컴퓨터소프트웨어학부
2022074057
홍바다

Project03 - Wiki

I. Design

A. Light-Weight Process (LWP)

Thread는 multitasking을 가능하게 하기 위해 다른 스레드와 자원, 주소 공간을 공유한다. 스케줄링 되기 위해 필수적인 자원들을 포함하여 스레드를 생성하기 때문에 보다 가볍고 병렬적으로 실행되며 정보교환에 효율적이라는 장점이 있다. 기본적으로 master thread가 process가 시작할 때 만들어지고, 이 메인 스레드를 통해 worker thread, 즉 자식 스레드가 생성된다. 그리고 이 master thread는 자식 스레드들이 종료가 될 때까지 대기해야한다. 프로세스의 실행 과정과 매우 비슷한 과정을 통해 작동하는 것을 볼 수 있다.

이러한 작동 방식을 보아 xv6에 존재하는 기본 시스템 콜들을 참고하고자 하였다. 먼저 과제 명세에 있는 함수들을 구현하기 전에 proc.h에 있는 프로세스 구조체를 수정하였다. 먼저 pid(process id)처럼 tid(thread id)가 필요했고, 현재 스케줄링된 것이 프로세스인지 스레드인지 구분하는 isThread라는 변수도 넣어주었다. 또한, 첫 단락에서 설명하였듯이 master thread의 역할이 중요해보여 자식 스레드들의 메인 스레드가 무엇인지 저장해주는 master라는 변수도 넣어줬다. 마지막으로 thread_join을 구현할때 필요해보이는 retval이라는 변수를 넣어 종료하는 스레드의 리턴값을 저장하게 해주었다. retval은 thread_exit에서도 마스터 스레드가 다른 스레드가 종료될때까지 대기하는 상황에서도 사용된다.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)

  int isThread; //1 if thread, 0 if process
  struct proc *master;  //master thread that may create sub threads
  int tid; //thread id
  void *retval; //return value from join
};
```

이렇게 구조체를 수정한 후 과제 명세에 적힌 pthread의 기본적인 API들의 기능을 다시 보며 xv6에 존재하는 시스템 콜들 중 어떤 걸 참고 할 수 있을지 생각해보았다. 먼저, thread_create()은 새 스레드를 생성하고 제공된 함수인 start_routine부터 실행을 시작한다. 이는 자식 프로세스를 새로 생성하고 실행하는 것과 비슷하기에 fork()와 exec() 코드를

참고하였다. 그리고 thread_exit()은 말 그대로 종료하는 것이기 때문에 exit() 함수를 참고하였다. 마지막으로 thread_join()에서는 스레드가 인자로 받은 명시된 스레드가 종료될때까지 기다려야하기 때문에 wait() 시스템 콜을 참고하였다.

이번 과제에서는 프로세스와 매우 흡사한 실행 과정을 구현하는 것이기 때문에 프로세스의 기본 코드들을 잘 이해하고 사용하는 것이 중요해보였다. 또한 master thread, 즉 프로세스와 함께 생성되는 메인 스레드를 통해 자식 스레드를 생성하고, 해당 프로세스의 자원, 주소 공간, 파일 디스크립터 등을 공유하기 때문에 마스터 스레드를 통한 스택 관리가 잘 이루어져야겠다고 생각했다. 배운대로 공유하는 주소 공간을 사용하기 위해서는 프로세스의 페이지 테이블을 스레드에 공유하면 된다는 것을 상기시켰다. 이는 master의 실제 주소를 가리킬수 있는 master->pgdir를 통해 주소 공간을 알아내어 자원 활용 및 추가를 할 수 있다고 생각했다.

또한, 프로세스와 함께 스레드도 스케줄링되어 실행되기 때문에 여러 시스템 콜들에 대한 수정도 필요해보였다. 먼저 스케줄러는 round-robin 방식을 따라 이전 과제와 상관없기 때문에 새로 깃 클론을 받은 후 진행했다. 과제 명세에 명시된 기존 시스템 콜들과 스레드 간의 상호작용을 위해 기본적으로 현재 실행된 것이 프로세스인지 스레드인지에 따라 다르게 구현해야되는 상황을 분석했다. 먼저 fork()에서는 스레드인 경우 해당 스레드의 주소 공간의 내용을 복사하고 새 프로세스를 시작해야하는데, 스레드의 주소 공간은 어차피 process와 공유된 상태이기 때문에 fork의 기존 코드로도 문제 없이 실행 될 것이라 판단하였다. exec()는 해당 스레드를 제외한 기존 프로세스의 모든 스레드들은 종료되고, 해당 스레드가 새로운 프로세스로써 시작해야하기 때문에 실행중인 것이 스레드인 경우 프로세스로써 작동하기 위해 proc 구조체 안에 새로 추가한 스레드에 대한 필드들의 값이 초기화되야 한다고 생각했다. 또한 나머지 스레드들을 종료하기 위한 코드도 필요해보였다. 그 외에는 기존 exec와 똑같이 작동한다. 그리고 sbrk, kill, sleep은 스레드와 프로세스 모두 같은 방식으로 처리되기 때문에 따로 처리할 필요가 없어보였다. 더불어 과제 명세에서 thread_exit에서는 시작 함수의 끝에 도달하여 종료하는 경우는 고려하지 않기 때문에 함수가 끝나 exit을 하는 경우에 대해 처리하기 위해 exit도 수정이 필요해보였다. exit()에서 스레드가 좀비인 경우로 인해 에러가 나면 안 되기 때문에 이에 대한 코드를 추가하기로 했다.

이러한 구현 방식을 생각한 후 코드를 작성했다.

B. Locking

먼저 locking 시스템을 구현하기 위해 수업시간에 배운 프로세스 동기화를 위한 알고리즘을 복습했다. 기본적으로 Peterson's Algorithm을 사용하면 동기화 문제를 해결할 수 있다. 하지만 peterson's algorithm은 스레드가 두개인 경우를 가정하고 두개의 스레드의 상호배제를 위해 작동한다. 만약 여러개의 스레드를 위해 동기화 문제를 해결하려면 flag 배열이 스레드 수만큼 늘어나고 flag 접근에 대한 상호배제도 필요하여 또 다른 상태 변수들이 필요할 것이다. 피터슨 알고리즘을 통해 N개의 스레드 race condition을 해결하기에는 너무 복잡하고 무리가 있어보였다. 그래서 그 다음으로 배운 하드웨어의 도움을 받아 동기화 문제를 해결하는 방법을 생각했다. 근본적으로 동기화 문제를 해결하기에는 소프트웨어보다는 하드웨어의 도움을 받는 것이 낫다고 판단하였다. Atomic hardware instructions를 사용한다면 한 cycle에 중간에 끊기지 않고 실행되는 것을 보장하여 한번에 하나의 스레드만이 접근할 수 있도록 한다. 그리하여 하드웨어적인 도움을 받아 원자성을 보장하는 Test-and-Set (TAS)와 Swap, 그리고 교수님께서 수업시간에 잠깐 언급하신 Compare-and-Swap (CAS)를 고려해보았다. 먼저 TAS는 lock이

걸렸는지 안 걸렸는지를 확인하여 critical section에 접근 가능한지를 나타낸다. Swap은 공유변수 lock과 의지를 나타내는 key값을 계속 바꾸어 lock이 풀릴때까지 대기하도록 해준다. CAS는 기대값과 비교하여 값이 일치하면 새로운 값을 할당해주어 여러 상태 변화의 상황에서 유용하게 사용된다. 이 중에서도 현재 상태를 반영할 수 있는 CAS를 이용해보고자 했다.

Compare-and-Swap은 원자성을 보장한다는 전제하에 동기화 문제를 해결한다. 이 원자성을 보장하기 위해서는 하드웨어의 도움을 받아야한다. 그리하여 직접적으로 하드웨어적으로 접근하기 위해 어셈블리어를 사용하기로 했다. 어셈블리어로 작성해야만 원자성을 보장해주는 instruction을 사용하여 구현할 수 있다. 그래서 CAS에 관한 함수를 하나 만들어야겠다고 생각했다. 이 함수는 현재 메모리에 저장되어 있는 값이 기대값과 같은지 비교하고, 서로 같으면 새로운 값을 저장해주고 기존의 값은 반환해주는 역할을 한다. swap과 비슷한 역할을 하여 만약 어떤 한 스레드가 lock을 잡고 있다면 다른 스레드들은 lock이 풀릴때까지 대기해야한다. 락이 한 스레드에게 점유되었을을 알기 위해 공유 변수 lock_flag가 필요할 것이다. 그리고 한 스레드가 락을 잡게 되면 해당 스레드는 critical section에 진입 가능하다. 임계영역에서 할일을 마치면 unlock을 통해 락을 놓게 된다. 이를 통해 mutual exclusion과 progress가 보장됨을 알 수 있다. 그리고 CAS는 원자성을 보장하므로 deadlock이 걸리지 않게 한다. 하지만 while문을 사용하기에 대기하는 동안에도 CPU 자원을 쓰게 되어 busy waiting이 일어난다. Busy waiting으로 인한 자원 낭비, 소요 시간, overhead를 줄이기 위해 exponential backoff strategy를 함께 사용할 것이다. Exponential backoff 전략은 CAS를 사용한 while문에 진입하게 되면 락을 잡을때까지 대기를 해야하는데, 대기할때 CPU를 사용하는 대신 usleep을 통해 딜레이를 만들어 잠시라도 overhead 및 자원 낭비를 하지 않는 것이다. 이 딜레이 되는 시간을 2배씩 증가하여 지수적으로 늘어나게 한다. 하지만 무한 대기와 무한으로 길어지는 대기 시간에 대비하기 위해 늘어나는 대기 시간에 제한을 두어 bounded waiting을 보장할 것이다. 이렇게 되면 성능향상과 함께 언젠가 모든 스레드들이 락을 잡을 수 있게 되고 한 스레드만이 critical section에 접근 가능하여 동기화 문제를 해결하기 위한 3가지 조건, mutual exclusion, progress, 그리고 bounded waiting을 보장하게 되어 하나의 솔루션이 될 수 있다.

II. Implement

1. Light-Weight Process (LWP)

A. Proc.h 수정

