Pintos Project 2: User Program (2)

담당 교수 / 분반 : 김영재 교수 / 1분반

이름 / 학번 : 정서영 / 20211589

개발 기간 : 25.10.02 ~ 25.10.29

I. 개발 목표

본 프로젝트는 Pintos 운영체제에서 파일 시스템 관련 시스템 콜을 구현하는 것을 목표로 한다. 이 시스템 콜들을 통해 User Program이 파일을 생성, 삭제, 열고 닫는 등 파일 관련 작업을 수행할 수 있도록 지원한다. 또한, 여러 프로세스가 동시에 파일 시스템에 접근할 때 발생할 수 있는 데이터 손상이나 일관성 문제를 방지하기 위해 락과 세마포어와 같은 동기화 기법을 사용하여 임계 영역을 보호하고 파일 시스템의 안정성을 확보한다.

II. 개발 범위 및 내용

A. 개발 범위

1. File Descriptor

   각 프로세스가 독립적인 파일 디스크립터 테이블을 관리하도록 구현하여 파일 접근의 독립성을 보장한다. 파일 디스크립터는 프로세스가 열어놓은 파일에 대한 참조를 정수 형태로 제공하며, 이를 통해 시스템 콜에서 효율적으로 파일을 식별하고 조작할 수 있다. STDIN(0), STDOUT(1)과 같은 표준 입출력도 일관된 방식으로 처리되며, 파일 디스크립터 2번부터 일반 파일에 할당된다. 이를 통해 한 프로세스의 파일 작업이 다른 프로세스에 영향을 주지 않으며, 각 프로세스는 자신만의 파일 상태를 유지할 수 있다.

2. System Calls

   User Program이 파일 시스템의 기능을 안전하게 활용할 수 있도록 create(), remove(), open(), close(), filesize(), read(), write(), seek(), tell() 시스템 콜을 구현한다. 이를 통해 프로그램은 파일을 생성하고, 삭제하며, 파일을 열어 데이터를 읽고 쓸 수 있고, 파일 내에서 읽기/쓰기 위치를 조정하는 등의 작업을 수행할 수 있다.

3. Synchronization in Filesystem

   파일 시스템의 공유 데이터에 여러 프로세스가 동시에 접근할 때 발생할 수 있는 경쟁 조건을 방지하기 위해 동기화 메커니즘을 구현한다. 현재 Pintos의 파일 시스템은 내부적으로 동기화를 제공하지 않기 때문에, 동시에 여러 프로세스가 파일 시스템 함수를 호출하면 데이터 손상이나 일관성 문제가 발생할 수 있다. 락이나 세마포어를 활용하여 파일 시스템의 원자성을 보장하고, 임계 영역을 보호함으로써 데이터의 일관성과 무결성을 유지한다. 또한 실행 중인 프로그램의 실행 파일이 수정되는 것을 방지하기 위해 파일 거부(file deny write) 메커니즘을 구현하여 시스템의 안정성을 보장한다.

B. 개발 내용

1. File Descriptor

파일 디스크립터의 관리를 위해 배열 기반 자료구조를 사용한다. thread 구조체에 고정 크기의 포인터 배열(fd_table)을 추가하여 각 프로세스가 최대 128개의 파일을 동시에 열 수 있도록 한다. 배열의 인덱스가 파일 디스크립터 번호가 되며, 각 배열 요소는 struct file 포인터를 저장한다. 배열을 선택한 이유는 파일 디스크립터를 통한 파일 접근이 O(1)의 시간 복잡도로 이루어져 효율적이며, 구현이 단순하고 메모리 오버헤드가 적기 때문이다. next_fd 변수를 추가하여 다음에 할당할 파일 디스크립터 번호를 추적하며, 파일을 열 때마다 증가시켜 유일한 파일 디스크립터 번호를 보장한다.

2. System Calls

이번 프로젝트에서 개발할 시스템 콜은 다음과 같다.

- create(); 지정된 이름과 초기 크기로 새로운 파일을 생성하고 성공 여부를 반환한다.
- remove(): 지정된 이름의 파일을 파일 시스템에서 삭제하고 성공 여부를 반환한다.
- open(): 지정된 이름의 파일을 열고 새로운 파일 디스크립터를 반환한다.
- close(): 주어진 파일 디스크립터를 닫고 해당 파일에 대한 연결을 해제한다.
- filesize(): 주어진 파일 디스크립터가 가리키는 파일의 크기를 바이트 단위로 반환한다.
- read(): 파일 디스크립터에서 지정된 크기만큼 데이터를 읽어 버퍼에 저장한다.
- write(): 버퍼의 데이터를 파일 디스크립터에 지정된 크기만큼 쓴다.
- seek(): 파일에서 다음 읽기 또는 쓰기가 수행될 위치를 변경한다.
- tell(): 파일에서 다음 읽기 또는 쓰기가 수행될 현재 위치를 반환한다.

3. Synchronization in Filesystem

파일 시스템 동기화를 위해 Lock과 Semaphore를 활용할 수 있다. Lock 방식에서는 전역 락을 선언하여 모든 파일 시스템 관련 시스템 콜의 시작 부분에서 lock_acquire()를 호출해 락을 획득하도록 하고, 종료 전에 lock_release()를 호출해 락을 해제하도록 한다. 이를 통해 하나의 스레드만 파일 시스템에 접근할 수 있도록 상호 배제를 보장하며, 간단하고 직관적인 구현이 가능하다. Semaphore 방식에서는 초기값이 1인 세마포어를 선언하여 바이너리 세마포어로 사용한다. 파일 시스템 작업 시작 전에 sema_down() 을 호출하여 세마포어를 획득하고, 작업 완료 후 sema_up()을 호출하여 세마포어를 해제한다. 이는 락과 유사하게 동작하지만 세마포어의 일반적인 인터페이스를 사용하며, 필요시 카운팅 세마포어로 확장할 수 있는 유연성을 제공한다.

