3️⃣

Project3_LWP & Locking

1. Design

LWP (Light Weight Process)

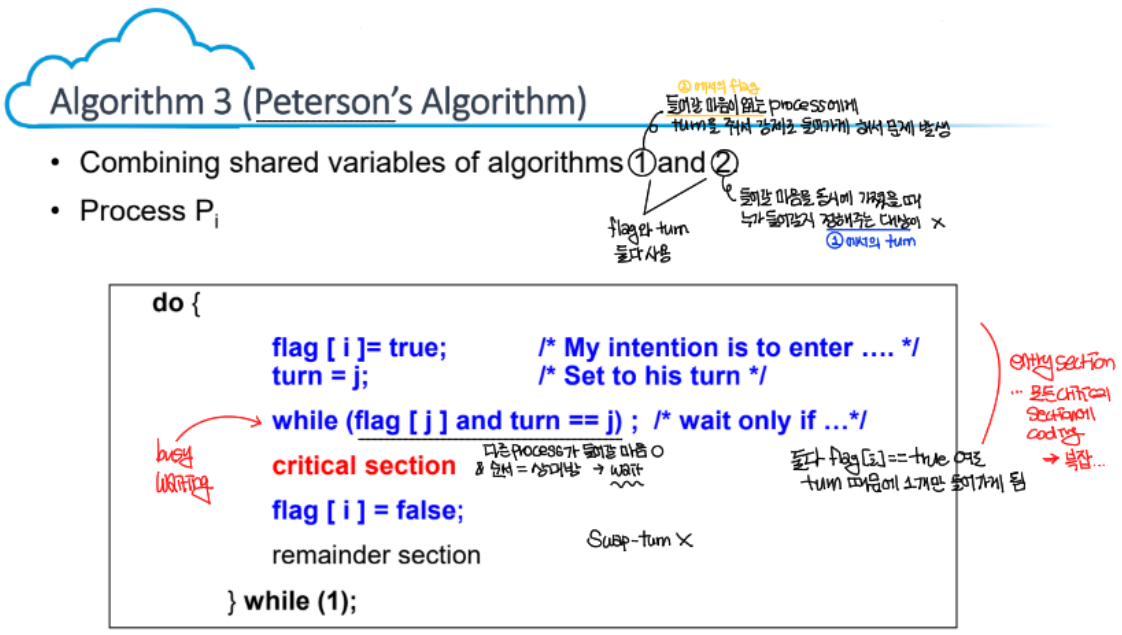
: 다른 LWP와 자원, 주소 공간 등을 공유하여 user level에서는 멀티태스킹을 가능하게 해주는 개념

명세에는 3개의 함수를 구현하도록 주어져 있습니다. 이 부분에서는 어떤 식으로 구현을 할지 설계한 내용을 다루었고, code 구현에 대한 내용은 **Implementation** 부분의 `proc.c` part에서 자세히 설명하였습니다.

LWP의 경우 thread를 구현하는 것이 목표이지만, ***"Light-Weight Process"***에서 알 수 있듯이 process를 구현하는 것과 비슷할 것이라고 보고 proc 구조체에 LWP를 구현하는데 필요한 변수를 추가하고 process처럼 구현을 하였습니다. Lab 5~7의 내용을 참고해본 결과 `fork`, `exec`, `wait`, `kill` 등 기존에 구현되어 있는 함수들과 system call을 참고해서 구현을 하면 될 것이라고 생각해서 기존에 구현되어 있는 내용을 참고해서 구현 명세에 맞게 내용을 수정하여 구현을 하였고 추가적으로 명세에는 없지만 필요한 내용을 새로운 함수를 통해 구현하였습니다.

Locking

Locking의 경우 **`Peterson's Algorithm`**을 참고하여 구현을 하였습니다.



Peterson Algorithm의 경우 2개 이상의 process 동기화를 해결할 수 있는 방법입니다. 처음에는 monitor로 구현을 하려고 하였으나, C언어에는 C++이나 Python 같은 객체지향 언어와 달리 class로 묶을 수 있는 방법이 없기 때문에 C언어로 구현을 할 방법을 생각해내지 못해서 이론 시간에 배웠던 Peterson's algorithm을 base로 구현을 시도하였습니다.
Peterson's algorithm의 설명을 보면 ***"Can be extended to support three or more processes, easily"*** 라는 문구가 있습니다. 이 문구를 통해 Peterson's algorithm을 base로 많은 process들을 다루면서 생길 수 있는 문제들을 대처해주면 될 것이라고 생각했습니다. 그 전에, Critical section problem을 해결하기 위한 3가지 조건에 대해 확인을 해보면 다음과 같습니다.