```
int isThread; //1 if thread, 0 if process
struct proc *master;  //master thread that may create sub threads
int tid; //thread id
void *retval; //return value from join
```

위에서 설명하였듯이 isThread는 실행되고 있는 것이 프로세스이면 0, 스레드이면 1을 저장한다. master는 해당 스레드가 어느 프로세스에서 시작되었는지를 알려주는 필드이다. tid는 pid처럼 해당 스레드의 id를 저장한다. retval은 join과 exit을 할때 사용되는 join의 반환값을 저장한다.

B. thread_create()

먼저 스레드들은 thread_create()을 통해 생성된다. 여기에서 tid 값을 저장하기 위해 nextpid와 함께 nexttid 전역 변수를 추가해줬다:

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

위에서 설명한 것과 같이 해당 API는 fork와 exec을 참고하여 구현하였다.

```
int thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg){
        struct proc *np;
        struct proc *curproc = myproc();
        uint sp, ustack[2];
        int i;
        struct proc *master;

        //reference from fork()
        if((np = allocproc()) == 0){ //allocate thread
                return -1;
        }
        nextpid--; //since allocated process was a thread
```

→ 먼저 fork를 참고하여 필요한 변수들을 선언해주었다. 그 다음 fork 함수의 시작과 같이 allocproc()을 통해 사용중이지 않는 프로세스 하나를 스레드로 할당해준다. 만약 성공적으로 스레드를 만들었다면 이제 해당 프로세스는 완전히 스레드이기 때문에 nextpid의 값을 하나 낮췄다. 여기에 allocproc()에 대한 설명을 덧붙이자면 아래에 보이는 것처럼 스레드에 관한 변수들을 초기화 해주었다. 이는 할당된 프로세스가 스레드가 아니고 아직 tid가 할당되기 전인 상태를 나타낸다. 물론 스레드가 아니기에 마스터 스레드 또한 없다. allocproc에서 이미 nextpid를 하기 때문에 thread_create에서 스레드 생성 후 nextpid--;를 해준 것이다.

→
```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  p->isThread = 0; //initializing variables for project03
  p->tid = -1;
  p->master = 0;

  release(&ptable.lock);
```
in allocproc()

```
        //analyze the master thread
        if(curproc->master == 0){
                master = curproc;
        }
        else{
                master = curproc->master;
        }
```

→ 다음으로 스레드는 프로세스에서 생성되는 master thread에서 자식 스레드가 생성되어야한다. 그리하여 해당 스레드의 master 스레드를 찾는 코드를 넣어주었다. master의 초기값은 0이기 때문에 만약 값이 0 이라면 기존 프로세스, 즉 마스터 스레드라는 뜻이다. 0이 아닌 다른 프로세스/스레드 주소를 가지고 있다면 현재 스레드의 마스터 스레드를 저장해준다.

```
        if(master->pgdir == 0){
                np->state = UNUSED;
                return -1;
        }
```

→ 그리고 또 다른 에러 처리를 위해 만약 마스터 스레드의 페이지 디렉토리가 0이면, 즉 마스터 스레드가 사용 불가능한 상태이면 스레드 생성에 실패한 것으로 -1을 반환하며 끝내게 해줬다.

```
master->sz = PGROUNDUP(master->sz);
if ((master->sz = allocuvm(master->pgdir, master->sz, master->sz + 2 * PGSIZE)) == 0) {
        return -1;
}
clearpteu(master->pgdir, (char*)(master->sz - 2 * PGSIZE));
sp = master->sz;
```

→ 다음은 exec를 참고하여 새 스레드를 위해 마스터의 주소 공간을 2페이지만큼 늘려주었다. allocuvm을 통해 해당 작업을 해주고 스택 포인터에 마스터의 스택 사이즈를 저장하였다.

```
np->parent = master;
np->pid = master->pid;
np->master = master;
np->pgdir = master->pgdir;
np->sz = master->sz;
*np->tf = *master->tf;

np->isThread = 1;
np->tid = nexttid++;
*thread = np->tid;
```

→ 그 후 생성된 스레드를 스택에 넣어 공유된 자원에 추가하기 전에 새 스레드에 대한 정보를 설정해주었다. 이건 프로세스가 아닌 스레드이기 때문에 프로세스에 대한 정보는 master에 대한 정보로 채워주고, 이제 스레드이기 때문에 isThread = 1, tid는 nexttid를 이용하고, 과제 명세의 요구조건처럼 인자로 받은 thread 주소에 tid를 저장했다.

```
ustack[0] = 0xffffffff;
ustack[1] = (uint)arg;
sp -= 8; //2 * 4 to decrease stack by 2
if(copyout(master->pgdir, sp, ustack, 8) < 0){
        return -1;
}

np->tf->eip = (uint)start_routine;
np->tf->esp = sp;
```

→ 다음은 선언해둔 임시 스택 ustack에 리턴 함수에 대한 정보를 담은 후 스택 포인터를 임시스택 사이즈만큼 줄여주었다. 스택 프레임에 있는 내용들이 copyout을 통해 해당 스레드의 공간에 저장된다. 그 후 instruction pointer인 eip에 시작 함수를 저장하고, 스택 포인터인 esp에 새롭게 조정된 sp 값을 넣어줬다.

