2020학년도 2학기

Operation System

Project #2

Design Report

담당교수 : 박찬익
학번 : 20180579 / 20180180
학과 : 컴퓨터공학과 / 컴퓨터공학과
이름 : 정혜일 / 최검기
POVIS ID : hyelie / cgg7777

1. Analysis on current Pintos system

1.1. process execution procecure

1) naive pintos - thread_exit()

naive pintos에서 process를 종료시키는 process_exit 함수는 아래와 같이 구현되어 있다. 현재 thread의 pointer를 넘겨받은 후, 해당 pointer의 pagedir 값을 받아 destroy 함수를 사용하고, 포인터 내부의 값들을 초기화하는 과정을 거치게 된다. 따라서 1번의 목표를 달성하기 위해서는 해당 함수에 필요한 동작들을 추가하여 구현할 수 있다.

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

  /* Destroy the current process's page directory and switch back
     to the kernel-only page directory. */
  pd = cur->pagedir;
  if (pd != NULL)
    {
      /* Correct ordering here is crucial.  We must set
         cur->pagedir to NULL before switching page directories,
         so that a timer interrupt can't switch back to the
         process page directory.  We must activate the base page
         directory before destroying the process's page
         directory, or our active page directory will be one
         that's been freed (and cleared). */
      cur->pagedir = NULL;
      pagedir_activate (NULL);
      pagedir_destroy (pd);
    }
}
```

그림 1. src/userprog/process.c

2) naive pintos - thread_execute()

위 함수는 user program을 실행시키는 thread를 start시키는 함수로, filename을 인자로 thread_create() 함수를 실행시키면서 동작한다. 2번의 argument parssing을 달성해야 하는 부분이 이 part로, file_name대신 parssing된 정보를 바탕으로 thread가 create 될 수 있게 만드는 것을 목표로 한다.

```
/* Starts a new thread running a user program loaded from
   FILENAME.  The new thread may be scheduled (and may even exit)
   before process_execute() returns.  Returns the new process's
   thread id, or TID_ERROR if the thread cannot be created. */
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);

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

그림 2. src/userprog/process.c

3) naive pintos - Denying Writes

실행 중인 프로그램의 데이터가 write로 인해 modified 되는 경우, 사용자가 예상하지 못하는 에러나 결과 값을 출력하게 될 위험이 매우 높다. 따라서 operating system은 실행중인 유저의 program에 대한 write를 disable 하게 만들어야 하는데, naive pintos에는 이 부분이 구현되어 있지 않다. 단, 아래과 같이 write를 disable하고, enable하게 만드는 함수 자체는 구현이 되어 있다.

```
8 void
9 file_deny_write (struct file *file)
0 {
1   ASSERT (file != NULL);
2   if (!file->deny_write)
3     {
4       file->deny_write = true;
5       inode_deny_write (file->inode);
6     }
7 }
8
```

그림 3. src/filesys.file.c

```
32 void
33 file_allow_write (struct file *file)
34 {
35   ASSERT (file != NULL);
36   if (file->deny_write)
37     {
38       file->deny_write = false;
39       inode_allow_write (file->inode);
40     }
41 }
42
```

그림 4. src/filesys.file.c

1.2. system call procedure

1) naive pintos - syscall handler

naive pintos의 system call은 아무것도 구현되어 있지 않다. 강의 중 배운 system call은 interrupt vector table로 들어가 interrupt의 종류에 따라 어떤 처리를 할지 고르는데, 현재 pintos는 syscall.c의 syscall_handler() 함수를 호출한다. 그러나 아래 그림과 같이 system call이라는 메시지만 출력하고 thread를 종료시킨다.

```
15 static void
16 syscall_handler (struct intr_frame *f UNUSED)
17 {
18   printf ("system call!\n");
19   thread_exit ();
20 }
```

그림 5. src/userprog/syscall.c

이후 halt, exit, wait, exec 등의 system call나 create, remove 등의 file manipulation call이 모두 구현되어 있지 않기 때문에 이들에 대한 구현을 추가해야 한다. src/devices/shutdown.h, src/threads/thread.h, /src/filesys/filesys.h 등에 필요한 기본 함수들이 정의되어 있으니 이들을 이용하면 된다.

