49003157 최우석
20100374 주봉조
CSED312-01 운영체제
November 7, 2019

PintOS Final Report
Project2 - User Program

프로젝트 원격저장소: https://github.com/wooseokyourself/pintos

목차

* Problem 1 과 Problem 4 는 모두 Problem 2 및 Problem 3 을 해결하는 과정에서 해결할 수 있으므로 순서를 위와 같이 했습니다.

Problem 2: Argument Passing

구현 방법

1. **[process_execute]** process_execute 함수에서, 인자로 전달된 file_name을 strtok_r 함수를 이용하여 “ “이 나오기 전까지의 문자를 thread_create의 호출에서 name 인자로 보낸다. process_execute의 인자로 넘어온 file_name은 프로그램을 실행하기 위해 입력된 한 줄의 커맨드라인이므로, 이를 분해해서 파일명만을 스레드생성 함수의 인자로 보내주기 위함이다.
2. **[start_process]** thread_create 호출시 인자로서 호출되는 start_process 함수에도 역시 process_execute와 동일한 file_name이 인자로 넘어온다. 여기에서 file_name을 strtok_r 함수를 이용하여 “ “ 을 기준으로 나눈다. 이렇게 나뉜 문자들은 char **argv에 저장되며, int argc에 나눠진 문자의 수를 저장한다. 이 때, argv[0] 엔 프로그램의 이름이 저장되며, argv[1] 은 프로그램 실행인자1, argv[2] 에는 프로그램 실행인자2, … 식으로 저장된다.
3. (start_process 함수 내에서 생성된 intr_frame if_ 의 멤버변수인 esp가 현재 프로세스의 유저스택을 가리키는 포인터이다. 이어서 호출되는 load 함수 내에서 이 esp는 setup_stack 함수를 통해 PHYS_BASE로 초기화됨으로써 유저스택에 데이터를 쌓기 위한 준비를 마친다.)
4. load 함수가 아무런 문제 없이 데이터를 쌓기 위한 모든 준비를 마치고 true를 리턴하였다면, esp를 이용하여 유저스택에 데이터를 쌓아준다. [argv에 저장된 문자열들 -> word-align -> NULL -> argv 문자열들의 주소값 -> argv의 주소값 -> argc값 -> return addr] 순으로 데이터를 쌓는다. 핀토스의 유저스택은 아래로 늘어나므로, 한 데이터를 쌓을 때마다 *esp -= sizeof(‘넣을데이터’) 로 스택을 확장한 뒤 *esp = ‘넣을데이터’ 식으로 진행했다. word-align 은 argv 문자열을 넣은 뒤 다음 데이터를 넣기 전 4의 배수를 맞추기 위해 넣는 데이터이므로, while ((PHYS_BASE - *esp) % 4 != 0) 로 반복문을 돌리며 4의 배수가 맞춰질 때까지 0을 집어넣었다.

Design에서 변경된 점

1. 기존의 계획에서는 file_name의 토큰화를 load 함수 내에서 진행한 뒤, 이를 통해 생성된 char **argv와 int argc를 esp와 함께 setup_stack의 인자로 보내서, setup_stack 내에서 데이터를 쌓고자 하였다. 하지만 그렇게 한다면 구현이 너무 복잡하고 기존 제공된 함수(setup_stack)의 정의를 변경해야 하므로, 그냥 이 모두를 start_process 내에서 구현했다.

