**Pintos Project 2: User Program (2)**

담당 교수 / 분반 : 박성용 / 3분반

이름 / 학번 : 김주호 / 20121277

개발 기간 : 2021.11.01 ~ 2021.11.14

**목 차**

[1. 개발 목표 3](#_Toc87783349)

[2. 개발 범위 및 내용 4](#_Toc87783350)

[A. 개발 범위 4](#_Toc87783351)

[B. 개발 내용 4](#_Toc87783352)

[3. 추진 일정 및 개발 방법 6](#_Toc87783353)

[A. 추진 일정 6](#_Toc87783354)

[B. 개발 방법 6](#_Toc87783355)

[4. 연구 결과 9](#_Toc87783356)

[A. Flow Chart 9](#_Toc87783357)

[B. 제작 내용 13](#_Toc87783358)

[C. 시험 및 평가 내용 23](#_Toc87783359)

# 1. 개발 목표

proj1에 이어 file system과 관련한 system call을 추가로 구현한다. (create, remove, open, close, filesize, read, write, seek, tell)

# 2. 개발 범위 및 내용

## A. 개발 범위

### 1. File Descriptor

* 파일 디스크립터(file descriptor)는 리눅스 혹은 유닉스 계열의 시스템에서 프로세스(process)가 파일(file)을 다룰 때 사용하는 개념으로 0 아닌 정수값을 가진다.
* file descriptor를 사용하면, 프로세스(혹은 thread)가 핸들링하고 있는 파일에 접근할 수 있게 된다.

### 2. System Calls

* user level에서 명령을 입력하였을 때, 이 명령이 kernel level의 함수 사용을 필요로 하는 명령일 수 있다.
* system call handler를 통해 argument들을 가져와 file system 관련 명령을 수행할 수 있다.

### 3. Synchronization in File system

* 여러 개의 thread가 concurrent하게 수행되면서 파일을 read 혹은 write할 때, 여러 thread에 의해 동일한 파일이 조작된다면, thread의 명령 실행이 의도한 바대로 동작하지 않는다.
* 파일을 읽고 쓰는 동안 thread가 그 해당 파일을 독점적으로 사용할 수 있도록하여 동기화할 수 있도록 한다.

## B. 개발 내용

### 1. File Descriptor

* file descriptor인 0, 1, 2, .. 인 integer가 곧바로 file 구조체를 지정할 수 있도록, 배열로 file descriptor를 관리한다.
* fd가 핸들링하는 file 구조체의 형태가 모두 동일하다는 면에서, 배열을 자료구조로 택할 수 있다.

### 2. System Calls

* create
  + 새로운 파일을 만든다. 성공적으로 만들어진다면, true를 반환
* remove
  + 파일을 삭제한다. 성공적으로 삭제된다면, true를 반환
* open
  + 파일을 연다. 성공적으로 파일을 연다면, file descriptor를 반환한다.
* close
  + file descriptor에 해당하는 파일을 닫는다.
* filesize
  + file descriptor에 해당하는 file의 byte size를 반환한다.
* read
  + buffer로부터 파일을 읽어들인다. read한 byte 수를 반환 한다.
* write
  + buffer의 byte size만큼 write를 수행한다. write 된 byte 수를 반환하게 된다.
* seek
  + 파일의 커서를 조정하는 함수이다.
* tell
  + 파일에서 현재 커서의 위치를 반환한다.

### 3. Synchronization in File system

* thread가 concurrent하게 수행되면서, 하나의 파일을 동시에 읽거나, 쓴다면 synchronization이 깨지게 된다.
* 파일을 읽거나 쓸 때, lock 구조체 변수를 선언하고 lock\_acquire()를 수행하여, thread가 파일을 독점적으로 사용하도록 만든다.
* 사용이 끝나면, lock\_release()로 다른 thread가 해당 파일에 access할 수 있도록 만든다.
* parent가 child보다 먼저 죽게 되어, child가 고아가 되는 것을 방지해야 한다.
* parent가 항상 child보다 나중에 죽을 수 있도록, Semaphore를 통해 synchronization을 달성한다. (sema\_down()을 한 뒤, child를 reap할 수 있도록)

# 3. 추진 일정 및 개발 방법

## A. 추진 일정

* **day1 ~ day3.** base file system 구현
  + file system과 관련한 system call의 기틀이 되는 코드를 작성한다.
* **day4 ~ day7.** synchronization 및 예외처리 구현
  + thread 구조체의 속성을 추가로 정의하고, 이를 핸들링하면서 synchronization이 달성되도록 코드를 작성
* **day8 ~ day10.** 코드 정리 및 오류 디버깅
  + 더 가독성 있는 코드가 되도록 변수명과 함수명을 정리하고, 다른 오류들도 디버깅한다.

