Pintos Project 2_1. User Program Basic

(설계 프로젝트 수행 결과)

과목명: [CSE4070-01] 운영체제
담당교수: 서강대학교 컴퓨터공학과 이혁준
조원: 54조 고창영

개발기간: 2014. 10. 04. - 2014. 10. 08.

최 종 보 고 서

프로젝트 제목: Pintos 프로젝트 2_1. User Program Basic
제출일: 2014. 10. 26.
참여조원: 54조 고창영

1 개발목표

최소한의 기능을 갖는 교육용 OS Pintos를 만들어나가는 과정에서 OS의 wait()과 exec(), 그리고 System Call의 실제 구현을 통해 Synchronization과 같은 이론을 학습한다.

2 개발 범위 및 내용

2.1 개발 범위

1. Argument Passing
2. User Memory Access
3. Process Wait
4. Process Execute
5. System Call

2.2 개발 내용

2.2.1 Argument Passing

`./a.out arg1 arg2 arg3` 와 같이 Argument들이 주어지면, 이들을 process에게 `argv[]={"arg1", "arg2"}`와 같은 식으로 stack을 이용해서 전달한다.

2.2.2 User Memory Access

User 프로세스가 자신의 메모리 영역 밖을 접근하거나, 아직 할당되지 않은 메모리 영역을 접근하려 할 경우에 잘못된 접근을 차단하고 프로세스를 강제 종료한다.

2.2.3 Process Wait

부모 프로세스가 자식 프로세스에 대해 wait()를 호출함으로써 자식 프로세스가 종료될 때까지 기다리고, 자식 프로세스의 종료 상태를 알 수 있도록 한다.

2.2.4 Process Execute

프로세스가 exec()(예:exec("name_of_executable arguments"))를 호출함으로써 주어진 arguments를 전달하여 자식 프로세스를 실행할 수 있도록 한다.

2.2.5 System Call

User 프로세스가 Interrupt를 통해 System call을 요청하면 이를 통해 System call 번호와 arguments를 받아서 이에 해당하는 system call(예:Write, Read 등)을 호출하도록 System call을 구현한다.

3 추진 일정 및 개발 방법

3.1 추진 일정

- 10.04 - 10.05: Pintos 매뉴얼 및 thread.c/h, synch.c/h 정독
- 10.06 - 10.06: Argument Passing 구현
- 10.07 - 10.07: Process Wait, Process Execute, System Call 구현
- 10.08 - 10.08: User Memory Access 구현

3.2 개발 방법

개발은 다음과 같은 원칙을 바탕으로 진행했다:

1. Pintos Manual의 내용을 숙지한다.
2. 이미 작성되어있는 Pintos 내부의 코드(thread.c 등)를 숙지한다.
3. 구현해야할 각 기능의 목표와 작동을 숙지한다.
4. 구현시 한 함수를 작성하면 해당 함수의 작동을 완벽히 분석한 다음 다른 함수를 작성한다.
5. 각 기능 구현의 기한을 정하고, 이에 맞춰 작성할 수 있도록 시간을 잘 안배한다.

또한, 다음과 같이 도표를 그려서 각 함수의 연결과 그 흐름을 확인했다.

3.3 연구원 역할 분담

혼자서 하는 프로젝트였으므로 역할을 분담하기보다는 위에서 적은 것과 같이 날짜를 기준으로 작업을 분배하였다.

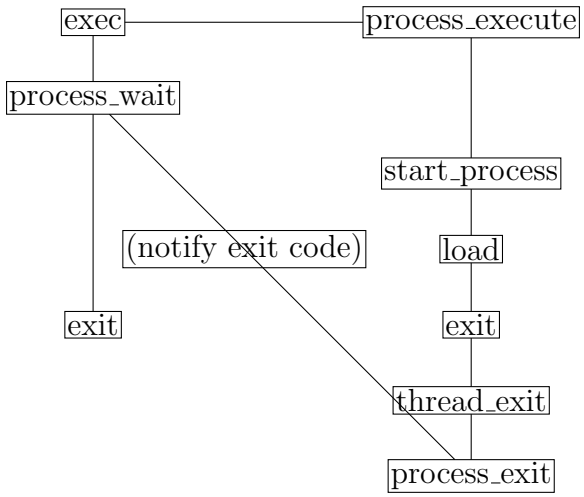

Figure 1: 흐름도

4 연구 결과

결과적으로 다음과 같은 테스트에 대해 pass를 받을 수 있었다:

```
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
```

