# OS Project03 - Wiki

**2019040591**
**컴퓨터소프트웨어학부**
**박진수**

## Design:

먼저 thread의 구현을 위해 proc.h 파일에 있던 기존의 struct proc에 아래와 같이 멤버변수를 추가했습니다. thread는 Light-Weight Process로, 많은 특징을 프로세스와 공유하기 때문에, 기존의 proc struct에 thread의 특성에 해당하는 멤버변수를 추가해주는 방식으로 구현하였습니다.

### struct proc (proc.h)

```
struct proc {
  ...
  // For Project3
  thread_t tid;                // Thread ID.
  int is_thread;               // 0 if process, 1 if thread
  struct proc *main_thread;    // Main thread in linked list
  struct proc *next_thread;    // Next thread in linked list(
  void *retval;                // Return value that join() wi
};
```

같은 프로세스 (메인 thread)에서 생성된 프로세스끼리는 같은 pid를 갖기 때문에, 각 프로세스를 구분해줄 수 있는 tid 멤버 변수를 추가했습니다. 그리고 해당 객체가 process인지, thread인지 구분하기 위해 is_thread 변수를 추가했습니다. is_thread가 0이면 일반적인 process, 1이면 thread를 나타내도록 하였습니다. 그리고 자원과 주소 공간을 공유하는 thread(main thread 포함)를 linked list로 연결하여 관리하기 위해, main_thread와 next_thread 변수를 추가했습니다. proc.c의 allocproc()에서 프로세스의 상태를 초기화해줄 때, main_thread를 자기 자신으로 설정하게 하고, next_thread는 0(NULL)로 초기화하여, 프로세스를 head로 시작하는 linked list가 되도록 설계했습니다. 마지막으로 retval 변수의 경우, thread가 종료될 때 thread_exit()에서 해당 멤버변수에 이 값을 넘겨주며, 이후 thread_join() 함수가 이 값을 읽어 받아갈 수 있도록 구현하였습니다.

이어서 세부적인 동작이나 코드들은 아래의 Implementation 부분에서 설명하도록 하겠습니다.

# Implementation:

## 1. proc.c

```
int nextpid = 1;
int nexttid = 1;
```

위의 코드는 proc.c의 첫부분의 코드로, 기존의 코드는 프로세스의 id를 생성하기 위한 nextpid만 존재하였으나, thread의 id를 관리하기 위해 nexttid변수를 만들고 1로 초기화하였습니다.

```
static struct proc*
allocthread(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  // p->pid = nextpid;
  // pid will be modified ouside the allocthread()

  release(&ptable.lock);

  // Allocate kernel stack.
```

```
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;

  return p;
}
```

위의 코드는 fork()나 initproc()에서 새로운 프로세스를 생성할 때 호출하는 allocproc()을 수정한 함수로, thread_create()에서 새로운 쓰레드를 생성할 때 사용하기 위해 추가했습니다. 기존의 allocproc()의 경우, ptable.proc 배열을 순회하다가 상태가 UNUSED인 빈 프로세스 구조체를 찾게되면 무조건 p→pid에는 nextpid가 할당되고, nextpid가 1 증가합니다. 그러나 쓰레드의 경우 동일한 pid를 공유하며, tid로 각 쓰레드를 구분하기 때문에, p→pid = nextpid++;를 삭제했습니다. 따라서, thread_create()에서 allocthread()를 호출하면, allocproc()과 마찬가지로 해당 쓰레드의 context를 초기화해주고, 페이지 테이블을 생성하고, 커널 스택을 할당하고 초기화해줍니다. pid의 처리 과정만 프로세스와 쓰레드 간의 차이가 있습니다. 각각의 쓰레드끼리는 위의 자원들을 서로 독립적으로 가지기 때문에, 프로세스와 동일한 처리 과정을 거칩니다.