III. 추진 일정 및 개발 방법

A. 추진 일정

기간	추진 내용
10/2 ~ 10/6	Pintos 소스 코드 분석 및 명세서 검토
10/7 ~ 10/13	File Descriptor 구조 설계 및 구현
10/14 ~ 10/20	System Calls 구현 및 테스트
10/21 ~ 10/25	Synchronization in Filesystem 구현 및 테스트
10/26 ~ 10/29	전체 테스트 검증 및 문서 작성

B. 개발 방법

1. File Descriptor

threads/thread.h의 thread 구조체에 파일 디스크립터 테이블 관련 멤버 변수들을 추가한다. fd_table은 struct file 포인터 배열로 선언하며, 크기는 128로 설정하여 각 프로세스가 최대 128개의 파일을 동시에 열 수 있도록 한다. next_fd는 다음에 할당할 파일 디스크립터 번호를 추적하는 정수 변수이며 초기값은 2로 설정한다. (0과 1은 표준 입출력용, 파일 크기는 핀토스 매뉴얼을 참고해 설정했다.)

threads/thread.c의 init_thread() 함수를 수정하여 추가한 멤버 변수들을 초기화한다. fd_table의 모든 항목을 NULL로 초기화하고, next_fd를 2로 설정한다. 이를 통해 새로 생성되는 모든 스레드가 빈 파일 디스크립터 테이블로 시작할 수 있다.

userprog/syscall.c에 파일 디스크립터 관리를 위한 헬퍼 함수들을 추가한다. allocate_fd() 함수는 새로운 파일을 열 때 빈 파일 디스크립터를 할당하는 역할을 하는 함수로, next_fd를 증가시키면서 fd_table에서 빈 슬롯을 찾아 파일 포인터를 저장하고 파일 디스크립터 번호를 반환하도록 한다. get_file() 함수는 주어진 파일 디스크립터 번호에 해당하는 파일 포인터를 찾아 반환하며, 유효하지 않은 파일 디스크립터인 경우 NULL을 반환한다.

userprog/process.c의 process_exit() 함수를 수정하여 프로세스 종료 시 열려있는 모든 파일을 닫도록 한다. fd_table을 순회하면서 NULL이 아닌 모든 항목에 대해 file_close()를 호출하여

파일을 닫고 자원을 해제한다.

2. System Calls

userprog/syscall.h에 각 파일 시스템 관련 시스템 콜 함수의 프로토타입을 선언하고, userprog/syscall.c에 실제 구현을 작성한다. syscall_handler() 함수의 switch 문에 새로운 시스템 콜 케이스들을 추가하여 각 시스템 콜 번호에 따라 적절한 함수를 호출하도록 한다.

create() 시스템 콜은 파일명과 초기 크기를 인자로 받아 filesys_create() 함수를 호출하여 구현한다. 호출 전 인자와 파일명 포인터의 유효성을 검사하고, 성공 여부를 eax에 저장한다.

remove() 시스템 콜은 파일명을 인자로 받아 filesys_remove() 함수를 호출하여 구현한다. 인자와 파일명 포인터의 유효성을 검사한 후 호출하며, 성공 여부를 eax에 저장한다.

open() 시스템 콜은 파일명을 인자로 받아 filesys_open() 함수로 파일을 연 후, allocate_fd() 함수를 호출하여 새로운 파일 디스크립터를 할당한다. 파일 열기에 실패하거나 파일 디스크립터 할당에 실패하면 -1을 반환하고, 성공하며 할당된 파일 디스크립터 번호를 반환한다.

close() 시스템 콜은 파일 디스크립터를 인자로 받아 get_file() 함수로 해당 파일 포인터를 찾고, file_close()를 호출하여 파일을 닫는다. 이후 fd_table에서 해당 항목을 NULL로 설정하여 파일 디스크립터를 해제한다.

filesize() 시스템 콜은 파일 디스크립터를 인자로 받아 get_file() 함수로 파일 포인터를 찾고, file_length() 함수를 호출하여 파일 크기를 반환한다.

read() 시스템 콜은 파일 디스크립터, 버퍼, 크기를 인자로 받는다. fd가 0이면 표준 입력으로 input_getc()를 반복 호출하여 키보드 입력을 읽고, fd가 2 이상이면 get_file()로 파일 포인터를 찾아 file_read()를 호출한다. 읽은 바이트 수를 반환하며, fd가 1이거나 유효하지 않으면 -1을 반환한다.

write() 시스템 콜은 파일 디스크립터, 버퍼, 크기를 인자로 받는다. fd가 1이면 표준 출력으로 putbut()를 호출하여 콘솔에 출력하고, fd가 2 이상이면 get_file()로 파일 포인터를 찾아 file_write()를 호출한다. 쓴 바이트 수를 반환하며, fd가 0이거나 유효하지 않으면 -1을 반환한다.

seek() 시스템 콜은 파일 디스크립터와 위치를 인자로 받아 get_file()로 파일 포인터를 찾고, file_seek() 함수를 호출하여 파일 포인터의 위치를 변경한다.

tell() 시스템 콜은 파일 디스크립터를 인자로 받아 get_file()로 파일 포인터를 찾고, file_tell() 함수를 호출하여 현재 파일 포인터의 위치를 반환한다.

3. Synchronization in Filesystem

userprog/syscall.c에 전역 락 변수 filesys_lock을 선언하고, syscall_init() 함수에서 lock_init()을 호출하여 초기화한다.

모든 파일 시스템 관련 시스템 콜 함수의 시작 부분에서 lock_acquire(&filysys_lock)을 호출하여 락을 획득하고, 함수 종료 전에 lock_release(&filysys_lock)을 호출하여 락을 해제한다. 이는 create(), remove(), open(), close(), filesize(), read(), write(), seek(), tell() 시스템 콜 모두에 적용한다. read()와 write() 시스템 콜에서 표준 입출력 처리 부분에도 동일하게 락을 적용한다.

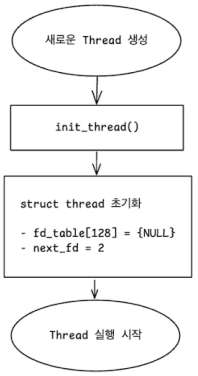
userprog/process.c의 load() 함수를 수정하여 실행 파일을 열 때 file_deny_write()를 호출하고, thread 구조체에 실행 파일 포인터를 저장한다. process_exit()에서는 저장된 실행 파일에 대해 file_allow_write()를 호출한 후 file_close()로 파일을 닫는다.

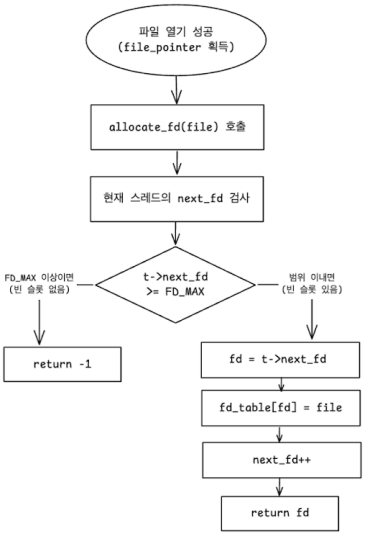
IV. 연구 결과

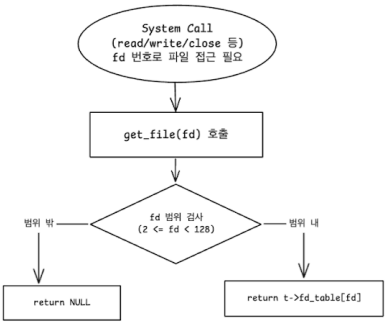
A. Flow Chart

1. File Descriptor

각각 init_thread()에서 thread 구조체의 fd_table, next_fd를 초기화하는 과정과, 이번에 구현한 allocate_fd(), get_fd() 함수에 관한 플로우 차트를 그렸다.

