운영체제 Project #2 Team05 보고서

박병준, 이동우, 정태호, 정현진

1. Process parent, child management design

Process가 처음 생성될 때, 우선 생성된 프로세스가 첫 프로세스인지를 확인한 뒤 그 결과에 따라 다르게 진행한다. 생성하려는 프로세스가 첫 프로세스인지의 여부는 t == initial_thread로 확인한다. 확인 결과가 첫 프로세스인 경우에는 생성하는 프로세스의 부모를 NULL로 설정한다. 만약 첫 프로세스가 아닌 경우에는 thread_current()로 확인한 현재의 프로세스를 부모 프로세스로 설정하고, 이에 관한 정보는 list_push_back을 통하여 현재 thread의 child 목록에 지금 생성하려는 프로세스의 child 목록도 추가한다. 이에 해당하는 부분의 코드는 아래와 같다.

```
void process_create(struct thread *t) {
        struct thread *cur = thread_current();
        enum intr_level old_level;

        old_level = intr_disable();

        // initialize semaphore
        sema_init(&t->sema_wait, 0);
        sema_init(&t->sema_exit, 0);

        if(t == initial_thread)
                t->parent = NULL;
        else {
                t->parent = cur;
                list_push_back(&cur->child_list, &t->child_elem);
        }

        t->exit_status = -1;

        intr_set_level(old_level);
}
```

process_create 함수는 thread_init/create 함수에서 호출되는데 이 역시 첫 프로세스인지의 여부에 따라 다음과 같이 경우가 나눠진다.

- thread_init -> process_create(initial_thread)
- thread_create -> process_create(t)

```
void
thread_init (void)
{
  ASSERT (intr_get_level () == INTR_OFF);

  lock_init (&tid_lock);
  list_init (&ready_list);
  list_init (&wait_list);
  list_init (&all_list);

#ifdef USERPROG
        process_init();
#endif
```

```
tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
  struct thread *t;
  struct kernel_thread_frame *kf;
  struct switch_entry_frame *ef;
  struct switch_threads_frame *sf;
  tid_t tid;
  // omitted part
#ifdef USERPROG
        process_create(t);
#endif
```

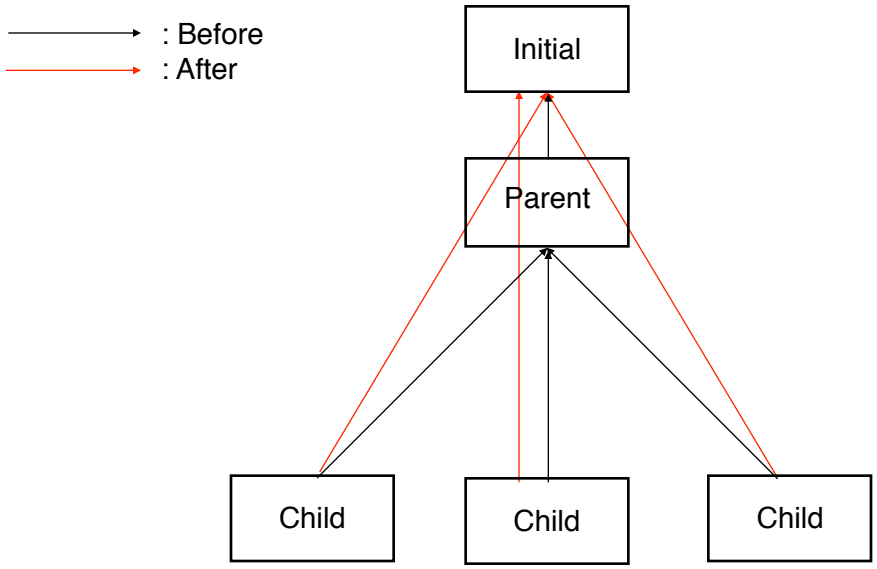
거꾸로 생성된 프로세스들을 종료시킬 때에는 우선 현재 프로세스에서 열려있는 모든 파일을 다 닫고 lock도 해제한 후에 현재의 프로세스에 child가 있는 경우에는 이들의 부모를 첫 프로세스로 설정한다. 프로세스의 child에서 parent를 재설정하는 것을 set_parent_init_thread라는 함수로 따로 구분지어서 구현했는데, 이 함수가 호출된 부분과 구현된 부분은 다음과 같다.

```
void
process_exit (void)
{
  struct thread *cur = thread_current ();
  uint32_t *pd;
      printf("%s: exit(%d)\n", cur->name, cur->exit_status);
      struct file_elem *fe;
      struct list_elem *e;
      struct thread *t;
      /// Close all files.
      while(!list_empty(&cur->open_files)) {
            e = list_pop_front(&cur->open_files);
            fe = list_entry(e, struct file_elem, elem);
            if(fe && fe->file) {
                  file_close(fe->file);
                  free(fe);
            }
      }
      // Release all locks
      for(e = list_begin(&cur->lock_list); e != list_end(&cur->lock_list); e = list_next(e)) {
            struct lock *l = list_entry(e, struct lock, elem);
            lock_release(l);
      }
      // Make parent of curr->child to initial_thread.
      set_parent_init_thread();
```

```
void set_parent_init_thread(void) {
      struct list_elem *e;
      struct thread *t;
      struct thread *cur = thread_current();
      // Set parent of children of current to initial_thread.
      for(e = list_begin(&cur->child_list); e != list_end(&cur->child_list); e = list_next(e)) {
            t = list_entry(e, struct thread, child_elem);
            t->parent = initial_thread;
            sema_up(&t->sema_wait);
      }
}
```