## thread_create()

```
int thread_create(thread_t *thread, void *(*start_routine)(vo
  struct proc *np;
  struct proc *curproc = myproc();
```

```
if ((np = allocthread()) == 0) {
  return -1;
}

// Shares same address space
np->sz = curproc->sz;
np->pgdir = curproc->pgdir;
np->parent = curproc->parent; // share same parent
*np->tf = *curproc->tf;
np->is_thread = 1;

np->tid = nexttid++;
np->pid = curproc->pid; // share same pid
curproc->next_thread = np; // linked list of threads
np->next_thread = 0;
np->main_thread = curproc->main_thread;

np->retval = 0;

// Adjust the instruction pointer in the trap frame to the
np->tf->eip = (uint)start_routine;

// set up the user stack for the new thread
uint sp = np->tf->esp;
sp -= 4;
*(uint*)sp = (uint)arg; // Push argument onto stack
sp -= 4;
*(uint*)sp = (uint)thread_exit;

// update the stack pointer
np->tf->esp = sp;

int i;
// Share open files and current working directory.
for (i = 0; i < NOFILE; i++)
  if (curproc->ofile[i])
    np->ofile[i] = filedup(curproc->ofile[i]);
```

```
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  *thread = np->tid;

  // Insert the new process (thread) into the runnable state
  acquire(&ptable.lock);
  np->state = RUNNABLE;
  release(&ptable.lock);

  return 0;
}
```

위의 thread_create는 fork()를 참고하여 구현하였습니다. 다만 새로 생성된 쓰레드의 경우, thread_create()를 호출한 쓰레드와 address spacef를 공유하기 때문에 fork()와는 달리 아래처럼 curproc의 sz와 pgdir를 그대로 참조하도록 하였습니다.

```
np->sz = curproc->sz;
np->pgdir = curproc->pgdir;
```

또한, 쓰레드끼리는 부모를 공유하므로(pid가 동일하므로) np→parent = curproc→parent를 통해 같은 부모를 가지도록 하였습니다. 그리고 각 쓰레드끼리는 서로 다른 일을 처리하기 때문에 독립적인 trapframe을 가져야 합니다. 그래서 trapframe을 설정하는 부분은 fork()와 동일하게 처리하였습니다. 그리고 tid의 경우, 원래는 프로세스가 가진 pid가 구분자가 되어, 서로 다른 프로세스의 쓰레드 끼리는 같은 tid를 가질 수 있지만, 편의를 위해 전역 변수로 nexttid를 설정해두고, pid와 동일하게 할당하고 1을 증가시키는 방식을 사용하였습니다. 그리고 curproc→next_thread = np의 경우, 쓰레드간 관리를 위한 linked list를 구현한 부분으로, thread_create()를 호출한 쓰레드의 바로 뒤에 해당 쓰레드를 연결하도록 하였습니다. 그리고 메인 쓰레드(프로세스)는 모든 쓰레드가 공유하므로 메인 쓰레드에 대한 정보도 추가하였습니다. retval주소에 할당되는 값은 thread_exit()에서 사용되므로 일단 0(NULL)로 초기화하였습니다. np→tf→eip의 경우, 해당 쓰레드가 바로 다음에 실행할 instruction을 가리키게 되므로, 해당 요구 조건의 경우, start_routine()을 실행해야 하므로 추가해주었습니다. 그리고 아랫 부분의 코드는 exec()를 참고하여 구현하였습니다.

```
// set up the user stack for the new thread
  uint sp = np->tf->esp;
```

```
  sp -= 4;
  *(uint*)sp = (uint)arg; // Push argument onto stack
  sp -= 4;
  *(uint*)sp = 0xffffffff;

  // update the stack pointer
  np->tf->esp = sp;
```

위의 코드는 user stack에 필요한 데이터를 설정해주고 sp의 위치를 다시 이에 맞춰 수정해준 부분인데, 사실 이 부분을 명확히 이해하고 작성하지 못해서 오류가 있을 수 있습니다. 제가 생각하기로는 thread_create()은 fork()에 대응되는 부분으로, 추후 eip에 해당하는 start_routine으로 이동하여 처리하면 된다고 생각했고, elf의 path의 형태로 전달받은 exec()와는 달리, 이미 code section에 start_routine이 포함되어 있다고 생각하여 메모리를 할당받는 부분은 생략했습니다. 작성을 완료하고 테스트 케이스를 실행했을 때, 실행 중에 초기 shell 화면으로 돌아가는 에러가 발생했는데, 아마 이것이 원인이 아닐까 생각하고 있습니다. 그리고 이후 파일 디스크립터도 공유해준 뒤, 인자로 넘겨받은 thread_t의 포인터를 참조하여 그 값에 새로 생성된 tid값을 추가하는 식으로 반환한 이후, state를 RUNNABLE로 바꾸고, 정상적으로 모든 과정이 끝났다면 0을 리턴하도록 하였습니다. 이 과정은 이후 시스템콜 핸들러가 그대로 받아 유저레벨 시스템콜 함수로 넘기도록 하였습니다.