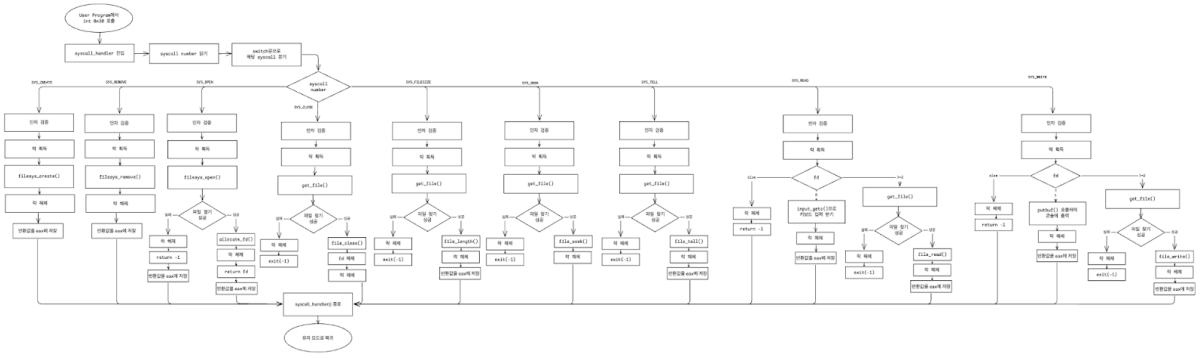






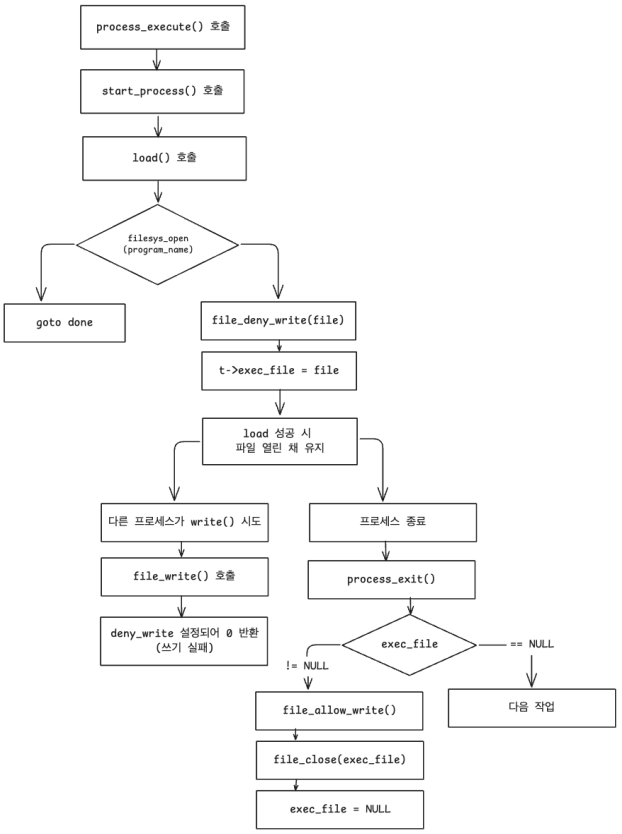
2. System Calls

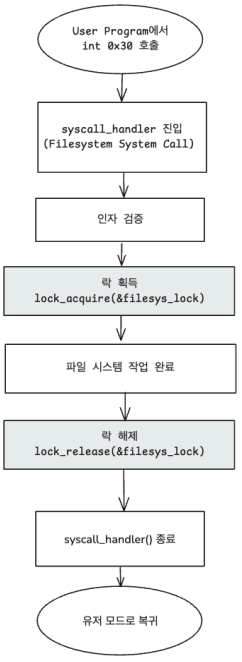
이번에 구현한 파일 시스템 관련 시스템 콜에 관한 플로우 차트를 그렸다.



3. Synchronization in Filesystem

다음은 각각 프로그램 로드 시 실행 파일을 보호하는 과정과 파일 시스템 관련 시스템 콜에서 여러 프로세스의 파일 시스템 접근 보호 과정에 관한 플로우 차트이다.





B. 제작 내용

1. File Descriptor

threads/thread.h의 struct thread에 파일 디스크립터 테이블 관리를 위한 멤버 변수를 추가했다. fd_table은 각 프로세스가 열 수 있는 파일들의 포인터를 저장하는 배열로, 크기를 128로 고정하여 최대 128개의 파일을 동시에 관리할 수 있도록 설계했다. 배열의 인덱스가 곧 파일 디스크립터 번호로 사용되며, 0과 1은 표준 입력과 표준 출력을 위해 사용된다. exec_file은 현재 실행 중인 프로세스의 실행 파일을 가리키며, 실행 중에는 해당 파일에 대한 쓰기를 방지하기 위해 사용된다.

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

    int exit_status; // 프로세스 종료 상태
    struct thread* parent; // 부모 스레드 포인터
    struct list child_list; // 자식 프로세스 리스트
    struct list_elem child_elem; // 부모의 child_list에 연결될 요소
    struct semaphore load_sema; // 로드 성공 여부 동기화
    struct semaphore wait_sema; // 프로세스 종료 동기화
    bool is_waited; // 이미 wait 되었는지 표시

    struct file *fd_table[128]; // 파일 디스크립터 테이블
    int next_fd; // 다음 할당될 fd 번호
    struct file *exec_file; // 실행 중인 파일
#endif
```

threads/thread.c의 init_thread() 함수에서 다음과 같이 초기화를 수행하였다. memset() 함수를 사용하여 128개의 포인터를 한 번에 0(NULL)으로 초기화하는 방식으로 구현했다. 그리고 next_fd를 2부터 시작하여 순차적으로 증가시키며 파일 디스크립터를 할당할 수 있도록 했고, exec_file은 NULL로 초기화했다.