2) naive pintos - process hierarchy

naive pintos의 process는 1개의 thread로 이루어져 있으며, 각 thread 간에는 hierarchy가 존재하지 않는다. 그래서 parent process가 child process의 종료를 기다리지 않고 먼저 종료되어 버리면 생기는 issue 중 하나인 zombie process, 또는 child process의 실행 전에 parent process가 종료되어 버리면 child process가 실행이 되지 않기 때문에 여러 문제가 생길 수 있다.

기존의 pintos의 경우, 아래 그림과 같이 src/threads/init.c 내의 run_task를 실행하면 내부에 있는 process_wait(process_execute(argv))로 argv로 받은 process를 process_execute() 함수는 ready list에 thread를 넣는다. 이후 pintos 내부의 thread scheduling으로 thread를 실행해야 하는데, process_wait 함수가 -1을 리턴하기 때문에 pintos가 바로 종료된다. 즉 user program이 어떠한 경우에도 실행되지 않는 문제가 있다.

```
static void
run_task (char **argv)
{
  const char *task = argv[1];

  printf ("Executing '%s':\n", task);
#ifdef USERPROG
  process_wait (process_execute (task));
#else
  run_test (task);
#endif
  printf ("Execution of '%s' complete.\n", task);
}
```

그림 6. src/threads/init.c

```
int
process_wait (tid_t child_tid UNUSED)
{
  return -1;
}
```

그림 7. src/userprog/process.c

1.3 File system

1) naive pintos - file descriptor

file descriptor가 필요하다. 현재 naive pintos의 thread 내에는 어떤 file을 관리하는지 나와 있지 않기 때문이다. create(), remove(), open(), filesize(), read(), write(), seek(), tell(), close() 총 9개의 file 관련 system call 중 create(), remove()는 file의 이름만 있으면 되지만 filesize나 read, write 등을 각 thread에서 이용하려면 어떤 thread가 어떤 file을 관리하는지에 대한 정보가 있어야 하기 때문에 system이 file로 접근하기 위한 중간자인 file descriptor를 구현해야만 한다. 현재의 naive pintos는 file descriptor가 없어 file에 접근 및 수정할 수 없다.

2) file synchronization

file에 대해 synchronization을 처리해 주어야 한다. 각 file에 대해 동시에 2개 이상의 process가 접근해 file이 동시에 2개의 process에서 열려 있는 경우 file에 read/write를 할 때 문제가 발생할 수 있다. 그렇기 때문에 lock 또는 semaphore를 이용해 synchronization을 처리해야 하는데, 이 경우에는 file은 오직 1개의 process에서만 열려 있어야 하므로 lock을 사용하는 것이 적절하다고 판단했다. src/threads/synch.h 내의 lock_init(), lock_acquire(), lock_release() 함수를 이용해 lock을 관리할 수 있다. pintos는 1번에 1개의 thread만 실행되므로 file에 read/write를 할 때 lock acquire / release를 해서 file에 접근하기 전에 lock을 얻어야만 접근할 수 있도록 하면 될 것이다.

lock에 관련해서는 pintos project 1에서 lock을 이미 다뤄 보았으므로 이들을 적절히 사용하면 될 것이다.

2. Solutions for each requirement

2.1. process termination message

간단하게 함수의 마지막 부분에 guideline에서 제시하는 다음과 같은 print문을 추가하여 문제를 해결할 수 있다.

print("%s: exit(%d)₩n", process_name, exit_code)

단, kernel thread 종료시에는 출력되지 않아야 하므로 이와 관련하여 처리를 진행해주어야 한다. TCB정보를 활용하거나, 구분을 위한 variable을 추가하여 해결할 수 있을 것으로 예상된다.

2.2. argument passing

guide line에는 다음과 같이 명시되어 있다.

> **1. *Argument Passing***
>
> *Add codes to support argument passing for the function process_execute(). Instead of taking a program file name as its only argument, process_execute() should split words using spaces and identify program name and its arguments separately. For instance, process_execute("grep foo bar") shoud run grep with two arguments foo and bar.*

따라서, thread_create의 인자로 file_name을 주는 것이 아니라, command line을 parsing하여 token의 형태로 넘겨주는 구현을 진행해야 한다.

이를 위해 user의 stack에 command line으로부터 입력받은 정보들을 parsing하여 저장하는 함수를 구현해야 한다. Process는 스택의 주소를 이동하며 함수-인자를 차례로 받아 함수가 실행되는 원리를 가지고 있으므로, 구현할 parsing 함수는 command line을 parsing한 후 스택에 차례대로 프로그램의 이름, 인자와 주소들을 담을 수 있도록 한다.