- **Mutual Exclusion**
- **Progress**
- **Bounded Waiting**

PPT에 나와있는 Algorithm 1, 2의 경우 **Mutual Exclusion**은 보장되지만, 들어갈 의지가 없는 process에게 turn을 주거나 여러 process의 flag가 true일 때 교통정리를 해주는 대상이 없어서 **progress** 문제가 생긴다는 점이 단점이였습니다.

Peterson algorithm은 이 두 algorithm의 문제점을 보완하기 위해 두 알고리즘에서 각각 사용하는 `turn`과 `flag` 변수를 동시에 사용하여 3가지 조건을 모두 만족시킬 수 있기 때문에 적합하다고 생각하였습니다.

저는 ppt에 나와있는 알고리즘도 물론 좋은 방법이지만, 이번 과제의 경우 **`NUM_THREAD`**, **`NUM_ITER`**에 의해 출력되는 shared variable의 값이 결정되므로 만약 이 두 값이 상당히 큰 값일 경우 **race condition**이 발생할 가능성이 매우 높아집니다. 따라서 Peterson's algorithm에 무언가를 더 추가해서 많은 process(thread)를 실행했을 때 **race condition**이 발생하는 것을 막아주도록 해야 한다고 생각했습니다. 먼저 `pthread_lock_linux.c` 파일에 구현되어 있던 `lock()`과 `unlock()` 함수에 인자로 thread_id를 받아와 `thread_func()`에서 인자로 받는 arg가 thread_id가 되도록 하였습니다.

```
/*pthread.h*/

/* Create a new thread, starting with execution of START-ROUTINE
   getting passed ARG.  Creation attributed come from ATTR.  The new
   handle is stored in *NEWTHREAD.  */
extern int pthread_create (pthread_t *__restrict __newthread,
               const pthread_attr_t *__restrict __attr,
               void *(*__start_routine) (void *),
               void *__restrict __arg) __THROWNL __nonnull ((1, 3));
```

```
/*pthread_lock_linux.c*/
...
int main(){
...
    for (int i = 0; i < n; i++) {
        tids[i] = i;
        pthread_create(&threads[i], NULL, thread_func, &tids[i]);
    }
...
}
```

그 이유는 `pthread.h` 파일의 `pthread_create()`의 주석을 보면 ***"Create a new thread, starting with execution of START_ROUTINE getting passed ARG"***라고 쓰여 있습니다. 함수를 보면 전달받은 **ARG**를 **START-ROUTINE**으로 설정하여 새로운 thread를 실행하는 것으로 해석할 수 있습니다. 실제로 main에서 쓰인 것을 보면 `threads[i]`와 `thread_func`, `tid[i]`를 인자로 받고 있는 것을 볼 수 있습니다. 이는 threads[i]에 대응하는 thread에 대해 tid[i]를 인자로 받는 `thread_func()`를 실행하는 것으로 볼 수 있습니다.
그런데,
`thread_func()`를 보면 `tid` 값에 인자로 받은 `arg`가 가리키는 주소에 존재하는 value를 int로 형변환을 한 뒤 대입하고 있는데 이 `tid`를 사용하고 있지 않고 있습니다.

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

    lock(); // -> lock(tid);

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

    unlock(); // -> unlock(tid);

    pthread_exit(NULL);
}
```

그래서 `lock()`과 `unlock()`에 인자로 `tid`를 받아와서 `pthread_create()`에서 `tid[i]`를 `thread_func()`에 넘겨주어 `tid`가 일치하는 thread만 critical section에 들어가 **`NUM_ITERS`** 만큼 반복문을 돌면서 shared_resource 값을 증가시킨 뒤 critical section을 빠져나오면서 lock을 해제할 수 있도록 하였습니다.

Lock / Unlock

기존의 Peterson algorithm의 경우 flag와 turn 변수를 이용하여 critical section에 들어갈 “의지”가 있는 process(thread)들만 turn에 맞게 진입을 허용해주었는데, 강의 자료에 설명되어 있는 내용은 i, j 2개의 process에 대해 비교를 하고 들어갈지 말지를 결정하기 때문에 process의 개수가 많아지면 비교 횟수가 많아지게 되고 이는 곧 시간복잡도의 증가로 이어지게 됩니다. 따라서 시간복잡도를 줄이기 위해 미리 모든 process에게 project2에서의 priority scheduling처럼 우선순위를 부여하여 labeling을 해놓은 뒤 flag 변수로 진입 여부를 결정해주면 될 것 같다고 생각을 하여 이 방향으로 구현하게 되었습니다.

그리고 peterson algorithm에서의 while문 조건(`while(flag[j] and turn==j)`)처럼 while문 조건을 구현하였는데 2개의 while문으로 구분해 주었습니다. 첫 번째 while문은 flag 변수를 검사하는 조건으로, true인 경우만 다음 조건으로 갈 수 있도록 해주었고 두 번째 while문은 turn 처럼 순서가 된 process를 critical section으로 들어가게 해주는 역할을 합니다. 그렇게 process가 critical section에 들어가고 `unlock()`이 호출되게 되면 자신의 순서를 0으로 설정해서 다음 process가 똑같은 메커니즘으로 critical section에 진입할 수 있도록 해주도록 하였습니다.

2. Implementation

LWP (Light Weight Process)

✅ types.h

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

`types.h` 파일에 `int` type `thread_t`를 선언하였습니다.

✅ proc.h

