Pintos Project 1: User Program (1)

담당 교수 : 김영재 교수

조 / 조원 : 20211589 정서영

개발 기간 : 25.09.11 ~ 25.10.08

1. 개발 목표

본 프로젝트는 Pintos 운영체제에서 User Program을 정상적으로 실행할 수 있도록 만드는 것을 목표로 한다. 현재는 시스템 콜과 시스템 콜 핸들러, 인자 전달 및 사용자 스택 구성, 메모리 접근 보호와 같은 핵심 기능들이 구현되어 있지 않아서 User Program이 제대로 동작하지 않는다. 따라서 이번 프로젝트에서는 User Program이 커널의 기능을 안전하게 요청하고 실행될 수 있도록 하는 데 필요한 OS 기능들을 구현한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Argument Passing

   User Program이 실행될 때 여러 개의 인자를 받아서 사용할 수 있다. 예를 들어 'echo x'를 실행할 때 프로그램 이름인 'echo'와 인자인 'x'가 분리되어 파싱되고, 이것들이 80x86 호출 규약에 맞춰서 사용자 스택에 적절한 형태로 배치된다. 이를 통해 프로그램 내부에서 전달받은 인자들을 참조할 수 있게 되어서, 프로그램이 인자를 활용한 동작을 수행할 수 있다.

2. User Memory Access

   User Program이 잘못된 포인터를 사용해서 시스템에 피해를 주는 것을 막을 수 있다. User Program이 NULL 포인터를 역참조하거나, 매핑되지 않은 가상 메모리 영역에 접근하거나, 커널 주소 공간을 가리키는 포인터를 사용하려고 하면 이를 감지해서 해당 프로세스를 안전하게 종료시킨다. 이를 통해 잘못된 메모리 접근으로 인해 커널이나 다른 프로세스가 손상되는 것을 방지할 수 있다.

3. System Calls

   User Program이 커널이 제공하는 기능을 사용할 수 있다. 이번 프로젝트에서는 halt(), exit(), exec(), wait(), read(), write() 시스템 콜을 구현한다. 파일 시스템 관련 기능은 이번 구현 범위에 포함하지 않고, read(), write()는 표준 입출력 동작만 구현한다. 이 시스템 콜들이 구현되면 User Program이 시스템 콜을 호출했을 때 커널이 요청받은 작업을 처리하고 결과를 User Program에 반환할 수 있다. 또한 프로세스가 종료될 때 "Process Name: exit(exit status)\n" 형식으로 종료 메시지를 출력하여 프로세스의 실행 결과를 확인할 수 있다. 추가적으로 fibonacci()와 max_of_four_int()라는 새로운 시스템 콜을 구현해서 피보나치 수열 계산 기능과 네 정수 중 최댓값을 찾는 기능을 제공한다.

B. 개발 내용

1. Argument Passing

User Program이 실행될 때 전달된 인자들을 사용자 스택에 배치하는 과정을 구현한다. 먼저 입력된 명령어를 파싱하여 프로그램 이름과 인자들을 분리한 뒤, 할당된 스택 페이지에 80x86 호출 규약에 따라 인자들을 할당한다.

사용자 스택은 높은 주소에서 낮은 주소 방향으로 성장하며, 우선 인자의 실제 문자열 데이터가 역순으로 차례대로 스택에 푸시되고, 이후 워드 정렬을 위한 패딩을 추가한다. 이어서 인자 포인터 배열의 끝을 나타내기 위한 NULL 포인터 센티널을 추가한 뒤, 각 인자의 주소를 역순으로 푸시한다. 마지막으로 인자 포인터 배열의 시작 주소, 인자 개수, 가짜 반환 주소를 차례대로 푸시한다.

가짜 반환 주소의 경우, 프로그램을 처음 실행할 때 실제 돌아갈 호출자가 없기 때문에 스택 구조를 맞추기 위해 넣는 임의의 값으로, 프로그램 종료 시 커널이 이 가짜 주소로 복귀하지 않고 종료 처리를 수행한다.

2. User Memory Access

Pintos 상에서의 invalid memory access는 User Program이 접근해서는 안 되는 메모리 영역에 접근하려는 시도를 의미한다. 구체적으로는 NULL 포인터 역참조, 매핑되지 않은 가상 메모리 영역 접근, 커널 주소 공간(PHYS_BASE 이상)에 대한 접근이 해당된다. 이러한 잘못된 메모리 접근을 허용하면 커널이나 다른 프로세스의 메모리가 손상되어 시스템 전체의 안정성이 무너질 수 있다.

이러한 invalid memory access를 막기 위해서는 두 가지 방법을 사용할 수 있다. 첫 번째 방법은 User Program이 제공한 포인터의 유효성을 사전 검증한 뒤 역참조하는 것이다. 두 번째 방법은 사후에 잘못된 접근으로 인해 page fault가 발생하면 예외를 처리하고 프로세스를 종료시키는 것이다.

3. System Calls

User Program은 사용자 모드에서 실행되기 때문에 메모리나 디스크 같은 하드웨어 자원에 직접 접근할 수 없다. 이러한 작업들은 커널 모드에서만 수행될 수 있으므로, User Program이 커널의 기능을 사용하려면 시스템 콜이라는 인터페이스를 통해 커널에게 작업을 요청해야 한다. 시스템 콜은 사용자 모드와 커널 모드의 권한을 분리하고 안전한 전환을 제공해주는 역할을 한다.

이번 프로젝트에서 개발할 시스템 콜은 다음과 같다.

- halt(): shutdown_power_off() 함수를 호출하여 Pintos를 종료시킨다.
- exit(): 현재 실행 중인 User Program을 종료하고 종료 상태를 커널에 반환한다.
- exec(): 새로운 자식 프로세스를 생성하여 지정된 프로그램을 실행시킨다.
- wait(): 자식 프로세스가 종료될 때까지 부모 프로세스를 대기시키고 자식의 종료 상태를 받아온다.
- read(): 표준 입력으로부터 데이터를 읽어오고, 읽은 byte 수를 반환한다.
- write(): 표준 출력으로 데이터를 쓰고, 쓴 byte 수를 반환한다.
- fibonacci(): 전달받은 정수 n에 대해 n번째 피보나치 수열 값을 계산하여 반환한다.
- max_of_four_int(): 네 개의 정수를 인자로 받아서 그중 가장 큰 값을 찾아 반환한다.

유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지의 요소는 다음과 같다.

(1) 먼저, User Program이 lib/user/syscall.c에 정의된 시스템 콜 API를 호출한다.
(2) 호출된 시스템 콜 API는 시스템 콜 번호와 인자들을 호출자 스택에 푸시한 후, 'int $0x30' 명령어로 인터럽트를 발생시켜 커널 모드로 진입한다.
(3) threads/intr-stubs.S의 인터럽트 스텁이 실행되어 인터럽트를 위한 스택을 설정하고, threads/interrupt.c의 intr_handler() 함수를 호출한다.
(4) intr_handler()는 등록된 시스템 콜 핸들러인 userprog/syscall.c의 syscall_handler()를 호출한다.
(5) syscall_handler()는 intr_frame 구조체의 esp 멤버를 통해 사용자 스택에 접근하여 시스템 콜 번호를 읽고, 이에 따라 해당하는 시스템 콜을 실행한다.
(6) 시스템 콜의 반환 값은 intr_frame 구조체의 eax 멤버에 저장되며, 인터럽트 핸들러가 종료되면 사용자 모드로 복귀한다.