```
#ifdef USERPROG
  t->parent = NULL;
  list_init(&t->child_list);
  sema_init(&t->load_sema, 0);
  sema_init(&t->wait_sema, 0);
  t->is_waited = false;

  memset(t->fd_table, 0, sizeof(t->fd_table));
  t->next_fd = 2;
  t->exec_file = NULL;
#endif
```

userprog/syscall.c에 파일 디스크립터를 관리하는 두 개의 헬퍼 함수(allocate_fd(), get_fd())와 관련 상수를 정의했다. 먼저 파일 디스크립터 관련 상수들을 매크로로 정의했다. FD_MIN은 2,

FD_MAX는 128, STDIN은 0, STDOUT은 1로 설정했다.

```
// 파일 디스크립터 범위 상수
#define FD_MIN 2
#define FD_MAX 128
#define STDIN 0
#define STDOUT 1
```

allocate_fd() 함수는 새로운 파일을 열 때 사용 가능한 파일 디스크립터를 할당한다. 이 함수는 먼저 next_fd가 FD_MAX(128) 이상인지 검사하여 파일 디스크립터 테이블이 가득 찼는지 확인한다. 공간이 있으면 현재 next_fd 값을 파일 디스크립터로 사용하여 fd_table에 파일 포인터를 저장하고, next_fd를 1 증가시킨 후 할당된 파일 디스크립터 번호를 반환한다. 테이블이 가득 찬 경우 -1을 반환하여 할당 실패를 알린다.

```
static int allocate_fd (struct file *file) {
  struct thread *t = thread_current();
  if(t->next_fd >= FD_MAX) {
    return -1;
  }
  int fd = t->next_fd;
  t->fd_table[fd] = file;
  t->next_fd++;
  return fd;
}
```

get_file() 함수는 파일 디스크립터 번호로 파일 포인터를 조회한다. fd가 유효한 범위(FD_MIN(2) 이상 FD_MAX(128) 미만)라면 해당하는 파일 포인터를 반환하고, 표준 입출력(0, 1)이나 범위를 벗어난 값은 NULL을 반환하여 처리한다.

```
static struct file *get_file (int fd) {
  struct thread *t = thread_current();
  if (fd < FD_MIN || fd >= FD_MAX) {
    return NULL;
  }
  return t->fd_table[fd];
}
```

userprog/process.c의 process_exit() 함수 시작 부분에 파일 디스크립터 테이블 정리 코드를 추가했다. 프로세스가 종료될 때 fd 2부터 127까지 순회하면서 NULL이 아닌 모든 파일 포인터에 대해 file_close()를 호출하여 파일을 닫고 파일 시스템 자원을 해제한다. 파일을 닫은 후 해당 슬롯을 명시적으로 NULL로 설정하여 중복 해제를 방지하고 테이블을 정리한다.

```
for (int fd = 2; fd < 128; fd++) {
  if (cur->fd_table[fd] != NULL) {
    file_close(cur->fd_table[fd]);
    cur->fd_table[fd] = NULL;
  }
}
```

2. System Calls

userprog/syscall.h에 다음과 같이 파일 시스템 관련 시스템 콜 함수들의 프로토타입을 선언했다.

```
bool create (const char *file, unsigned initial_size);
bool remove (const char *file);
int open (const char *file);
void close (int fd);
int filesize (int fd);
void seek (int fd, unsigned position);
unsigned tell (int fd);
```

userprog/syscall.c의 syscall_handler() 함수에 파일 시스템 관련 시스템 콜 케이스를 추가했다. 각 시스템 콜은 스택 포인터(esp)를 통해 전달된 인자들의 유효성을 check_valid_uaddr() 함수로 먼저 검사한 후, 해당하는 파일 시스템 함수를 호출하도록 구현했다. 각 시스템 콜은 인자의 개수만큼 esp+1, esp+2 등의 주소를 검증하며, 포인터 타입 인자의 경우 포인터가 가리키는 주소 자체도 추가로 검증한다. 반환값이 있는 시스템 콜은 결과를 f->eax 레지스터에 저장하여 유저 프로그램에 전달한다.

```
case SYS_CREATE:
  check_valid_uaddr (esp + 1);
  check_valid_uaddr (esp + 2);
  check_valid_uaddr ((void *) *(esp + 1));
  f->eax = create ((const char *) *(esp + 1), *(esp + 2));
  break;

case SYS_REMOVE:
  check_valid_uaddr (esp + 1);
  check_valid_uaddr ((void *) *(esp + 1));
  f->eax = remove ((const char *) *(esp + 1));
  break;

case SYS_OPEN:
  check_valid_uaddr (esp + 1);
  check_valid_uaddr ((void *) *(esp + 1));
  f->eax = open ((const char *) *(esp + 1));
  break;

case SYS_CLOSE:
  check_valid_uaddr (esp + 1);
  close (*(esp + 1));
  break;
```

```
    case SYS_FILESIZE:
      check_valid_uaddr (esp + 1);
      f->eax = filesize (*(esp + 1));
      break;

    case SYS_SEEK:
      check_valid_uaddr (esp + 1);
      check_valid_uaddr (esp + 2);
      seek (*(esp + 1), *(esp + 2));
      break;

    case SYS_TELL:
      check_valid_uaddr (esp + 1);
      f->eax = tell (*(esp + 1));
      break;

    default:
      exit (-1);
      break;
  }
}
```

구체적인 각 시스템 콜의 구현 내용은 다음과 같다.

create() 시스템 콜은 파일명과 초기 크기를 받아 새 파일을 생성한다. filesys_create() 함수를 호출하여 실제 파일 생성 작업을 수행하며, 이 함수는 파일명과 초기 크기를 인자로 받아 성공 시 true, 실패 시 false를 반환한다.

```
bool create (const char *file, unsigned initial_size) {
  lock_acquire(&filesys_lock);
  bool result = filesys_create (file, initial_size);
  lock_release(&filesys_lock);
  return result;
}
```

remove() 시스템 콜은 파일명을 받아 파일을 삭제한다. filesys_remove() 함수를 호출하여 파일 시스템에서 파일을 제거하며, 성공 시 true, 실패 시 false를 반환한다.

```
bool remove (const char *file) {
  lock_acquire(&filesys_lock);
  bool result = filesys_remove (file);
  lock_release(&filesys_lock);
  return result;
}
```

open() 시스템 콜은 파일을 열고 파일 디스크립터를 할당한다. filesys_open() 함수로 파일을 열어

파일 포인터를 얻고, 성공하면 앞서 구현한 allocate_fd() 함수를 호출하여 파일 디스크립터를 할당한다. 파일 열기에 실패하거나 파일 디스크립터 할당에 실패하면 -1을 반환하고, 성공하며 할당된 파일 디스크립터 번호를 반환한다.