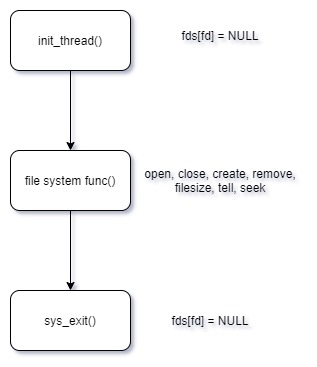
## B. 개발 방법

* src/threads/thread.h
  + struct thread 에 속성 추가
    - struct semaphore load\_sema 를 추가하여, 자식의 load 성공 여부를 알기 전까지 parent가 먼저 죽는 경우를 방지한다.
    - struct file\* fds[FILE\_NUM] 추가
      * thread가 핸들링하는 file descriptor를 관리한다.
    - struct file\* handling\_fp 추가
      * thread가 핸들링하는 file descriptor가 가리키는 file 구조체를 속성으로 추가한다.
    - struct thread\* parent 추가
* src/threads/thread.c
  + struct thread에 새로 추가한 속성에 관하여도 initialize를 수행하는 코드라인을 추가 작성
* src/userprog/syscall.c
  + 2개의 함수를 추가
    - void sys\_open(const char\* file)
      * 파일을 여는 함수
    - void sys\_close(int fd)
      * 파일을 닫는 함수
  + 함수에 코드 추가, 보완
    - void sys\_exit (int status)
      * file system 관련해 돌고 있는 thread에 대해, file을 close하고 reap하는 코드를 추가 작성
    - int sys\_read (int fd, void \*buffer, unsigned length)
      * proj1에서는 fd = 0 인 상황 즉, std input에 대해서 read하는 함수를 작성하였다.
      * proj2에서는 fd >= 3 인 상황에서도 file을 read할 수 있게 코드를 보완한다.
    - int sys\_write (int fd, const void \*buffer, unsigned length)
      * proj1에서는 fd = 1 인 상황 즉, std output에 대해서 write하는 함수를 작성하였다.
      * proj2에서는 fd >= 3 인 상황에서도 file에 write할 수 있게 코드를 보완한다.
* src/userprog/process.c
  + static void start\_process (void \*file\_name\_) 함수에 코드 추가
    - parent thread의 대기 상태를 풀어준다. (sema\_up(parent->tid))
    - child의 load가 성공 여부를 parent에게 알린다.
  + tid\_t process\_execute (const char \*file\_name) 함수에 코드 추가
    - thread\_create() 직후에 child thread의 load 상태를 전달 받기 전까지, 대기한다. (sema\_down(tid))
    - child thread가 parent thread의 대기 상태를 풀면, child의 load 성공 여부를 보고, 혹 실패시(filename이 invalid하거나 등) parent가 이를 수거할 수 있도록 한다.