3. 추진 일정 및 개발 방법

A. 추진 일정

기간	추진 내용
9/11 ~ 9/15	Pintos 소스 코드 분석 및 명세서 검토

9/16 ~ 9/22	Argument Passing 구현 및 테스트
9/23 ~ 9/26	User Memory Access 구현 및 System Call Handler 기본 구조 작성
9/27 ~ 9/30	주요 System Calls(halt, exit, exec, wait, read, write) 구현 및 테스트
10/1 ~ 10/4	Additional System Calls(fibonacci, max_of_four_int) 구현 및 테스트
10/5 ~ 10/8	전체 테스트 검증 및 문서 작성

B. 개발 방법

1. Argument Passing

   userprog/process.c의 load() 함수에서 먼저 파일명을 파싱하는 부분을 추가한다. 현재는 전체 명령줄 문자열이 그대로 전달되는데, strtok_r() 함수를 사용하여 공백을 기준으로 프로그램 이름과 인자들을 분리한다. 분리된 프로그램 이름만을 filesys_open() 함수에 전달하여 실행 파일을 열고, 인자들은 argv 배열에, 인자 개수는 argc에 저장한다.

   load() 함수에서 setup_stack() 함수가 호출된 이후 부분에 인자들을 사용자 스택에 배치하는 내용을 추가한다. 먼저 palloc_get_page()로 할당받은 스택 페이지의 최상단(PHYS_BASE)부터 시작하여 각 인자 문자열을 역순으로 복사한다. 스택 포인터를 문자열 길이만큼 감소시키면서 memcpy()로 문자열을 복사하고, 각 문자열의 시작 주소를 별도로 저장해둔다. 모든 문자열을 복사한 후 스택 포인터를 4바이트 단위로 정렬하기 위해 필요한 만큼 패딩을 추가한다. 그 다음 NULL 포인터 센티널을 푸시하고, 저장해둔 문자열 주소들을 역순으로 푸시한다. 마지막으로 argv의 주소, argc 값, 가짜 반환 주소 0을 순서대로 푸시하여 80x86 호출 규약을 완성한다. esp에는 최종 스택 포인터가 저장될 수 있도록 한다.

2. User Memory Access

   userprog/syscall.c에 포인터의 유효성을 검사하는 새로운 함수 check_valid_uaddr()를 작성한다. 이 함수는 순차적으로 주어진 포인터가 NULL이 아닌지 확인한 후, threads/vaddr.h의 is_user_vaddr() 함수로 포인터가 사용자 주소 공간에 있는지 확인하고, userprog/pagedir.c의 pagedir_get_page() 함수로 실제 물리 페이지에 매핑되어 있는지 검사한다. 검사에 실패하면 exit(-1)을 호출하여 프로세스를 종료시킨다.

   userprog/syscall.c의 syscall_handler()에서는 시스템 콜 번호와 모든 인자에 접근하기 전에 앞서 만든 check_valid_uaddr() 함수를 호출한다. 이를 통해 사용자 스택 포인터, 각 인자의 스택 위치,

포인터 인자가 가리키는 주소, 버퍼의 시작과 끝 주소 등 User Program이 전달한 모든 포인터에 대해 검증을 수행한다.

추가로 userprog/exception.c의 page_fault() 함수를 수정한다. page fault가 발생 시 사용자 모드에서 발생했고, 물리 페이지에 매핑되지 않아 발생한 경우 exit(-1)을 호출하여 프로세스를 종료시킨다. 이를 통해 사전 검증을 통과했더라도 실제 접근 시 발생할 수 있는 예외 상황을 안전하게 처리할 수 있도록 한다.

3. System Calls

먼저 threads/thread.h의 thread 구조체에 프로세스 관리를 위한 멤버 변수들을 추가한다. 부모-자식 관계를 관리하기 위한 parent 포인터, child_list, chil_elem을 추가하고, 동기화를 위한 load_sema와 wait_sema, is_waited를 추가한다. 이후 threads/thread.c의 init_thread()에서 추가한 멤버들을 초기화하고, thread_create()에서 새로 생성된 자식 프로세스를 부모의 child_list에 추가하는 코드를 작성한다. 또한 threads/thread.c에 get_child_thread() 함수를 추가하여 tid로 자식을 찾을 수 있도록 한다.

userprog/syscall.h에 각 시스템 콜 함수의 프로토타입을 선언하고, userprog/syscall.c에 실제 구현을 작성한다. 이후 userprog/syscall.c의 syscall_handler() 함수를 수정하여 intr_frame의 esp에서 시스템 콜 번호를 읽고, switch 문으로 각 시스템 콜을 분기한다. 각 시스템 콜에 필요한 인자들은 4바이트 단위로 읽어온다.

각 시스템 콜 구현은 다음과 같다. halt()는 devices/shutdown.c의 shutdown_power_off()를 호출하여 구현한다. exit()는 현재 스레드의 exit_status를 저장하고 thread_exit()를 호출한다. exec()는 process_execute()를 호출하여 자식 프로세스를 생성하고 tid를 반환한다. wait()는 process_wait()를 호출하여 자식 프로세스가 종료될 때까지 대기한다. read()는 fd가 0일 때 devices/input.c의 input_getc()를 반복 호출하여 구현한다. write()는 fd가 1일 때 lib/kernel/console.c의 putbuf()를 호출하여 구현한다. 각 시스템 콜의 반환값은 intr_frame 구조체의 eax에 저장한다.

프로세스 관리를 위해 userprog/process.c의 함수를 수정한다. 먼저, process_execute() 함수를 수정하여 부모-자식 프로세스 간 동기화를 구현한다. thread_create()로 자식 프로세스를 생성한 후, get_child_thread()로 생성된 자식을 찾는다. 자식의 load_sema를 활용하여 부모가 자식의 실행 파일 로드 완료를 기다리도록 구현한다. 자식의 start_process() 함수에서는 load() 함수 실행 후 부모에게 신호를 보내도록 수정한다. 로드 성공 여부는 자식의 exit_status를 통해 확인하며, 실패 시 TID_ERROR를, 성공 시 자식의 tid를 반환한다. 다음으로 process_waite() 함수를

수정하여 자식 프로세스의 종료 대기 기능을 구현한다. get_child_thread()로 대상 자식을 찾고 is_waited 플래그로 중복 wait를 방지한다. wait_sema를 사용하여 자식이 종료될 때까지 대기하고, 종료 후 exit_status를 읽어 반환한다. 읽기가 완료되면 child_list에서 자식을 제거하고 메모리를 해제한다. 마지막으로 process_exit()를 수정하여 프로세스 종료 메시지를 출력하도록 하고, 부모가 존재하면 wait_sema로 신호를 보낸 뒤, 모든 자식의 parent 포인터를 NULL로 설정하여 고아 프로세스로 만든다.

추가 시스템 콜 구현을 위해선 먼저, lib/syscall-nr.h에 새로운 시스템 콜인 SYS_FIBONACCI와 SYS_MAX_OF_FOUR_INT의 시스템 콜 번호를 추가한다. lib/user/syscall.h에 fibonacci()와 max_of_four_int() 함수의 프로토타입을 선언하고, lib/user/syscall.c에 이들의 사용자 레벨 API를 구현한다. max_of_four_int()는 인자가 4개이므로 syscall4() 매크로를 새로 정의하여 사용한다.

