Operating System (Fall 2014) Project 2
User Program and System Call

Jongwook Choi

Introduction

두 번째 pintos 프로젝트의 목표는 user program을 pintos 커널에 로드하고 user program이 올바르게 실행될 수 있도록 각종 system call 을 구현하는 것이다.

- **Argument Passing**: user program을 적재하고 실행할 때, command line argument이 stack에 쌓여 올바르게 전달될 수 있도록 해야 한다.
- **System Call**: user program에서 사용할 수 있는 각종 system call들을 구현해야 한다. 주로 process 관련 system call 및 file 관련 system call들을 포함하는데, 특히 `wait()`, `exec()` 등 UNIX 시멘틱의 process relationship을 처리할 수 있는 구조를 디자인해야 한다.

1. User Program Execution

가장 첫 번째 단계는 user program이 로드될 수 있는 기반 구조를 작성하는 것이다.

Argument Passing

프로세스를 생성 및 시작하는 `process_execute (const char *cmdline)` 함수가 `cmdline`을 받도록 해야 한다. PintOS 에서는 하나의 process는 하나의 thread에 매핑되어 있는데, 따라서 user process를 로드하고 시작하는 thread function인 `thread_create()` 에서 command line `cmdline`를 파싱하여 executable의 filename과 argument를 알아내고, 해당 파일을 load(적재)후 argument를 전달해주면 된다.

Parsing

`strtok_r`를 이용해서 공백을 구분자로 한번만 끊어준 후에, `process_start()`에서 아래와 같이 모든 인자들을 띄어쓰기를 기준으로 끊어 임시로 `cmdline_tokens` 버퍼에 저장한다. 페이지 영역은 4KB이므로 최대 2048개의 인자를 저장할 수 있는데, 특별히 buffer overflow 등을 처리하지는 않았다.

```
char *file_name = (char*) pcb->cmdline;
const char **cmdline_tokens = (const char**) palloc_get_page(0);
/* ... Some code omitted (error handling, etc) ... */

char* token;
char* save_ptr;
int cnt = 0;
for (token = strtok_r(file_name, " ", &save_ptr); token != NULL;
    token = strtok_r(NULL, " ", &save_ptr))
{
  cmdline_tokens[cnt++] = token;
}
```

이후 `load()`를 호출하고, 이것이 성공하면 매개변수들을 (`if`가 지정하는 주소 영역) 스택에 넣는 `push_arguments()` 함수를 호출한다.

```
success = load (file_name, &if_.eip, &if_.esp);
if (success) {
  push_arguments (cmdline_tokens, cnt, &if_.esp);
}
```

Pushing Arguments into Stack

구현은 straightforward 하므로 코드를 모두 첨부하지는 않았다. 자세한 코드는 `process.c:619`를 참고한다. Calling convention에 따라 (`esp`가 높은 순서에서 낮은 순서대로) 다음과 같이 설정하면 된다.

1. 가장 먼저 argument 들의 string pool을 스택에 쌓는다. `memcpy`로 스택 영역에 `cmdline_tokens`의 스트링들을 모조리 복사한다. 이 때 각 토큰별로, 스트링이 시작하는 스택의 시작 주소들을 기억해 둔다.
2. word align: string pool이 끝났으므로, `esp`를 4의 배수로 맞추어 준다.
3. argv의 마지막 (`argv[argc]`)은 0 을 셋팅한다.
4. `argv[argc-1]`, $\cdots$, `argv[0]`를 셋팅한다 — 1.에서 각 token들이 시작하는 스택의 시작 주소를 집어넣는다.
5. `argv`의 주소를 세팅한다.
6. `argc`를 세팅한다.
7. return addr (0)을 세팅한다.

System Call Infrastructure

System call 구현을 위한 infrastructure를 간단하게 설명하면 다음과 같다. User program에서 사용하는 시스템 콜들은 모두 인터럽트로 구현되는데, PintOS 커널에서 system call 인터럽트를 처리하는 핸들러가 바로 `syscall.c: syscall_handler()` 이다. 시스템 콜의 번호나 argument 들은 모두 스택에 쌓여 전달되고, (user program의) 스택을 참조하여 접근할 수 있다.