Problem 3: System Call

구현 방법

1. syscall.h 와 syscall.c 에 우리가 구현해야 할 모든 시스템콜의 정의를 써준다.
2. syscall_handler 함수에서 각 시스템콜은, 인자로 전달된 intr_frame *f 의 스택포인터인 f의 값을 기준으로 switch되게 하여 호출했다.
3. 시스템콜 함수 호출시 전달되는 인자의 위치는 1개일 경우 esp+4, 2개일 경우 esp+4 및 esp+8, … 에 위치한다. 이를 각 시스템콜 함수의 호출에 인자로 넣어준다. 이 때, 각 인자로 들어갈 값들에 대해 is_user_vaddr 함수를 이용하여 스택포인터가 커널을 참조하고 있는건 아닌지 확인한다. 이 is_user_vaddr 함수는 check_user_vaddr 함수를 이용하여 호출하였고, false 리턴시 -1과 함께 종료하도록 했다.
4. 시스템콜 함수가 리턴값을 가질경우, 이 값을 f->eax 에 넣어준다.
5. 파일디스크립터(fd)를 구현하기 위해, 스레드의 구조체에 struct file *fd[128] 을 추가하였다. 각 파일의 fd는 이 배열의 인덱스와 매핑된다.

User Process Manipulation

halt()

shutdown_power_off 함수를 호출한다.

exit()

스레드의 구조체에 exit_code를 추가하였다(원래 제공되는 것일수도 있다). 스레드의 이름과 exit_code 를 출력한 뒤, 스레드의 exit_code 를 인자로 전달된 status에 대입하고, 스레드의 모든 파일을 fd를 이용하여 닫아준 뒤(또 다른 시스템함수인 close 함수 이용), 마지막으로 스레드를 종료한다.

exec()

process_execute 함수를 호출한다.

wait()

process_wait 함수를 호출한다. 인자로 전달되는 pid는 자식의 pid를 의미하며, 이를 process_wait 함수의 인자로 전달한다.

스레드는 자식스레드들과 부모스레드를 가지고 있어야 하므로, 이를 스레드 구조체에 list children과 thread *parent 로 정의하였고, 프로세스간의 synchronization 을 구현하기 위해, child_lock 과 memory_lock 이라는 세마포어를 정의하였다.

process_wait 함수의 인자로 전달된 pid는 자식스레드를 의미한다. 따라서 현재 스레드의 children 리스트를 순회하면서 인자로 전달된 tid를 발견하면, 그 녀석의 child_lock을 down 시켜서 자식프로세스가 종료하기를 기다린다. 이후 종료되면 list_remove를 이용해 제거한다. 한 편, 이 child 스레드는 process_exit 에서 up을 해야하므로, process_exit에서 현재 스레드에 대해 sema_up(child_lock) 을 추가한다.

한 편, 자식스레드가 종료되면 이 자식스레드의 메모리가 사라진다. 당장은 안 사라질 수도 있지만, 사라질 경우도 존재한다. 그러므로 이후 이 녀석에 대해 list_remove를 수행할 수 없게 된다. 그러므로 process_exit에서

sema_up(child_lock) 을 통해 내가 현재 종료된다는 것을 알린 뒤, sema_down(memory_lock) 을 통해 list_remove를 수행할 때까지 기다린다. 이렇게 기다리는 동안, 다시 process_wait에서 list_remove가 실행되면, 이후 sema_up (memory_lock) 을 통해 process_exit 을 진행할 수 있게 했다.

File Manipulation

모든 *NULL* 체크가 *true* 라면 *exit(-1)*을 호출한다.

create()
NULL을 체크하고, filesys_create를 호출한다.

remove()
NULL을 체크하고, filesys_remove를 호출한다.

open()
인자로 전달된 file을 인자로 filesys_open을 호출한 뒤, 이 리턴값이 NULL인지를 체크한 뒤 아니라면 현재 스레드의 가장 앞쪽의 비어있는 fd배열(file* 자료형)에 넣어준다. 이후 이 fd(int) 를 리턴한다. (file_deny_write)

_fd_를 인자로 파일을 가져오는 시스템함수를 위해, _struct file *getfile (int fd)_ 함수를 정의하였다.
인자로 _buffer_가 들어오면 _check_user_vaddr_를 이용해 체크하였다.

filesize()
NULL을 체크하고, file_length를 호출한다.