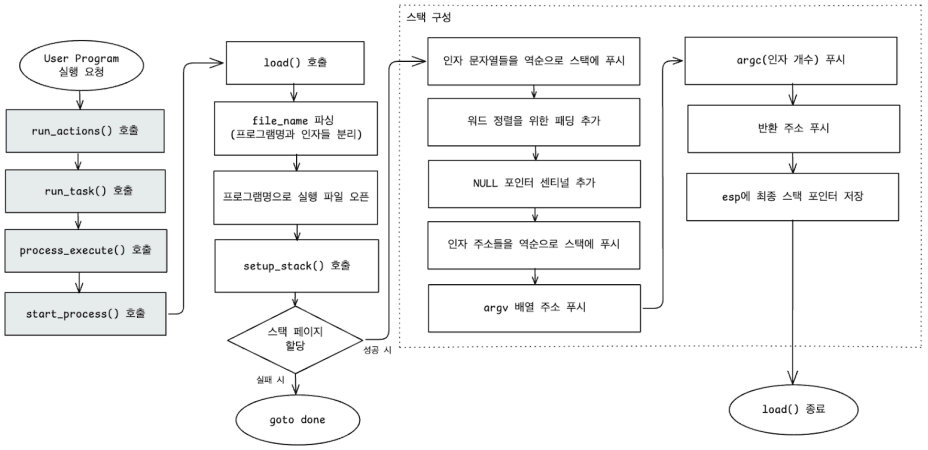
이후 userprog/syscall.h에서 두 함수의 프로토타입을 선언하고, userprog/syscall.c에 해당 구현을 추가한다. fibonacci()는 반복문으로 n번째 피보나치 수를 계산하고, max_of_four_int()는 네 인자를 비교하여 최댓값을 반환한다. syscall_handler()의 switch문에 두 케이스를 추가하고 반환값을 eax에 저장한다.

src/examples 디렉토리에 additional.c 파일을 생성하여 fibonacci()와 max_of_four_int() 시스템 콜을 테스트하는 프로그램을 작성하고, src/examples/Makefile을 수정하여 컴파일되도록 한다.