```
// 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)

  struct proc *main_thread;                // Main thread
  thread_t tid;                            // Thread ID
  void *retval;                            // Return value of the thread
};
```

LWP를 구현하기 위해 기존의 proc 구조체에서 3개의 구조체 변수를 추가해주었습니다.

✅ proc.c

```
int thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg)
```

> - 새 thread 생성 및 시작
> - parameter
>   1. thread : 해당 주소에 thread의 id를 저장
>   2. start_routine : thread가 시작할 함수를 지정. 즉, 새로운 thread가 만들어지면 그 thread는 start_routine이 가리키는 함수에서 시작
>   3. arg : thread의 start_routine에 전달할 인자
>   4. return : thread가 성공적으로 만들어졌으면 0, error가 있다면 0이 아닌 값을 return

```
int
thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg)
{
  struct proc *p, *np, *main_thread, *curproc = myproc();
  uint sz, sp, ustack[3+MAXARG+1];
  pde_t* pgdir;

  // Allocate thread
  if((np = allocproc()) == 0){
    return -1;
  }
  nextpid--;

  acquire(&ptable.lock);

  if(curproc->main_thread == 0){
    main_thread = curproc;
  }else{
    main_thread = curproc->main_thread;
  }

  // Allocate stack
  sz = PGROUNDUP(main_thread->sz);
  main_thread->sz += 2*PGSIZE;
  pgdir = main_thread->pgdir;


  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0){ // stack 할당
    release(&ptable.lock);
    return -1;
  }

  clearpteu(pgdir, (char*)(sz - 2*PGSIZE)); // 새로운 stack을 위한 page table 초기화
  sp = sz;
  main_thread->sz = sz;

  np->pgdir = main_thread->pgdir; // Page table 공유

  // Start_routine에서 사용할 tid 설정
  np->tid = nexttid;
  *thread = np->tid;
  nexttid++;
```

```
  // process state 복사
  np->sz = main_thread->sz;
  np->parent = main_thread->parent;
  np->main_thread = main_thread;
  np->pid = main_thread->pid;
  *np->tf = *main_thread->tf;

  // main_thread의 pid와 같은 pid를 갖는 process를 찾아서 sz와 page table을 설정
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == main_thread->pid){
      p->sz = sz;
      switchuvm(p);
    }
  }

  ustack[0] = 0xffffffff;  // fake return PC
  ustack[1] = (uint)arg;   // Start_routine의 인자

  sp -= 8;
  // ustack에 start_routine의 주소와 인자를 저장
  if(copyout(pgdir, sp, ustack, 8) < 0){
    release(&ptable.lock);
    return -1;
  }

  // Context 설정
  np->tf->eip = (uint)start_routine;
  np->tf->esp = sp;
  np->tf->ebp = sp;

  // file descriptor 복사
  for(int i = 0; i < NOFILE; i++)
    if(main_thread->ofile[i])
      np->ofile[i] = filedup(main_thread->ofile[i]);
  np->cwd = idup(main_thread->cwd); // current working directory 복사

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

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

    return 0;
  }
```

`thread_create()`는 process 생성과 관련된 함수인 `fork()`와 `exec()`를 참고해서 구현을 하였습니다.