set_parent_init_thread는 현재 thread의 child list의 시작점에서부터 마지막 원소까지 하나씩 거쳐가면서 parent를 initial thread로 설정하는 것을 확인할 수 있다. 각 함수들의 호출 과정은 아래로 정리할 수 있다.

- thread_exit -> process_exit -> set_parent_init_thread



2. Process wait-exit design

프로세스의 wait과 exit의 과정을 살펴보자. 과정을 간단하게 그림으로 나타내면 다음과 같다.

부모(P)와 자식(C)이 있을 때, 부모가 exec을 실행하면 process_execute() 함수에 의해 C가 P의 자식으로 생성된다. 이 때 부모가 C에 대해 wait을 호출하면 P는 block되어 C가 exit할 때까지 기다리게 된다. 그 뒤 C가 exit하게 되면 기다리고 있던 P는 C의 exit status를 받고, 다시 작업을 할 수 있게 된다.

우리 조는 이 과정을 두 개의 semaphore를 이용해 구현했다. 두 semaphore는 자식이 exit할 때 사용할 sema_exit과 부모가 wait할 때 사용할 sema_wait으로, process_create() 함수에서 다음과 같이 0으로 초기화되어 semaphore를 획득하려고 할 때 block이 될 수 있게 하였다.

```
234   // initialize semaphore
235   sema_init(&t->sema_wait, 0);
236   sema_init(&t->sema_exit, 0);
```

Wait과 exit를 하는 흐름을 따라가보자. 부모가 wait을 호출할 경우, sema_down(sema_exit)을 요청해 자식이 exit를 하면서 semaphore를 up시켜줄 때까지 기다리게 된다. 이제 자식은 명령들을 수행하다가 exit를 하게 되면서 exit할 때 필요한 여러가지 처리들을 해준 뒤, 부모가 작업을 수행할 수 있도록 sema_up(sema_exit)을 해준다. 이 때 자식이 바로 종료되어 버리면 부모가 exit status를 얻기 전에 종료가 될 수 있다. 따라서 부모가 필요한 작업을 모두 마칠 때까지 기다리도록 자식이 기다릴 필요가 있다. 이를 자식에서 sema_down(sema_wait)을 요청함으로써 구현하였다. 이제 다시 부모로 넘어와서, 자식의 exit status를 받고, 자식이 종료될 수 있도록 sema_up(sema_wait)을 수행해준 뒤 exit status를 return해준다. 자식은 semaphore를 얻었으므로 남은 작업들을 다시 수행한 뒤 종료할 수 있게 된다.

이제 특수한 경우에 대해 생각해보자. 먼저 같은 자식에 대해 여러 번의 wait를 할 경우이다. 이 경우, 첫 번째 wait는 정상적으로 자식의 exit status를 반환하여야 하고 두 번째부터는 이미 종료된 자식을 wait하는 것과 같기 때문에 error로 처리하여 -1을 반환하여야 한다. 이 때 자식이 종료가 된 후 두 번째 wait이 실행이 된다면 문제가 없지만(child를 못 찾아 -1 return) 자식이 종료가 되기 전에 두 번째 wait이 실행될 수가 있는데, 그럴 경우 문제가 생기게 된다. 이 경우를 방지하기 위해 process_wait에서 sema_exit을 얻고 난 후, return할 자식의 exit status를 저장한 다음에 자식의 exit status를 -1로 바꿔 주었다. 그 뒤 wait에서 직접 sema_up(sema_exit)을 해 주어 다음 wait이 실행될 수 있게 하여 2번째부터의 wait이 -1의 값을 return하게 하여 error 처리를 하게 된다. 이 경우는 etl에 있던 B의 경우에 속한다. 이 외에도 A의 경우는 pid가 자식 프로세스가 아니거나, 자식 관계가 끊어진 경우인데 그 경우는 wait의 첫 부분에서 자식을 못 찾기 때문에 if문에 걸려 -1로 바로 return이 되게 된다.