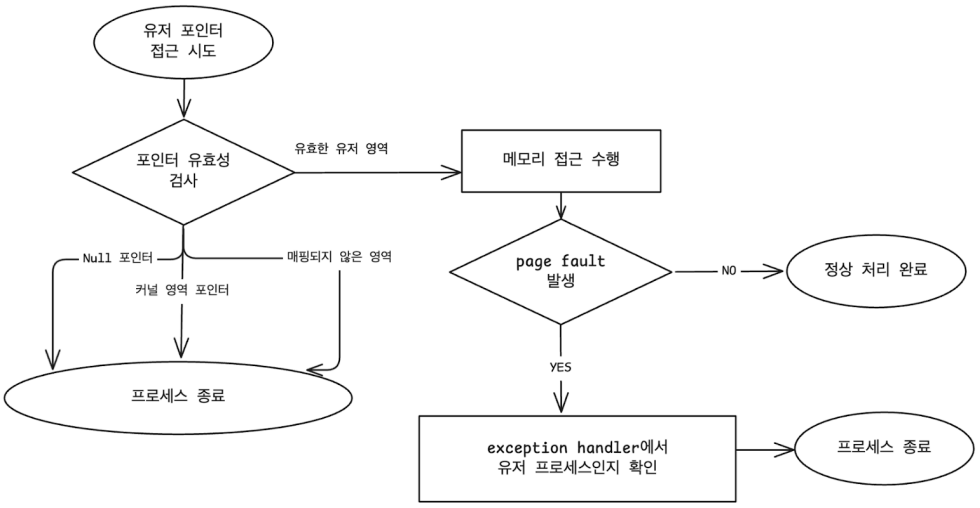
4. 연구 결과

A. Flow Chart

1. Argument Passing

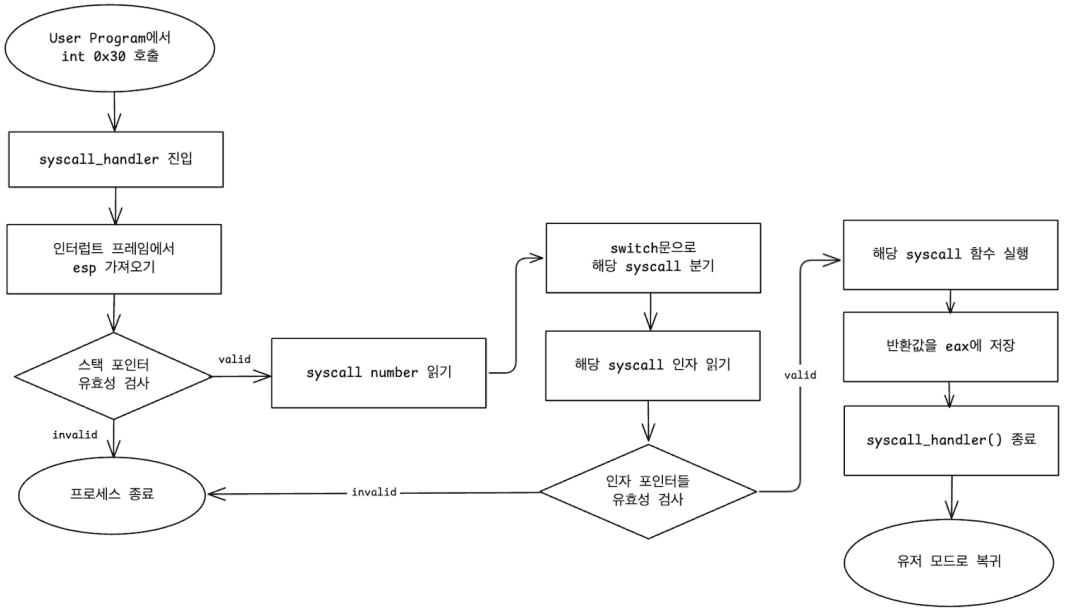


2. User Memory Access

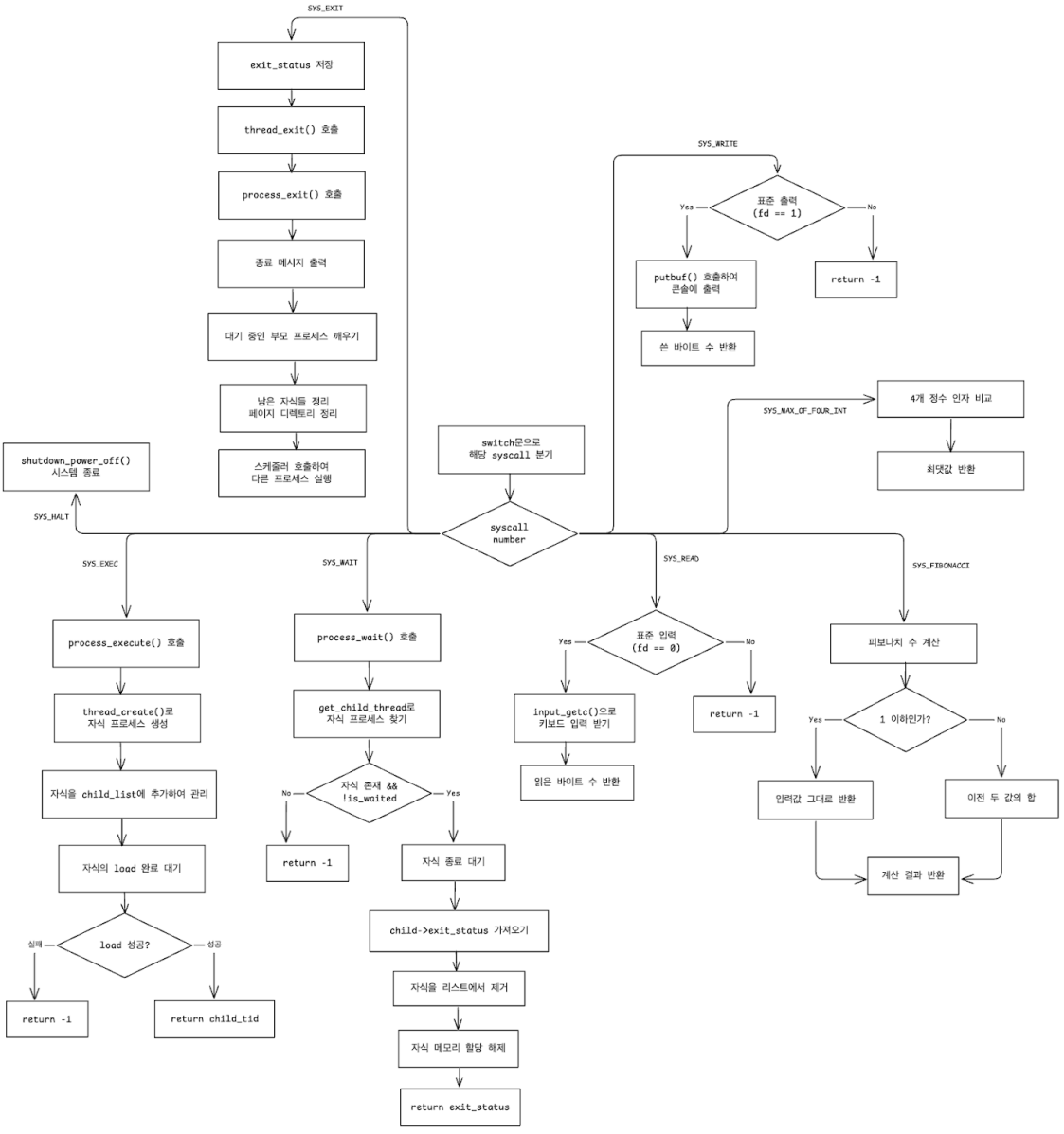


3. System Calls

다음은 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지의 플로우이다.



다음은 각 시스템 콜 함수의 실행 과정에 대한 구체적인 플로우를 나타낸 것이다.



B. 제작 내용

1. Argument Passing

먼저 userprog/process.c의 load() 함수에서 전달받은 file_name 문자열을 파싱하는 부분을 추가하였다. lib/string.c의 strtok_r()은 원본 문자열을 수정하므로 file_name의 복사본 fn_copy를 만들어 사용했고, 공백을 구분자로 프로그램 이름과 인자들을 분리하였다. argv 배열은 일반적인 프로그램의 인자 개수를 고려하여 충분한 크기인 128로 설정했고, argc에는 실제 인자

개수를 저장했다. 파싱된 토큰들을 argv 배열에 저장하고, 첫 번째 토큰(argv[0])인 프로그램 이름만을 filesys_open() 함수에 전달하여 실행 파일을 열도록 하였다.

```
char *fn_copy;
fn_copy = palloc_get_page (0);
if (fn_copy == NULL)
  goto done;
strlcpy(fn_copy, file_name, PGSIZE);

char *token, *save_ptr;
char *argv[128];
int argc = 0;

for (token = strtok_r(fn_copy, " ", &save_ptr); token != NULL;
     token = strtok_r(NULL, " ", &save_ptr)) {
  argv[argc++] = token;
}

char *program_name = argv[0];

/* Open executable file. */
file = filesys_open (program_name);
if (file == NULL)
```

setup_stack() 함수에서 스택 페이지가 할당되므로, setup_stack() 함수 호출 이후 부분에서 80x86 호출 규약에 따라 사용자 스택을 구성하였다. 먼저 스택 포인터 esp는 PHYS_BASE로부터 시작하여, 각 인자 문자열들을 역순으로 스택에 푸시하였다. 이때 문자열 길이에 널 종료 문자를 더한 만큼 스택 포인터를 내리고, 스택에 lib/string.h의 memcpy() 함수를 사용해 문자열을 지정된 바이트 수만큼 그대로 복사한 후, argv 배열에 각 인자들의 주소를 저장하도록 했다.

```
// 인자들을 역순으로 스택에 푸시
for (int i = argc - 1; i >= 0; --i) {
  int len = strlen(argv[i]) + 1;
  *esp -= len; // 문자열 길이 + 널 종료 문자(1)만큼 스택 포인터를 내림
  memcpy(*esp, argv[i], len); // 스택에 문자열을 복사
  argv[i] = *esp; // argv 배열 포인터를 스택에 복사된 문자열의 주소로 업데이트
}
```

모든 문자열을 복사한 후 워드 정렬을 수행하였다. 현재 스택 포인터 주소를 uintptr_t로 변환하고 비트 AND 연산(& 0xfffffffc)을 사용하여 주소의 하위 2비트를 0으로 만들었다. 이를 통해 스택 포인터가 가장 가까운 4의 배수 주소로 정렬될 수 있도록 했다.

```
// 스택 포인터를 4바이트(워드) 경계에 맞게 정렬
uintptr_t esp_uint = (uintptr_t) *esp;
esp_uint &= 0xfffffffc; // 4바이트 정렬
*esp = (void *) esp_uint;
```

그 다음 NULL 포인터 센티널, 인자 주소들, argv 배열의 시작 주소, argc, 가짜 반환 주소를 순서대로 푸시하였다. 포인터나 정수를 푸시할 때는 스택 포인터를 4바이트씩 감소시키고 해당 위치에 값을 저장하는 방식으로 구현하였다.

```
// argv 배열의 마지막 요소(NULL 포인터 센티널)을 스택에 푸시
*esp -= sizeof (char *);
*(char ***) *esp = NULL;

// 인자 주소들을 역순으로 스택에 푸시
for (int i = argc - 1; i >= 0; --i) {
  *esp -= sizeof (char *);
  *(char **) *esp = argv[i];
}

// argv 배열의 시작 주소를 푸시
char **argv_base = (char **) *esp;
*esp -= sizeof (char **);
*(char ***) *esp = argv_base;

// argc(인자 개수) 푸시
*esp -= sizeof (int);
*(int *) *esp = argc;

// 가짜 반환 주소 푸시
*esp -= sizeof (void *);
*(void **) *esp = 0;
```

다음은 Argument Passing의 구현을 확인하기 위해 임시로 process_wait()에 무한 루프를 넣고 hex_dump()를 활용해 ‘echo x’를 명령어로 전달했을 때의 스택 덤프를 출력해본 결과이다.

```
Boot complete.
Extracting ustar archive from scratch device into file system...
Putting 'echo' into the file system...
Erasing ustar archive...
Executing 'echo x':
--- Stack dump after Argument Passing ---
bffffe0  00 00 00 00 02 00 00 00-ec ff ff bf f9 ff ff bf |................|
bfffff0  fe ff ff bf 00 00 00 00-00 65 63 68 6f 00 78 00 |.........echo.x.|
----------------------------------------
system call!
```

결과를 분석해보면 다음과 같고, 의도한 대로 스택이 구성되었음을 확인할 수 있었다.