## thread_exit()

```
void thread_exit(void *retval) {
  struct proc *curthread = myproc();

  if(curthread == initproc) {
    panic("init exiting");
  }

  acquire(&ptable.lock);

  curthread->retval = retval;

  // Wakeup any process or thread waiting on this thread's te
  wakeup1(curthread);

  curthread->state = ZOMBIE;
  sched();
```

```
  panic("zombie exit");
}
```

위 함수의 경우 caller thread를 종료할 수 있도록 하였습니다. exit()함수를 참고하였고, thread_exit()으로 전달받은 retval 값을 현재 쓰레드의 retval 멤버변수에 저장하여, 이후 thread_join이 이 쓰레드에 접근하여 이 값을 받아갈 수 있도록 하였습니다. 파일 디스크립터의 경우, 다른 쓰레드들이 아직 사용하고 있을 수 있으니 닫지 않았고, 해당 쓰레드를 기다리는 다른 쓰레드에게 wakeup1(curthrea)로 signal을 보내줍니다. 이후 해당 쓰레드의 상태를 ZOMBIE로 바꿔줍니다.

## thread_join()

```
int thread_join(thread_t thread, void **retval) {
  struct proc *p;
  struct proc *prev = 0;
  int tid_found = 0;
  struct proc *foundthread = 0;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);

  for(;;){
    // Scan through table looking for exited children.
    tid_found = 0;
    struct proc *start = curproc->main_thread;
    for(p = start; p != 0; prev = p, p = p->next_thread){
      if(p->tid == thread){
        tid_found = 1;
        foundthread = p;
        if(p->state == ZOMBIE){
          // Found one.
          if (retval != 0) {
            *retval = p->retval;
          }
          kfree(p->kstack);
          p->kstack = 0;

          prev->next_thread = p->next_thread;
```

```
            p->tid = 0;
            p->is_thread = 0;
            p->retval = 0;

            p->pid = 0;
            p->parent = 0;
            p->name[0] = 0;
            p->killed = 0;
            p->state = UNUSED;
            release(&ptable.lock);
            return 0;
          }
          if(tid_found)
            break;
        }

        // No point waiting if we don't have any children.
        if(!tid_found){
          release(&ptable.lock);
          return -1;
        }

        // Wait for the thread to exit  (See wakeup1 call in pr
        sleep(foundthread, &ptable.lock);  //DOC: wait-sleep
      }
    }
  }
```

종료된 쓰레드를 join할 수 있는 쓰레드는 해당 쓰레드를 포함하는 linked list에 속해있기에, 해당 linked list의 main_thread부터 시작하여 thread들을 탐색하여, thread에 해당하는 tid값을 가진 쓰레드가 있는지 확인합니다. 만약 발견했다면, 해당 쓰레드에 담겨있던 retval 값을 thread_join()의 인자로 넘어온 retval의 주소를 참조하여 반환합니다. 이후 kernel stack에 해당하는 메모리를 free하고 초기화해줍니다. 이후, 해당하는 thread는 linked list에서 삭제 해야하므로, 나머지 쓰레드끼리 이어줍니다. 그리고 나머지 멤버변수들을 초기화해줍니다. 만약에 해당하는 쓰레드를 발견하지 못했다면, 0이 아닌 다른 값을 반환해줍니다. 그리고 찾았는데, ZOMBIE 상태가 아닌 경우. 해당 쓰레드가 종료되어 foundthread에 해당하는 signal을 보내줄 때까지 sleep 합니다. 이후 다시 깨어나면 for loop를 순회하며 자원 정리를 위한 부분을 실행합니다.