Calling convention에 따르면 system call 번호는 `esp`가 가리키는 4byte 값이므로, 이를 참조하여 적절한 (kernel 쪽의) system call 함수들이 실행되도록 dispatch를 하는 방식으로 코드가 구성되어 있다. 예를 들면 `syscall_number`가 `SYS_WAIT` 면 `sys_wait(pid)` 등의 핸들러를 호출한다 (trivial한 내용이므로 자세한 설명은 생략하고 `syscall_handler()` 함수를 참고하면 된다). 마찬가지로 각 시스템 콜마다 필요한 argument들이 있다면, user stack (`esp+4`, `esp+8` 주소 등)에 있는 값을 참조하여 얻어올 수 있다.

Accessing User Memory

커널이 유저 메모리에 접근할 때에는, 그 주소가 올바른지를 검사하는 것이 필요하다. 사용한 방법은 system call 등의 루틴에서 유저 메모리에 접근할 때마다, user address가 `PHYS_BASE` 이하인지를 체크하는 것이다.

먼저 user memory에서 1byte를 얻어오는 루틴인 `get_user()`는 다음과 같다. 이 함수는 `uaddr` 주소에서 1byte를 읽어, `int32` 으로 그 값을 리턴한다. 단 잘못된 메모리 영역이면 -1을 리턴해야 한다.

```
static int32_t get_user (const uint8_t *uaddr) {
  // check that a user pointer `uaddr` points below PHYS_BASE
  if (! ((void*)uaddr < PHYS_BASE)) {
    return -1;
  }

  // as suggested in the reference manual, see (3.1.5)
  int result;
  asm ("movl \$1f, \%0; movzbl \%1, \%0; 1:"
      : "=\&a" (result) : "m" (*uaddr));
  return result;
}
```

모든 유저 메모리 접근은 이 함수를 통하기 때문에, 올바른 메모리 영역인지를 체크한다. 할당되지 않은 page에 접근하게 되면 `asm` 라인에서 page fault (interrupt)가 발생하는데, 이 때에는 `get_user()` 함수가 -1를 리턴하도록 해야 한다. 커널 모드에서 발생한 page fault인 경우, `exception.c` 의 `pagefault()` 인터럽트 핸들러에서, eax를 -1로 설정 (리턴값)하고 eip (program counter)를 eax로 설정하여 `get_user()`으로 반환하도록 처리하였다.

```
/* (3.1.5) a page fault in the kernel merely sets eax to 0xffffffff
 * and copies its former value into eip */
if(!user) { // kernel mode
  f->eip = (void *) f->eax;
  f->eax = 0xffffffff;
  return;
}
```

마찬가지로 유저 메모리 영역에 접근하여 쓰는 경우에는, 비슷한 역할을 하는 `put_user()` 를 사용한다. 구조는 거의 같고 구현은 레퍼런스 매뉴얼에 나와 있으므로 생략한다.

2. System Call: Process Management

PintOS는 일반적인 OS와 달리, 쓰레드가 곧 user program 인 간단한 모델을 사용하고 있다. 이 과제의 핵심이 되는 `wait()`, `exit()` 등의 구현을 위해서는, 프로세스를 관리하는 적절한 자료 구조의 설계가 필요하고 parent-child process 관계를 고려하여 `wait()`, `exit()` 의 시멘틱을 잘 설계해야 한다.

Design

먼저 추가/수정된 자료구조는 다음과 같다. 먼저 쓰레드(`struct thread`)는 process와 관련된 정보를 저장할 수 있도록 다음과 같은 필드들이 추가되었다.

```
--- a/src/threads/thread.h
+++ b/src/threads/thread.h
@@ -103,6 +103,13 @@ struct thread
 #ifdef USERPROG
     /* Owned by userprog/process.c. */
     uint32_t *pagedir;                  /* Page directory. */
+
+    // Project 2: file descriptors and process table
+    /* Owned by userprog/process.c and userprog/syscall.c */
+
+    struct process_control_block *pcb;  /* Process Control Block */
+    struct list child_list;             /* List of children processes of this thread,
+                                           each elem is defined by pcb#elem */
 #endif

     /* Owned by thread.c. */
```

- `struct list child_list` : 이 process를 부모 프로세스로 하는 자식 프로세스(정확히는 그들의 `pcb` 객체들)의 목록을 관리하는 linked list.
- `struct process_control_block pcb` : 해당 thread에 연관된 process에 관한 각종 정보들을 관리하는 레코드(아래 참고).