그리고 다음과 같은 추가기능도 정상 작동을 확인할 수 있었다:

```
$ pintos -v -k -T 60 --bochs  --filesys-size=2 -p ../../examples/sum -a sum
  -- -q  -f run 'sum 5 4 5 1'
...
Putting 'sum' into the file system...
Erasing ustar archive...
Executing 'sum 5 4 5 1':
5 15
sum: exit(0)
Execution of 'sum 5 4 5 1' complete.
...
```

4.1 합성 내용

4.1.1 Argument Passing

`char[]="executable arg1 arg2"` 형태로 주어지는 C-Style String을 tokenize해서 다음과 같은 형태로 stack에 push한다.

”executable”	1000
”arg1”	990
”arg2”	985
(word-align)	984
null pointer	980
990	976
985	972
972	968
2	964
null pointer	960

Table 1: argument와 stack

4.1.2 User Memory Access

User 프로세스가 자신의 메모리 영역 밖을 접근하거나, 아직 할당되지 않은 메모리 영역을 접근하려 할 경우에 잘못된 접근을 차단하고 프로세스를 강제 종료한다.
이는 User 프로세스가 접근하려는 메모리 영역에 대한 page가 해당 프로세스에게 할당되었는지를 page directory를 이용해서 확인하면 된다.

4.1.3 Process Wait

부모 프로세스가 자식 프로세스에 대해 wait()를 호출함으로써 자식 프로세스가 종료될 때까지 기다리고, 자식 프로세스의 종료 상태를 알 수 있도록 한다.
이는 thread를 block 및 unblock시키는 것으로 구현할 수 있다.

4.1.4 Process Execute

프로세스가 exec()(예:exec("name_of_executable arguments"))를 호출함으로써 주어진 arguments를 전달하여 자식 프로세스를 실행할 수 있도록 한다.
이는 `process_execute`를 호출하는 것으로 구현 가능한다.

4.1.5 System Call

User 프로세스가 Interrupt를 통해 System call을 요청하면 이를 통해 System call 번호와 arguments를 받아서 이에 해당하는 system call(예:Write, Read 등)을 호출하도록 System call을 구현한다.
이는 다음과 같이 `intr_frame *f`의 구조를 이해함으로써 구현할 수 있다:

syscall return value	f→eax
syscall number	f→esp+0
syscall argument1	f→esp+4
syscall argument2	f→esp+8
syscall argumentK	f→esp+4K

Table 2: intr_frame *f의 구조

4.2 제작 내용

4.2.1 Argument Passing

구현은 새로운 함수를 작성하여 해결했다. load() 함수 내에서 setup_stack()이 성공하면, 그 때 argument passing을 위한 함수가 수행된다.

1. `command line`을 복사한다(원본을 손상시키지 않기 위해).
2. `strtok_r`을 이용, `argument`를 `parse`함. 이 때, 이 `argument`들은 `char **argv[2]`에 저장되는데,
   이는 `realloc()`이 없는 환경에서 이를 대신하기 위함이다.
   `argument`의 총 개수가 몇 개인지 알 수 없으므로 `char *argv[1000]`과 같이 사용하지 않고, `**argv`를 사용하는데,
   매번 새로 `malloc()`을 하기 위해 `**argv`가 두 개가 된 것이다(한 번은 `argv[0]`, 한 번은 `argv[1]` 이렇게 교대로 사용한다).
   물론 이 과정에서 `arument`의 총 개수인 `argc` 또한 저장한다.
3. 다음엔 위에서 설명한 표(`Table 1`)와 같이 `stack`에 값들을 `push`한다.