```c
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }
  acquire(&ptable.lock);

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd); // cwd : current working directory

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

  pid = np->pid;

  np->state = RUNNABLE;
```

exec.c > exec(char *, char **)

```c
exec(char *path, char **argv)

  // Allocate two pages at the next page boundary.
  // Make the first inaccessible.  Use the second as the user stack.
  sz = PGROUNDUP(sz);
  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
  sp = sz;

  // Push argument strings, prepare rest of stack in ustack.
  for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
      goto bad;
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
      goto bad;
    ustack[3+argc] = sp;
  }
  ustack[3+argc] = 0;

  ustack[0] = 0xffffffff;  // fake return PC
  ustack[1] = argc;
  ustack[2] = sp - (argc+1)*4;  // argv pointer

  sp -= (3+argc+1) * 4;
  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
    goto bad;

  // Save program name for debugging.
  for(last=s=path; *s; s++)
    if(*s == '/')
      last = s+1;
  safestrcpy(curproc->name, last, sizeof(curproc->name));

  // Commit to the user image.
  oldpgdir = curproc->pgdir;
  curproc->pgdir = pgdir;
  curproc->sz = sz;
  curproc->tf->eip = elf.entry;  // main
  curproc->tf->esp = sp;
  switchuvm(curproc);
```

가장 먼저 proc 구조체를 참조하는 4개의 변수 (p, np, main_thread, curproc)를 생성했습니다. 먼저 np는 자식 process와 같은 개념으로 thread_create()에 의해 생성되는 thread입니다. main_thread와 공유하는 data는 공유하고 공유하지 않는 data의 경우 thread_create()의 인자인 start_routine과 arg를 받아와서 np가 scheduling 되었을 때 arg를 받은 start_routine을 실행하도록 하는 역할 입니다. main_thread의 경우 main_thread가 항상 존재할 수 있도록 현재 process에 main thread가 존재하면 그 thread를 main_thread로 하고 main_thread가 존재하지 않는다면 현재 실행중인 process를 main_thread로 설정하여 주었습니다. 그 다음 stack을 할당하고 np와 main_thread끼리 page table을 공유하도록 만들어서 thread끼리 자원을 공유하도록 하였습니다. 인자로 받은 thread 변수에 start routine에서 사용할 tid 값을 할당해주는데 여기서 nexttid의 경우 기존에 전역변수로 존재하는 nextpid 처럼 전역변수로 선언하였고 초기값은 1로 설정되어 있습니다. 다음 thread를 위해 nexttid를 np의 tid에 할당해준 뒤 1만큼 늘려주고 np에 main_thread의 정보를 복사해줍니다. user stack에 **fake return address**와 start_routine의 인자로 주어질 **arg**를 저장하고 stack pointer를 8만큼 내려서 user stack에 start routine의 주소와 인자를 저장한 뒤 trap frame의 **eip**에 start_routine을 저장해서 다음에 실행할 함수를 load한 뒤 np가 scheduling 되었을 때 start_routine과 arg를 불러와서 함수를 실행할 수 있도록 하였습니다.

void thread_exit(void *retval)

> - Thread를 종료한 뒤 값을 반환하는데 모든 thread는 반드시 이 함수를 통해 종료하고, 시작 함수의 끝에 도달하여 종료하는 경우는 고려하지 않는다.
> - retval: 스레드를 종료한 후 join 함수에서 받아갈 값

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

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

  acquire(&ptable.lock);
```

```
  if(curproc->main_thread != 0){
    wakeup1(curproc->main_thread);
  }

  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_exit()`의 경우 `exit()`을 참고해서 구현하였습니다.

```
237  // Exit the current process.  Does not return.
238  // An exited process remains in the zombie state
239  // until its parent calls wait() to find out it exited.
240  void
241  exit(void)
242  {
243    struct proc *curproc = myproc();
244    struct proc *p;
245    int fd;
246
247    if(curproc == initproc)
248      panic("init exiting");
249
250    acquire(&ptable.lock);
251    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
252      if(p->pid == curproc->pid && p != curproc){
253        kfree(p->kstack);
254        p->kstack = 0;
255        p->pid = 0;
256        p->tid = 0;
257        p->parent = 0;
258        p->name[0] = 0;
259        p->killed = 0;
260        p->state = UNUSED;
261      }
262    }
263    release(&ptable.lock);
```

```
265    // Close all open files.
266    for(fd = 0; fd < NOFILE; fd++){
267      if(curproc->ofile[fd]){
268        fileclose(curproc->ofile[fd]);
269        curproc->ofile[fd] = 0;
270      }
271    }
272
273    begin_op();
274    iput(curproc->cwd);
275    end_op();
276    curproc->cwd = 0;
277
278    acquire(&ptable.lock);
279    // Parent might be sleeping in wait().
280    wakeup1(curproc->parent);
281
282    // Pass abandoned children to init.
283    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
284      if(p->parent == curproc){
285        p->parent = initproc;
286        if(p->state == ZOMBIE)
287          wakeup1(initproc);
288      }
289    }
290
291    // Jump into the scheduler, never to return.
292    curproc->state = ZOMBIE;
293    sched();
294    panic("zombie exit");
295  }
```

`exit()`과 다른 점은 자원을 회수하는 부분을 구현하지 않았다는 점입니다. 그 이유는 `thread_exit()`의 경우 thread가 종료된 뒤 `thread_join()`에서 `retval`을 받아가는데 `thread_join()`에서 thread에 할당된 자원들을 회수하고 정리하기 때문에 `thread_exit()`에서도 자원을 회수하고 정리하게 되면 이중으로 정리하는 셈이 되기 때문에 `thread_exit()`에서는 정리하지 않고 현재 process의 `retval`에 인자로 받은 `retval`을 할당해주었습니다.