```
        for(i = 0; i < NOFILE; i++)
                if(master->ofile[i])
                        np->ofile[i] = filedup(master->ofile[i]);
        np->cwd = idup(master->cwd);

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

        acquire(&ptable.lock);
        np->state = RUNNABLE;
        release(&ptable.lock);

        return 0;
}
```

→ 마스터 스레드의 파일 디스크립터 내용들을 새 스레드에 복사해준 후 새 스레드를 스케줄링에 포함시키기 위해 runnable 상태로 만들어주었다.

C. thread_exit()

```
void thread_exit(void *retval){
        struct proc *curproc = myproc();
        struct proc *p;
        int fd;

        //reference from exit()
        if(curproc == initproc)
                panic("init exiting");

        for(fd = 0; fd < NOFILE; fd++){
                if(curproc->ofile[fd]){
                        fileclose(curproc->ofile[fd]);
                        curproc->ofile[fd] = 0;
                }
        }

        begin_op();
        iput(curproc->cwd);
        end_op();
        curproc->cwd = 0;

        acquire(&ptable.lock);
        wakeup1(curproc->master);

        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                if(p->parent == curproc){
                        p->parent = initproc;
                        if(p->state == ZOMBIE)
                                wakeup1(initproc);
                }
        }

        curproc->retval = retval;
        curproc->state = ZOMBIE;
        sched();
        panic("zombie exit");
}
```

thread_exit은 위에서 설명한 바와 같이 exit()를 참고하여 작성하였다. 스레드를 종료하고 값을 반환하면 되기 때문에 exit 코드와 많이 비슷하다. 현재 스레드의 자원을 회수한 후 현재 스레드를 기다리고 있을지도 모르는 마스터 스레드를 깨운다. 오직 모든 스레드만이 해당 함수를 통해 종료되기에 그럴 경우는 없겠지만 만약 curproc이 마스터인 경우에 대비하여 좀비 스레드를 없애기 위해 loop을 이용하여 initproc을 기다리고 있는 좀비 상태의 스레드를 initproc을 깨워 없애주었다. 그 후 반환값인 retval을 저장해주고 좀비 상태로 만들어 종료시켜준다.

시작함수의 끝에 도달하여 종료하는 경우는 해당 시스템 콜에서 고려하지 않기 때문에 아래 exit 부분에서 수정한 부분을 설명하겠다.

D. thread_join()

```
int thread_join(thread_t thread, void **retval) {
        struct proc *p;
        int havekid;
        struct proc *curproc = myproc(); // get the current thread

        //reference from wait()
        acquire(&ptable.lock);
        for (;;) {
                havekid = 0;
                for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
                        if (p->master != curproc) // thread is not a sub thread of current thread
                                continue;
                        if(p->tid != thread)
                                continue;
                        havekid = 1; // found the specified thread
                        if (p->state == ZOMBIE) { // thread already terminated
                                *retval = p->retval; // retrieve the return value
                                kfree(p->kstack);
                                p->kstack = 0;
                                p->pid = 0;
                                p->parent = 0;
                                p->name[0] = 0;
                                p->killed = 0;

                                p->isThread = 0;
                                p->tid = -1;
                                p->master = 0;
                                p->state = UNUSED; //make it unused to safely terminated
                                release(&ptable.lock);
                                return 0;
                        }
                }
                // no specified thread to wait for
                if (!havekids || curproc->killed) {
                        release(&ptable.lock);
                        return -1;
                }
                // wait for the specified thread to terminate
                sleep(curproc, &ptable.lock);
        }
}
```

thread_join은 인자로 받은 스레드가 종료되기까지 대기하는 역할이기 때문에 wait()를 참고하였다. 지정된 함수에만 기다리면 되기에 tid를 통해 해당 스레드가 좀비 상태, 즉 종료된 상태인지 확인해주었다. 아직 종료되지 않았다면 for loop에서 sleep을 통해 계속 대기하게 된다. 또한 현재 스레드가 자식 스레드를 생성한 적이 없거나 이미 종료된 상태라면 이는 잘못된 종료이기 때문에 -1을 반환하며 종료한다. 만약 지정된 스레드가 무사히 종료되었다면 ZOMBIE 상태인지 확인 후 thread_exit에서 받아온 반환값을 저장하고 자원을 회수하여 초기화시켜준다. UNUSED 상태로 전환시켜 완전히 종료된 스레드임을 알린 후 정상적으로 join시켰다는 의미로 0을 반환한다.

E. System Calls 수정

위 세가지 API와 상호작용하기 위해 기존의 시스템 콜들에도 수정이 필요했다. 위에서 언급한 것처럼 nexttid 변수 추가와 함께 allocproc에서 프로세스 상태일때 스레드에 대한 변수들의 값을 초기화 해주는 코드를 추가 해주었다.

growproc:

```
int
growproc(int n)
{
  uint sz;
  struct proc *curproc = myproc();

  if(curproc->isThread == 0){ //if process
    sz = curproc->sz;
    if(n > 0){
      if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
        return -1;
    } else if(n < 0){
      if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
        return -1;
    }
    curproc->sz = sz;
  }
  else{ //if thread
    sz = curproc->master->sz; // use stack size of master thread
    if(n > 0){
      if((sz = allocuvm(curproc->parent->pgdir, sz, sz + n)) == 0)
        return -1;
    } else if(n < 0){
      if((sz = deallocuvm(curproc->parent->pgdir, sz, sz + n)) == 0)
        return -1;
    }
    curproc->master->sz = sz;
  }

  switchuvm(curproc);
  return 0;
}
```

→ 프로세스일때와 스레드인 때를 구분하여 프로세스의 경우 원래 코드대로 작동하고, 스레드인 경우 안전하게 해당 스레드의 마스터 스레드를 이용하여 스텍 사이즈를 조정할 수 있도록 했다.

fork:

```
  np->sz = curproc->sz;
  //edited for project03
  if(np->isThread == 1) //if thread
    np->parent = curproc->master;
  else
    np->parent = curproc;
  *np->tf = *curproc->tf;
```

→ fork에서도 프로세스와 스레드인 경우를 나누어 처리해주었다. 만약 스레드면 마스터 스레드를 부모로 저장해주었다. 이는 스레드에서 fork가 호출될때 기존의 fork 루틴을 문제없이 실행할 수 있게 하기 위해 구분하여 처리해주었다. 나머지 부분은 수정없이도 스레드가 프로세스처럼 주소공간을 복사하고 새로운 프로세스를 시작하도록 해준다.

exit:

```
if(curproc == initproc)
  panic("init exiting");

int havekids;
acquire(&ptable.lock); //edited for project03
//used wait() as reference
if(curproc->tid == -1){ //if it is a master thread or a process
  for(;;){
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->pid != curproc->pid || p == curproc || p->isThread == 0) //if process
        continue;
      //found a thread
      if(p->state == ZOMBIE){
        kfree(p->kstack);
        p->kstack = 0;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->tid = -1;
        p->isThread = 0;
        p->master = 0;
        p->state = UNUSED;
      }
      else{
        havekids++;
        p->killed = 1;
        wakeup1(p);
      }
    }
    if(havekids == 0){ //no need to wait if there's no thread
      break;
    }
    sleep(curproc, &ptable.lock);
  }
}
release(&ptable.lock);
```

→ 위에서 설명하였듯이 시작함수의 끝에 도달하여 종료하는 경우는 exit에서 처리해주었다. 여기서는 wait()을 참고하여 종료하려는 프로세스의 스레드들이 종료가 될때까지 기다려주는 역할을 하게 했다. 먼저 현재 실행중인 프로세스가 thread를 갖고 있는지 확인해주어 만약 스레드가 없다면 안전하게 loop을 빠져나가게 해주었다. 만약 스레드를 포함하고 있다면 해당 스레드가 좀비인지, 즉 종료상태인지 확인해준다. 만약 좀비라면 자원을 회수하고 값을 초기화한 후 UNUSED 상태로 만들어 완전히 종료되게 해준다. 아직 종료 상태 전이라면 killed = 1을 하여 종료를 원함을 표시하고 wakeup1()을 이용하여 종료 전 필요한 동작들을 수행하게 해준다. 여기에서 좀비 상태로 바뀌어 다시 ptable을 순회했을때 좀비 상태에서 UNUSED 상태로 바뀌어 안전하게 자원이 회수되고 종료될 수 있다. 이 동작은 좀비 스레드를 예방하기 위한 코드이다.