구조체 `process_control_block`(PCB)는 `process.c`에 새로 정의되었다.

```
/* PCB : see initialization at process_execute(). */
struct process_control_block {

  pid_t pid;                /* The pid of process */

  const char* cmdline;      /* The command line of this process being executed */

  struct list_elem elem;    /* element for thread.child_list */

  bool waiting;             /* indicates whether parent process is waiting on this. */
  bool exited;              /* indicates whether the process is done (exited). */
```

```
    bool orphan;              /* indicates whether the parent process has terminated before. */
    int32_t exitcode;         /* the exit code passed from exit(), when exited = true */

    /* Synchronization */
    struct semaphore sema_initialization;   /* the semaphore used between start_process() and process_execute() */
    struct semaphore sema_wait;             /* the semaphore used for wait() : parent blocks until child exits */

  };
```

- `pid` : 프로세스의 process id. 쓰레드의 tid와 natural mapping[1] 을 사용한다.
- `cmdline` : 해당 프로세스를 실행시킬 때의 command line string.
- `elem` : `struct thread`의 `child_list`에 담기 위한 list element.
- `waiting` : parent process가 `wait()` 시스템 콜을 하여 이 process를 기다리고 있다면 `true`. 중복 `wait()`시 바로 리턴하기 위해 필요하다.
- `exited` : 현재 process가 종료했는지 상태를 나타낸다 (terminated이면 `true`).
- `orphan` : 현재 process가 orphan인지를 나타낸다 (부모 프로세스가 자신보다 먼저 종료된 경우 `true`).
- `exitcode` : process가 종료한 경우, exit(return) code 를 저장한다.
- `sema_initializtion` 및 `sema_wait` : 프로세스 시작 혹은 `wait()`에서 적절한 두 프로세스 간의 동기화/커뮤니케이션을 위해 사용하는 semaphore로, 아래에서 자세히 설명한다.

`child_list` 라는 링크드 리스트에 의해, process의 부모-자식 관계(tree structure)가 정의된다. 이제 프로세스를 생성하거나 (`process_execute()`), `wait()` 또는 `exit()` 시스템 콜의 동작마다 위의 정보들을 올바르게 관리해주는 것이 필요하다. 크게 프로세스 생성 부분과 wait-exit 시스템 콜의 구현으로 나누어 설명해보자.

[1] thread t에 associated 된 process p에 대해 t.tid $= p$.tid.

Process Execution

프로세스 생성 시의 동작에 관여하는 두 가지 주요 routine은 `process_execute()` 와 `start_process()` 이다. 설명을 간단히 하기 위해 예외 처리, 리소스 해제 등을 제외한 핵심 동작만을 기술하면 아래와 같다.

```
pid_t process_execute (const char *cmdline) {
  /* ... (Omitted) ... */

  // Create a PCB, along with file_name, and pass it into thread_create
  // so that a newly created thread can hold the PCB of process to be executed.
  struct process_control_block *pcb = NULL;
  pcb = palloc_get_page(0);

  // pid is not set yet. Later, in start_process(), it will be determined.
  // so we have to postpone afterward actions (such as putting 'pcb'
  // alongwith (determined) 'pid' into 'child_list'), using context switching.
  pcb->pid = PID_INITIALIZING;

  pcb->cmdline = cmdline_copy;
  pcb->waiting = false;
  pcb->exited = false;
  pcb->orphan = false;
  pcb->exitcode = -1; // undefined

  // create thread, and wait until initialization inside start_process() is complete.
  tid = thread_create (file_name, PRI_DEFAULT, start_process, pcb);
  if(tid == TID_ERROR) return -1; // fail
  sema_down(&pcb->sema_initialization);

  // process successfully created, maintain child process list
  if(pcb->pid >= 0)
    list_push_back (&(thread_current()->child_list), &(pcb->elem));
  return pcb->pid;
}
```