시작 주소	바이트 데이터				값	내용
0xbffffe0	00	00	00	00	0x00000000	가짜 반환 주소
0xbffffe4	02	00	00	00	0x00000002	argc: 2
0xbffffe8	ec	ff	ff	bf	0xbfffffec	argv 시작 주소

0xbfffffec	f9	ff	ff	bf		0xbffffff9	argv[0] 주소
0xbffffff0	fe	ff	ff	bf		0xbfffffffe	argv[1] 주소
0xbffffff4	00	00	00	00		0x00000000	NULL 포인터
0xbffffff8	00					0x00	word-align 패딩
0xbffffff9	65	63	68	6f	00	“echo\0”	argv[0] 문자열
0xbfffffffe	78	00				“x\0”	argv[1] 문자열

초기 구현에선 문자열 복사 시 널 종료 문자를 포함하지 않아 hex_dump()로 스택을 확인했을 때 문자열이 제대로 구분되지 않는 문제가 발생하였다. 이는 strlen()의 결과에 1을 더하여 널 종료 문자까지 복사하도록 수정하여 해결했다.

```
Executing 'echo x':
Initial esp: 0xc0000000
Final esp: 0xbffffe0
--- Stack dump after Argument Passing ---
bffffe0  00 00 00 00 02 00 00 00-ec ff ff bf fb ff ff bf |................|
bfffff0  ff ff ff bf 00 00 00 00-00 00 00 65 63 68 6f 78 |...........echox|
----------------------------------------
Page fault at 0xc0000000: rights violation error reading page in user context.
echo x: dying due to interrupt 0x0e (#PF Page-Fault Exception).
Interrupt 0x0e (#PF Page-Fault Exception) at eip=0x8049607
```

또한 thread_create()에 전체 명령줄 문자열(예: “echo x”)을 전달하면 스레드 이름이 “echo x”로 설정되어 프로세스 종료 시 “echo: exit(0)”으로 출력되어야 하나, “echo x: exit(0)” 형태로 출력되는 문제가 있었다. 이를 해결하기 위해 userprog/process.c의 process_execute() 함수에서 명령줄 문자열을 파싱하여 프로그램 이름만 추출한 후 thread_create() 함수에 전달하도록 수정했다. 이는 strlcpy() 함수로 명령줄 문자열을 복사해온 뒤 strtok_r() 함수를 사용하여 공백을 기준으로 첫 번째 토큰만 분리해서 사용했다.

```
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 program_name[128];
  char *save_ptr;
  strlcpy(program_name, file_name, 128);
  strtok_r(program_name, " ", &save_ptr);

  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create (program_name, PRI_DEFAULT, start_process, fn_copy);
  if (tid == TID_ERROR) {
    palloc_free_page (fn_copy);
    return TID_ERROR;
  }
```

2. User Memory Access

userprog/syscall.c에 check_valid_uaddr() 함수를 작성하여 포인터를 역참조하기 전에 유효성을 사전검증하도록 했다. 이 함수는 먼저 포인터가 NULL을 가리키는지 확인하고, threads/vaddr.h의 is_user_vaddr() 함수로 주소가 사용자 영역인지 확인하고, userprog/pagedir.c의 pagedir_get_page() 함수로 해당 가상 주소가 실제 물리 페이지에 매핑되어 있는지 검사한다. 유효하지 않으면 exit(-1)을 호출해 프로세스를 종료하도록 했다.

```
void
check_valid_uaddr (const void *uaddr)
{
  if (uaddr == NULL ||
      !is_user_vaddr (uaddr) ||
      pagedir_get_page (thread_current ()->pagedir, uaddr) == NULL)
  {
    exit (-1);
  }
}
```

syscall_handler() 함수에서는 시스템 콜 처리 전에 모든 포인터에 대해 check_valid_uaddr()를 호출하여 검증을 수행하도록 했다. 먼저 intr_frame 구조체의 esp를 검사했는데, 이는 시스템 콜 번호가 저장된 스택의 최상단 주소를 가리킨다.

```
static void
syscall_handler (struct intr_frame *f UNUSED)
{
  check_valid_uaddr (f->esp);

  uint32_t *esp = (uint32_t *) f->esp;
  uint32_t syscall_num = *esp;
```

이때 intr_frame 구조체의 esp는 void* 타입이지만, 시스템 콜 번호와 인자들은 4바이트(32비트) 정수로 스택에 저장된다. esp를 uint32_t* 타입으로 캐스팅하면 포인터 연산 시 자동으로 4바이트 단위로 이동하게 된다. 따라서 esp+1은 실제로 esp의 주소에서 4바이트 떨어진 위치를 가리키며, 이는 첫 번째 인자의 위치와 동일하다. 그래서 편의상 esp를 uint32_t*로 캐스팅해서 사용했다.

```
  /* Pushed by the CPU.
     These are the interrupted task's saved registers. */
  void (*eip) (void);         /* Next instruction to execute. */
  uint16_t cs, :16;           /* Code segment for eip. */
  uint32_t eflags;            /* Saved CPU flags. */
  void *esp;                  /* Saved stack pointer. */
  uint16_t ss, :16;           /* Data segment for esp. */
};
```

```
void halt (void);
void exit (int status);
tid_t exec (const char *cmd_line);
int wait (tid_t pid);
int read (int fd, void *buffer, unsigned size);
int write (int fd, const void *buffer, unsigned size);
```

다음으로 각 시스템 콜별로 인자 개수에 따라 검증을 수행했다. halt()는 인자가 없어 검사할 내용이 없었고, exit()와 wait()는 인자가 1개이므로 esp+1을 검증했다. exec()는 인자가 1개이지만 문자열 포인터이므로 check_valid_uaddr(esp + 1)로 스택 주소 자체의 유효성을 검사하고, check_valid_uaddr((void *) *(esp + 1))로 스택에서 읽은 포인터가 가리키는 문자열의 주소를 검증하도록 했다. 이는 포인터를 담고 있는 스택 위치와 그 포인터가 가리키는 실제 데이터 위치를 모두 검증하기 위함이다.

```
static void
syscall_handler (struct intr_frame *f UNUSED)
{
  check_valid_uaddr (f->esp);

  uint32_t *esp = (uint32_t *) f->esp;
  uint32_t syscall_num = *esp;

  switch (syscall_num) {
    case SYS_HALT:
      break;

    case SYS_EXIT:
      check_valid_uaddr (esp + 1);
      break;

    case SYS_EXEC:
      check_valid_uaddr (esp + 1);
      check_valid_uaddr ((void *) *(esp + 1));
      break;

    case SYS_WAIT:
      check_valid_uaddr (esp + 1);
      break;
```

read()와 write()는 인자가 3개이므로 esp+1, esp+2, esp+3를 먼저 각각 검증했다. 그리고 버퍼 포인터는 시작 주소와 끝 주소(buffer + size - 1)를 모두 검증하여 버퍼 전체가 유효한 메모리 영역에 있는지 확인했다. Pintos는 페이지(4KB) 단위로 메모리를 관리하므로, 버퍼의 시작 주소와 끝 주소만 검사해도 해당 범위의 페이지들이 모두 유효함을 보장할 수 있다.

```
case SYS_READ:
  check_valid_uaddr (esp + 1);
  check_valid_uaddr (esp + 2);
  check_valid_uaddr (esp + 3);
  {
    int fd = *(esp + 1);
    void *buffer = (void *) *(esp + 2);
    unsigned size = *(esp + 3);

    check_valid_uaddr (buffer);
    if (size > 0) check_valid_uaddr (buffer + size - 1);
  }
  break;

case SYS_WRITE:
  check_valid_uaddr (esp + 1);
  check_valid_uaddr (esp + 2);
  check_valid_uaddr (esp + 3);
  {
    int fd = *(esp + 1);
    const void *buffer = (const void *) *(esp + 2);
    unsigned size = *(esp + 3);

    check_valid_uaddr (buffer);
    if (size > 0) check_valid_uaddr (buffer + size - 1);
  }
  break;

default:
  exit (-1);
  break;
```