```
// Parent might be sleeping in wait().
if(!(curproc->master)) //edited for project03
  wakeup1(curproc->parent);
else{ // if it is a thread
  curproc->master->killed = 1; // kill the master thread to terminate
  wakeup1(curproc->master);
}
```

→ 그리고 fork에서 대기상태인 부모를 위한 코드를 수정해줬다. 만약 마스터 스레드거나 프로세스인 경우 master 값은 0이다. 이 경우 그들의 부모 프로세스를 깨워준다. 만약 스레드인 경우 마스터 스레드를 killed = 1 해주어 종료 상태로 만들어 wakeup1()을 했을때 그의 모든 자식 스레드가 모두 종료될때까지 대기하고 무사히 종료하게 해준다. 이는 스레드가 시작함수의 끝에 도달하여 종료하는 경우를 처리해준다. kill 시스템 콜을 참고하여 작성하였다.

Sleep, kill은 따로 수정하지 않아도 스레드와 알맞게 상호작용한다.

exec in exec.c:

```
struct proc *curproc = myproc();
struct proc *p;

acquire(&ptable.lock);
if(curproc->isThread == 1){ //make the current thread to be the next executed process
  curproc->tid = -1;
  curproc->isThread = 0;
  curproc->parent = curproc->master->parent;
  curproc->master = 0;
}

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->pid == curproc->pid && p != curproc){ //terminate all other threads, also including the original process
    kfree(p->kstack);
    p->kstack = 0;
    p->pid = 0;
    p->parent = 0;
    p->name[0] = 0;
    p->killed = 1;
    p->isThread = 0;
    p->tid = -1;
    p->master = 0;
    p->state = ZOMBIE;
  }
}
release(&ptable.lock);
```

→ exec가 실행되면 기존 프로세스의 모든 스레드가 종료되어야한다. 또한 한 스레드에서 새로운 프로세스로써 시작해야한다. 그래서 만약 현재 실행중인 것이 스레드라면 프로세스처럼 작동하기 위해 스레드에 대한 변수들을 초기화해주었다. 여기서 부모 프로세스에는 마스터 스레드의 부모 프로세스가 저장된다. 그 다음 다른 모든 스레드들을 종료해주기 위해 pid가 같은, 즉 같은 프로세스에 포함된 스레드들을 좀비 상태로 만들어주고 다른 정보들은 초기화시켜주었다. 여기에서 UNUSED가 아닌 ZOMBIE를 사용한 이유는 좀비로 만들어야 부모 프로세스가 좀비 프로세스 또는 스레드의 자원을 회수하고 종료될때까지 기다를 수 있기 때문이다. 이 외에 다음 기존 코드들을 따라가면 프로세스와 스레드가 정상적으로 상호작용하며 작동한다.

sys_sbrk in sysproc.c:

```
int
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;
  if(myproc()->isThread == 0) //if it is process
    addr = myproc()->sz;
  else //if it is thread
    addr = myproc()->master->sz;
  if(growproc(n) < 0)
    return -1;
  return addr;
}
```

→ sbrk에도 스레드와 프로세스 사이에 혼동을 주지 않기 위해 스레드인 경우 마스터의 sz를 사용하게 해주었다.

System calls for thread_create, thread_exit, thread_join:

```
//edited for project03
int
sys_thread_create(void){
  thread_t *thread;
  void *(*start_routine)(void *);
  void *arg;

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

int
sys_thread_exit(void){
  int retval;
  if(argint(0, &retval) < 0)
    return -1;
  thread_exit((void*)retval);
  return 0;
}

int
sys_thread_join(void){
  int thread;
  void **retval;

  if(argint(0, &thread) < 0 || argptr(1, (char**)&retval, sizeof retval) < 0)
    return -1;
  return thread_join((thread_t)thread, retval);
}
```

→ 위 코드는 thread_create, thread_exit, thread_join의 wrapper functions로 유저 프로그램에서도 사용할 수 있도록 해준다. 시스템 콜로 등록하기 위해 defs.h, Makefile, syscall.h, syscall.c, user.h, usys.S를 수정해주었다. 또한 Makefile에 테스트를 위한 파일들(thread_test, thread_exec, thread_exit, thread_kill, hello_thread)을 등록해주었다.

2. Locking

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int shared_resource = 0;
volatile int lock_flag = 0;

#define NUM_ITERS 10
#define NUM_THREADS 10
#define MAX_ATTEMPTS 10 //set the maximum number of backoff attempts
#define INITIAL_BACKOFF_INTERVAL 10 //set the initial backoff interval in microseconds
#define MAX_BACKOFF_INTERVAL 10000 //set the maximum backoff interval in microseconds
```

→ 위에서 언급한대로 lock을 누군가가 잡고 있는지를 확인하기 위해 공유 변수인 lock_flag를 선언해주었다. 한 스레드가 점유하고 있다면 1, 사용가능한 상태이면 0 값을 갖게 된다. 여기에 하드웨어 인터럽트, 여러 스레드에 인한 변화 등 어떠한 방해에 의해 예기치 못하게 컴파일러에 의해 값이 변화하는걸 막기 위해 변수에 volatile을 붙여주었다. 그리고 lock()에서 exponential backoff를 구현할 때 사용할 usleep을 위해 unistd 헤더파일을 포함해주었다. 그 아래에 딜레이 시간을 늘릴 수 있는 기회를 최대 10으로 두어 무기한 대기시간으로 이어지지 않게 해주어 bounded waiting을 보장해주었다. 딜레이하는 시간의 범위는 10~10000으로 두었다. 최대 10000으로 둔 이유는 10 microseconds로 시작해 최대 10번까지 지수적으로 시간이 늘어난다면: 10 * 2^10 = 10240이 된다. 그래서 최대치를 10240과 비슷하게 10000으로 설정해주었다.

```c
int compare_and_swap(volatile int*, int, int);
void lock();
void unlock();

int compare_and_swap(volatile int *ptr, int expected, int new_value) {
        int old_value;
        __asm__ volatile(
                "lock cmpxchg %2, %1\n\t"
                : "=a"(old_value), "+m"(*ptr)
                : "q"(new_value), "0"(expected)
                : "memory"
        );
        return old_value;
}
```

→ 해당 알고리즘을 구현하여 동기화 문제를 해결하기 위해서 세가지 함수를 사용했다. Lock, unlock에 더불어 compare_and_swap 함수를 추가해주었다. 이 함수에는 전에 설명했다시피 원자성을 하드웨어의 도움을 받아 보장하기 위해 어셈블리어로 작성하였다. 기본적으로 CAS는 주어진 메모리 위치에 현재 값이 기대한 값과 동일한 경우에만 새로운 값을 저장하게 해준다. 따라서 ptr에 있는 값, 즉 lock_flag가 점유가능한 상태이면 expected, 기대값과 같아 해당 스레드가 lock을 차지할 수 있게 된다. 위 inline assembly 언어를 해석해보자면, 먼저 "lock"이라는 prefix를 사용하여 CPU 명령어 레벨에서 비교 및 교환을 해주는 cmpxchg 명령어를 수행한다. "lock"은 해당 명령어가 원자적으로 실행되도록 보장하여 여러 스레드가 lock_flag에 대해 CAS를 시도하여도 오직 하나의 스레드만 접근하여 lock 차지에 성공할 수 있다. ("lock"은 c library에 포함된 동기화 API가 아닌 하드웨어적으로 제공하는 instruction이다.)

"__asm__ volatile"을 통해 인라인 어셈블리 코드의 시작을 나타낸다. volatile을 붙여 컴파일러가 이 코드에 대해 최적화하지 않도록 한다.

"lock cmpxchg %2, %1\n\t"에서 %1은 ptr, %2는 expected를 나타낸다. (new_value는 %3이다.) lock cmpxchg를 통해 ptr의 값과 expected의 값이 동일한 경우에만 ptr에 new_value를 저장한다. 이 이후의 코드는 입출력 제약에 관한 내용을 담고 있다.

: "=a"(old_value), "+m"(*ptr)에서 출력에 관하여 "=a"(old_value)는 eax(=a)에 있는 값을 old_value에 반환하고, 입력에 관하여 "+m"(*ptr)은 ptr에 대해 읽기 및 쓰기를 수행한다. old_value는 나중에 리턴값으로 쓰인다.

: "q"(new_value), "0"(expected)는 new_value와 expected를 해당 register에 각각 할당한다.

: "memory"는 어셈블리 코드가 메모리에 직접 접근할때 컴파일러가 해당 영역을 최적화하지 않게 해준다. 따라서 항상 메모리의 실제 값을 읽고 쓰도록 보장해준다.