이 함수 내에서 `pcb` 객체를 생성 및 초기화하고, 쓰레드를 생성하여 pid를 리턴해주어야 한다. 새로운 프로세스의 executable binary를 load 하는 것은, `process_execute()`가 실행되는 쓰레드 (부모 프로세스)가 아닌 새로이 생성된 쓰레드 (자식 프로세스)에서 수행되는데 이 때의 성공 여부에 따라 pid가 결정된다 (성공 시 tid, 실패 시 -1). 따라서 부모 프로세스의 입장에서는 자식 쓰레드의 초기화가 완료될 때까지 대기해야 한다. 이를 위해 `pcb->sema_initialization` 세마포어가 사용되는데, `start_process()`가 수행되는 자식 쓰레드에서 초기화를 수행 후 실패/성공 시 이 세마포어를 release함으로써 그 이후에 부모 프로세스가 다시 re-schedule 되어 실행될 수 있도록 설계되었다.

```
void start_process (void *pcb_) {
  /* ... Omitted ... */
  struct process_control_block *pcb = pcb_;
  char *file_name = /* Parse and tokenize cmdline */;

  /* ... Omitted ... */

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

```
  if (success) {
    push_arguments (cmdline_tokens, cnt, &if_.esp);
  }

  /* Assign PCB */
  pcb->pid = success ? (pid_t)(t->tid) : PID_ERROR;
  t->pcb = pcb;
  sema_up(&pcb->sema_initialization);  // wake up sleeping in process_execute()

  if (!success) sys_exit(-1);
  /* ... Omitted ... (jump, return from an interrupt) */
}
```

만약 프로세스 초기화가 실패한 경우는 (없는 파일이라 load가 실패하거나, argument가 잘못되어 오류가 발생하거나, 메모리 부족으로 page allocation이 실패하는 경우 등) `pcb->pid`을 -1 (`PID_ERROR`)로 세팅하고 semaphore를 해제 (signal, semaphore up)한다. 이 경우에는 새로 생성된 thread는 `sys_exit()` 를 통해 자원을 모두 해제한 뒤 종료되어 (내부적으로 `thread_exit()` 호출) 부모 프로세스 입장에서는 `process_execute()`가 -1 을 리턴하게 된다. 자식 프로세스를 성공적으로 초기화한 경우에는, 새로 자식 생성된 프로세스의 `pcb`가 부모 프로세스의 `child_list`에 삽입된다.

따라서 `exec()` 시스템 콜의 구현은, `process_execute()`를 통해 새 프로세스를 생성하면 된다 (요구사항). process가 새로 생긴다는 점에서 standard UNIX의 시멘틱과 다르긴 하지만, 위의 과정을 통해서 프로세스를 올바르게 실행시킬 수 있음을 알 수 있다.

Wait-Exit System Call

`wait(pid)` 시스템 콜은 현재 프로세스의 자식 프로세스 중에서 pid가 `pid`인 것을 찾아 종료할 때까지 대기하도록 한다. 시스템 콜 자체는 `process_wait()` 함수로 delegate 하고, 이 함수에 모든 핵심이 구현되어 있다.

- 현재 쓰레드 (프로세스)의 자식 프로세스들은 `child_list`라는 리스트에 저장되어 있으므로 리스트를 순회하여 자식 프로세스 (의 `pcb` 객체)를 찾는다. 그런 자식 프로세스가 없는 경우에는 스펙에 따라 -1을 리턴한다.
- 자식 프로세스가 이미 종료했거나, 중복으로 `wait`을 호출했는지는 `pcb`의 `exited` 이나 `waiting` 필드를 보면 알 수 있으므로, 이런 경우에도 마찬가지로 blocking 없이 -1을 리턴한다.
- 자식 프로세스가 아직 종료하지 않은 경우, 현재 쓰레드 (부모 프로세스)는 자식 프로세스가 `exit()`을 호출하여 종료할 때까지 기다려야 한다. 이를 위해 자식 프로세스의 `sema_wait` 세마포어를 사용하는데, 현재 쓰레드는 이 세마포어에서 wait하고 자식 프로세스가 종료될 때 signal이 발생할 때까지 대기할 수 있다. (아래 `exit()` 시스템 콜 설명 참고)
- `sema_wait`에 의한 blocking에서 깨어났다면, 이 상태는 자식 프로세스가 종료한 뒤이므로 (zombie status), `pcb`로부터 exit code를 얻을 수 있다. 뒤에서 다시 설명하겠지만 이 때 자식 프로세스의 리소스 (page) 해제가 일어난다.

그러면 이와 대응되는, 자식 프로세스가 `exit()`을 호출했을 때 일어나는 일을 살펴보자.

- 먼저 요구사항에 따라, 현재 쓰레드 이름과 return code를 호출한다.
- `pcb` 자료 구조에 exit code를 설정하고, 프로세스 상태가 zombie가 된다 (`exited`를 true로 설정함).

- `thread_exit()`를 호출하여, 쓰레드가 PintOS 커널에서 종료 및 제거될 수 있도록 한다. 이 과정에서 `process_exit()` 가 불린다.
- `process_exit()`에서는 먼저 이 프로세스가 점유한 리소스를 해제한다 (뒤에서 자세히 설명). 해제되는 리소스들은, file descriptor 밑 이 프로세스의 '종료된' 자식 프로세스들의 pcb 등이다. 종료되지 않은 자식 프로세스들은 orphan process로 처리된다.
- 마지막으로, (자신의 `pcb` 내) `sema_wait`에 signal을 준다. 이를 통해 자신을 기다리는 부모 프로세스가 있었다면 깨어나서 부모 프로세스의 `wait()`가 리턴될 수 있도록 한다.
- 만약 자신이 orphan process이면, 자신의 `pcb`를 해제한다 (아래 장에서 자세하게 설명).

이와 같은 구조로 `wait()` 및 `exit()` 시스템 콜이 동작하며, 부모-자식 프로세스 간의 상호작용이 이루어지게 된다. 결국 핵심은 적절한 시점까지 부모 프로세스를 기다리게 하는 `sema_wait` 세마포어의 사용이라고 할 수 있다.

`halt()`

한가지 빠진 시스템 콜이 있는데, `halt()` 시스템 콜이다. 이것은 그냥 `shutdown_power_off()` 를 호출하는 것으로 끝이다.

3. System Call: File Descriptor Management

이번에는 시스템 콜 중에서 파일과 관련된 create, remove, open, filesize, seek, tell, close, read, write 등을 설명한다.

Design

파일 관련 시스템 콜을 구현하기 위해서는, 프로세스별로 열고 있는 파일들을 관리할 수 있도록 하는 자료 구조들을 도입해야 한다.

