하는 경우 등에는 오류 처리하여야 한다.

1. System Calls

System call(halt, exit, exec, wait, read, write)을 구현하고 이 System call을 호출하기 위한 system\_call\_handler()를 구현한다. Multi-thread 환경임을 유의하여 parent thread와 child thread의 synchronization을 구현한다. 이를 위해 thread 구조체에 parent thread를 식별할 수 있는 변수와 semaphore 등을 추가한다.

* 1. **개발 내용**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명

Kernel 내에 있는 Stack에 argument를 쌓기 위해서 먼저 command line으로 입력 받은 인자 중 어떤 것이 실행하고자 하는 파일이고 어떤 것이 그 파일에 넘겨 줄 인자인지를 파악해야 한다. 이를 위해 입력 받은 문자열을 공백(“ “)을 기준으로 tokenize한 후, Stack Pointer인 esp를 활용해 Stack을 쌓는다.

이 과정은 push하면 밀려나는 느낌이 아닌 해당 주소에 마지막 토큰을 넣고 그 주소에 들어간 토큰의 byte수 만큼 아래로 내려간 주소에 그 이전 토큰을 넣는 것을 반복하여 인자의 내용을 Stack에 넣어준다. 인자를 모두 넣었으면, word-align을 추가하는데, 이는 데이터의 접근 속도를 빠르게 하기 위해 주소를 4의 배수로 맞춰주는 역할을 한다. 이후 토큰들의 주소 위와 같은 방식으로 아래로 쌓고, 전체 주소와 인자의 수, 그리고 반환할 return 주소를 쌓아준다.

이 과정을 마치면 esp는 가장 아래에 있을 것이고, 참조할 시에는 위로 올라가며 활용할 수 있다.

* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명

Pintos OS 상에서 각 프로세스는 고유의 virtual memory를 갖는데, 이를 위해 page로 실제 physical memory와 mapping하고 mapping된 page table과 page directory를 이용해 실제 메모리를 참조한다. 하지만 프로세스는 고유의 영역을 갖고 있고, 커널의 메모리 영역은 user process에 의해 접근되면 안 된다. 각 프로세스에게 할당된 영역을 벗어난 메모리에 접근하거나, 커널의 메모리 영역에 접근할 경우 오류로 처리될 수 있도록 한다.

* + Invalid memory access를 어떻게 막을 것인지 설명

이를 위해서 현재 접근하고 있는 주소가 user memory address인지, kernel memory address인지, 또 이 접근을 시도하는 주체가 user인지 kernel인지를 판단해 커널에 의한 접근이거나, user process가 kernel에 접근하려 하면 오류로 처리하여 종료하도록 한다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

앞서 설명했듯 user는 kernel의 메모리에 접근할 수 없다. 그렇기 때문에 kernel의 기능들, 예를 들어 I/O, 프로세스 제어 등을 user가 사용하게 하기 위해 OS는 system call을 제공하고, system call을 통해 kernel mode에 진입하여 필요한 기능을 사용할 수 있게 한다.

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

halt : pintos 프로그램을 종료시키는 함수이다.

exec : 인자로 받은 command를 하나의 프로세스로 실행시키는 함수이다. 프로세스에 주어진 인자를 넘기고 성공했을 경우 새로운 프로세스의 id를 반환하고 정상적이지 않은 경우 -1을 반환한다..

exit : 현재 user program을 종료시키는 함수이고, 인자로 받은 status 값을 반환한다. 만약 parent thread가 기다리고 있으면 parent에게 자신의 상태를 알려준다. 반환 값은 정상적은 종료일 경우 0이고, 비정상적일 경우 0이 아닌 수를 반환한다.

wait : child thread의 id를 받아 해당 child thread의 종료를 기다리고 child의 종료 status 값을 수거한다. 비정상적인 status 값으로 -1을 반환한다.

read : 정해진 size만큼 file descriptor가 0인 stdin으로부터 입력을 받아 buffer에 저장하는 함수로 input\_getc()를 이용해 한 글자씩 추가한다. 실제로 읽은 개수를 반환한다.

write : 정해진 size만큼 file desciptor가 1인 stdout에 buffer로부터 출력한다. putbuf()를 통해 출력한다.

fibonacci : 피보나치 수열 중 입력으로 받은 n 번째에 있는 수를 반환한다.

max\_of\_four\_int : 입력으로 받은 4개의 정수 중 최대값을 반환한다.

* + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

커널은 각 시스템 콜을 구분하기 위해 기능 별로 고유번호를 할당해 놓는다. 각 번호에 해당하는 서비스 루틴을 내부적으로 정의해 놓는다. 유저 레벨에서 시스템 콜 API를 호출하면 인터럽트를 발생시키며 커널 모드로 넘어가며 제어가 커널에게 넘겨진다. 요청된 시스템 콜에 해당하는 번호를 확인하고 그에 맞는 서비스 루틴을 수행한다. 루틴을 모두 처리하고 나면 다시 사용자 모드로 전환된다. Pintos에선 이를 구현하기 위해 /src/lib/syscall-nr.h에 각 시스템 콜을 enumerate하였고, 각 시스템 콜에 맞는 서비스 루틴을 /userprog/syscall.c에 명시했다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

10.5 (월) ~ 10/28 (수) : ppt/강의자료 및 pintos manual 분석

10.29 (목) ~ 10/30 (금) : Argument Passing / User memory access 구현

10/31 (토) ~ 11/01 (일) : System Call Handler / threads synchronization / System calls 구현(Additional Implement 포함)

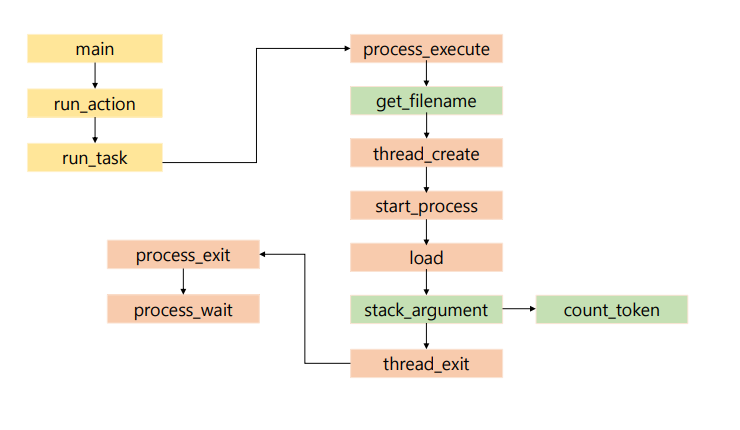
* 1. **개발 방법**

Argument Passing을 구현하기 위하여 userprog/process.c에 stack\_argument 함수와 count\_token 함수, get\_filename 함수를 구현한다. process\_execute에서 입력 받은 command line 중 filename을 get\_filename 함수로 filename을 parsing하고, start\_process에서 filename의 프로그램을 load한다. load를 성공적으로 마칠 경우 stack\_argument를 호출하여 인자를 parsing하고 스택에 쌓는다. 이후 스택에 정상적으로 쌓였는지 확인하기 위해 hex\_dump 함수를 이용해 디버깅한다.

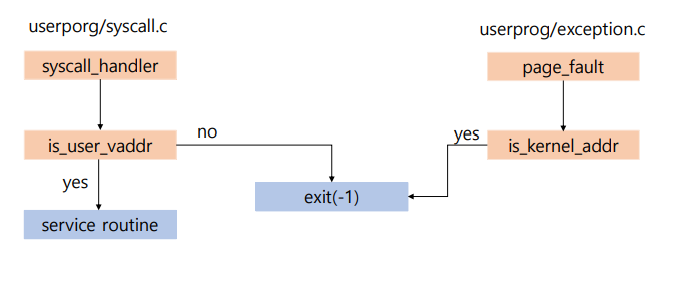
User Memory Access를 구현하기 위해 threads/vaddr.h에 있는 is\_kernel\_vaddr 함수 또는 is\_user\_vaddr 함수를 사용해 esp > PHYS\_BASE인 경우 (커널 메모리를 접근하는 경우) 오류 처리하여 종료하도록 한다.

System Call을 구현하기 위해 src/syscall.c의 syscall\_handler 함수 내부에 switch-case 문으로, lib/syscall-nr.h에 enumerate 되어 있는 시스템 콜의 번호에 맞게 루틴을 구현한다. 시스템 콜의 번호는 stack pointer인 esp가 가리키는 값을 사용해 구분한다.