```
int open (const char *file) {
  lock_acquire(&filesys_lock);
  struct file *f = filesys_open (file);
  if (f == NULL) {
    lock_release(&filesys_lock);
    return -1;
  }
  int fd = allocate_fd (f);
  lock_release(&filesys_lock);
  return fd;
}
```

close() 시스템 콜은 열린 파일을 닫고 파일 디스크립터를 해제한다. 앞서 구현한 get_file() 함수로 파일 디스크립터에 해당하는 파일 포인터를 찾는다. 해당되는 파일을 찾지 못하면 get_file()은 NULL을 반환하며, 이 경우 exit(-1)을 호출하여 프로세스를 종료한다. 해당되는 파일을 찾으면 file_close() 함수로 파일을 닫아 관련 자원을 해제한 후 fd_table의 해당 슬롯을 NULL로 설정하여 파일 디스크립터를 명시적으로 해제한다.

```
void close (int fd) {
  lock_acquire(&filesys_lock);
  struct file *f = get_file (fd);
  if (f == NULL) {
    lock_release(&filesys_lock);
    exit(-1);
  }
  file_close (f);
  thread_current()->fd_table[fd] = NULL;
  lock_release(&filesys_lock);
}
```

filesize() 시스템 콜은 파일의 크기를 바이트 단위로 반환한다. 앞서 구현한 get_file() 함수로 파일 디스크립터에 해당하는 파일 포인터를 찾는다. 해당되는 파일을 찾지 못하면 get_file()은 NULL을 반환하며, 이 경우 exit(-1)을 호출하여 프로세스를 종료한다. 유효한 파일 포인터를 얻으면 file_length() 함수를 호출하여 파일의 전체 크기를 얻고 이 값을 반환한다.

```c
int filesize (int fd) {
  lock_acquire(&filesys_lock);
  struct file *f = get_file (fd);
  if (f == NULL) {
    lock_release(&filesys_lock);
    exit(-1);
  }
  int result = file_length (f);
  lock_release(&filesys_lock);
  return result;
}
```

read() 시스템 콜은 파일이나 표준 입력으로부터 데이터를 읽는다. 파일 디스크립터 값이 STDIN(0)이면 Project1에서 구현한 방식대로 표준 입력으로부터 읽기 위해 input_getc() 함수를 size만큼 반복 호출하여 키보드로부터 한 문자씩 입력을 받아 버퍼에 저장한다. 파일 디스크립터 값이 2 이상이면 get_file() 함수로 먼저 파일 포인터를 얻고, NULL인 경우 잘못된 파일 디스크립터로 판단하여 프로세스를 종료한다. 유효한 파일 디스크립터를 얻으면 file_read() 함수를 호출하여 파일의 현재 위치에서 size 바이트만큼 데이터를 읽어 buffer에 저장하고, 실제로 읽은 바이트 수를 반환한다. fd가 1(표준 출력)이거나 유효하지 않은 범위면 -1을 반환한다.

```c
int read (int fd, void *buffer, unsigned size) {
  lock_acquire(&filesys_lock);
  if (fd == STDIN) {
    uint8_t *buf = (uint8_t *) buffer;
    for (unsigned i = 0; i < size; i++) {
      buf[i] = input_getc ();
    }
    lock_release(&filesys_lock);
    return size;
  }
  if (fd >= 2) {
    struct file *f = get_file(fd);
    if (f == NULL) {
      lock_release(&filesys_lock);
      exit(-1);
    }
    int result = file_read (f, buffer, size);
    lock_release(&filesys_lock);
    return result;
  }
  lock_release(&filesys_lock);
  return -1;
}
```

write() 시스템 콜은 파일이나 표준 입력으로부터 데이터를 쓴다. 파일 디스크립터 값이 STDOUT(1)이면 Project1에서 구현한 방식대로 putbuf() 함수를 호출해 버퍼의 내용을 size 바이트만큼 콘솔에 출력하고, 쓴 바이트 수인 size를 반환한다. 파일 디스크립터 값이 2 이상이면 파일에 쓰기 위해 get_file() 함수로 파일 포인터를 얻고, NULL인 경우 프로세스를 종료한다. 유효한 파일 포인터를 얻으면 file_write() 함수를 호출하여 buffer의 데이터를 파일의 현재 위치에 size 바이트만큼 쓰고, 실제로 쓴 바이트 수를 반환한다. fd가 0(표준 입력)이거나 유효하지 않으면 -1을 반환한다.

```
int write (int fd, const void *buffer, unsigned size) {
  lock_acquire(&filesys_lock);
  if(fd == STDOUT) {
    putbuf(buffer, size);
    lock_release(&filesys_lock);
    return size;
  }
  if (fd >= 2) {
    struct file *f = get_file(fd);
    if (f == NULL) {
      lock_release(&filesys_lock);
      exit(-1);
    }
    int result = file_write (f, buffer, size);
    lock_release(&filesys_lock);
    return result;
  }
  lock_release(&filesys_lock);
  return -1;
}
```

seek() 시스템 콜은 파일 내에서 다음 읽기 또는 쓰기 작업이 수행될 위치를 변경한다. 먼저 get_file() 함수로 파일 디스크립터에 해당하는 파일 포인터를 얻고, NULL이면 프로세스를 종료한다. 유효한 파일 포인터를 얻으면 file_seek() 함수를 호출하여 파일의 내부 포인터를 position 바이트 위치로 이동시킨다.

```
void seek (int fd, unsigned position) {
  lock_acquire(&filesys_lock);
  struct file *f = get_file (fd);
  if (f == NULL) {
    lock_release(&filesys_lock);
    exit(-1);
  }
  file_seek (f, position);
  lock_release(&filesys_lock);
}
```

tell() 시스템 콜은 파일 내에서 다음 읽기 또는 쓰기 작업이 수행될 현재 위치를 바이트 오프셋으로 반환한다. get_file() 함수로 파일 포인터를 얻고, NULL이면 프로세스를 종료한다. 유효한 파일 포인터를 얻으면 file_tell() 함수를 호출하여 파일의 현재 포인터 위치를 얻어 반환한다.