```
--- a/src/threads/thread.h
+++ b/src/threads/thread.h
@@ -103,6 +103,17 @@ struct thread
 #ifdef USERPROG
     /* Owned by userprog/process.c. */
     uint32_t *pagedir;                  /* Page directory. */

     // Project 2: file descriptors and process table
     /* Owned by userprog/process.c and userprog/syscall.c */

     struct process_control_block *pcb;  /* Process Control Block */
     struct list child_list;             /* List of children processes of this thread,
                                            each elem is defined by pcb#elem */
+
+    struct list file_descriptors;       /* List of file_descriptors the thread contains */
+
+    struct file *executing_file;        /* The executable file of associated process. */
 #endif

     /* Owned by thread.c. */
```

`struct thread`에 추가된 필드들은 다음과 같다.

- `struct list file_descriptors` : 현재 프로세스의 file descriptor table. 열려진 모든 파일 엔트리들을 관리한다. 리스트의 각 element들은 아래서 설명할 `file_desc` 구조체이다.
- `file* executing_file` : 실행중인 프로세스의 executable 파일에 write하는 것을 방지하기 위해, executable 파일에 대한 파일 레퍼런스를 들고 있는다.

File descriptor table의 엔트리인 구조체 `file_desc`는 `process.h`에 새롭게 정의되었다.