## sysproc.c

여기에 작성된 함수들은 시스템 콜 핸들러 함수들로, 유저 프로그램에서 시스템 콜을 호출하면 usys.S에서 만들어진 stub 함수들을 통해 미리 설정해둔 각각의 시스템콜에 해당하는 인터럽트를 호출하며, 이 인터럽트 번호가 syscall.c에서 아래의 함수들과 맵핑되어 아래의 핸들러 함수들이 실행되게 됩니다.

```
// Project 3
int
sys_thread_create(void)
{
  thread_t *thread;
  void *(*start_routine)(void *);
  void *arg;

  if(argptr(0, (void*)&thread, sizeof(thread_t *)) < 0 ||
      argptr(1, (void*)&start_routine, sizeof(void *(*)(void
      argptr(2, (void*)&arg, sizeof(void *)) < 0)
      return -1;

  return thread_create(thread, start_routine, arg);
}

int
sys_thread_exit(void)
{
  void *retval;

  if(argptr(0, (void*)&retval, sizeof(void *)) < 0)
      return -1;

  thread_exit(retval);
  return 0;
}

int
sys_thread_join(void)
```

```
{
  thread_t thread;
  void **retval;

  if(argint(0, &thread) < 0 || argptr(1, (void*)&retval, size
      return -1;

  return thread_join(thread, retval);
}
```

위의 함수들의 경우, 유저 레벨에서 호출한 시스템콜의 인자들이 제대로 들어왔는지 판단하는 예외처리 기능을 수행합니다. argptr()의 리턴 값이 음수인 경우 제대로 인자가 넘겨지지 않았다는 의미이므로 -1을 반환하도록 하였습니다. 그리고 인자들이 정상적인 값으로 판단된 경우, 넘겨진 인자들을 이용하여 proc.c에 있는 해당하는 커널 함수를 호출하게 하였습니다.

## user.h

```
// project 3
int thread_create(thread_t *thread, void *(*start_routine)(vo
void thread_exit(void *retval);
int thread_join(thread_t thread, void **retval);
```

유저 프로그램에서는 위의 user.h에서 선언된 함수를 호출하게 되며, 위의 형태가 과제의 명세에서 요구하는 함수의 형태와 정확히 일치합니다. 위의 함수를 호출하면 아래와 같이 usys.S에서 만들어진 stub 함수가 호출되게 됩니다.

## usys.S

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
...
```

```
SYSCALL(thread_create)
SYSCALL(thread_exit)
SYSCALL(thread_join)
```

위의 경우 각각의 SYSCALL(함수이름)으로 만들어진 stub 함수(Assembly code)는 각 시스템콜에 할당된 번호에 맞는 interrupt를 발생시킵니다. 각 시스템콜에 매핑된 번호는 아래의 파일에 작성되어 있습니다.

## syscall.h

```
...
#define SYS_thread_create  22
#define SYS_thread_exit    23
#define SYS_thread_join    24
```

위처럼 SYS_thread_create는 22로 매핑되어 있습니다. 그래서 위의 usys.S에서 thread_create()가 호출되었을 경우, eax레지스터에 SYS_thread_create에 해당하는 22를 저장시키고, T_SYSCALL이라는 interrupt number를 가진 시스템 인터럽트를 호출합니다. 여기서 시스템 콜에 해당하는 인터럽트를 발생시키면, 커널 레벨 핸들러에서는 이 eax 레지스터에 들어있는 숫자를 파악하여 어떤 시스템콜이 호출되었는지 판별할 수 있습니다.

## syscall.c

아래의 코드는 커널 레벨의 시스템 콜 핸들러 함수들을 extern으로 설정하여, 다른 파일들에서도 해당 함수를 호출할 수 있도록 만든 부분입니다

```
extern int sys_thread_create(void);
extern int sys_thread_exit(void);
extern int sys_thread_join(void);
```

아래는 앞서 usys.S에서 설정했던 어셈블리 함수에서 시스템 콜 인터럽트가 호출되면 그것을 처리하여 해당하는 시스템콜 핸들러 함수를 호출하는 부분으로, eax레지스터에 담겨있는 값에 따라 해당하는 sys_{시스템콜명} 함수를 실행합니다. 이렇게 되면 앞서 sysproc.c에서 만들어 두었던 함수들이 실행되게 됩니다.