read()
NULL을 체크한다. 만약 fd == 0 이라면 input_getc 함수를 이용해 키보드 입력을 버퍼에 넣는다. 그리고 입력된 사이즈를 리턴한다. 만약 fd != 0 이라면, file_read를 호출한다.

write()
fd == 1 이라면 putbuf를 이용해 버퍼의 내용을 콘솔에 입력한다. fd != 1 이라면 NULL을 체크하고 file_write 를 호출한다. (file_deny_write)

seek()
NULL을 체크하고, file_seek 를 호출한다.

tell()
NULL을 체크하고, file_tell 을 호출한다.

close()
NULL 을 체크하고, file_close 를 호출한다.

Design에서 변경된 점

wait()

기존에는 cond_wait 및 cond_signal 을 이용하여 synchronization을 구현하려 하였으나, semaphore 를 이용하는 것이 더 직관적이어서 바꿨다.

자식스레드가 커널에 종료되는 상황, 직계자식이 아닌 상황 등에 대응하기 위해 스레드에 isRun 변수를 추가해서 이 변수와 status를 이용하여 모든 경우에 대해 조건문을 붙이려고 했지만, 사실 결국 모두 -1을 리턴해야 하는 것이므로, 즉 그냥 직계자식이 아닌 경우에만 -1을 리턴하면 되는 것이므로 굳이 그럴 필요가 없었다. 이렇게 생각하게 된 계기는 자식스레드를 포인트하는 멤버변수를 상속관계에 가진 모든 자식들을 포인트하는 list children 이 아닌 오로지 하나의 직계자식만을 포인트하는 file *child 를 가지도록 하려 했기 때문이다. 단지 children 리스트를 이용하여, 이 리스트를 순회하며 자식스레드면 synchronization을 적용하며 종료를 기다리고, 아예 자식스레드가 없는 경우에만 -1을 리턴하면 됐던 것이다.

open()

fd에 대한 개념이 잡히지 않아서 file_open_count 전역변수를 이용하여 열린 모든 파일에 대해 모두 독립된 값을 가지도록 하려고 하였으나, fd가 중복되지 않아야 한다는 조건은 결국 그 파일들을 실행하는 스레드 내에서만 한정된 이야기이므로, fd를 스레드에서 열린 파일들을 저장하는 멤버변수인 file *fd[128]의 인덱스와 매핑시킴으로써 file_open_count도 필요 없게 되었다.

write()

필요한 사이즈만큼 반복문을 돌지 않아도 되었다. 왜 그런지에 대한 이유는 아직 찾지 못했다..

Problem 1: Process Termination Messages

구현 방법

Argument Passing 에서 스레드를 생성할 때 파일 이름을 스레드의 이름으로 지정한다. 이후 System Calls의 exit 에서 스레드가 종료할 때 그 스레드의 이름과 exit_code를 형식에 맞게 출력한다.

Design에서 변경된 점

exit_code 가 thread_status 라고 착각했었다.

Problem 4: Denying Writes to Executables

구현 방법

System Calls의 read 함수와 write 함수에서 file -> deny_write 를 통해 파일을 쓸 수 있는지 확인하고, 만약 쓸 수 없다면 -1을 반환하며, 쓸 수 있다면 file_deny_write 함수를 호출하여 못 쓰게 막은 뒤 바로 read 혹은 write를 진행하고 이후 다시 file_allow_wirte 를 호출하여 원상태로 복귀시킨다.
하지만 read 함수에서 이를 진행하면 몇몇 테스트를 통과하지 못해서 read에는 적용하지 않았다..

Design에서 변경된 점

없음.