```
/* File descriptor */
struct file_desc {
  int id;
  struct list_elem elem;
  struct file* file;
};
```

- `int id` : fd 번호.
- `struct list_elem elem` : `file_descriptors` list에 담기 위한 list element
- `file* file` : 커널에서 관리하는 (유저 프로그램의) 파일 객체

Implementation

`open()`

- `file_desc` 객체가 새로 생성된다. 이 때 메모리 영역은 유저 프로세스의 page 영역에 할당되며, 구현을 간단히 하기 위해 file descriptor 엔트리 1개당 1개의 page를 사용하였다(약간 낭비가 있지만).
- 열고자 하는 파일을 `filesys_open()`를 통해 open한다 (실패한 경우는, 리소스 해제를 해주고 -1 리턴).
- fd 번호를 할당해야 하는데, 사용한 방식은 단순히 (`file_descriptors` 리스트에 존재하는 file descriptor 중 가장 큰 fd) $+1$를 새로운 fd 번호로 사용하는 것이다. 이렇게 하면 (race condition이 생기지 않는다는 전제 하에) 열려진 모든 파일별로 고유한 fd 번호가 할당된다. 물론 기존에 열려진 파일이 없을 수도 있는데, 이 때에는 fd가 3부터 할당되게 하였다. (0, 1, 2는 stdin/stdout/stderr에 예약된 것으로 간주)
- 새로 추가된 file descriptor 는 file descriptors 테이블 (linked list) 에 추가된다.

`close()`

- 인자로 주어지는 fd 를 통해, file descriptors 테이블에서 해당 file descriptor 엔트리를 찾는다 (리스트 순차 탐색).
- 열려진 file 를 닫고 (`file_close()`), 리스트에서 제거한 후 메모리를 해제한다.

`create()`, `remove()`

- Straightforward. `filesys_create()`, `filesys_remove()`에 그대로 delegate 한다.
- 파일시스템 동작이 실패할수도 있는데 이 때에는 프로세스를 에러와 함께 종료하지 않고 파일시스템 함수가 리턴하는 값을 시스템 콜의 반환값으로 처리한다.

`filesize()`, `tell()`, `seek()`

- 역시 Straightforward. 다만 인자로 fd를 받으므로, 역시 list에서 file descriptor 엔트리를 검색하고, file 객체를 file system API를 활용하여 다룬다.
- 잘못된 fd가 주어진 경우는 에러를 내고 프로그램을 종료시킬 수도 있지만, 특별한 지시사항이 없어서 그냥 termination 없이 -1을 결과값으로 반환하도록 처리하였다.

`read(fd, buffer, size)`

- 먼저 유저 메모리 영역의 `buffer` 가 올바른지를 검사한다 (양쪽 boundary 모두 체크).

- `fd`가 0인 경우, 표준 입력인 경우는 `input_getc()` 함수를 통해 `size` 바이트 만큼 버퍼를 입력으로 채운다. 이 때, 앞서 언급한 `put_user()` 를 사용하여 잘못된 메모리 영역에 접근(segfault)시 에러와 함께 종료할 수 있도록 한다.
- 그 외에는 `fd`로 file descriptor를 찾아(없으면 -1) file 객체를 얻고, `file_read()` 함수를 통해 버퍼를 채운다.

`write(fd, buffer, size)`

- `read()`와 대칭이다. 마찬가지로 `buffer` 가 올바른지 검사한다.
- `fd`가 1인 경우, 표준 출력이므로 `putbuf()` 를 사용하여 콘솔에 버퍼 내용을 출력한다.
- 그 외에는 역시 `fd`로 file descriptor와 file 객체를 찾고, `file_write()` 함수를 통해 버퍼의 내용을 파일에 기록한다.

위 시스템 콜의 구현에서는 공통적으로 file descriptor 를 찾는 연산들이 필요하기 때문에, `find_file_desc(struct thread *t, int fd)` 헬퍼 함수를 두어 쓰레드 `t` 의 file descriptors list에서 번호가 `fd` 인 file descriptor 엔트리를 검색할 수 있도록 하였다.

4. Several Issues

각종 예외 처리

정상적인 실행 경로 말고도, 유저 프로그램이 잘못된 argument를 전달하거나 잘못된 메모리를 접근하는 경우, 커널에 리소스가 부족한 경우 등에 여러가지 예외 케이스에 대해 올바른 처리를 하는 것이 중요하다.

모두 각 함수마다 이를 모두 감안하여 구현하였는데, 큰 방향들은 아래와 같다.

- `palloc_get_page()` 등을 사용하여 메모리 영역을 할당받는 경우, allocation에 실패하면 `NULL`이 리턴될 수 있다. 이 때에는 구현해야 하는 각 함수의 요구사항에 따라 특정 리턴값(주로 -1를 리턴)을 반환할 수 있도록 처리해야 한다.
- 잘못된 메모리 영역을 접근하는 경우 등 프로세스가 비정상적으로 종료되어야 하는 경우는, `sys_exit(-1)` 를 통해 프로세스가 종료하도록 처리한다. exit system call 핸들러를 태움으로써, 점유하고 있던 모든 자원을 해제하고 부모 프로세스가 자신을 기다리는 경우 등에는 적절한 동기화 처리가 일어날 수 있도록 한다.

Resource Management

시스템 콜의 구현 및 프로세스의 lifecycle 동안 일어나는 모든 과정에 대해, 리소스와 메모리가 누수(leak)되지 않는 것이 중요하다.

실제로 메모리 릭 여부는 `no-vm/multi-oom` 테스트 케이스에서 검증이 된다. 이 테스트는 자식 프로세스를 recursion으로 계속적으로 spawn 하는데 (더이상 process 생성이 안될 때까지), 동시에 특정 깊이 이상에서는 잘못된 메모리 접근 등으로 crash가 발생하도록 한다. 이 깊이가 30 이상이어야 통과하며, 이 과정을 10번 정도 반복하여 같은 깊이를 반복하는지를 검증함으로써 정상적인 프로세스의 종료 및 비정상적인 모든 경우에 대해 메모리가 새지 않아야만 이 테스트를 통과할 수 있다.

이를 위해, 할당한 리소스(`palloc_get_page()`)가 가능한 모든 실행 흐름의 경우에 대해 올바르게 해제(`palloc_free_page()`)됨을 구현 레벨에서 모두 확인해야 했고, 아래에 그 내용들을 요약해보기로 한다.

`sys_open()`

file descriptor 할당을 위해 새로 page를 allocate 하는 부분.