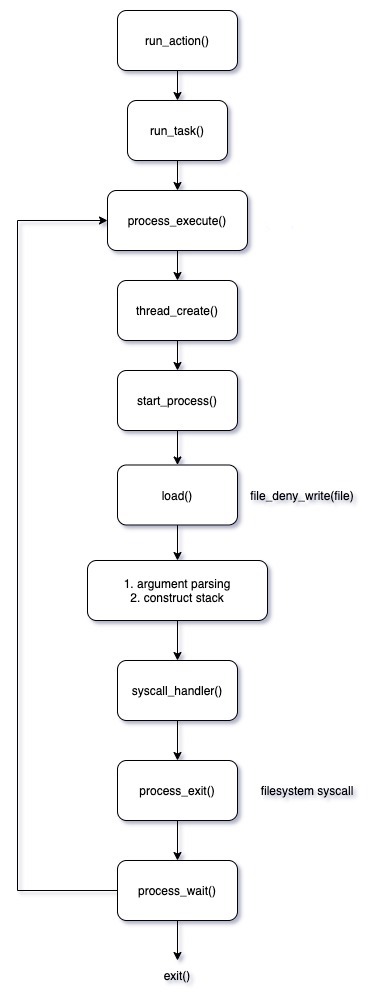
# 4. 연구 결과

## A. Flow Chart

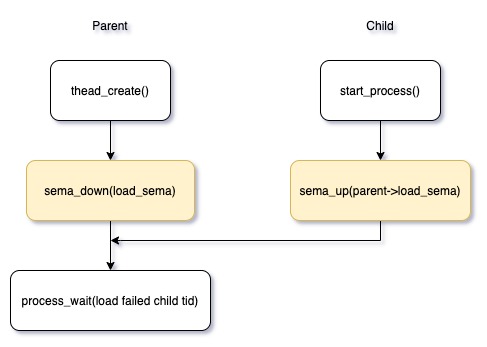
### 1. File Descriptor

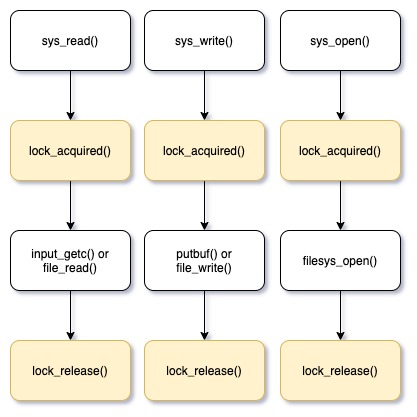


### 2. System Calls



### 3. Synchronization in File system





## B. 제작 내용

* src/threads/thread.h
  + struct thread 에 5가지의 속성을 추가로 부여한다.
    - struct semaphore load\_sema 를 추가하여, 자식의 load 성공 여부를 알기 전까지 parent가 먼저 thread 실행을 마치는 경우를 방지한다.
      * child가 혹여 filename이 invalid하여 load가 실패 했을 시, parent가 이를 기다렸다가 reap해줄 수 있다.
      * parent는 thread\_create() 직후에 대기한다. (sema\_down(sema))
      * child는 start\_process()를 수행하면서, load() 직후에 parent의 sema를 up한다.(sema\_up(parent->sema))
      * 이로써, parent는 자식이 load가 완료가 되면, 혹여 filename이 잘못되어 load에 실패한 child를 수거할 수 있게 된다.
    - struct file\* fds[FILE\_NUM] 추가
      * file 구조체에 고유한 index를 부여하는 셈이다.
      * file descriptor를 통해 O(1)로 file 구조체에 접근할 수 있다.
    - struct file\* handling\_fp 추가
      * thread가 현재 핸들링하는 file descriptor가 가리키는 file 구조체를 속성으로 추가한다.
      * 현 thread가 핸들링하는 file의 속성에 편리하게 접근할 수 있다.
    - struct thread\* parent 추가
      * 현재 thread의 parent를 속성으로 가져, child에서 parent로 접근할 수 있게 한다.
      * parent의 thread 속성들에 편리하게 접근할 수 있다.
      * 이를테면, start\_process()에서 sema\_up(&(thread\_current())->parent->load\_sema) 하는 방식으로 간단히 자식 thread에서 부모 thread의 load\_sema 속성에 접근할 수 있게 된다.
    - bool load\_success 추가
      * thread가 load될 때마다, load의 성공 여부를 속성값으로 저장하고 있는다.
      * 부모 thread는 자식의 이 속성을 보면서, 성공적으로 load되지 못한 child를 수거한다.

|  |
| --- |
| * struct thread { ...  #ifdef USERPROG  /\* Owned by userprog/process.c. \*/  uint32\_t \*pagedir; /\* Page directory. \*/  struct list child;  struct semaphore child\_sema;  struct semaphore memory\_sema;  struct semaphore load\_sema;  struct list\_elem child\_elem;  int exit\_status;   // proj2 - thread 구조체에 속성들 추가  struct file\* fds[FILE\_NUM];  bool load\_success; // load의 성공 여부  struct file\* handling\_fp;  struct thread\* parent; #endif  ... }; |

* src/threads/thread.c
  + struct thread에 새로 추가한 속성에 관하여도 initialize를 수행하는 코드라인을 추가 작성한다.
  + 배열 fds와 handling\_fp는 모두 NULL 값으로 초기화하고, load\_success는 true로 초기화 하며, thread의 parent는 running\_thread()를 디폴트 값으로 부여한다.

|  |
| --- |
| * static void init\_thread (struct thread \*t, const char \*name, int priority) { ...  #ifdef USERPROG  sema\_init(&(t->child\_sema), 0);  sema\_init(&(t->memory\_sema), 0);  list\_init(&(t->child));  list\_push\_back(&(running\_thread()->child), &(t->child\_elem));   //////////////////// 추가  for (int fd = 0; fd < FILE\_NUM; fd++) {  t->fds[fd] = NULL;  }  sema\_init(&(t->load\_sema), 0);  t->load\_success = true;  t->handling\_fp = NULL;  t->parent = running\_thread();  #endif } |