```
unsigned tell (int fd) {
  lock_acquire(&filesys_lock);
  struct file *f = get_file (fd);
  if (f == NULL) {
    lock_release(&filesys_lock);
    exit(-1);
  }
  unsigned result = file_tell (f);
  lock_release(&filesys_lock);
  return result;
}
```

3. Synchronization in Filesystem

먼저 userprog/syscall.c에 전역 락 변수 filesys_lock을 정의하고, syscall_init() 함수에서 lock_init(&filesys_lock)을 호출하여 초기화했다. 그리고, userprog/syscall.h에 extern struct lock filesys_lock;을 선언하여 다른 소스 파일에서도 동일한 락을 사용할 수 있도록 구성했다. extern 키워드는 다른 파일에 정의된 전역 변수를 참조하기 위한 선언으로, 실제 변수는 syscall.c에 정의되어 있지만 여러 파일에서 공유하여 사용할 수 있게 한다.

```
struct lock filesys_lock;

void
syscall_init (void)
{
  lock_init(&filesys_lock);
  intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
}
```

```
// 전역 파일 시스템 락
extern struct lock filesys_lock;
```

모든 파일 시스템 관련 시스템 콜에서는 critical section에 진입하기 전에 lock_acquire() 함수를

호출하여 락을 획득한다. 락을 성공적으로 획득한 프로세스만이 파일 시스템 작업을 수행할 수 있으며, 다른 프로세스들은 락이 해제될 때까지 블록된다. 파일 시스템 작업이 완료되면 lock_release() 함수를 호출하여 락을 해제하고, 대기 중인 다른 프로세스가 락을 획득할 수 있도록 한다. 또한 예외 처리 부분에서 에러가 발생하여 exit(-1)을 호출하거나 조기 반환하는 경우에도 반드시 lock_release()를 먼저 호출하도록 구현했다. 이러한 패턴은 create(), remove(), open(), close(), filesize(), read(), write(), seek(), tell()의 모든 파일 시스템 콜에 일관되게 적용했다.

```
int read (int fd, void *buffer, unsigned size) {
  lock_acquire(&filesys_lock);
  if (fd == STDIN) {
    uint8_t *buf = (uint8_t *) buffer;
    for (unsigned i = 0; i < size; i++) {
      buf[i] = input_getc ();
    }
    lock_release(&filesys_lock);
    return size;
  }
  if (fd >= 2) {
    struct file *f = get_file(fd);
    if (f == NULL) {
      lock_release(&filesys_lock);
      exit(-1);
    }
    int result = file_read (f, buffer, size);
    lock_release(&filesys_lock);
    return result;
  }
  lock_release(&filesys_lock);
  return -1;
}
```

```
int write (int fd, const void *buffer, unsigned size) {
  lock_acquire(&filesys_lock);
  if(fd == 1) {
    putbuf(buffer, size);
    lock_release(&filesys_lock);
    return size;
  }
  if (fd >= 2) {
    struct file *f = get_file(fd);
    if (f == NULL) {
      lock_release(&filesys_lock);
      exit(-1);
    }
    int result = file_write (f, buffer, size);
    lock_release(&filesys_lock);
    return result;
  }
  lock_release(&filesys_lock);
  return -1;
}
```

```c
bool create (const char *file, unsigned initial_size) {
  lock_acquire(&filesys_lock);
  bool result = filesys_create (file, initial_size);
  lock_release(&filesys_lock);
  return result;
}

bool remove (const char *file) {
  lock_acquire(&filesys_lock);
  bool result = filesys_remove (file);
  lock_release(&filesys_lock);
  return result;
}

int open (const char *file) {
  lock_acquire(&filesys_lock);
  struct file *f = filesys_open (file);
  if (f == NULL) {
    lock_release(&filesys_lock);
    return -1;
  }
  int fd = allocate_fd (f);
  lock_release(&filesys_lock);
  return fd;
}
```

```c
void close (int fd) {
  lock_acquire(&filesys_lock);
  struct file *f = get_file (fd);
  if (f == NULL) {
    lock_release(&filesys_lock);
    exit(-1);
  }
  file_close (f);
  thread_current()->fd_table[fd] = NULL;
  lock_release(&filesys_lock);
}

int filesize (int fd) {
  lock_acquire(&filesys_lock);
  struct file *f = get_file (fd);
  if (f == NULL) {
    lock_release(&filesys_lock);
    exit(-1);
  }
  int result = file_length (f);
  lock_release(&filesys_lock);
  return result;
}
```

```c
void seek (int fd, unsigned position) {
  lock_acquire(&filesys_lock);
  struct file *f = get_file (fd);
  if (f == NULL) {
    lock_release(&filesys_lock);
    exit(-1);
  }
  file_seek (f, position);
  lock_release(&filesys_lock);
}

unsigned tell (int fd) {
  lock_acquire(&filesys_lock);
  struct file *f = get_file (fd);
  if (f == NULL) {
    lock_release(&filesys_lock);
    exit(-1);
  }
  unsigned result = file_tell (f);
  lock_release(&filesys_lock);
  return result;
}
```

다음으로, 실행 중인 프로그램의 바이너리 파일이 수정되는 것을 방지하기 위한 read-only executable 기능을 구현했다. userprog/process.c의 load() 함수에서 먼저 lock_acquire 함수로 filesys_lock을 획득한 후 실행 파일을 연다. 파일 열기가 성공하면 file_deny_write() 함수를 호출하여 해당 파일에 대한 쓰기를 금지하고, 이 파일 포인터를 thread 구조체의 exec_file 필드에 저장한다. 이후 ELF 헤더를 읽고 프로그램 세그먼트를 로드하는 과정에서도 락을 유지하여 파일 시스템 작업의 원자성을 보장한다. 각 프로그램 헤더를 처리하는 switch문에서 에러가 발생할 경우 lock_release로 락을 해제한 후 done 레이블로 이동하여 정리 작업을 수행한다. 모든 세그먼트 로딩이 완료되면 lock_release로 락을 해제하고, 스택 설정 등 나머지 작업을 진행한다.