이 후 이 함수를 start_process에서 실행시켜 입력받은 command를 parsing하고, 해당 정보를 바탕으로 execution 함수가 실행될 수 있도록 한다.

2.3. system call

1) system call handler

아래 그림과 같이, syscall-nr.h에 어떤 system call의 종류가 있는지 정의되어 있다. 1.2. - 1)에 있는 syscall_handler() 함수와 아래에 있는 call number와 swich-case문 또는 if-else문을 이용해 각 system call에 따른 올바른 처리를 해 주어야 한다. 각 call에 해당하는 함수는 2), 3)에 설명했다.

```
/* System call numbers. */
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. */
```

그림 8. src/lib/syscall_nr.h

한편, syscall_handler는 intr_frame *f를 parameter로 받는다. 이 f는 frame이며, 함수의 호출 정보를 저장하는 data structure이다. 만약 이 frame이 가리키는 stack, 또는 주소가 user stack이 아닌 경우, kernel stack을 가리킬 경우에는 OS에 치명적인 문제가 생길 수 있고 빈 공간을 가리킬 경우에는 system call을 처리하더라도 return했을 때 이상한 에러가 생길 수 있다. 따라서 먼저 frame에 있는 stack pointer가 user section에 있는지 확인해야 한다. 이 함수의 이름을 isValidFrame()이라고 하겠다.

다음으로, user program에서 kernel로 context switch가 일어나면 현재 register의 값을 user stack 등으로 back-up 해 두어야 한다. 이후 system call 처리가 끝난 후 kernel에서 user program으로 다시 context switch가 일어날 때 원래 user program의 register를 가지고 있어야 하기 때문이다. 이 함수의 이름을 backup_user_register()라고 하겠다.

2) user process manipulation

- halt() : pintos를 종료하는 system call이다.
  src/devices/shutdown.h 내의 shutdown_power_off() 함수 호출해서 pintos 종료하면 된다.

- exit() : 현재 user program 종료하고 kernel로 status 리턴. kernel은 받은 status를 parent에게 넘기는 system call이다.
  pintos의 process는 1개의 thread로 이루어져 있기 때문에, 현재 실행중인 thread를 가져오는 함수, src/threads/thread.h의 thread_current() 함수를 이용해 현재 실행중인 thread를 가져와 src/threads/thread.h의 thread_exit() 함수를 이용해 thread를 종료해야 한다.

- exec() : cmd_line에 있는 프로그램 실행하고 new process id 리턴함. 실패하면 pid로 -1 리턴. synchronization은 만족되어야 한다.
  1.2. - 2)에 언급되었듯 각 process 간의 hierarchy가 존재해야 한다. 그러나 아래 그림과 같이 기존의 naive pintos의 thread structure 내에는 그러한 자료구조가 없다. 따라서 기존 thread structure에 child thread를 담을 수 있는 새로운 list를 추가해야 한다. 추가한 list는 src/threads/thread.c 내의 init_thread() 함수에서 list_init() 함수를 이용해 list를 초기화를 해 주어야 할 것이다. child thread의 list를 child_list라고 하겠다.

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

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

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */


    long long sleep_tick;
    // time for sleep current thread


    // for donation
    int original_priority;            // for clear priority
    struct lock *lock_waiting;        // a lock's address which this thread wait
    struct list donated_list;         // thread list which donate to this thread
    struct list_elem donated_list_elem;       // for donated_list