```
static int (*syscalls[])(void) = {
...
[SYS_thread_create]  sys_thread_create,
[SYS_thread_exit]    sys_thread_exit,
```

```
[SYS_thread_join]    sys_thread_join,
};
```

## defs.h

```
// For project3
int             thread_create(thread_t *thread, void *(*start
void            thread_exit(void *retval);
int             thread_join(thread_t thread, void **retval);
```

다른 C 파일들에서 해당하는 함수에 접근할 수 있도록 위의 코드를 추가해주었습니다.

## types.h

```
#ifndef TYPES_H
#define TYPES_H

typedef unsigned int   uint;
typedef unsigned short ushort;
typedef unsigned char  uchar;
typedef uint pde_t;
typedef int thread_t;

#endif // TYPES_H
```

thread_t type의 경우, proc.c와 sysproc.c 파일에서 접근하여, 공유되는 헤더파일인 위에서 한번에 설정하여 중복 선언을 방지하였습니다.

## Makefile

주어진 테스트 파일인 mlfq_test.c를 xv6 코드에서 빌드하기 위해, 아래와 같이 추가하여 줍니다.

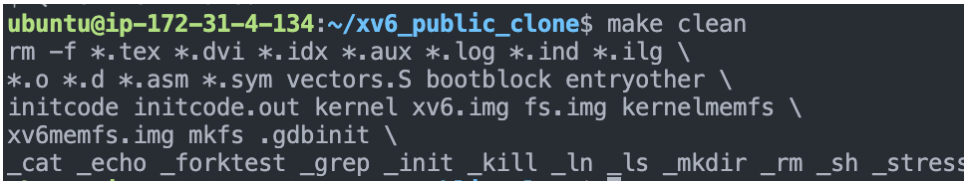
```
UPROGS=\
    ...
```

```
    _thread_test\
    _thread_exec\
    _thread_exit\
    _thread_kill\
    _hello_thread\
```

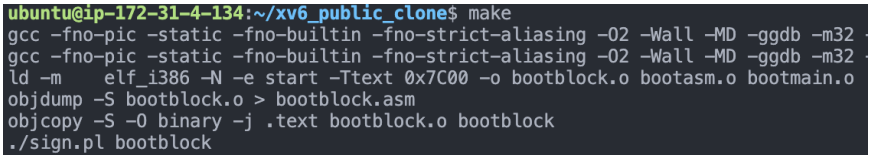
```
EXTRA=\
..thread_test.c thread_exec.c thread_exit.c thread_kill.c hel
```

## Result

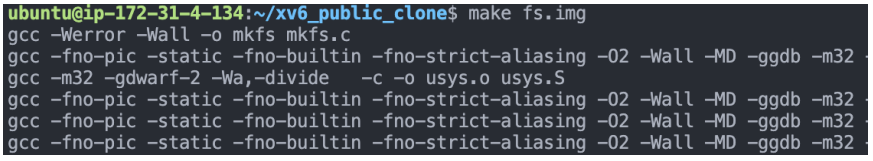
먼저 콘솔창에 **make clean**을 입력합니다.




그 다음 **make**를 입력합니다.




그 다음 **make fs.img**를 입력해줍니다.

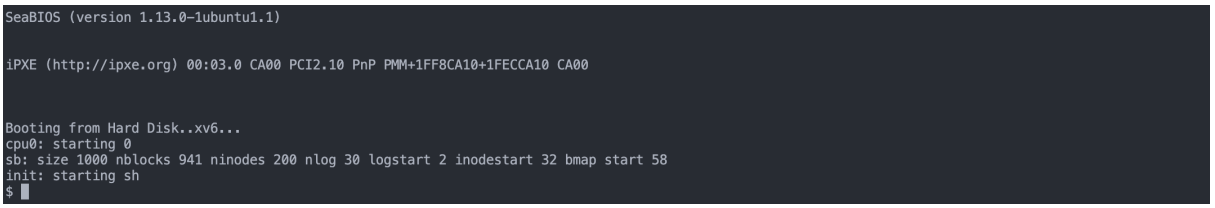



그 다음 **./bootxv6.sh**를 입력합니다. bootxv6.sh 파일의 내용은 아래와 같습니다.

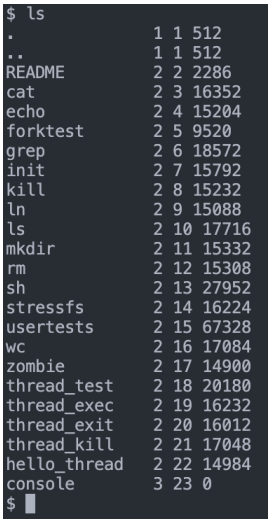
```
#!/bin/bash

qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6
```

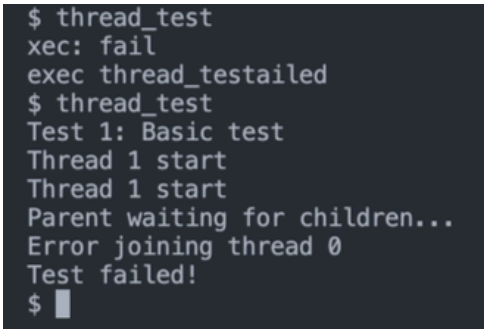
**./bootxv6**를 입력하면 아래와 같은 화면이 출력됩니다.



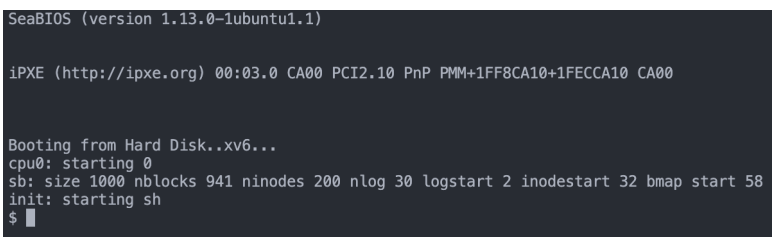

**ls**를 입력해보면, 테스트 파일들이 나타나 있는 것을 확인할 수 있습니다.




그 다음 테스트를 위해 **thread_test**를 입력합니다.




아래와 같이 출력된 이후, 다시 초기의 shell 화면으로 돌아오는 현상이 나타났습니다.



그 다음으로 **thread_exec**를 입력합니다.

```
$ thread_exec
Thread exec test start
Thread 4 start
Thread 4 start
ThrThread 4 start
Thread 4 start

This code shouldn't be executed!!
$
```

마찬가지로 이 이후 바로 shell 화면으로 돌아갑니다

그 다음으로 **thread_exit**를 입력합니다.

```
$ thread_exit
Thread exit test start
Thread 4 start
Thread 4 startThread 4 start
Thread 4 start
Thread 4 start
This code shouldn't be executed!!
```

이 경우에도 마찬가지로 바로 shell 화면으로 돌아갑니다.

그 다음으로 **thread_kill**를 입력합니다.

```
$ thread_kill
Thread kill test start
This code should bepid 3 thread_kill: trap 14 err 5 on cpu 0 eip 0xffffffff addr 0xff
pid 3 thread_kill: trap 14 err 5 on cpu 0 eip 0xffffffff addr 0xffffffff--kill proc
```

이 과정에서도 마찬가지로 shell 화면으로 돌아가는 것을 확인할 수 있었습니다.

# Trouble shooting

thread_exec 과정에서부터 정상적인 동작을 하지 않고 처음으로 돌아가는 것으로 보아, thread의 생성부터 문제가 있는 것 같습니다. thread의 kernel stack부분은 정확하게 구현하였다고 생각하나, user stack의 경우에 이해가 조금 미흡했던 것 같습니다. proc.c파일의 전체적인 동작 흐름과, 각 함수의 코드 내용들은 이해하였으나, 메모리 관리와 레지스터에 대한 이해가 부족했다고 생각합니다. 저는 user stack의 제일 윗 부분을 exec()의 형태로 구현하면, 약간의 메모리 문제는 발생하겠지만 맨 위에 담겨있는 인자의 정보를 이용하여

eip의 위치로 PC가 이동하고, start_routine이 실행되리라고 생각했습니다. 하지만, 예상과는 다르게, start_routine으로 이동하지 않았으며, 무언가 잘못된 실행 흐름이 발생했고, 예상과 달리 디버깅 할수도 없이 모든 테스트마다 처음의 쉘로 이동하는 것을 확인할 수 있었습니다. 이런 과정에서 느꼈던 점은, 무작정 fork(), exit(), exec()와 유사한 형태로 코드를 작성하는 것이 중요한게 아니라, 먼저 메모리 접근과 생성이 어떻게 일어나는지 먼저 완벽하게 이해한 후 코드를 작성하는 것이 중요하다는 것을 알게되었습니다. 사실 먼저 개념을 완벽하게 이해한 이후에 코드 작성을 진행하였다면, 훨씬 시간을 덜 소모하면서 정확한 코드를 작성할 수 있었을 것 같다는 생각이 듭니다. 이후 진행할 과제의 경우에는, 미리 xv6관련 document를 읽어보면서 각 operation이 어떻게 동작하고 메모리가 어떻게 관리되는지 파악해보려고 합니다. 단순히 코딩하는 것 보다 이해와 설계가 훨씬 더 중요하다는 사실을 알게 되었습니다.

# Locking