4.2.2 User Memory Access

구현은 위에서 설명한 것과 같이 User 프로세스의 pagedir를 이용해서 해당 메모리 영역이 할당된 page가 존재하는지를 확인한다.
이는 `pagedir_get_page()` 함수로 어렵지 않게 해결할 수 있다:

```
static bool
is_valid_user_addr (const uint8_t *uaddr)
{
        if (uaddr >= PHYS_BASE) return false;
        if (pagedir_get_page(thread_current()->pagedir, uaddr) == NULL)
                return false;
        return true;
}
```

4.2.3 Process Wait 및 Process Execute

이들의 구현은 thread.h의 struct thread 구조체를 수정하고, 새로운 struct child를 만듦으로써 쉽게 할 수 있다.

```
#ifdef USERPROG
struct child
{
    tid_t tid;
    int exit_code;
    struct list_elem elem;
};
#endif

struct thread {
...
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;        /* Page directory. */
    int exit_code;            /* exit code of this thread */

    /* for synchronization. */
    bool is_alive;            /* true if this thread is alive, otherwise dying. */
    tid_t waiting_for;        /* the tid of child I am waiting for */
    /* thread(process) tree. */
struct lock wait_lock;    /* lock for wait() */
    struct thread *parent;    /* parent thread */
    struct list children;     /* list of children, either dead or alive. */
#endif
...
}
```

위의 구조체를 이용하면, process_wait()를 다음과 같은 논리로 구현할 수 있다:

process가 자식 process를 만들 때 thread_create()에서 자식의 parent를 thread_current(), 재 exec를 요청한 thread로 설정하고, struct list children을 초기화(init_list)

한 다음, 자신을 나타내는 struct child를 새로 만들고(malloc()), tid를 현재 자신의 tid로 설정한다. 그 다음엔 자신의 parent의 children에 이 struct child를 추가한다. 이 struct child는 thread가 죽은 뒤에도 부모 process가 wait()를 호출할 수 있기 때문에 thread_exit()에서 thread를 free()한 후에도 exit code를 저장하기 위해 만든 것이다. 마지막 초기화는 부모의 wait_lock을 acquire해놓는 것인데, 이는 wait을 구현하기 편하게 해준다.

이제 wait에 대해 살펴보면, 부모가 wait를 호출할 경우, 이 process가 자식인지를 일단 자신의 children list를 순회하면서 일치하는 tid를 갖는 child가 존재하는지 확인하는 것으로 수행한다. 만일 존재하지 않는다면 -1을 리턴하고, 그렇지 않다면 자식의 is_alive를 확인한 다음 자신이 기다리는 thread가 이 자식임을 표시하기 위해 waiting_for를 자식의 tid로 변경한 다음, lock을 acquire하기 위해 기다렸다가 list에서 이 자식을 삭제한 다음(한 process를 두 번 기다리면 안 된다) exit_code를 리턴한다.

이 lock은 자식이 thread_exit()를 수행할 때 struct child에 exit_code를 저장한 다음 release하도록 구현하면 자식이 종료되었을 때 부모가 다시 깨어나게 되므로 wait()이 성공적으로 수행된다.
코드는 다음과 같다:

```
...
  /* Initialize exit code. */
  t->exit_code = 0;
  /* Indicate the thread is alive. */
  t->is_alive = true;
  /* I'm currently waiting for no one. */
  t->waiting_for = -1;
  /* Initialize process tree. */
  t->parent = NULL;
  list_init(&t->children);
  lock_init(&t->wait_lock);
  if (!list_empty(&all_list)) {
      t->parent = thread_current();
  }
...
```

```
...
#ifdef USERPROG
  if (t->parent != NULL) {
      struct child *newborn = malloc(sizeof(*newborn));
      newborn -> exit_code = 0;
      newborn -> tid = tid;
      list_push_back(&t->parent->children, &newborn->elem);
  }
#endif
...
```

```
...
    lock_acquire(&me->wait_lock);

    exit_code = list_entry(target, struct child, elem) -> exit_code;
    list_remove(target);
    me -> waiting_for = TID_ERROR;

    lock_release(&me->wait_lock);

    return exit_code;
...
```

4.2.4 System Call

구현은 위의 Table 2에서 본 intr_frame의 구조를 이용해서 간단하게 할 수 있고, 코드도 복잡하지 않다:

1. System Call Number를 통해 호출하려는 함수를 알아낸다.
2. 그 다음 주어진 argument를 통해 함수를 호출한다.
3. 마지막으로는 return value를 f-¿eax에 대입하고 끝낸다.
4. 없는 system call을 호출하거나 메모리가 영역 밖을 참조하거나 없는 메모리 영역을 참조하면 exit(-1)를 호출한다(User Memory Access).

```
...
#define kth(esp,k) (esp+k*sizeof(char*))
#define get_arg(esp,k,type) *(type*)(kth(esp,k))
#define is_user_vaddr_after(esp,k,type) (is_user_vaddr(kth(esp,k+1)-1))
#define SYSTEMCALL_HANDLER_LIST 0
#define syscall_return (f->eax)
...
    char *esp = f -> esp;
    int syscall_number = -1;

    if (is_valid_user_addr(esp) &&
        is_user_vaddr_after(esp, 0, int)) {
        syscall_number = *(int*)esp;
    }
...
    } else if (syscall_number == SYS_WRITE &&
            is_valid_user_addr(kth(esp, 1)) &&
            is_user_vaddr_after(esp, 1, int) &&
            is_valid_user_addr(kth(esp, 2)) &&
```

```
            is_user_vaddr_after(esp, 2, char*) &&
            is_valid_user_addr(kth(esp, 3)) &&
            is_user_vaddr_after(esp, 3, unsigned)) {
        syscall_return = write_handler(get_arg(esp, 1, int), get_arg(esp, 2, char*),
...
```

4.3 시험 및 평가내용

평가는 기본적으로 make check의 pass / fail로 평가했으며, 특정 문제를 해결하기 힘든 경우에는 특정 프로그램만 시험해보도록 다음과 같은 커맨드를 이용했다.
`$ pintos -v -k -T 60 --bochs --filesys-size=2 -p tests/userprog/args-none -a args-none -- -q -f run args-none` pass / fail에 대한 결과는 다음과 같이 21개 pass이다. 이 평가 내용들은 각각 개발 목표의 네 부분, 그리고 생산성과 내구성으로 나눌 수 있다.

```
1. argument passing(생산성:프로그램의 작동이 의미있나)
pass tests/userprog/args-none
pass tests/userprog/args-single
pass tests/userprog/args-multiple
pass tests/userprog/args-many
pass tests/userprog/args-dbl-space
2. user memory access(내구성:프로그램이 에러 상황에 대처가능한가)
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/exec-bad-ptr
3. system call(생산성)
pass tests/userprog/halt
pass tests/userprog/exit
4. process execute(생산성 및 내구성)
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
5. process wait(생산성 및 내구성)
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
```

4.3.1 보건 및 안정

OS는 기본적으로 어떤 환경에서도 죽어서는 안 된다. 그러므로 Page Fault를 일으키는 경우에는 죽지 않도록 `Exception.c`에서 Kernel Page Fault의 경우에도 `eip:=eax;eax:=-1;`를 하고 User process를 kill 하는것으로 처리했다. 또한, 잘못된 user memory에 접근해서 kernel 메모리를 더럽히지 못하도록 미리 이 주소가 valid한지 테스트한 다음 접근하도록 구현했으며, argumen passing을 구현할 시에도 stack의 영역을 벗어나지 않도록 최대 4096 바이트만 push 가능하도록 하고 초과하는 경우에는 프로세스를 종료시켰다.

5 기타

5.1 Cascading termination

부모 프로세스가 죽을 때(exit할 때), 자식 프로세스를 모두 같이 죽이는(exit) 것이다.
Windows와 같은 운영체제에서 기본적으로 지원하는 기능인 것으로 보여서 내 프로그램에도 구현했다.

5.2 Page

OS에서 가장 작은 메모리 할당 단위이며, Virtual Memory의 연속적인 메모리 영역을 나타낸다.
이는 일부 메모리를 Main Memory 대신 HDD에 메모리 내용을 저장해놓는 용도로도 사용된다.
즉, 연속된 영역의 Virtual Memory와 실제 Main Memory(혹은 보조 디스크) 상의 블록들과의 대응이다.
실제 연속되지 않은 메모리(혹은 디스크)영역을 연속된 것처럼, 특정 주소를 이용해서 접근 가능하게 하기 위한 것이다.

5.3 연구 조원 기여도

고창영 : 설계 및 코드 작성, 그리고 보고서(100%).

5.4 느낀점

OS의 원리와 이론들을 직접 구현하면서 실습할 수 있는 기화가 되어 유익했다.
특히 wait()의 구현에서 많은 것을 배울 수 있었고 Cascading termination같은 개념도 생각해볼 수 있어서 좋은 기회였다.
하지만 test가 획일적이고 적은 것과, pintos의 에러를 잡기 힘든 것이 좀 아쉽고 힘들었다.