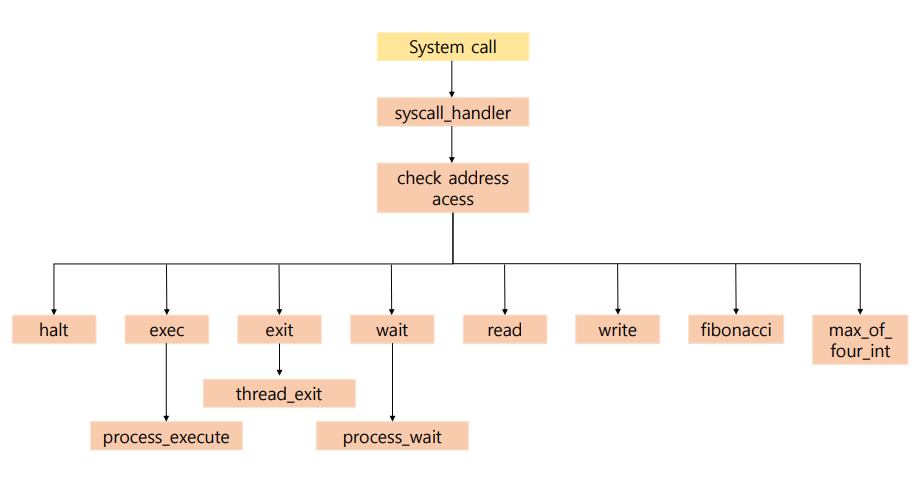
1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing



1. User Memory Access



1. System Calls



* 1. **제작 내용**

1. Argument Passing

/userprog/process.c

**void stack\_argument(char\* filename, void\*\* esp)**

입력받은 argument를 stack에 쌓는 함수로 count\_token 함수를 호출해 argument의 개수를 센 후, 그 수 만큼 배열을 할당한다. 마지막 argument부터 차례로 stack에 쌓은 후 wordalign으로 4의 배수를 맞춰주고, NULL값과 argument들의 각 주소, argument 전체 주소와 argument의 개수 마지막으로 return address를 쌓아주는 과정이다. esp와 argv\_len 변수를 통해 각 argument token의 길이만큼 아래로 내리면서 쌓는다.

|  |
| --- |
| void  stack\_argument(char\* file\_name, void \*\*esp){  char\*\* argv;  int i = 0;  unsigned len\_filename = strlen(file\_name)+1, total\_len = 0;  unsigned argv\_len = 0;  char\* temp = (char\*)malloc(len\_filename);  char\* ptr;  int argc = count\_token(file\_name);  argv = (char\*\*)malloc(sizeof(char\*)\*argc);  strlcpy(temp, file\_name, len\_filename);  argv[i++] = strtok\_r(temp, " ", &ptr);  for(; i<argc; i++) {  argv[i] = strtok\_r(NULL, " ", &ptr);  if(argv[i] == NULL) break;  }  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;  }  //pushing word\_align  int wordalign;  if (total\_len % 4 != 0) {  wordalign = total\_len % 4;  }  else {  wordalign = 0;  }  for(i = 0; i < wordalign; i++) {  \*esp-=1;  \*\*(uint8\_t\*\*)esp = 0; // 1byte  }  // push null pointer  \*esp -= 4;  \*\*(uint32\_t\*\*)esp = 0;    // push address  for(i = argc-1; i >= 0; i--) {  \*esp -= 4;  \*\*(uint32\_t\*\*)esp = argv[i];  }  //push \*\*argv address  \*esp-=4;  \*\*(uint32\_t\*\*)esp = \*esp+4;  //push argc  \*esp-=4;  \*\*(uint32\_t\*\*)esp = argc;  //return address  \*esp -=4;  \*\*(uint32\_t\*\*)esp = 0;  // hex\_dump(\*esp, \*esp, 1000, 1);  free(temp);  free(argv);  } |

**int count\_token(char\* filename)**

입력받은 argument의 token 개수를 세어주는 함수이다. 기존의 FILENAME을 훼손하지 않기 위해 strlcpy 함수를 사용해 임시 배열에 저장한다. 이후, strtok\_r 함수를 사용해 token의 개수를 센다.

|  |
| --- |
| int  count\_token(const char\* filename) {  unsigned count = 0;  char\* ptr, \*token;  char temp[256];  strlcpy(temp, filename, strlen(filename)+1);  token = strtok\_r(temp, " ", &ptr);  while(token!=NULL) {  count++;  token = strtok\_r(NULL, " ", &ptr);  };  return count;  } |

**void get\_filename(const char\* src, char\* dest)**

입력받은 argument로부터 filename만을 parsing하기 위한 함수이다. process\_execute에서 thread\_create로 파일 이름을 넘겨줄 때에 사용된다.