이 과정들은 process_wait과 process_exit에서 구현했다. 이 때 exit의 경우 시스템 콜에서 exit이 호출 될 때 syscall_exit() -> thread_exit() -> process_exit() 순으로 호출이 되어 process와 thread의 종료 작업을 모두 처리할 수 있도록 했다. Wait과 exit을 구현한 코드는 각각 다음과 같다.

```c
int
process_wait (tid_t child_tid)
{
  struct thread* target = find_thread_tid(thread_current(), child_tid);
  if(target==NULL)  return -1; //if there is no matched child, return -1

  // wait for child's exit
  sema_down(&target->sema_exit);
  int exit_status = target->exit_status;

  // These two lines are for multi wait
  target->exit_status = -1;
  sema_up(&target->sema_exit);

  // Now up child's wait semaphore
  sema_up(&target->sema_wait);

  return exit_status;
}
```

```c
  printf("%s: exit(%d)\n", cur->name, cur->exit_status);
  /// Close all files
  while(!list_empty(&cur->open_files)) {
    e = list_pop_front(&cur->open_files);
    fe = list_entry(e, struct file_elem, elem);
    if(fe && fe->file) {
      file_close(fe->file);
      free(fe);
    }
  }
  // Release all locks
  for(e = list_begin(&cur->lock_list); e != list_end(&cur->lock_list); e = list_next(e)) {
    struct lock *l = list_entry(e, struct lock, elem);
    lock_release(l);
  }
  // Make parent of curr->child to initial_thread
  set_parent_init_thread();
  /* 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);
    }
  // If there is open file, then close.
  if(cur->open_file) {
    file_close(cur->open_file);
  }
  if(cur->parent) {
    // Unblock parent
    sema_up(&cur->sema_exit);
    // Wait parent to up semaphore.
    sema_down(&cur->sema_wait);
    list_remove(&cur->child_elem);
  }
}
```

3. File descriptor management design

```
struct file_elem {
  int fd;                 /* file descriptor */
  struct file *file;      /* file pointer */
  struct list_elem elem;  /* list element */
};
```

우선 File Element structure의 구조는 위와 같이 file descriptor와 file pointer로 구성되어 있다. list_elem은 위 구조체를 리스트로 사용할 수 있게 하기 위해 추가하였다. File System에 관련된 System call중에서, file open, create, remove 등은 file name을 인자로 받으며, 나머지 read, write, seek, filesize, tell, close등은 file desciptor를 인자로 사용한다. file descriptor는 현재 수행 중인 프로세스의 file descriptor table의 number이며, 해당 number를 이용하여 현재 열려있는 파일에 접근 할 수 있도록 한다.

1) File Open

   우선 filesys/file.c에 있는 Filesys_open() 함수를 이용해 file을 open한다. 그 뒤 file descriptor를 정해준다. Fd는 기존에 열린 파일이 없을 경우 3을 주고(0: stdin, 1: stdout, 2: stderr), 있는 경우엔 마지막으로 열린 파일의 fd에 1을 더한 값으로 정해준다. 그 뒤 위에서 정의한 file_elem을 위한 메모리를 할당해주고, fiel_elem 구조체의 구성 요소들을 정의해준다. 마지막으로 현재 열린 file의 list인 open_files 에 파일을 넣어주고 fd를 return한다.

2) File Close ->

```
void
file_close (struct file *file)
{
  if (file != NULL)
    {
      file_allow_write (file);
      inode_close (file->inode);
      free (file);
    }
}
```

   Syscall_close는 우선 file desciptor를 이용하여 위에서 선언한 file_elem를 찾은 후에, file_close 함수를 이용하여 해당 파일의 쓰기 허용을 하고, inode를 close한 후, file pointer에 할당된 메모리를 free해준다. 그 후, file_element list에서 해당 파일 element를 제거 한 후, 마찬가지로 file element에 할당된 메모리를 free해준다. file descriptor로 file_elem를 찾는 과정은, 현재 thread에 open file list에서 loop를 돌려서 file descriptor가 같은 file_elem를 찾아 return하는 함수를 별도로 추가한다.

3) File Remove -> file_name을 인자로 받아, filesys_remove()함수를 호출하여 파일을 삭제한다.

4) File Create -> file_name을 인자로 전달 받아, filesys_create()함수를 호출하여 파일을 생성한다.

5) File Write -> File Write의 경우, 만약 File Descriptor가 stdout인 경우에는, putbuf()를 이용하여 stdout에 출력을 해주고, 그 외의 경우에는 file descriptor를 이용하여 file pointer를 찾아 file_write함수를 이용하여 해당 파일에 써준다.

6) File Read -> File Read 또한 Write와 유사하게, 만약 File Descriptor가 stdin인 경우에 input_getc() 함수를 호출하여, command line을 읽으며, 그 외의 경우에는 file descriptor를 이용하여, file pointer를 찾아 file_read함수를 호출하여 해당 파일을 읽는다.

7) File Size -> file descriptor를 이용하여 파일을 찾은 후, file_length()함수를 호출하여 결과를 리턴한다.

8) Seek -> file_seek 함수를 호출 하여, 인자로 받은 position만큼을 offset으로 둔다. 파일 내에서 읽거나 쓰고자 하는 위치를 변경 할 때 사용된다.

9) Tell -> file_tell 함수를 호출 한 결과를 리턴한다. 현재 파일에서 읽고 있는 위치를 알고자 할 때 사용된다.