다음으로 userprog/exception.c의 page_fault()를 수정하여 예외 처리를 보강하였다. 사용자 모드에서 not_present(물리 페이지 부재)인 page_fault가 발생하면 User Program이 매핑되지 않은 메모리에 접근한 것이므로 exit(-1)을 호출하여 프로세스를 종료시켰다.

```
if (user && not_present) {
    exit(-1);
 }
```

3. System Calls

(1) 스레드 구조체 확장, 스레드 초기화 및 관리

먼저 threads/thread.h의 struct thread에 프로세스 관리를 위한 멤버 변수들을 추가했다. exit_status는 프로세스의 종료 상태를 저장하고, parent는 부모 스레드를 가리키는 포인터이다. child_list는 자식 프로세스들을 관리하기 위한 리스트이며, child_elem은 이 스레드가 부모의 child_list에 연결될 때 사용되는 리스트 요소이다. 또한 동기화를 위해 두 개의 세마포어를 추가했다. load_sema는 자식 프로세스의 실행 파일 로드가 완료될 때까지 부모가 대기하도록 하며, wait_sema는 자식 프로세스가 종료될 때까지 부모가 대기하도록 한다. is_waited 플래그는 같은 자식에 대해 중복으로 wait를 호출하는 것을 방지한다.

```
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */

    int exit_status; // 프로세스 종료 상태
    struct thread* parent; // 부모 스레드 포인터
    struct list child_list; // 자식 프로세스 리스트
    struct list_elem child_elem; // 부모의 child_list에 연결될 요소
    struct semaphore load_sema; // 로드 성공 여부 동기화
    struct semaphore wait_sema; // 프로세스 종료 동기화
    bool is_waited; // 이미 wait 되었는지 표시
#endif
```

그리고 threads/thread.c의 init_thread() 함수를 수정하여 추가한 멤버 변수들을 초기화했다. 이후 thread_create() 함수를 수정하여 새로 생성되는 스레드를 부모의 child_list에 추가했다. 이때 parent 포인터를 현재 실행 중인 스레드로 설정했고, 새로운 스레드의 child_elem을 부모의 child_list에 추가했다.

```
#ifdef USERPROG
  t->parent = NULL;
  list_init(&t->child_list);
  sema_init(&t->load_sema, 0);
  sema_init(&t->wait_sema, 0);
  t->is_waited = false;
#endif
```

```
  /* Initialize thread. */
  init_thread (t, name, priority);
  tid = t->tid = allocate_tid ();

#ifdef USERPROG
  t->parent = thread_current ();
  list_push_back(&thread_current()->child_list, &(t->child_elem));
#endif

  /* Stack frame for kernel_thread(). */
  kf = alloc_frame (t, sizeof *kf);
  kf->eip = NULL;
```

또한 get_child_thread() 함수를 새로 추가하여 주어진 tid를 가진 자식 스레드를 찾을 수 있도록 했다. 이 함수는 현재 스레드의 child_list를 순회하면서 tid가 일치하는 자식을 찾아 반환하고, 찾지 못하면 NULL을 반환한다.

```c
/* 현재 스레드의 자식 리스트에서 주어진 tid를 가진 자식 스레드를 찾아 반환,
찾지 못하면 NULL 반환 */
struct thread*
get_child_thread(tid_t tid)
{
#ifdef USERPROG
  struct thread *cur = thread_current ();
  struct thread *child;
  struct list_elem *e;

  for(e = list_begin(&(cur->child_list)); e != list_end(&(cur->child_list));
      e = list_next(e))
  {
    child = list_entry(e, struct thread, child_elem);
    if(tid == child->tid) return child;
  }
#endif

  return NULL;
}
```

(2) 시스템 콜 구현

```c
void halt (void);
void exit (int status);
tid_t exec (const char *cmd_line);
int wait (tid_t pid);
int read (int fd, void *buffer, unsigned size);
int write (int fd, const void *buffer, unsigned size);
```

userprog/syscall.h에 각 시스템 콜 함수의 프로토타입을 선언하고, userprog/syscall.c에 실제 구현을 작성하였다. 그리고 syscall_handler() 함수를 수정해 switch문으로 분기된 각 시스템 콜에서 유효성 검사 후 해당 시스템 콜 함수를 호출하도록 했다. 각 시스템 콜에 필요한 인자들은 스택에서 4바이트 단위로 읽어오고, 반환값이 있는 시스템 콜 함수의 경우 반환값을 intr_frame의 eax 멤버에 저장하여 User Program에 전달되도록 했다.

```c
static void
syscall_handler (struct intr_frame *f UNUSED)
{
  check_valid_uaddr (f->esp);

  uint32_t *esp = (uint32_t *) f->esp;
  uint32_t syscall_num = *esp;

  switch (syscall_num) {
    case SYS_HALT:
      halt ();
      break;

    case SYS_EXIT:
      check_valid_uaddr (esp + 1);
      exit (*(esp + 1));
      break;

    case SYS_EXEC:
      check_valid_uaddr (esp + 1);
      check_valid_uaddr ((void *) *(esp + 1));
      f->eax = exec ((const char*) *(esp + 1));
      break;

    case SYS_WAIT:
      check_valid_uaddr (esp + 1);
      f->eax = wait(*(esp + 1));
      break;
```

```c
case SYS_READ:
  check_valid_uaddr (esp + 1);
  check_valid_uaddr (esp + 2);
  check_valid_uaddr (esp + 3);
  {
    int fd = *(esp + 1);
    void *buffer = (void *) *(esp + 2);
    unsigned size = *(esp + 3);

    check_valid_uaddr (buffer);
    if (size > 0) check_valid_uaddr (buffer + size - 1);

    f->eax = read (fd, buffer, size);
  }
  break;

case SYS_WRITE:
  check_valid_uaddr (esp + 1);
  check_valid_uaddr (esp + 2);
  check_valid_uaddr (esp + 3);
  {
    int fd = *(esp + 1);
    const void *buffer = (const void *) *(esp + 2);
    unsigned size = *(esp + 3);

    check_valid_uaddr (buffer);
    if (size > 0) check_valid_uaddr (buffer + size - 1);

    f->eax = write (fd, buffer, size);
  }
  break;
```

주요 시스템 콜 구현 내용은 다음과 같다.

halt()는 devices/shutdown.h의 shutdown_power_off() 함수를 호출하여 구현했다. 이 함수는 Pintos를 완전히 종료시킨다.

```
void halt() {
  shutdown_power_off();
}
```

exit()는 현재 스레드의 종료 상태를 앞에서 thread 구조체에 추가한 exit_status 멤버 변수에 저장하고 thread_exit()를 호출하여 구현했다.

```
void exit(int status) {
  thread_current()->exit_status = status;
  thread_exit();
}
```

exec()는 process_execute() 함수를 호출하여 새로운 자식 프로세스를 생성하고 tid를 반환한다.

```
tid_t exec(const char *cmd_line) {
  return process_execute(cmd_line);
}
```

wait()은 process_wait() 함수를 호출하여 구현했다.

```
int wait(tid_t tid) {
  return process_wait(tid);
}
```

read()는 fd가 0이면 devices/input.h의 input_getc() 함수를 호출하여 키보드 입력을 한 바이트씩 읽는다. 이를 요청된 size만큼 반복하여 버퍼에 저장한 후, 읽은 바이트 수를 반환한다.

```
int read (int fd, void *buffer, unsigned size) {
  if (fd == 0) {
    uint8_t *buf = (uint8_t *) buffer;
    for (unsigned i = 0; i < size; i++) {
      buf[i] = input_getc ();
    }
    return size;
  }
  return -1;
}
```

write()는 fd가 1이면 lib/kernel/console.h의 putbuf() 함수를 호출하여 버퍼의 내용을 표준 출력으로 출력한다. putbuf()는 주어진 버퍼와 크기를 받아 한 번에 출력하므로 간단히 호출만 한 후, 쓴 바이트 수를 반환하도록 했다.