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

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

argument를 parsing한 후, validity를 검사한다. 정상적인 파일 이름이라면, file\_name을 이름으로 하는 child thread를 생성하고, 함수 포인터로 start\_process를 넘겨 이 thread에서 start\_process를 실행할 수 있도록 한다. 이후 child thread의 parent\_tid로 현재 실행 중인 thread를 설정해준다.

|  |
| --- |
| tid\_t  process\_execute (const char \*file\_name)  {  char \*fn\_copy;  char cmd[256];  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);  get\_filename(file\_name, cmd);  // check the validity of FILE\_NAME  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);  find\_thread(tid)->parent\_tid = thread\_current()->tid;  return tid;  } |

**static void start\_process(void \*file\_name)**

child thread에서 실행하는 함수이다. FILENAME의 이름을 가진 파일을 load 함수를 통해 로드한다. 이때 load함수가 정상적으로 종료됐다면, stack\_argument 함수를 호출해 argument를 stack에 쌓는다.

|  |
| --- |
| static void  start\_process (void \*file\_name\_)  {  char \*file\_name = file\_name\_;  struct intr\_frame if\_;  bool success;  char cmd[256];  get\_filename(file\_name, cmd);  /\* 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 (cmd, &if\_.eip, &if\_.esp);  if (success) {  stack\_argument(file\_name, &if\_.esp);  }  /\* If load failed, quit. \*/  palloc\_free\_page (file\_name);  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 ();  } |

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

child thread가 종료되길 기다리는 함수이다. synchronization을 위해 semaphore를 이용하는데, sema\_down을 통해 child thread가 아직 수행 중임을 알고 종료를 기다린다. child thread가 어떤 thread인지 알기 위해 인자로 받은 child\_tid와 find\_thread 함수를 통해 찾는다.

또 child thread는 작업을 종료했다면, process\_exit 함수 내에서 sema\_up을 통해 parent thread에게 종료를 알린다.

|  |
| --- |
| int  process\_wait (tid\_t child\_tid)  {  struct thread\* t = find\_thread(child\_tid);  if(t == NULL) return -1;  sema\_down(&(t->sync));  return thread\_current()->child\_status;  } |

**/threads/thread.h**

**struct thread**

syncronization을 위한 몇 가지 변수를 구조체에 추가한다. semaphore를 추가하고, child의 종료 상태를 확인할 수 있는 child\_sync 변수와 parent가 어떤 thread인지 확인할 수 있는 parent\_thread를 추가하였다.

|  |
| --- |
| struct thread  {  ....  #ifdef USERPROG  /\* Owned by userprog/process.c. \*/  uint32\_t \*pagedir; /\* Page directory. \*/  tid\_t parent\_tid;  struct semaphore sync;  int child\_status;  struct list childs;  struct list\_elem child\_elem;  #endif  /\* Owned by thread.c. \*/  unsigned magic; /\* Detects stack overflow. \*/  }; |

**/threads/thread.c**

**struct thread\* find\_thread(tid\_t tid)**

인자로 받은 tid를 활용해 tid에 해당하는 thread를 찾는 함수이다. 현재 존재하는 thread는 자료구조 list인 all\_list에 저장되어 있고, 이 list를 탐색하며 인자로 받은 tid와 같은 thread가 있는지 확인하고 반환한다.

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

1. User Memory Access

**/userprog/exception.c**

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

page fault 예외 처리를 하는 함수에서, 사용자가 정당한 메모리 영역에 접근하고 있는지에 대해 검사한다. 만약 접근이 커널에 의해 행해지거나, 사용자가 커널의 메모리에 접근하고 있다면 오류 처리하며 종료한다.

|  |
| --- |
| static void  page\_fault (struct intr\_frame \*f)  {  ....  ....  /\* Determine cause. \*/  not\_present = (f->error\_code & PF\_P) == 0;  write = (f->error\_code & PF\_W) != 0;  user = (f->error\_code & PF\_U) != 0;  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);  } |

**/userprog/syscall.h**

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

요청된 시스템 콜을 syscall-nr.h 에 나열되어 있는 번호에 따라 정해진 서비스 루틴을 수행하는 함수이다. 각 case에 해당 메모리 영역으로의 접근이 정당한지에 대한 검사가 이루어진다.

|  |
| --- |
| static void  syscall\_handler (struct intr\_frame \*f)  {  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(\*(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)) 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)) 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\_FIBO:  if(!is\_user\_vaddr(f->esp+4)) exit(-1);  f->eax = fibonacci((int)\*(uint32\_t\*)(f->esp+4));  break;  case SYS\_MAX:  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;  } |

1. System Calls

**/userprog/syscall.c**

**void halt(void)**

pintos를 종료시키는 시스템 콜이다. devices/shutdown.h 에 선언된 shutdown\_power\_off()함수를 호출하여 수행한다.