* src/userprog/syscall.c
  + void sys\_open(const char\* file)
    - Pintos에 내장된 filesys\_open() 함수를 사용하여, file을 open한다.
    - thread의 fds 배열에서 빈 자리를 순회하면서, 빈 자리를 발견하면, open한 file의 정보를 저장한다.
    - 파일을 열고 fds 배열에 저장하는 일련의 과정을 lock\_acquire(), lock\_release()로 감싸면서, 파일을 여는 동작이 다른 thread의 침범을 받지 않고 독점적으로 진행될 수 있도록 만든다.

|  |
| --- |
| int sys\_open(const char\* file) {  lock\_acquire(&file\_lock);  struct file \*fp = filesys\_open(file);  if (fp == NULL) {  lock\_release(&file\_lock);  return -1;  }   for (int fd = 3; fd < FILE\_NUM; fd++) {  if (!(thread\_current()->fds[fd])) {  thread\_current()->fds[fd] = fp;  lock\_release(&file\_lock);  return fd;  }  }   lock\_release(&file\_lock);  return -1; } |

* void sys\_close(int fd);
  + Pintos에 내장된 file\_close() 함수를 사용하여, file을 close한다.
  + 현재 thread의 fds 배열에 저장된 file 정보도 NULL 값으로 초기화한다.

|  |
| --- |
| void sys\_close(int fd) {  if (thread\_current()->fds[fd] == NULL) sys\_exit(-1);  file\_close(thread\_current()->fds[fd]);  // thread\_current()->handling\_fp = NULL;  thread\_current()->fds[fd] = NULL;  } |

* void sys\_exit (int status)
  + file system 관련한 system handler를 추가로 구현하였으므로, thread를 종료할 때, thread가 핸들링하고 있는 file들을 모두 close & reap하는 코드를 추가로 작성한다.

|  |
| --- |
| void sys\_exit (int status){  printf("%s: exit(%d)\n", thread\_name(), status);  thread\_current()->exit\_status = status;    /////////////////// 추가  // close  for (int fd = 3; fd < FILE\_NUM; fd++){  if (thread\_current()->fds[fd] != NULL) sys\_close(fd);  }  // reap  for (struct list\_elem \*cur = list\_begin(&(thread\_current())->child);  cur != list\_end(&(thread\_current()->child)); cur = list\_next(cur)) {    process\_wait(list\_entry(cur, struct thread, child\_elem)->tid);  }  file\_close(thread\_current()->handling\_fp);  //////////////////////    thread\_exit();  } |

* int sys\_read (int fd, void \*buffer, unsigned length)
  + proj1에서는 fd = 0 인 상황 즉, std input에 대해서 read하는 함수를 작성하였다.
  + 이 때, Pintos에 내장된 input\_getc() 함수를 이용하였다.
  + proj2에서 file system 관련 로직들을 추가하면서, fd >= 3 인 상황에서도 file을 read할 수 있게 분기문을 추가하여, 코드를 보완한다.
  + fd >= 3인 경우에 대해서는 Pintos에 내장된 file\_read() 함수를 이용하여 file을 buffer에 읽어들인다.
  + 위의 모든 로직들을 lock\_acquire(), lock\_release()로 감싸면서, 파일을 읽어들이는 작업이 다른 thread의 침범을 받지 않고 독점적으로 진행될 수 있도록 만든다.
  + 유효하지 않은 buffer 주소에 접근하여 read를 시도하는 경우 종료될 수 있도록 한다.

|  |
| --- |
| * int sys\_read (int fd, void \*buffer, unsigned length){  if (!is\_valid\_ptr(buffer)) sys\_exit(-1);    lock\_acquire(&file\_lock);  if (fd == 0) { // std input  int cnt;  for (cnt = 0; cnt < (int)length; cnt++) {  if (input\_getc() == '\0') break;  }  lock\_release(&file\_lock);  return cnt;  }    if (!(thread\_current()->fds[fd])) {  lock\_release(&file\_lock);  sys\_exit(-1);  }  if (fd >= 3) { // not std input  int ret;  ret = file\_read(thread\_current()->fds[fd], buffer, length);  lock\_release(&file\_lock);  return ret;  }    lock\_release(&file\_lock);  return -1;  } |