    // for mlfqs
    int nice;
    int recent_cpu;
  };
```

그림 9. src/threads/thread.h

만약 src/threads/thread.c의 thread_create() 함수가 실행될 때, 어떤 process가 해당 thread를 create했는지에 대한 정보를 저장해야만 한다. call한 process의 thread에 새로 만든 thread를 child list에 추가해야 한다.

한편, child process program이 모두 load될 때까지 parent process는 대기해야 한다. 그렇지 않으면 1.2. - 2)에 언급했듯 child process가 만들어지기 전에 parent process가 종료될 수 있기 때문이다. 이는 lock 또는 semaphore를 이용해 해결할 수 있는데, 이 경우에는 child process가 1개가 아닐 수 있기 때문에 semaphore를 사용하는 것이 더 낫다고 판단했다. 추가한 semaphore들 또한 thread_create() 함수에서 sema_init() 함수를 이용해 초기화해야 할 것이다. 이후 child process가 만들어지기 전에는 sema_down()을 이용해 대기하고, child process가 만들어 진 이후 sema_up을 이용해 parent process를 다시 실행시켜야 할 것이다.

- wait() : pid로 주어진 child process가 종료하길 기다린다.

naive pintos는 child process가 종료되기 전에 parent process를 모두 종료한다. 그러나 이 경우 1.2. - 2)에 언급되었듯 문제가 생길 수 있기 때문에 child process의 종료를 모두 기다려야 할 것이다. thread structure 내부에 child list를 넣어두었기 때문에 child_list의 모든 child process가 종료되었을 때 parent process를 종료할 수 있을 것이다. 말인즉슨 parent process의 종료는 모든 child process의 종료까지 기다려야 한다는 것인데 이 또한 exec()와 유사하게 semaphore를 이용해 구현할 수 있다. child process가 모두 종료되어야 sema_up() 함수를 이용해 parent process가 terminate할 수 있게 해야 할 것이다. 이를 위해서 src/thread/thread.c의 thread_exit()함수를 수정해야 한다.

3) file manipulation

filesys/filesys.c.와 file.c에 기본적인 file system function은 구현되어 있다. 이를 이용해 create(), remove(), open(), filesize(), read(), write(), seek(), tell(), close()를 구현하면 된다.

먼저 각 process마다 다른 file을 관리할 수 있으므로 thread struct 내부에 file descriptor라는 배열을 하나 가진다. 그리고 src/threads/thread.c - thread_create() 함수에서 이 배열을 초기화해야 한다. 한편, file을 추가하거나 삭제할 때마다 모든 file desciptor 배열을 탐색할 수는 없기 때문에 file descriptor의 최대값을 thread에 따로 추가한다.

또 process 종료가 일어나면 모든 file을 close해야 하므로 src/userprog/process.c - process_exit()함수에서 file descriptor를 모두 정리(모든 파일 close)해주는 기능을 추가해야 할 것이다.

- create() : 인자로 받은 file의 이름과 file size를 초기화한다. 성공하면 true, 아니면 false 리턴.

기존에 있는 /src/filesys/filesys.h - filesys_create()를 이용해 인자로 받은 file name과 fize size에 해당하는 file을 생성하면 된다.

- remove() : 인자로 받은 file을 삭제한다. 성공하면 true, 아니면 false 리턴. file이 열려있든 닫혀있든 삭제되어야 하지만 삭제하는 것이 file close를 하는 것은 아니다.

기존에 있는 /src/filesys/filesys.h - filesys_remove()를 이용해 file 이름에 해당하는 file을 제거한다.

- open() : 인자로 받은 file을 open한다. 실패하면 -1, 성공하면 file descriptor에서 사용할 숫자를 리턴한다. file descriptor 숫자 0, 1은 이미 할당되어 있다.

만약 file open에 성공하면 해당 thread의 file descriptor table에 file을 추가하고 file descriptor max value를 1 증가해야 한다.

- filesize() : 인자로 받은 fd에 해당하는 opened file의 size를 리턴한다.

인자로 받은 fd값을 이용해서 현재 thread의 file 목록을 검사한다. fd 값이 file descriptor 내에 존재한다면 opened file이므로 해당 file 객체의 length를 리턴하면 될 것이고, 아닌 경우 -1을 리턴하면 된다.

- read() : fd에 해당하는 열린 파일의 byte 크기를 읽고 리턴, 실패하면 -1로 리턴한다. 만약 fd가 0이라면 input_getc()를 이용해 읽어야 한다.

인자로 받는 fd가 0인 경우, input_getc()를 이용해 키보드로 입력받아 처리해야 한다. 아닌 경우 fd에 해당하는 file object를 리턴해야 한다.

한편 1.3. - 2)에 언급했듯 여러 file에 동시에 read를 하면 문제가 발생할 수 있기 때문에 lock을 이용해, read 전후에 lock_acquire(), lock_release() 함수를 이용해 critical section을 만들면 될 것이다.

- write() : fd에 해당하는 file의 buffet에 size byte를 적는다. byte가 실제로 적혔다면 해당 숫자를 리턴한다. 만약 fd가 1이라면 console로 작성되어야 한다. putbuf() 함수를 이용한다.

  fd가 1인 경우, putbuf() 함수를 이용해 console 창에 파일을 처리해 주어야 한다. 아닌 경우 file descriptor에서 fd에 해당하는 file에 값을 write해야 할 것이다.

  read()와 마찬가지로 여러 file에 동시에 write를 하면 문제가 발생할 수 있기 때문에 lock을 이용해, write 전후에 lock_acquire(), lock_release() 함수를 이용해 critical section을 만들면 될 것이다.

- seek() : 인자로 받은 fd에 해당하는 open file의 position의 next byte를 read/written으로 변경한다.

  인자로 받은 fd로 file descriptor 내부에 있는 file에 접근하고 해당 file의 position 만큼 이동해야 한다.

- tell() : open fild fd에 해당하는 read/written된 next byte position을 리턴한다.

  인자로 받은 fd로 file descriptor 내부에 있는 file에 접근하고 해당 file의 position 을 리턴해야 한다.

- close() : fd에 해당하는 파일을 삭제한다. file descriptor도 삭제한다.

  만약 file close에 성공하면 해당 thread의 file descriptor table에 file을 삭제하고 file descriptor max value를 1 감소시켜야 한다.

2.4. denying writes to executables

file에 대한 write 권한을 제한하는 함수는 위에서 살펴본 바와 같이 이미 정의되어 있으므로, 해당 함수를 사용하기 위한 흐름을 구현해주면 문제를 해결할 수 있다.

우선 아래와 같은 thread 구조체에 open된 file의 정보를 저장하기 위한 variable을 새로 정의한다. 다음으로, file을 직접적으로 open하는 함수인 load에 위에서 살펴본 deny 함수를 활용하여 file에 대한 write를 disable하게 설정한다.