```
/* Open executable file. */
// 락 획득
lock_acquire(&filesys_lock);

file = filesys_open (program_name);
if (file == NULL)
  {
    lock_release(&filesys_lock); // 락 해제
    printf ("load: %s: open failed\n", program_name);
    goto done;
  }

// 실행 중인 파일의 쓰기 금지
file_deny_write(file);
t->exec_file = file;
```

```
    case PT_SHLIB:
      lock_release(&filesys_lock); // 락 해제
      goto done;
    case PT_LOAD:
      if (validate_segment (&phdr, file))
        {
          bool writable = (phdr.p_flags & PF_W) != 0;
          uint32_t file_page = phdr.p_offset & ~PGMASK;
          uint32_t mem_page = phdr.p_vaddr & ~PGMASK;
          uint32_t page_offset = phdr.p_vaddr & PGMASK;
          uint32_t read_bytes, zero_bytes;
          if (phdr.p_filesz > 0)
            {
              /* Normal segment.
                 Read initial part from disk and zero the rest. */
              read_bytes = page_offset + phdr.p_filesz;
              zero_bytes = (ROUND_UP (page_offset + phdr.p_memsz, PGSIZE)
                            - read_bytes);
            }
          else
            {
              /* Entirely zero.
                 Don't read anything from disk. */
              read_bytes = 0;
              zero_bytes = ROUND_UP (page_offset + phdr.p_memsz, PGSIZE);
            }
          if (!load_segment (file, file_page, (void *) mem_page,
                             read_bytes, zero_bytes, writable)) {
            lock_release(&filesys_lock); // 락 해제
            goto done;
          }
        }
      else {
        lock_release(&filesys_lock); // 락 해제
        goto done;
      }
      break;
    }
}

// 락 해제
lock_release(&filesys_lock);
```

이후 프로세스가 종료될 때 process_exit() 함수에서 exec_file이 NULL이 아닌 경우 file_allow_write() 함수를 호출하여 쓰기 금지를 해제하고 file_close()로 파일을 닫는다. 이를 통해 실행 중인 프로그램이 자기 자신의 바이너리를 수정하는 것을 방지한다.

```
if (cur->exec_file != NULL) {
  file_allow_write(cur->exec_file);
  file_close(cur->exec_file);
  cur->exec_file = NULL;
}
```

4. 구현 중 발생한 문제점 및 해결 방안

process.c의 load()에서 done 레이블에서 메모리 누수를 방지하기 위해 자원 관리 과정을 수정했다. 기존엔 done 레이블에서 fn_copy를 해제하지 않아 load 함수가 호출될 때마다 한 페이지씩 메모리가 누수되었다. 이를 해결하기 위해 먼저 fn_copy가 NULL이 아닌 경우 palloc_free_page()로 해제하도록 했다. 그 다음 success가 false인 경우에 파일을 닫도록 조건을 추가했다. 구체적으로 load가 실패했고 file이 NULL이 아닌 경우에만 file_close()를 호출하고 exec_file을 NULL로 설정한다.

```
done:
 /* We arrive here whether the load is successful or not. */
if (fn_copy != NULL)
  palloc_free_page (fn_copy);

if (!success) {
  if (file != NULL) {
    file_close (file);
    t->exec_file = NULL;
  }
}
 return success;
```

다음으로, 테스트 과정에서 커널이 page fault로 종료되는 문제가 있었다. Project1에서 수정한 userprog/exception.c의 page_fault() 함수에서 user 변수가 true이고 not_present 변수가 true인 경우에만 exit(-1)을 호출했다. 즉 유저 모드에서 발생한 page fault이면서 해당 페이지가 존재하지 않는 경우에만 프로세스를 종료시켰다. 그러나 이 조건이 너무 제한적이어서 다른 유형의 잘못된 메모리 접근을 처리하지 못했다.

그래서 이를 해결하기 위해 page_fault() 함수의 조건을 수정했다. 먼저 user가 true인 경우 exit(-1)을 호출해 유저 모드에서 발생한 모든 page_fault를 처리하도록 했다. 그리고 user가

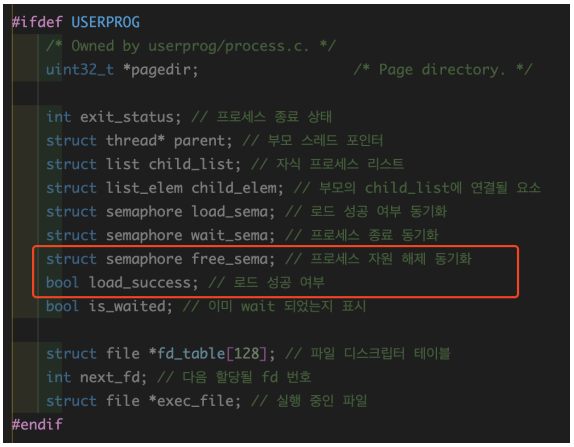
false인 경우에도 is_user_vaddr() 함수로 fault_addr이 유저 공간의 주소인지 확인하는 조건을 추가했다. 시스템 콜 핸들러에서 잘못된 유저 포인터를 참조하면 user는 false이지만 fault_addr은 유저 공간의 주소이므로, 이 경우에도 exit(-1)을 호출하여 프로세스를 안전하게 종료시킨다. 이렇게 수정함으로써 유저 모드에서 직접 발생한 page fault 뿐만 아니라 커널이 유저로부터 전달받은 잘못된 포인터를 역참조할 때 발생하는 page fault도 안전하게 처리할 수 있게 되었다. 결과적으로 커널 패닉 없이 해당 프로세스만 종료되도록 하여 시스템의 안정성을 확보할 수 있었다.

```
// 유저의 모든 page fault
if (user) {
    exit(-1);
}

// 커널이 유저 포인터 역참조 중 page fault (syscall에서 잘못된 유저 포인터 접근)
if (!user && is_user_vaddr(fault_addr)) {
    exit(-1);
}
```

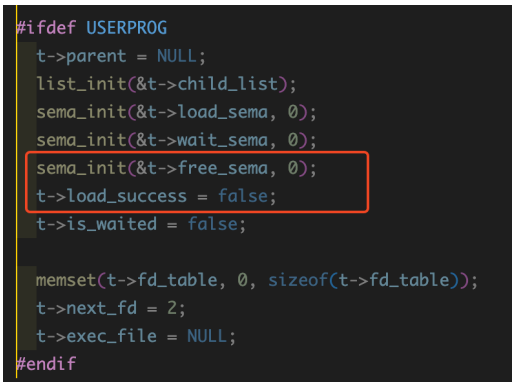
또한, multi-oom 테스트에서 예상보다 적은 수의 프로세스만 생성되는 문제가 있었다. 이 테스트는 재귀적으로 자기 자신을 실행하여 메모리가 부족할 때까지 프로세스를 생성하는데, 예상 깊이 60에 도달해야 하지만 실제로는 52에서 멈췄다. 이는 약 8개 프로세스 분량의 메모리가 누수되고 있음을 의미했다.

원인을 분석한 결과 threads/thread.c의 thread_schedule_tail() 함수에서 종료된 thread의 페이지를 해제할 때 parent가 NULL인 고아 프로세스만 해제하고 있었다. 부모가 있는 프로세스는 페이지가 해제되지 않아 thread 하나당 4KB씩 메모리가 누수되었다. 처음에는 부모가 있는 경우 부모가 페이지를 해제해야 한다고 생각했으나, 실제로는 부모는 자식의 정보만 읽고 페이지 해제는 스케줄러가 담당해야 했다.

Project1에서 일부 구현했던 세마포어 동기화 메커니즘을 수정해 프로세스 간의 동기화를 강화하여 이 문제를 해결했다. 먼저 free_sema 세마포어와 load_success 불리언 멤버를 thread 구조체에 추가하고, init_thread()에서 초기화를 수행했다. free_sema는 특정 조건이 만족될 때까지 자식 프로세스의 자원 해제를 지연시키는 역할을 한다.

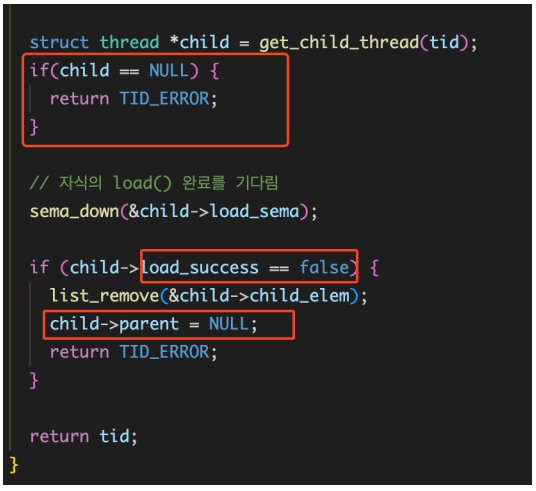