이렇게 하드웨어적인 도움을 받아 구현하여 원자성을 보존하여 동기화 문제도 해결할 수 있다.

```
void lock() {
        int expected = 0;
        int attempts = 0; //attempts of backoff
        int delay = INITIAL_BACKOFF_INTERVAL; //start the delay time with initial backoff interval

        while (compare_and_swap(&lock_flag, expected, 1) != expected) {
        //exponential backoff: increase wait time exponentially up to a maximum number of attempts
                if (attempts < MAX_ATTEMPTS) {
                        usleep(delay); // delay without using CPU
                        // double the backoff interval, but till MAX_BACKOFF_INTERVAL
                        delay = delay * 2 < MAX_BACKOFF_INTERVAL ? delay * 2 : MAX_BACKOFF_INTERVAL;
                        attempts++;
        } else {
            // if maximum backoff attempts reached, use a fixed wait time
            usleep(MAX_BACKOFF_INTERVAL);
        }
    }
}
```

→ 다음은 compare_and_swap 함수를 이용한 lock 함수 구현이다. 일단 스레드 id와 무관하게 진행되므로 인자로 아무것도 받지 않는다. 먼저 기대값을 0으로 설정 후 시작하고 exponential backoff를 시도하는 변수도 초기화해주었다. 그리고 가정 처음 backoff를 시도할때의 딜레이 시간을 위에서 선언해준 값으로 할당해준다. 이제 while문을 통해 compare_and_swap을 사용하여 락을 잡을 때까지 대기하게 해준다.

먼저 lock_flag 변수를 0으로 설정하여 lock을 해제하고 시작한다. 한 스레드가 compare_and_swap 함수를 사용하여 lock_flag를 1로 설정하려고 시도하려고 할때, 기대값으로는 0을 사용하고, new_value에는 1을 사용한다. 만약 lock_flag가 이미 1로 설정되어 있다면 다른 스레드가 임계 영역을 보호하고 있는 것이므로 lock_flag = 1, expected = 0 으로 값이 달라 lock을 차지하지 못하게 되어 대기하게 된다. 이때 대기 시간 동안 CPU 자원을 소비하지 않는 대신 주기적으로 compare_and_swap 함수를 호출해서 임계 영역에 접근할 수 있는지 확인한다. 대기 시간은 지수적으로 증가하는 백오프 알고리즘을 사용하여 점차 대기 시간을 늘린다. 처음에는 짧은 대기 시간 (=10 microseconds)부터 시작해서 여러 번 시도하면서 대기 시간을 지수적으로 늘려가다가 시도 가능한 횟수(=10번)에 다다르면 최대 대기 시간(= 10000 microseconds)으로 대기 후 다시 시도한다. 이렇게 최대 대기 시간을 설정해두어

무기한으로 대기하지 않도록 보장해준다. 이렇게 공유변수 lock_flag에 대해 atomic 함수인 compare_and_swap을 사용하면서 대기시간을 정해두어 mutual exclusion, progress, 그리고 bounded waiting을 보장할 수 있게 된다.

```
void unlock() {
        lock_flag = 0;
}
```

→ 마지막으로 unlock에서는 원자적으로 lock을 차지했던 것을 해제하는 작업을 한다. 이미 lock_flag를 원자적으로 점유했기 때문에 하나의 스레드만이 lock_flag의 값을 바꿀 수 있다. 그렇기에 해제할 때에는 atmoic 함수를 사용하지 않아도 된다고 판단하여 lock_flag = 0을 통해 값을 바꿔주어 다른 스레드가 lock_flag를 차지할 수 있도록 했다.

III. Result

A. Light-Weight Process (LWP): 컴파일 및 실행

Compile:

make → make fs.img → ./bootxv6.sh 순으로 명령어를 작성하여 컴파일하면 아래와 같이 실행되었다.

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ thread_test
Test 1: Basic test
```

여러번 실행을 하였을때, 과제 명세에 명시된 것처럼 출력문이 꼬이고 순서가 달라져 뒤죽박죽 나올때가 있었다. 하지만 논리에 따라 에러없이 그대로 실행되니 문제는 없었다.

Test 1. Thread_test 1

```
$ thread_test
Test 1: Basic test
Thread 0 start
Thread 0 end
Thread 1 start
Parent waiting for children...
Thread 1 end
Test 1 passed
```

위 결과를 보면 과제 명세의 테스트 출력문 예시와 같게 나온 걸 볼 수 있다. 이는 위에서 구현한 thread_create, thread_exit, thread_join이 잘 적용되어 스레드 사이에서 메모리가 잘

공유된 걸 뜻한다. 스레드 0이 시작 후 바로 종료하고, 스레드 1은 잠깐 대기 후 종료된걸 볼 수 있다. thread_create을 통해 두 스레드가 생성되고, thread_basic이라는 함수에서 시작하여 두번째 스레드만 2초 대기 후 둘 다 thread_exit을 통해 종료된다. 그 후 thread_join을 통해 마스터 스레드가 두 스레드의 종료를 기다리고 무사히 종료하여 테스트가 완전히 통과한 걸 볼 수 있다. 다만 여러번 실행하였을때 출력문이 꼬이는 현상이 있었지만 그래도 스레드 0,1의 시작과 스레드 0의 종료는 항상 "parent waiting for children…" 문구 이전에 출력되는 것을 보아 문제가 되진 않는다.

Test 2. Thread_test 2

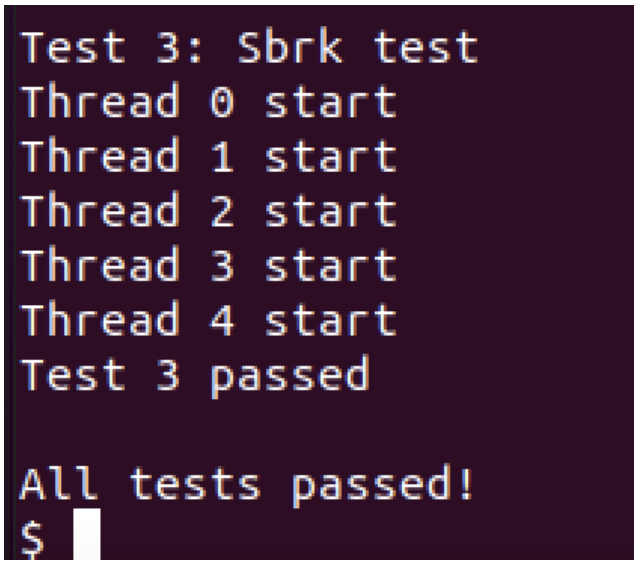
```
Test 2: Fork test
Thread 0 startThread 1 start