* int sys\_write (int fd, const void \*buffer, unsigned length)
  + proj1에서는 fd = 1 인 상황 즉, std output에 대해서 write하는 함수를 작성하였다.
  + 이 때, Pintos에 내장된 putbuf() 함수를 이용하였다.
  + proj2에서 file system 관련 로직들을 추가하면서, fd >= 3 인 상황에서도 file에 write할 수 있게 분기문을 추가하여, 코드를 보완한다.
  + fd >= 3인 경우에 대해서는 Pintos에 내장된 file\_write() 함수를 이용하여 buffer로부터 file에 write 한다.
  + 위의 모든 로직들을 lock\_acquire(), lock\_release()로 감싸면서, 파일을 쓰는 작업이 다른 thread의 침범을 받지 않고 독점적으로 진행될 수 있도록 만든다.

|  |
| --- |
| int sys\_write (int fd, const void \*buffer, unsigned length){  lock\_acquire(&file\_lock);  if (fd == 1) { // std output  putbuf((char\*)buffer, (size\_t)length);  lock\_release(&file\_lock);  return length;  }    if (thread\_current()->fds[fd] == NULL) {  lock\_release(&file\_lock);  sys\_exit(-1);  }  if (fd >= 3) { // not std output  int ret;  ret = file\_write(thread\_current()->fds[fd], buffer, length);  lock\_release(&file\_lock);  return ret;  }    lock\_release(&file\_lock);  return -1;  } |

* src/userprog/process.c
  + static void start\_process (void \*file\_name\_) 함수에 코드 추가
    - parent thread는 child thread의 load 성공 여부를 알기 위해 process\_execute()에서 대기 하고 있다.
    - child thread의 load가 완료된 직후, parent에 대한 sema\_up(parent->tid)을 수행한다.
    - child thread는 load의 결과에 따라 자신의 bool load\_success 속성에 true or false를 할당하면서, parent에게 load 성공 여부를 알린다.
    - 이로써, parent thread는 child thread의 load 성공 여부를 알 수 있게 된다.

|  |
| --- |
| 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);   // cmd Parsing and Create a new thread  char CMD[256];  char\* nxt;  strlcpy(CMD, file\_name, strlen(file\_name) + 1);  char \*thread\_name = strtok\_r(CMD, " ", &nxt);   if (!filesys\_open(thread\_name)) return -1;   tid = thread\_create (thread\_name, PRI\_DEFAULT, start\_process, fn\_copy);   /////////////////// proj2 - semaphore 처리  // P, sem\_wait() => child가 load된 뒤, sema\_up 해줄 때까지 대기한다.  sema\_down(&thread\_current()->load\_sema);   // 아제 child의 load 성공 여부를 보면서, load 실 child는 reap 시킴  if (tid == TID\_ERROR) palloc\_free\_page (fn\_copy);  for (struct list\_elem \*cur = list\_begin(&(thread\_current())->child);  cur != list\_end(&(thread\_current()->child)); cur = list\_next(cur)) {   if (list\_entry(cur, struct thread, child\_elem)->load\_success == false) return process\_wait(tid);  }  //////////////////////   return tid; } |

* tid\_t process\_execute (const char \*file\_name) 함수에 코드 추가
  + thread\_create() 직후, sema\_down()을 수행하여, 자식 thread의 load가 완료 되기를 기다린다. (sema\_down(tid))
  + 자식 thread는 start\_process()에서 sema\_up(parent->tid)을 수행한다.
  + 이로써, 자식의 load가 완료된 뒤, 부모가 해당 자식의 load 성공 여부를 볼 수 있다.
  + child thread는 자신의 load 성공 여부를 bool load\_success에 저장하고 있기 때문에 이를 살피면서, 부모가 자식의 load 성공 여부를 알 수 있다
  + child의 load가 성공적이지 못한 경우, 해당 child thread를 찾아 reap 시킨다.

|  |
| --- |
| static void start\_process (void \*file\_name\_)  {  char \*file\_name = file\_name\_;  struct intr\_frame if\_;  bool success;    /\* Initialize interrupt frame and load executable. \*/  memset (&if\_, 0, sizeof if\_);  if\_.gs = if\_.fs = if\_.es = if\_.ds = if\_.ss = SEL\_UDSEG;  if\_.cs = SEL\_UCSEG;  if\_.eflags = FLAG\_IF | FLAG\_MBS;      success = load (file\_name, &if\_.eip, &if\_.esp);    /\* If load failed, quit. \*/  palloc\_free\_page (file\_name);    //////////////////  // V, sem\_signal() => child가 parent의 load\_sema를 풀어줌.  if (success){  thread\_current()->load\_success = true; // 생략해도 무방  sema\_up(&(thread\_current())->parent->load\_sema);  }  else {  thread\_current()->load\_success = false;  sema\_up(&(thread\_current())->parent->load\_sema);  sys\_exit(-1);  //thread\_exit ();  } |

## C. 시험 및 평가 내용