```
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */

    int exit_status; // 프로세스 종료 상태
    struct thread* parent; // 부모 스레드 포인터
    struct list child_list; // 자식 프로세스 리스트
    struct list_elem child_elem; // 부모의 child_list에 연결될 요소
    struct semaphore load_sema; // 로드 성공 여부 동기화
    struct semaphore wait_sema; // 프로세스 종료 동기화
    struct semaphore free_sema; // 프로세스 자원 해제 동기화
    bool load_success; // 로드 성공 여부
    bool is_waited; // 이미 wait 되었는지 표시

    struct file *fd_table[128]; // 파일 디스크립터 테이블
    int next_fd; // 다음 할당될 fd 번호
    struct file *exec_file; // 실행 중인 파일
#endif
```



```
#ifdef USERPROG
  t->parent = NULL;
  list_init(&t->child_list);
  sema_init(&t->load_sema, 0);
  sema_init(&t->wait_sema, 0);
  sema_init(&t->free_sema, 0);
  t->load_success = false;
  t->is_waited = false;

  memset(t->fd_table, 0, sizeof(t->fd_table));
  t->next_fd = 2;
  t->exec_file = NULL;
#endif
```

userprog/process.c의 load() 함수에서 child가 NULL일 때 호출하던 palloc_free_page() 부분을 삭제했다. 또한 자식의 load 완료를 기다린 후 exit_status를 검사하는 대신 load_success로 검사하도록 변경했으며, load_success가 false인 경우에도 palloc_free_page()를 호출하는 대신 child의 parent만 NULL로 설정했다. start_process()에서는 load() 호출 이후 load_success에 그 성공 여부 값을 설정하도록 했다.



```
  struct thread *child = get_child_thread(tid);
  if(child == NULL) {
    return TID_ERROR;
  }

  // 자식의 load() 완료를 기다림
  sema_down(&child->load_sema);

  if (child->load_success == false) {
    list_remove(&child->child_elem);
    child->parent = NULL;
    return TID_ERROR;
  }

  return tid;
}
```

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

  palloc_free_page (file_name);
  cur->load_success = success;
  sema_up(&cur->load_sema);

  /* If load failed, quit. */
  if (!success) {
    cur->exit_status = -1;
    thread_exit ();
  }
```

process_wait()에서 부모가 자식의 exit_status를 읽은 후 fress_sema를 올리고, 자식은 process_exit()에서 free_sema를 내리도록 해서, free_sema를 받은 후에야 페이지 디렉토리를 해제하고 종료하도록 했다. 구체적으로는 process_wait()에서 자식이 종료되어 세마포어가 올라오면 exit_status를 읽고 child_elem을 리스트에서 제거한 후, free_sema에 대해 sema_up을 호출하여 자식이 자원을 해제할 수 있도록 신호를 보낸다. process_exit()에서는 부모가 NULL이 아닌 경우에만 wait_sema를 올려 부모를 깨우고, free_sema에 대해 sema_down을 호출하여 부모가 정보를 읽을 때까지 대기한다.

```c
child = get_child_thread(child_tid);
if(child == NULL || child->is_waited) return -1;

child->is_waited = true;
sema_down(&(child->wait_sema)); // 자식이 종료될 때까지 대기
exit_status = child->exit_status;
list_remove(&(child->child_elem)); // 자식을 child_list에서 제거

sema_up(&(child->free_sema)); // 자식이 자원 해제하도록 신호

return exit_status;
```

```c
// exit status 출력
printf("%s: exit(%d)\n", cur->name, cur->exit_status);

// 부모가 wait 중이면 깨워줌
if (cur->parent != NULL) {
  sema_up(&cur->wait_sema);
  sema_down(&cur->free_sema); // 자식이 자원 해제할 때까지 대기
}
```

이렇게 하면 자식이 THREAD_DYING 상태가 될 때는 부모가 이미 필요한 정보를 모두 읽은 후이므로 안전하게 페이지를 해제할 수 있다. 따라서 thread_schedulte_tail()에서는 parent 여부를 확인하지 않고 THREAD_DYING 상태의 모든 thread에 대해 pallic_free_page()를 호출하도록 수정했다.

```c
if (prev != NULL && prev->status == THREAD_DYING && prev != initial_thread)
  {
    ASSERT (prev != cur);
    palloc_free_page (prev);
  }
```

C. 시험 및 평가 내용

다음은 src/userprog에서 'make check' 명령어를 실행했을 때의 결과로, 80개의 테스트에 대해 모두 pass한 것을 확인할 수 있다.

```
pass tests/userprog/bad-jump2
pass 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
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 80 tests passed.
make[1]: warning:  Clock skew detected.  Your build may be incomplete.
make[1]: Leaving directory '/sogang/under/cse20211589/pintos/src/userprog/build'
make: warning:  Clock skew detected.  Your build may be incomplete.
cse20211589@cspro6:~/pintos/src/userprog$
```

다음은 'make check'의 결과로 생성된 results 파일 내용이다.

```
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/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
pass tests/userprog/exec-arg
pass 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
pass tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
pass tests/userprog/rox-simple
pass tests/userprog/rox-child
pass 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
pass 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
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write

```