CThread 2 start
Thread 3 starthild of thread
Thread 4 start
1 Child of thread 0 start
Child of thread Child of thread Child of thread start
2 start
3 start
4 start
Child of thread 0 end
Child of thread 1 end
Child of thread 2 end
Child of thread 3 end
Child of thread 4 end
Thread 0 end
Thread 1 end
Thread 2 end
Thread 3 end
Thread 4 end
Test 2 passed
```

출력이 조금 꼬였지만 0~4까지 5개 스레드 모두 무사히 생성되어 시작한 걸 볼 수 있다. 그리고 각각의 스레드마다 fork를 통해 자식 스레드를 생성하였다. 여기에서도 출력문이 꼬였지만 1~4까지 자식 스레드가 생긴걸 볼 수 있다. 그리고 1초 뒤 자식 스레드들이 끝나고 thread_exit을 통해 부모 스레드들도 종료된다. 여기서 부모 스레드들은 thread_join에서 자식 스레드들이 종료되길 기다리는데 이미 자식 스레드들이 종료되었으므로 바로 뒤이어 "Thread # end" 출력문을 볼 수 있었다. 이렇게 해서 두번째 테스트도 통과하였다. 이때 fork를 통해 자식 스레드를 생성하였으므로 부모와 자식 스레드 간에 주소 공간이 분리되어있어 메모리 상 내용은 같지만 주소 공간을 공유하지는 않기에 아무 에러가 발생하지 않는 것을 확인 할 수

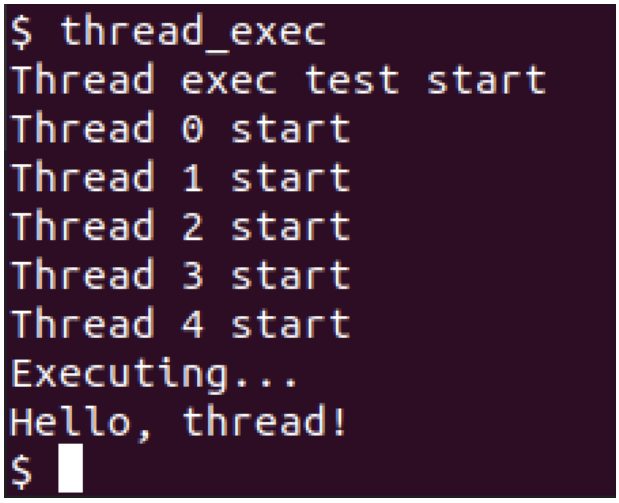
있다. 여러번 실행하였을때 출력문이 대부분 꼬이는 경우가 많지만 그대로 대체로 스레드 0~4가 먼저 시작을 하고, 자식 스레드 0~4가 시작을 한 후 종료를 하며 그 뒤에 부모 스레드 0~4가 종료하는 것을 보아 문제가 되진 않아 보인다.

Test 3. Thread_test 3



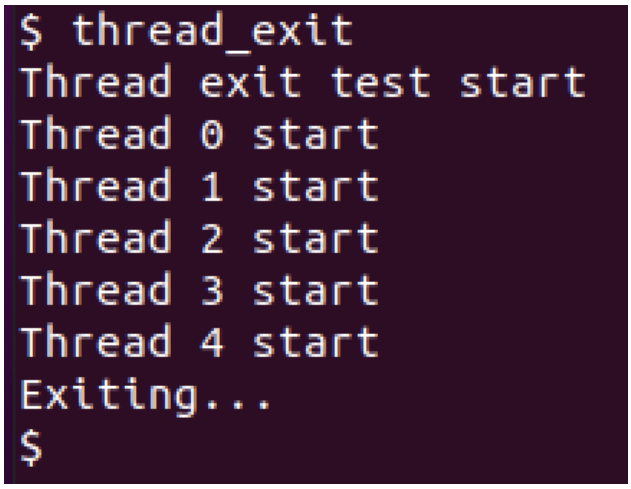
thread_test의 마지막 테스트는 malloc을 하였을때 sbrk가 알맞게 작동하는지에 대한 것이다. 스레드들은 주소 공간을 공유하기 때문에 메모리가 할당될 때 모든 스레드들이 접근하는데 문제가 없어야 한다. 또한 주소 공간을 공유하는만큼 같은 주소에 중복 할당 해서는 안된다. 실행 도중 주소공간에서 하나의 스레드가 다른 스레드를 찾았다는 에러 문구가 뜨지 않으므로 무사히 sbrk가 내부적으로 호출되어 중복 할당되지 않았음을 알 수 있다. 이것 또한 출력문이 간혹 꼬일때가 있었지만 에러없이 테스트를 통과하였으므로 문제가 되진 않는다. 다만 "Test 3 passed"를 출력하는데 약 5초 정도 걸리는 현상이 있다. 이는 유저 프로그램의 함수에서 malloc을 수행하고 free 하고 sleep을 하여 시간이 걸리는 듯하다. 하지만 정상적인 출력을 하는데에는 문제가 없다.

Test 4. Thread_exec



스레드에서도 exec가 잘 실행되어야한다. 만약 스레드에서 다른 프로그램을 실행한다면 다른 스레드들은 모두 종료되어야 한다. 스레드 5개가 생성된 후 스레드 0에서 exec를 통해 hello_thread 프로그램을 실행시킨다. 이때 exec에서 스레드 0을 제외한 다른 스레드들을 모두 종료시키기 때문에 스레드 0이 hello_thread에서 무사히 종료된다. 그렇기에 에러 메시지가 없고, 다른 스레드들이 종료되지 않아 재부팅되거나 하는 현상은 없는 것을 볼 수 있다.

Test 5. Thread_exit



스레드에서도 exit이 잘 동작해야 한다. 위에서 exit의 수정된 코드를 설명한 바와 같이 스레드에서 exit을 하는 경우 그의 마스터 스레드를 killed = 1 해주어 종료 상태로 만들고 그의 모든 자식 스레드, 0~4가 모두 종료될때까지 대기하고 무사히 종료하게 해준다. 그리하여 아무 에러 문구 없이 "Exiting..." 출력 후 대기없이 바로 쉘로 빠져나가는 것을 볼 수 있다.

Test 6. Thread_kill

```
$ thread_kill
Thread kill test start
Killing process 4
This code should be executed 5 timeThis code should beThis code should beThis code should be This code should be es.
 executed 5 times.
 executed 5 times.
executed 5 times.
xecuted 5 times.
Kill test finished
$
```

부모 프로세스에서 fork를 통해 자식 프로세스를 생성한 후 각각 5개의 스레드를 생성한다. 부모 프로세스의 스레드 0이 자식 프로세스를 kill한다. 이때 자식 프로세스가 종료되기 때문에 그의 스레드들이 모두 종료될 때까지 대기 후 완전히 종료된다. 위에 exit 코드에서 설명한 바와 같이 프로세스 내에 스레드가 있는 경우에 대비하여 좀비 상태인 스레드들을 찾아 종료시킨 후에 끝내기 때문에 좀비 프로세스/스레드나 다른 에러 사항없이 무사히 테스트가 끝난 걸 볼 수 있다. 다만 출력이 많이 꼬인 것을 볼 수 있지만 여전히 아무 에러 없이 특정 문구를 정확히 5번 출력하기 때문에 문제는 없어 보인다.

B. **Locking:** 컴파일 및 실행

gcc -o pthread_lock_linux pthread_lock_linux.c -lpthread → ./pthread_lock_linux -lpthread 순으로 우분투 내에서 명령어를 작성하여 컴파일하면 아래와 같이 실행 되었다. 해당 코드는 어셈블리어를 포함하고 있어 우분투 같은 x86 아키텍처에서 에러없이 실행 가능하다.

```
bada@bada-Standard-PC-Q35-ICH9-2009:~/Downloads/pthread_lock$ gcc -o pthread_lock_linux pthread_lock_linux.c -lpthread
bada@bada-Standard-PC-Q35-ICH9-2009:~/Downloads/pthread_lock$ ./pthread_lock_linux -lpthread
```

- NUM_ITERS = 10, NUM_THREADS = 10:

```
bada@bada-Standard-PC-Q35-ICH9-2009:~/Downloads/pthread_lock$ ./pthread_lock_linux -lpthread
shared: 100
```

NUM_ITERS * NUM_THREADS = 10 * 10 = 100으로 race condition 없이 잘 출력되었다.
딜레이 없이 출력하는 것을 보아 busy waiting과 overhead가 크지 않았고, 결과값이 잘 나온 것을 보아 mutual exclusion, progress, bounded waiting 모두 보장되어 동기화 문제가 해결된 것으로 확인됐다. 스레드와 Iterator 수를 늘려가며 동기화 문제가 잘 해결됐는지 보겠다.

- NUM_ITERS = 100, NUM_THREADS = 100:

```
bada@bada-Standard-PC-Q35-ICH9-2009:~/Downloads/pthread_lock$ ./pthread_lock_linux -lpthread
shared: 10000
```

- NUM_ITERS = 1000, NUM_THREADS = 1000:

```
bada@bada-Standard-PC-Q35-ICH9-2009:~/Downloads/pthread_lock$ ./pthread_lock_linux -lpthread
shared: 1000000
```

- NUM_ITERS = 10000, NUM_THREADS = 10000:

```
bada@bada-Standard-PC-Q35-ICH9-2009:~/Downloads/pthread_lock$ ./pthread_lock_linux -lpthread
shared: 100000000
```

→ 2초의 딜레이 후 출력값이 나왔다.

- NUM_ITERS = 15000, NUM_THREADS = 15000:

```
bada@bada-Standard-PC-Q35-ICH9-2009:~/Downloads/pthread_lock$ ./pthread_lock_linux -lpthread
shared: 225000000
```

→ 3초의 딜레이가 있었다. NUM_ITERS와 NUM_THREADS의 숫자가 커질수록 출력시간이 오래 걸리지만 그래도 결과는 제대로 나오는 것으로 보아 busy waiting으로 인한 시간 소요로 보인다. 하지만 오래 걸린다 하더라도 3-4초 이내이고, 이것 또한 NUM_ITERS 또는 NUM_THREADS의 수가 15000에 근접한 경우이다.

- NUM_ITERS = 10, NUM_THREADS = 100000:

```
bada@bada-Standard-PC-Q35-ICH9-2009:~/Downloads/pthread_lock$ ./pthread_lock_linux -lpthread
Segmentation fault (core dumped)
```

→ 실행 후 5초 후에 segmentation fault가 나왔다. 아래 사진과 같이 ulimit -u 명령어를 통해 우분투 내에서 사용 가능한 유저 프로세스의 수를 확인하니 15314개인 것으로 확인됐다. 따라서 NUM_THREADS를 15314보다 큰 100000으로 할당하였을 때 segmentation fault가 일어났다. 그러니 이것은 작성한 코드상의 문제가 아닌 메모리에 대한 문제로 일어난 exception이다.