```
struct file_desc* fd = palloc_get_page(0);
```

- 파일을 닫아줄 때 (`sys_close()`) 해제된다.
- 파일을 닫지 않고 프로세스가 결국에 종료된다면, 프로세스가 종료할 때 (`process_exit()`) 열려있는 모든 파일들을 닫아줄 때 해제된다.

```c
/* Resources should be cleaned up */
// 1. file descriptors
struct list *fdlist = &cur->file_descriptors;
while (!list_empty(fdlist)) {
  struct list_elem *e = list_pop_front (fdlist);
  struct file_desc *desc = list_entry(e, struct file_desc, elem);
  file_close(desc->file);
  palloc_free_page(desc); // see sys_open()
}
```

`process_execute()`

프로세스를 생성할 때, process status를 담는 `pcb` 구조체를 allocate함.

```c
// Create a PCB, along with file_name, and pass it into thread_create
// so that a newly created thread can hold the PCB of process to be executed.
pcb = palloc_get_page(0);
if (pcb == NULL) {
  goto execute_failed;
}
```

`pcb` 는 wait-and-exit 에서도 설명했듯이, 부모 프로세스가 종료되는 자식 프로세스의 return code를 알아내기 위해서 프로세스가 종료된 이후에도 남아 있어야 한다(zombie status). 따라서 `pcb` 구조체가 해제(free)되는 것은 부모 프로세스가 담당한다. 편의상 자식 프로세스를 c, 부모 프로세스를 p 라고 하자. 우리가 보여야 할 것은 c의 `pcb`가 (모든 경우에) 잘 해제된다는 것이다.

- 첫째, c가 p보다 먼저 종료하는 경우, c가 `exit()`을 수행했고, `pcb`는 아직 해제되지 않았다. 두 가지로 분기하는데
  - (1) p가 c에 대해 `wait()`을 호출한 적이 없는 경우: 추후에 p가 종료할 때, p는 `process_exit()` 루틴에서 자신의 자식 프로세스들 중 terminate된 것이 있다면 이들을 해제하는데, 이 때 c의 `pcb`가 해제된다.

    ```c
    // 2. clean up pcb object of all children processes
    struct list *child_list = &cur->child_list;
    while (!list_empty(child_list)) {
      struct list_elem *e = list_pop_front (child_list);
      struct process_control_block *pcb;
      pcb = list_entry(e, struct process_control_block, elem);
      if (pcb->exited == true) {
        // pcb can freed when it is already terminated
        palloc_free_page (pcb);
      } else {
        // the child process becomes an orphan.
        // do not free pcb yet, postpone until the child terminates
        pcb->orphan = true;
      }
    }
    ```

  - (2) p가 c에 대해 `wait()`을 호출한 경우: 이 때에는 p의 `wait()` 실행 중, (c가 종료한 이후에) p가 c를 child list에서 제거하고 c의 `pcb`를 해제한다.

- 둘째, c가 p보다 나중에 종료하는 경우 : 이 때 c는 orphan process가 된다. 먼저 p가 `exit()`을 수행할 때, c는 위의 코드에서도 볼 수 있듯이 아직 종료된 상태가 아니다. 이 때에는 c의 `pcb`를 해제하지 않는다 (c가 종료하기 전까지는, c의 `pcb`는 유효해야 한다). 추후 c가 종료될 때, 자신이 orphan process 이므로 (자신의 `pcb`를 거둘 프로세스가 없음) 자기 스스로 `pcb`를 해제한다.