```
#include <stdio.h>
#include <pthread.h>
#include <stdbool.h>

volatile int shared_resource = 0;

#define NUM_ITERS 100
#define NUM_THREADS 100

volatile bool entering[NUM_THREADS] = {false};
volatile bool flag[NUM_THREADS] = {false};

void lock(int tid);
void unlock(int tid);


void lock(int tid) {
    // Show intention to enter the critical section
    entering[tid] = true;

    // Wait until no other thread is in the critical section
    for (int j = 0; j < NUM_THREADS; j++) {
```

```
        if (j != tid) {
            while (flag[j]) {
                // Busy wait
            }
        }
    }

    // Declare that it is in the critical section
    flag[tid] = true;

    // Check for any thread with lower index
    for (int j = 0; j < tid; j++) {
        if (entering[j]) {
            flag[tid] = false;
            while (entering[j]) {
                // Busy wait
            }
            j = -1; // restart the loop
        }
    }

    // Now safe to enter
    entering[tid] = false;
}

void unlock(int tid) {
    flag[tid] = false;
}

void* thread_func(void* arg) {
    int tid = *(int*)arg;

    lock(tid);

        for(int i = 0; i < NUM_ITERS; i++)    shared_resource

    unlock(tid);
```

```
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int tids[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++) {
        tids[i] = i;
        pthread_create(&threads[i], NULL, thread_func, &tids[
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("shared: %d\n", shared_resource);

    return 0;
}
```

처음에는 hardware support를 통해 mutual exclusion을 구현하면 몇 줄이면 구현 할 수 있을 것이라고 생각하였으나, swap 이나 test and set의 경우 전부 API를 사용하도록 되어 있어 요구조건에 부합하지 않을 것 같아서busy waiting을 사용하는 software solution을 사용하였습니다. 저는 그 중 Szymanski’s algorithm을 사용하여 해결하였습니다. 해당 알고리즘은 들어오는 문과 나가는 문이 있는 대기실을 모티브로 동작합니다. 처음에는 입구가 열려있고, 출구가 닫혀있습니다. 그리고 거의 동일한 순간에 들어온 쓰레드들만 대기실로 진입하게 하고, 대기실에 성공적으로 들어온 쓰레드의 tid에 해당하는 flag를 true로 바꾸어 줍니다. 마지막에 들어온 thread가 들어오면, 이제 못 들어오게 입구를 닫습니다(flag 값을 통해). 이제 들어온 쓰레드들의 flag는 true가 됩니다. 이 이후에는 출구를 열어 tid가 작은 것에서부터 높은 순서로 critical section으로 접근합니다. 대기실을 빠져나가면 해당 쓰레드의 entering 값을 다시 false로 바꾸어 줍니다. 그리고 critical section을 빠져나가면 unlock()을 통해 해당 tid에 해당하는 flag를 다시 false로 바꾸어줍니다. 그리고 또 마지막 프로세스가 critical section을 빠져나가면 출구가 닫히고 입구가 열려 다시 쓰레드들이 대기실로 진입할 수 있게 되는 구조입니다. 추가적으로 공유되는 변수들은 volatile로 설정하여, 컴파일러가 최적화를 위해 원래 흐름에서 벗어나서 실행하도록 하는것을 방지하였습니다.

## 결과

```
#define NUM_ITERS 10
#define NUM_THREADS 10
```

```
ubuntu@ip-172-31-4-134:~/xv6_public_clone$ ./pthread_lock_linux
shared: 100
```

```
#define NUM_ITERS 100
#define NUM_THREADS 100
```

```
ubuntu@ip-172-31-4-134:~/xv6_public_clone$ gcc -o pthread_lock_linux pthread_lock_linux.c -lpthread
ubuntu@ip-172-31-4-134:~/xv6_public_clone$ ./pthread_lock_linux
shared: 10000
```

```
#define NUM_ITERS 1000
#define NUM_THREADS 1000
```

```
ubuntu@ip-172-31-4-134:~/xv6_public_clone$ gcc -o pthread_lock_linux pthread_lock_linux.c -lpthread
ubuntu@ip-172-31-4-134:~/xv6_public_clone$ ./pthread_lock_linux
shared: 1000000
```

감사합니다.