```
bada@bada-Standard-PC-Q35-ICH9-2009:~/Downloads/pthread_lock$ ulimit -u
15314
```

- NUM_ITERS = 15314, NUM_THREADS = 15314:

```
bada@bada-Standard-PC-Q35-ICH9-2009:~/Downloads/pthread_lock$ gcc -o prac3 prac3.c -lpthread
bada@bada-Standard-PC-Q35-ICH9-2009:~/Downloads/pthread_lock$ ./prac3 -lpthread
shared: 234518596
```

→ 3초의 딜레이가 있었지만, 최대 프로세스 수를 넘어가지 않아 정확한 값이 나온 것을 볼 수 있다.

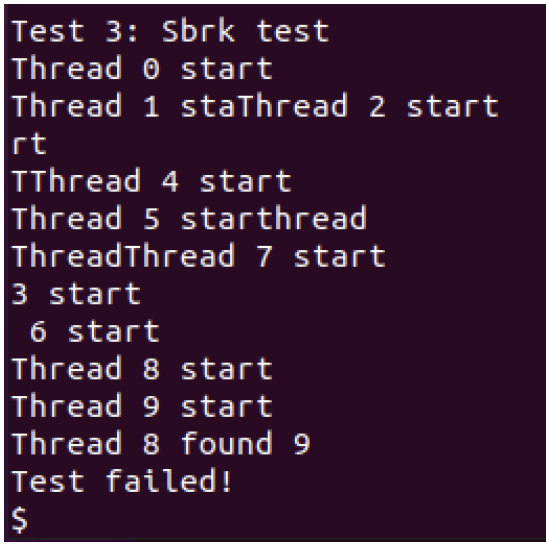
위의 여러 테스트 케이스들을 통해 컴퓨터의 메모리 상 문제를 제외하고 코드 상의 문제가 전혀 없이 출력되는 것을 볼 수 있다. 구현한 코드가 원자성을 보장하며 동기화 문제를 해결하여 race condition이 일어나지 않는다는 것을 확인할 수 있다.

IV. Trouble Shooting

1. Light-Weight Process (LWP)

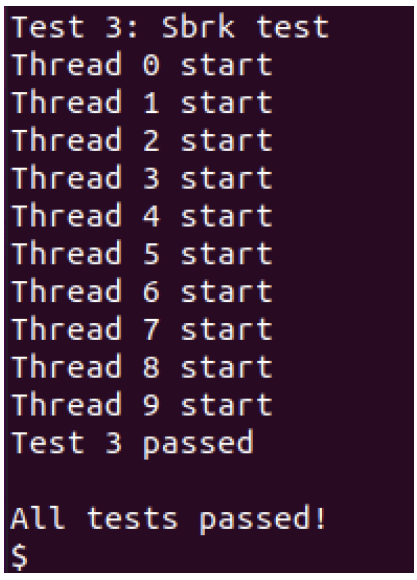
A. thread_test의 sbrk test에서 에러 발생

```
Test 3: Sbrk test
Thread 0 start
Thread 1lapicid 0: panic: remap
 80107107 80107539 8010461e 8010602d 801051ad 801063a1 801060ec 0 0 0QEMU: Termd
```



→ (NUM_THREADS 수를 늘려 실행한 결과이다.)

위와 같이 thread_test에서 테스트 1,2번을 잘 통과한 후 sbrk에서 에러가 발생하였다. 특히 두번째 사진과 같이 공유하는 주소공간에서 다른 스레드에 의해 할당된 공간에 접근하여 테스트가 실패한 경우는 스레드 숫자를 늘렸을때 더 자주 일어났다. panic: remap이 일어났다는 것은 메모리 할당이 잘못되었거나 잘못된 주소 공간에 접근한 것이다. 일단 sys_sbrk에서 확실하게 하기 위해 스레드인 경우에 대해 마스터 스레드를 통해 스택 사이즈에 접근하도록 해주었다. 그리고 growproc에서 프로세스의 메모리 공간을 늘릴때 스레드인 경우를 고려하여 마스터 스레드의 스택 사이즈를 이용하여 pgdir에 변화를 주도록 하였다. 또한 혹시 모를 경우를 위해 thread_create에서 스택사이즈를 늘려 메모리를 할당해준 후 guard 함수인 clearpteu를 통해 page table entries를 정리하도록 해주었다. 이렇게 sz와 pgdir에 접근하는 방식을 바꾸어 다시 실행시키니 아래와 같이 깔끔히 해결되었다.



B. thread_exec에서 재부팅 발생

thread_exec을 실행했을때 "Executing…" 다음에 "This code shouldn't be executed!!"을 출력 후 명시된 다른 유저 프로그램을 실행하지 않고 쉘이 재부팅되는 현상이 일어났다. exec와 thread_exec에 디버깅을 위해 여러 출력문을 넣은 결과 exec에서 제대로 프로그램을 실행시키고 있지 않다는 것을 발견했다. 이는 해당 스레드 외에 다른 스레드들이 제대로 종료되지 않아 문제가 된 것으로 생각해 exec 코드를 수정했다. 처음에는 해당 스레드를 제외한 모든 스레드들을 종료시킬 때 ZOMBIE 상태를 쓰지 않고 UNUSED로 종료시켰다. 하지만 자원을 회수하는데 있어 문제가 생길 것 같아 좀비 상태로 바꿔주었고, killed 값 또한 1로 바꿔주었다. 또한 만약 스레드가 exec을 하는거라면 프로세스처럼 작동하도록 스레드에 대한 필드값들을 초기화 시켜주었다. 이를 수정한 후 다시 실행시키니 "This code shouldn't be executed!!" 출력문은 더이상 뜨지 않았지만 재부팅이 되는 것은 여전했다. 이 문제를 해결하기 위해 수정한 proc.c 코드를 살펴보니 master 스레드에 대한 로직이 잘 정립되지 않은 것 같았다. 그리하여 프로세스인 경우와 스레드인 경우를 나누어 스레드는 항상 마스터 스레드를 통해 메모리에 접근하도록 해주었다. 전체적인 코드 수정 후 다시 실행시키니 재부팅 문제가 해결되었다.

C. thread_exit에서 에러 발생