테스트 및 make check

```
.
==========================================================================
                    Bochs x86 Emulator 2.6.2
            Built from SVN snapshot on May 26, 2013
              Compiled on Oct  7 2019 at 06:05:54
==========================================================================
00000000000i[     ] reading configuration from bochsrc.txt
00000000000e[     ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keyboard' option.
00000000000i[     ] installing nogui module as the Bochs GUI
00000000000i[     ] using log file bochsout.txt
PiLo hda1
Loading.....
Kernel command line: -q run 'echo coocoocoo'
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer...  163,400 loops/s.
hda: 1,008 sectors (504 kB), model "BXHD00011", serial "Generic 1234"
hda1: 197 sectors (98 kB), Pintos OS kernel (20)
hdb: 5,040 sectors (2 MB), model "BXHD00012", serial "Generic 1234"
hdb1: 4,096 sectors (2 MB), Pintos file system (21)
filesys: using hdb1
Boot complete.
Executing 'echo coocoocoo':
echo coocoocoo
echo: exit(0)
Execution of 'echo' complete.
Timer: 136 ticks
Thread: 0 idle ticks, 126 kernel ticks, 13 user ticks
hdb1 (filesys): 39 reads, 0 writes
Console: 658 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...



==========================================================================
                    Bochs x86 Emulator 2.6.2
            Built from SVN snapshot on May 26, 2013
              Compiled on Oct  7 2019 at 06:05:54
==========================================================================
00000000000i[     ] reading configuration from bochsrc.txt
00000000000e[     ] bochsrc.txt:8: 'user_shortcut' will be replaced by new 'keyboard' option.
00000000000i[     ] installing nogui module as the Bochs GUI
00000000000i[     ] using log file bochsout.txt
PiLo hda1
Loading.....
Kernel command line: -q run echo
Pintos booting with 4,096 kB RAM...
383 pages available in kernel pool.
383 pages available in user pool.
Calibrating timer...  163,400 loops/s.
hda: 1,008 sectors (504 kB), model "BXHD00011", serial "Generic 1234"
hda1: 197 sectors (98 kB), Pintos OS kernel (20)
hdb: 5,040 sectors (2 MB), model "BXHD00012", serial "Generic 1234"
hdb1: 4,096 sectors (2 MB), Pintos file system (21)
filesys: using hdb1
Boot complete.
Executing 'echo':
echo
echo: exit(0)
Execution of 'echo' complete.
Timer: 133 ticks
Thread: 0 idle ticks, 124 kernel ticks, 12 user ticks
hdb1 (filesys): 39 reads, 0 writes
Console: 626 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
Powering off...
--------------------------------------------------------------------------
```

```
pass tests/filesys/base/syn-write
pass tests/userprog/args-none
FAIL tests/userprog/args-single
FAIL tests/userprog/args-multiple
FAIL tests/userprog/args-many
FAIL 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/sc-boundary-3
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
pass tests/userprog/open-normal
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr
pass tests/userprog/open-twice
pass tests/userprog/close-normal
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
FAIL tests/userprog/exec-arg
FAIL tests/userprog/exec-bound
pass tests/userprog/exec-bound-2
pass tests/userprog/exec-bound-3
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
FAIL tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
pass tests/userprog/rox-simple
FAIL tests/userprog/rox-child
FAIL tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
FAIL tests/userprog/no-vm/multi-oom
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
FAIL tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
11 of 80 tests failed.
```

느낀점

프로젝트를 띄엄띄엄 하게 되면 지난번에 하며 익힌 내용이 기억이 나지 않아 적더라도 꾸준히 하는게 중요함을 깨달았다. 특히 프로젝트를 진행하며, 수업시간에 이론적으로 배운 내용들을 내가 확실히 이해하고 있는지에 대한 의문이 들었다. 또한 multi-oom 테스트는 아무리 해봐도 통과되지 않았는데, 그 이유를 제발 알고 싶다.

또한 args-none은 괜찮은데 args-single, args-multiple, args-many 등은 항상 TIMEOUT 에러가 났다. 이것도 왜 이런지 알고싶다. ‘echo x’의 실행은 되는데 왜 이건 안되는건지 궁금하다.