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

read()와 write() 두 시스템 콜 모두 파일 시스템 관련 기능은 이번 프로젝트의 범위에 포함되지 않으므로 표준 입출력이 아닌 경우 -1을 반환하도록 했다.

(3) 프로세스 생성 및 로드

userprog/process.c의 process_execute() 함수를 수정하여 부모-자식 간 동기화를 구현했다. process_execute() 함수를 보면 우선 thread_create()를 호출하여 새로운 스레드를 생성하고, get_child_thread()로 방금 생성한 자식 스레드를 찾는다. 그리고 load_sema를 사용해 동기화를 하는데, 먼저 부모는 sema_down()을 호출하여 자식의 실행 파일 로드가 완료될 때까지 대기한다.

```c
/* Create a new thread to execute FILE_NAME. */
tid = thread_create (program_name, PRI_DEFAULT, start_process, fn_copy);
if (tid == TID_ERROR) {
  palloc_free_page (fn_copy);
  return TID_ERROR;
}

struct thread *child = get_child_thread(tid);
if(child == NULL) {
  palloc_free_page(fn_copy);
  return TID_ERROR;
}

// 자식의 load() 완료를 기다림
sema_down(&child->load_sema);

if (child->exit_status == -1) {
  list_remove(&child->child_elem);
  palloc_free_page(child);
  return TID_ERROR;
}

return tid;
}
```

```c
success = load (file_name, &if_.eip, &if_.esp);

/* If load failed, quit. */
palloc_free_page (file_name);
if (!success) {
  cur->exit_status = -1;
  sema_up(&cur->load_sema); // 로드 실패 알림
  thread_exit ();
}

// 로드 성공 시 부모에게 알림
sema_up(&cur->load_sema);
```

자식 스레드의 start_process() 함수에서는 load() 함수로 실행 파일을 로드한 후, sema_up()으로 부모를 깨우도록 수정했다. 부모는 sema_down()에서 깨어난 후 자식의 exit_status를 확인하여 -1이면 로드 실패로 판단하고 TID_ERROR를 반환한다. 로드가 성공했다면 자식의 tid를 반환한다. 이러한 방식으로 부모는 자식의 로드가 완료되고 성공했는지 확인한 후에만 다음 작업을 진행할 수 있다.

(4) 프로세스 대기 및 종료

다음으로 userprog/process.c의 process_wait() 함수를 수정했다. process_wait() 함수는 지정된

자식 프로세스가 종료될 때까지 대기하고 그 종료 상태를 반환한다. 먼저 get_child_thread()로 해당 tid를 가진 자식을 찾는다. 자식을 찾지 못하거나 이미 wait한 자식이라면 -1을 반환한다. is_waited 플래그를 true로 설정하여 같은 자식에 대한 중복 wait를 방지한다. 그 다음 wait_sema에 대해 sema_down()을 호출하여 자식이 종료될 때까지 대기한다.

```
int
process_wait (tid_t child_tid UNUSED)
{
  struct thread *child;
  int exit_status;

  child = get_child_thread(child_tid);
  if(child == NULL || child->is_waited) return -1;

  child->is_waited = true;
  sema_down(&(child->wait_sema)); // 자식이 종료될 때까지 대기
  exit_status = child->exit_status;
  list_remove(&(child->child_elem)); // 자식을 child_list에서 제거
  palloc_free_page(child);

  return exit_status;
}
```

```
/* Free the current process's resources. */
void
process_exit (void)
{
  struct thread *cur = thread_current ();
  uint32_t *pd;

  // exit status 출력
  printf("%s: exit(%d)\n", cur->name, cur->exit_status);

  // 부모가 wait 중이면 깨워줌
  if (cur->parent != NULL) {
    sema_up(&cur->wait_sema);
  }

  // 자식들 처리
  struct list_elem *e = list_begin(&cur->child_list);
  while (e != list_end(&cur->child_list)) {
    struct thread *child = list_entry(e, struct thread, child_elem);
    struct list_elem *next_e = list_next(e);
    list_remove(e);
    child->parent = NULL;
    e = next_e;
  }

  /* Destroy the current process's page directory and switch back
     to the kernel-only page directory. */
  pd = cur->pagedir;
  if (pd != NULL)
    {
```

자식 프로세스의 process_exit() 함수에서 sema_up()이 호출되면 부모가 깨어나도록 수정했다. 부모는 깨어나면 자식의 exit_status를 읽어온 후, child_list에서 자식을 제거하고, 자식 메모리를 해제한다. process_exit() 함수는 프로세스가 종료될 때 호출되는데, 먼저 현재 스레드의 name과 exit_status를 활용해 “Process Name: exit(exit status)\n” 형식으로 종료 메시지를 출력하도록 했다. 그리고 부모가 존재한다면 wait_sema에 대해 sema_up()을 호출하여 부모를 깨운다. 그 다음 자신의 모든 자식 프로세스를 순회하면서 parent 포인터를 NULL로 설정하여 고아 프로세스로 만든다.

(5) 메모리 해제 타이밍 처리

초기 구현에서 프로세스 종료 후 시스템이 응답하지 않고 무한 대기 상태에 빠지는 문제가 발생했다. 이는 자식 프로세스(echo)가 부모(main)보다 먼저 종료했고, threads/thread.c의 thread_schedule_tail() 함수에서 THREAD_DYING 상태의 이전 스레드를 즉시 해제하기 때문에, 부모가 process_wait()에서 자식의 exit_status를 읽으려 할 때 이미 메모리가 해제되어 깨진 데이터에 접근하게 된 것이다.

```
Putting 'echo' into the file system...
Erasing ustar archive...
Executing 'echo x':
[EXEC] thread_create returned tid=3
[GET_CHILD] Searching for tid=3 in main's child_list
[GET_CHILD] child_list empty? 0
[GET_CHILD] Child #0: name=idle, tid=2, status=2
[GET_CHILD] Child #1: name=echo, tid=3, status=1
[GET_CHILD] Found!
[EXEC] Waiting for load_sema...
echo x
echo: exit(0)
[EXIT] Signaling wait_sema for parent
[EXEC] load_sema acquired, success=1
[EXEC] Returning tid=3
[WAIT] main (tid=1) waiting for child_tid=3
[GET_CHILD] Searching for tid=3 in main's child_list
[GET_CHILD] child_list empty? 0
[GET_CHILD] Child #0: name=idle, tid=2, status=2
[GET_CHILD] Child #1: name=0000000000000000000000000000000000000000000000000000000000000000
qemu-system-i386: terminating on signal 2
```