```
$ thread_exit
Thread exit test start
Thread 0 stThread 1 start
Thread Thread 3 start
2Thread 4 starart
 start
t
Exiting...
This code shouldn't be executed!!
$
```

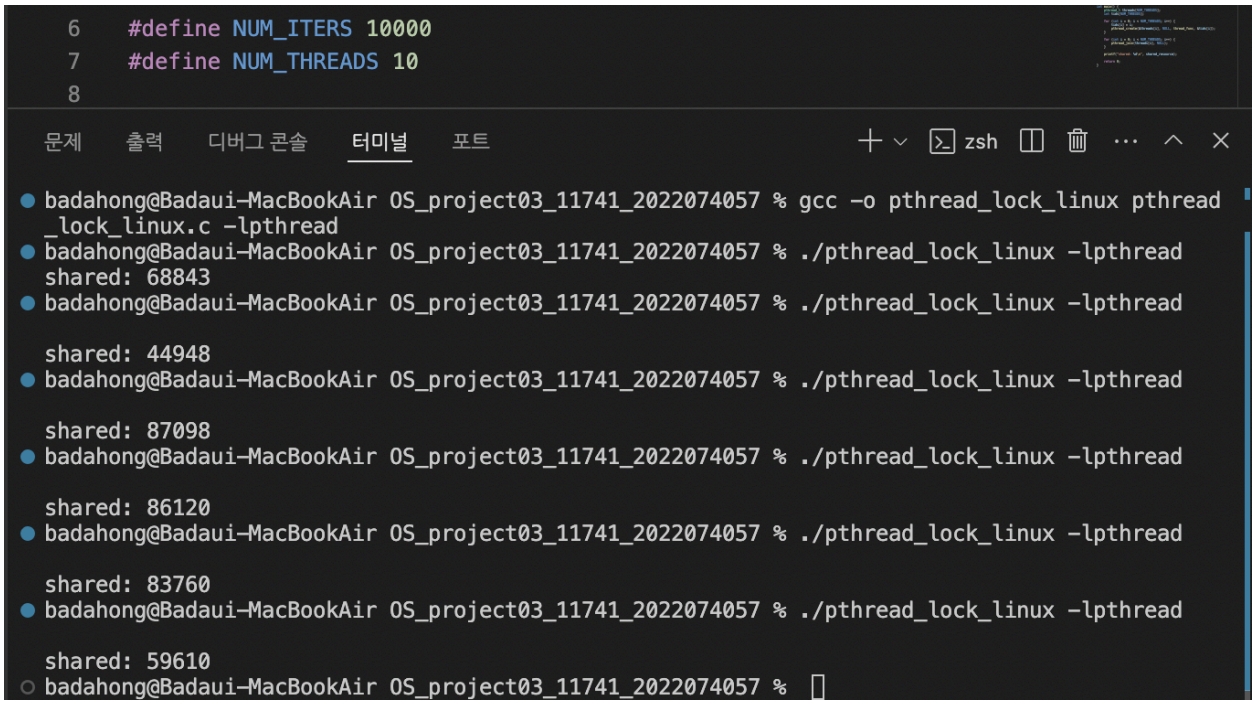
위에서 thread_exec와 같이 재부팅하는 현상이 똑같이 있었지만 전체적인 스레드에 대한 코드 수정 후 재부팅 문제는 해결되었다. 하지만 이것 역시 "This code shouldn't be executed!!"을 출력하는 문제가 있었다. 이는 기존의 exit 코드와 상호작용이 잘 안되고 있는 것 같아 exit을 수정하였다. 위 코드 구현 설명에서 본 것과 같이 wait()를 참고하여 종료하기 전 자식 스레드가 종료될 때까지 기다리는 코드 구현과 스레드인 경우 curproc->parent가 아닌 curproc->master를 통해 wakeup1을 하여 마스터 스레드 또한 종료될 수 있도록 하였다. 이렇게 하면 스레드에서 종료하는 경우에도 모든 스레드가 무사히 종료된 후 해당 스레드도 종료될 수 있다. 코드 수정 후 다시 실행시키니 문제가 해결되어 "Exiting…" 이후 바로 쉘로 빠져나갈 수 있었다.

2. Locking

여러 알고리즘 구현 실패**:**

사실 가장 먼저 Peterson's algorithm을 이용하여 구현해보았다. 그 결과 작은 숫자에서는 결과값이 잘 나왔지만 스레드의 숫자가 커질수록 race condition이 일어나 동기화 문제가 있어

아래처럼 결과가 다르게 나오는 걸 볼 수 있었다. 이를 해결하고자 또 다른 상태 변수를 이용하고자 했지만 매우 복잡하여 다른 소프트웨어적으로 해결할 알고리즘을 생각해보았다.



그리하여 다음으로는 교수님께서 수업시간에 잠깐 언급하신 Bakery Algorithm이 생각나 조사를 해보았다. 베이커리 알고리즘이란 빵집에서 번호표를 나누어주어 순서대로 입장가능한 것처럼, 스레드마다 번호표를 부여하고 번호표 순으로 critical section에 접근가능하도록 한다. Locking system의 근본적인 목적은 하나의 프로세스에게만 열쇠 획득이 가능하도록 하는 것이다. 이러한 취지에 베이커리 알고리즘이 적합해보였고, 해당 알고리즘은 N개의 스레드에 대해 상호배제성을 부여하기 때문에 해당 과제에서 사용해보았다. 그 결과 peterson's algorithm과는 다르게 결과가 일정하게 잘 나왔고, 이로써 mutual exclusion과 progress를 잘 보장한다는 것을 알았다. 하지만 NUM_THREADS와 NUM_ITERS의 수가 1000을 넘어가는 경우 busy waiting이 심해보였고, overhead가 매우 커 로딩시간이 15초 이상으로 매우 길었고, 심한 경우 프로그램이 종료되었다.

따라서 busy waiting과 overhead를 줄이기 위해서는 blcok/wakeup 방법을 쓰거나 하드웨어의 도움을 받아야했다. blcok/wakeup은 semaphore를 사용해야하는데 이는 구현할때 어차피 mutex가 필요하여 하드웨어의 도움이 필요해보였다. 그래서 소프트웨어적으로 해결하는 것 말고 하드웨어적으로 해결할 방법을 모색하였다. TAS, Swap, CAS 중 여러 스레드에 관한 race condition을 해결하기 위해서는 현재 상태를 반환해주고 반영하는 CAS가 가장 적합해 보여 CAS를 선택했다.

하지만 또 다른 어려움이 있었는데, 바로 inline assembly language를 사용하는 것이었다. 원자성을 보장하기 위해서는 소프트웨어상에서 함수를 구현하는 것만으로는 충족시킬 수 없었다. 원초적으로 원자성을 보장하기 위해 c 파일에 어셈블리어를 쓰는 방법을 택했고, 이에 대해서 공부를 하는데에도 시간이 꽤 필요했다.

또한, 처음에 VSCode를 이용하여 코드를 작성하는데 내가 작성하는 어셈블리어가 우분투의 x86에서 지원한다는 것을 깨닫기까지 꽤 오랜 시간이 걸려 VSCode의 ARM64 아키텍처에 맞게 코드를 수정하고자 낭비한 시간이 많았다. 결국 우분투에 코드를 옮겨 실행시키자 에러 없이 돌아가 어셈블리어를 이용한 코드를 완성할 수 있었다.

```
int compare_and_swap(volatile int *ptr, int expected, int new_value) {
        int old_value;
        __asm__ volatile(
                "lock cmpxchg %2, %1\n\t"
                : "=a"(old_value), "+m"(*ptr)
                : "q"(new_value), "0"(expected)
                : "memory"
        );
        return old_value;
}

void lock() {
        int expected = 0;
        while (compare_and_swap(&lock_flag, expected, 1) != expected) {
                expected = 0; // Reset expected value for next attempt
        }
}

void unlock() {
        lock_flag = 0;
}
```

또한, 위 코드처럼 원래는 CAS만을 이용하여 동기화 문제를 해결하고자 했다. 하지만 큰 수의 NUM_ITERS와 NUM_THREADS를 지정했을 때 실행시간이 4초 정도로 조금 걸려 busy waiting과 대기시간을 줄이고자 exponential backoff strategy를 도입했다. 이는 동기화 문제를 해결 후 단지 busy waiting과 실행시간을 줄이고자 사용한 전략이다. 그렇게 해서 위 구현 설명에서 본 최종 코드를 완성하였다. 하지만 생각보다 실행시간이 눈에 띄게 단축되진 않았다. 코드 상 구현 문제는 없었고, 논리대로라면 busy waiting이 줄어 CPU 자원을 쓰는 일도 줄어들고, overhead도 조금 줄어야 한다. 그리고 while문에서 usleep을 통해 딜레이를 주어 CPU를 쓰는 시간이 줄어들어 실행시간도 단축해야한다. 하지만 컴퓨터가 허용하는 최대 유저 프로세스량은 15314이었으므로 이보다 더 많은 스레드의 동기화 문제를 테스트해볼 순 없었다. 최대치(15314)로 두고 실행하였을 때 눈에 띄는 차이점은 없는 걸로 보아 실행시간에 영향을 줄만큼 큰 overhead가 일어나지 않았거나 이를 테스트하기에는 허용된 프로세스의 최대값이 부족한 듯 하였다. 더 높은 스레드 수로 시험해보지 못해 아쉬웠다.
하지만 출력결과물이 항상 NUM_ITERS * NUM_THREADS로 일정하여 높은 수의 스레드 사이에서도 동기화 문제가 해결되었다는 것은 확실하였고, 눈에 보이진 않지만 busy waiting을 줄여 CPU 자원을 덜 낭비하고 overhead도 조금 줄인 것이 분명하기에 CAS에 exponential backoff 전략을 사용한 코드를 최종본으로 제출하게 되었다.