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

. **void exit(int status)**

halt가 pintos를 종료한다면 exit는 현재의 thread를 종료시키는 시스템 콜이다. 현재 실행 중인 thread를 current라는 구조체 변수로 선언하고 thread\_current를 통해 초기화한다. parent thread가 child thread의 종료 status를 알기 위해 parent\_tid 변수와 find\_thread 함수로 해당 thread를 찾아준다. 그리고 child\_status를 업데이트한다.

|  |
| --- |
| void exit(int status){  find\_thread(thread\_current()->parent\_tid)->child\_status = status;  printf("%s: exit(%d)\n", current->name, status);  thread\_exit();  } |

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

인자로 받은 cmd\_line을 file\_name으로 하는 프로세스를 process\_execute를 통해 실행한다.

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

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

process\_wait를 호출하여 인자로 받은 pid의 thread가 끝날 때까지 기다린다. 해당 thread가 종료되면 process\_wait로부터 status를 받는다.

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

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

fd가 0인 stdin으로부터 size만큼 읽어 buffer에 저장한다. devices/input.h에 선언된 input\_getc()를 통해 한글자씩 읽어온다.

|  |
| --- |
| int read(int fd, void\* buffer, unsigned size){  if(fd == 0) {  //call input\_getc()  //prototype : uint8\_t input\_getc(void)  //return a char from keyboards  unsigned i;  for(i=0;i<size;i++){  \*(uint8\_t\*)buffer = input\_getc();  if(((char\*)buffer)[i] == '\0') break;  }  return i;  }  return -1;  } |

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

fd가 1인 stdout에 size만큼 buffer에 있는 내용을 출력한다. devices/input.c 에 선언된 putbuf를 호출해 출력한다. 출력한 내용의 size를 반환한다.

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

1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**

**int fibonacci(int n)**

피보나치 수열에서 n 번째에 있는 수를 출력한다. 입력된 n만큼의 배열을 할당 받고 반복문을 통해 해당 배열을 피보나치 수열로 초기화했다. 그 후 n-1번째 인덱스에 있는 값을 반환했다.

|  |
| --- |
| int fibonacci(int n){  int i;  int\* seq = (int\*)malloc(sizeof(int)\*n);  seq[0] = 1;  seq[1] = 1;  for(i=2; i<=n; i++){  seq[i] = seq[i-1]+seq[i-2];  }  i = seq[n-1];  free(seq);  return i;  } |

**int max\_of\_four\_int(int n0, int n1, int n2, int n3)**

입력 받은 4개의 변수 중 최대값을 출력한다.

|  |
| --- |
| int max\_of\_four\_int(int n0, int n1, int n2, int n3){  int max = n0;  int arr[4] = {n0,n1,n2,n3};  int i;  for(i=1; i<4; i++){  if(max <= arr[i]) max = arr[i];  }  return max;  } |

**/examples/additional.c**

Fibonacci와 max\_of\_four\_int의 시스템 콜을 호출하는 user program을 작성했다. 인자의 개수를 먼저 타당한지 검사한다. 정상적이라면 filename까지 포함해 5개의 인자가 입력되어야 한다.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <syscall.h>  int main(int argc, char\* argv[]){  int n0,n1,n2,n3, ret\_fibo, ret\_max;  if(argc != 5) {  printf("usage: additional [num1] [num2] [num3] [num4]\n");  exit(-1);  }  n0 = atoi(argv[1]); n1 = atoi(argv[2]);  n2 = atoi(argv[3]); n3 = atoi(argv[4]);  ret\_fibo = fibonacci(n0);  ret\_max = max\_of\_four\_int(n0, n1, n2, n3);  printf("%d %d\n", ret\_fibo, ret\_max);  return 0;  } |

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

syscall\_handler에서 Fibonacci 와 max\_of\_four\_int의 시스템 콜 번호를 구분할 수 있도록 enum에 추가헀다.

|  |
| --- |
| 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. \*/  SYS\_FIBO, /\* Fibonacci Sequence \*/  SYS\_MAX, /\* Max of Four int \*/    ………  };  #endif /\* lib/syscall-nr.h \*/ |

**/lib/user/syscall.c**

기존에는 인자를 3개까지 받는 매크로밖에 없었고, max\_of\_four\_int는 4개의 인수를 받기 때문에 새로운 매크로를 정의해주었다.

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

**/examples/Makefile**

새로 만든 additional.c 파일이 같이 make 되도록 makefile을 수정하였다.

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

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