PintOS 커널은 single-processor이므로 시스템 콜의 작동 순서만 잘 보장된다면 (without interleaving), 위 두 가지 경우 중 한가지에 반드시 속하므로, 모든 경우에 대해 — 각 프로세스와 그 부모 프로세스가 종료한 이후에는 반드시 — `pcb` 메모리가 잘 해제됨을 알 수 있다.

Synchronization and Lock

현재 구현되어 있는 filesystem은 내부 동기화가 없기 때문에, 최대 하나의 프로세스만 파일시스템에 접근하는 것을 보장하기 위하여 파일 접근을 하는 시스템 콜의 전후로는 lock을 잡아주어야 한다. 이 때 사용한 lock은 `syscall.c`에 `filesys_lock`로 선언되어 있으며, `exec()`를 비롯하여 (내부적으로 `load()`에서 파일시스템에 접근하기 때문) 다른 파일시스템 관련 시스템 콜을 수행하기 전에는 lock을 acquire, 이후에는 (실패한 경우도 포함) lock을 release 하도록 구현하였다.

5. Test Results

성공적으로 모든 테스트 케이스를 통과하였다. PintOS의 테스트 케이스가 잡아주지 못하는 여러가지 시나리오들도 있겠지만 전반적으로 옳은 방향의 설계로 각종 문제점이 발생할 수 있는 부분들을 챙겨가며 성공적으로 옳은 구현을 완성지을 수 있었다.

```
TOTAL TESTING SCORE: 100.0%
ALL TESTED PASSED -- PERFECT SCORE

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

SUMMARY BY TEST SET

Test Set                                    Pts Max  % Ttl  % Max
------------------------------------------- --- --- ------ ------
tests/userprog/Rubric.functionality          108/108  35.0%/ 35.0%
tests/userprog/Rubric.robustness              88/ 88  25.0%/ 25.0%
tests/userprog/no-vm/Rubric                    1/  1  10.0%/ 10.0%
tests/filesys/base/Rubric                     30/ 30  30.0%/ 30.0%
------------------------------------------- --- --- ------ ------
Total                                                100.0%/100.0%
```

부록으로 `multi-oom` 테스트의 실행 결과 일부를 첨부한다. 메모리 관리를 특별히 비효율적으로 한 부분은 없어서, Recursion depth는 60회까지 일어났다.

```
Copying tests/userprog/no-vm/multi-oom to scratch partition...
qemu-system-i386 -device isa-debug-exit -hda /tmp/user/1002/o9kBpuTXko.dsk -m 4 -net none -nographic -monitor null
PiLo hda1
Loading..........
Kernel command line: -q -f extract run multi-oom
/* ... (omitted) ... */
multi-oom: exit(60)
(multi-oom) success. program forked 10 times.
(multi-oom) end
multi-oom: exit(60)
Execution of 'multi-oom' complete.
Timer: 7081 ticks
Thread: 4043 idle ticks, 113 kernel ticks, 2925 user ticks
hda2 (filesys): 268445 reads, 242 writes
hda3 (scratch): 118 reads, 2 writes
Console: 139265 characters output
Keyboard: 0 keys pressed
Exception: 320 page faults
Powering off...
```

Conclusion

이로써 PintOS 커널에 기본적인 유저 프로그램이 실행될 수 있는 구조가 완성되었다. 자, Virtual Memory여 어서 오라!