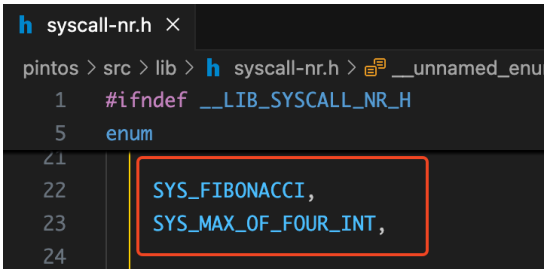
따라서 USERPROG 모드에서는 thread_schedule_tail()에서 THREAD_DYING 상태 스레드를 즉시 해제하지 않도록 했고, 대신 부모가 process_wait()에서 자식의 exit_status를 안전하게 읽은 후에 palloc_free_page()를 호출하여 메모리를 해제했다. 이렇게 하면 부모가 필요한 정보를 모두 읽을 때까지 자식의 메모리가 유지되어 안전한 동기화가 가능하다.

```
  if (prev != NULL && prev->status == THREAD_DYING && prev != initial_thread)
    {
      ASSERT (prev != cur);
#ifdef USERPROG
      // USERPROG에서는 부모가 process_wait()에서 해제
#else
      palloc_free_page (prev);
#endif
    }
}
```

```
int
process_wait (tid_t child_tid UNUSED)
{
  struct thread *child;
  int exit_status;

  child = get_child_thread(child_tid);
  if(child == NULL || child->is_waited) return -1;

  child->is_waited = true;
  sema_down(&(child->wait_sema)); // 자식이 종료될 때까지 대기
  exit_status = child->exit_status;
  list_remove(&(child->child_elem)); // 자식을 child_list에서 제거
  palloc_free_page(child);

  return exit_status;
}
```

4. Additional System calls

먼저, lib/syscall-nr.h에 SYS_FIBONACCI와 SYS_MAX_OF_FOUR_INT를 정의해 시스템 콜 번호를 추가했다.



이후 다음과 같이 lib/user/syscall.h에 함수 프로토타입을 선언하고, lib/user/syscall.c에 사용자 레벨 API를 구현했다.

```c
int fibonacci (int n);
int max_of_four_int (int a, int b, int c, int d);
```

```c
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, d);
}
```

이때 max_of_four_int()는 인자가 4개이므로 syscall4 매크로를 새로 정의하여 사용했다.

```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;                                               \
        })
```

다음으로 userprog/syscall.h에 커널 레벨 함수 프로토타입을 선언하고, userprog/syscall.c에 실제 구현을 추가하였다. fibonacci()는 반복문을 사용하여 n번째 피보나치 수를 계산하도록 구현했고, max_of_four_int()는 단순히 4개의 인자를 비교하여 최댓값을 반환하도록 구현했다.

```
int fibonacci (int n);
int max_of_four_int (int a, int b, int c, int d);

#endif /* userprog/syscall.h */
```

```
int fibonacci (int n) {
  if (n <= 1) return n;
  int prev = 0, cur = 1;
  for(int i = 2; i<= n; i++) {
    int next = prev + cur;
    prev = cur;
    cur = next;
  }
  return cur;
}

int max_of_four_int (int a, int b, int c, int d) {
  int ans = a;
  if(b > ans) ans = b;
  if(c > ans) ans = c;
  if(d > ans) ans = d;
  return ans;
}
```

userprog/syscall.c의 syscall_handler()에 있는 switch문에 앞서 만든 두 가지 케이스에 대한 내용을 추가했다. 각각 인자들을 check_vaild_uaddr()로 유효성을 검사하도록 했고, 반환 값은 eax에 저장하도록 했다.

```
case SYS_FIBONACCI:
  check_valid_uaddr (esp + 1);
  f->eax = fibonacci(*(esp + 1));
  break;

case SYS_MAX_OF_FOUR_INT:
  check_valid_uaddr (esp + 1);
  check_valid_uaddr (esp + 2);
  check_valid_uaddr (esp + 3);
  check_valid_uaddr (esp + 4);
  f->eax = max_of_four_int(*(esp + 1), *(esp + 2), *(esp + 3), *(esp + 4));
  break;
```

그리고 테스트 프로그램인 additional.c를 src/examples 디렉토리에 작성했다. 명령행 인자로 4개의 정수를 받아서 첫 번째 인자로 fibonacci()를 호출하고, 4개의 인자로 max_of_four_int()를 호출하여 결과를 출력하도록 했다.

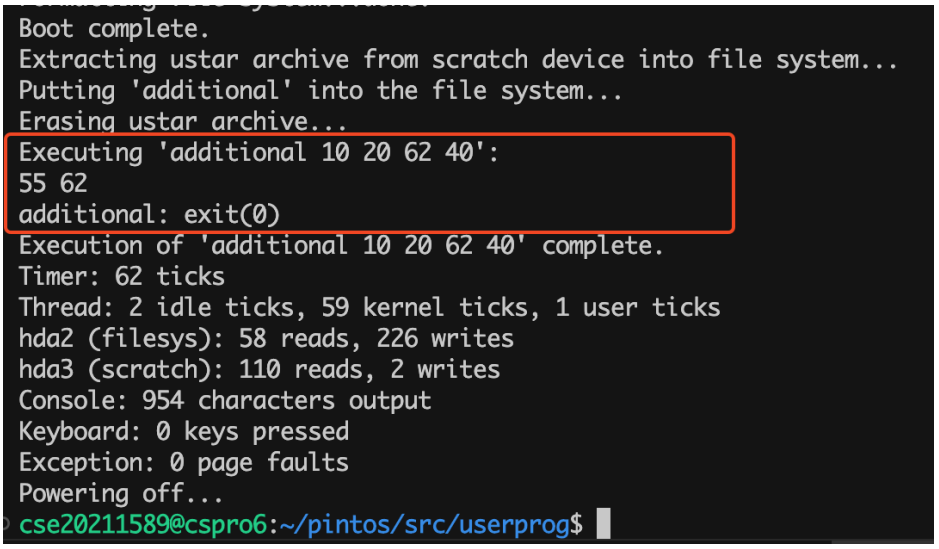
```c
#include <syscall.h>
#include <stdio.h>
#include <stdlib.h>

int
main (int argc, char *argv[])
{
  if (argc != 5) {
    printf("Usage: additional [num1] [num2] [num3] [num4]\n");
    return EXIT_FAILURE;
  }

  int num1 = atoi(argv[1]);
  int num2 = atoi(argv[2]);
  int num3 = atoi(argv[3]);
  int num4 = atoi(argv[4]);

  int fib_val = fibonacci(num1);
  int max_val = max_of_four_int(num1, num2, num3, num4);

  printf("%d %d\n", fib_val, max_val);

  return EXIT_SUCCESS;
}
```

src/examples/Makefile을 수정하여 additional 프로그램이 컴파일되도록 했다. 기존 프로그램들과 동일한 방식으로 PROGS에 추가하고, additional_SRC = additional.c 부분을 추가했다.

```
PROGS = cat cmp cp echo halt hex-dump ls mcat mcp mkdir pwd rm shell \
  bubsort lineup matmult recursor additional

# Should work from project 2 onward.
cat_SRC = cat.c
cmp_SRC = cmp.c
cp_SRC = cp.c
echo_SRC = echo.c
halt_SRC = halt.c
hex-dump_SRC = hex-dump.c
lineup_SRC = lineup.c
ls_SRC = ls.c
recursor_SRC = recursor.c
rm_SRC = rm.c
additional_SRC = additional.c
```

C. 시험 및 평가 내용

다음은 fibonacci()와 max_of_four_int() 시스템 콜 수행 결과를 캡처한 것이다. src/userprog에서 'pintos -filesys-size=2 -p ../examples/additional -a additional - -f -q run 'additional 10 20 62 40' 명령어를 실행시켰을 때의 결과로, 올바르게 출력된 것을 확인할 수 있다.



다음은 src/userprog에서 'make check'를 실행한 결과로, 이번 프로젝트의 평가 기준인 21개의 테스트 케이스에 대해 pass한 것을 확인할 수 있다.