```
   blocked state is on a semaphore wait list. */
struct thread
 {
   /* Owned by thread.c. */
   tid_t tid;                          /* Thread identifier. */
   enum thread_status status;          /* Thread state. */
   char name[16];                      /* Name (for debugging purposes). */
   uint8_t *stack;                     /* Saved stack pointer. */
   int priority;                       /* Priority. */
   struct list_elem allelem;           /* List element for all threads list. */

   /* Shared between thread.c and synch.c. */
   struct list_elem elem;              /* List element. */

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

   /* Owned by thread.c. */
   unsigned magic;                     /* Detects stack overflow. */


   long long sleep_tick;
   // time for sleep current thread


   // for donation
   int original_priority;            // for clear priority
   struct lock *lock_waiting;        // a lock's address which this thread wait
   struct list donated_list;         // thread list which donate to this thread
   struct list_elem donated_list_elem;       // for donated_list

   // for mlfqs
   int nice;
   int recent_cpu;
 };

```

그림 10. src/threads/thread.h

```
bool
load (const char *file_name, void (**eip) (void), void **esp)
{
  struct thread *t = thread_current ();
  struct Elf32_Ehdr ehdr;
  struct file *file = NULL;
  off_t file_ofs;
  bool success = false;
  int i;

  /* Allocate and activate page directory. */
  t->pagedir = pagedir_create ();
  if (t->pagedir == NULL)
    goto done;
  process_activate ();

  /* Open executable file. */
  file = filesys_open (file_name);
  if (file == NULL)
    {
      printf ("load: %s: open failed\n", file_name);
      goto done;
    }

  /* Read and verify executable header. */
  if (file_read (file, &ehdr, sizeof ehdr) != sizeof ehdr
      || memcmp (ehdr.e_ident, "\177ELF\1\1\1", 7)
      || ehdr.e_type != 2
      || ehdr.e_machine != 3
      || ehdr.e_version != 1
      || ehdr.e_phentsize != sizeof (struct Elf32_Phdr)
      || ehdr.e_phnum > 1024)
    {
      printf ("load: %s: error loading executable\n", file_name);
      goto done;
    }

```

그림 11. src/userprog/process.c

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

  /* Destroy the current process's page directory and switch back
     to the kernel-only page directory. */
  pd = cur->pagedir;
  if (pd != NULL)
    {
      /* Correct ordering here is crucial.  We must set
         cur->pagedir to NULL before switching page directories,
         so that a timer interrupt can't switch back to the
         process page directory.  We must activate the base page
         directory before destroying the process's page
         directory, or our active page directory will be one
         that's been freed (and cleared). */
      cur->pagedir = NULL;
      pagedir_activate (NULL);
      pagedir_destroy (pd);
    }
}
```

그림 12. src/userprog/process.c

마지막으로, process가 exit하기 전 파일을 닫고, 다시 file이 writable하도록 변경하는 과정을 거치면 문제를 해결할 수 있다.