```
int thread_join(thread_t thread, void **retval)
```

> - 인자로 받은 thread가 종료되기를 기다리고, thread가 thread_exit()을 통해 반환한 값(retval)을 받아오는데 만약 thread가 이미 종료되었다면 즉시 반환.
> - thread가 종료된 후, thread에 할당된 자원들 (Page table, Memory, Stack 등)을 회수하고 정리
> - thread: join할 스레드의 id

> - retval: 스레드가 반환한 값을 저장
> - return: 정상적으로 join했다면 0을, 그렇지 않다면 0이 아닌 값을 반환

```
int
thread_join(thread_t thread, void **retval)
{
  struct proc *p, *curproc = myproc();
  int havekids;

  acquire(&ptable.lock);
  for(;;){
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->tid != thread)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        *retval = p->retval; // return value
        clear_threads(p);

        release(&ptable.lock);
        return 0;
      }
    }

    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }
    sleep(curproc, &ptable.lock);
  }
}
```

`thread_join()`의 경우 `wait()`을 참고해서 구현하였습니다.

```
299  int
300  wait(void)
301  {
302    struct proc *p;
303    int havekids, pid;
304    struct proc *curproc = myproc();
305
306    acquire(&ptable.lock);
307    for(;;){
308      // Scan through table looking for exited children.
309      havekids = 0;
310      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
311        if(p->parent != curproc)
312          continue;
313        havekids = 1;
314        if(p->state == ZOMBIE){
315          // Found one.
316          pid = p->pid;
317          kfree(p->kstack);
318          p->kstack = 0;
319          freevm(p->pgdir);
320          p->pid = 0;
321          p->tid = 0;
322          p->parent = 0;
323          p->name[0] = 0;
324          p->killed = 0;
325          p->state = UNUSED;
326          release(&ptable.lock);
327          return pid;
328        }
329      }
330
331      // No point waiting if we don't have any children.
332      if(!havekids || curproc->killed){
333        release(&ptable.lock);
334        return -1;
335      }
336
337      // Wait for children to exit.  (See wakeup1 call in proc_exit.)
338      sleep(curproc, &ptable.lock);  //DOC: wait-sleep
339    }
340  }
```

그 이유는 명세에도 주어져 있듯이 인자로 받은 thread가 종료되기를 ***"기다렸다가"*** thread가 종료되면 자원을 회수하고 정리하는 역할이기 때문에 `wait()`을 참고해서 구현하였습니다. `wait()`과 다른 점은 `wait()`의 경우 인자를 받지 않지만, `thread_join()`의 경우 `thread`와 `retval`을 인자로 받아 ptable에 있는 process들 중 tid 값이 일치하는 process를 찾지 못한 경우 반복문을 빠져나오게 했고, retval에 proc 구조체에 선언했던 retval을 할당해주었습니다. 또한 process와 달리 thread는 page table을 공유하기 때문에 `freevm(p->pgdir);` 도 제외하여 page table을 해제하지 않도록 하였습니다. 이 부분을 제외하면 `wait()`함수와 동일합니다.

`void clear_threads(struct proc* p)`

```
void
clear_threads(struct proc* p)
{
  kfree(p->kstack);
  p->kstack = 0;
  p->pid = 0;
  p->tid = 0;
  p->parent = 0;
  p->main_thread = 0;
  p->name[0] = 0;
  p->killed = 0;
  p->state = UNUSED;
}
```

자원을 회수하고 정리하는 기능을 하는 함수가 `kill_threads()`와 `thread_join()` 2개나 있어서 자원을 회수하는 부분을 함수로 만들었습니다. 이 두 함수 이외에도 `wait()`과 `exit()`에서도 자원을 회수하는 부분이 있는데 이 두 함수의 경우 process를 다루는 system call이고 `wait()`의 경우 page table을 해제하는 함수 (`freevm()`)가 존재하는데, thread는 page table을 공유하기 때문에 해제하면 안되므로 이 부분은 `clear_threads()`를 사용하지 않았습니다.

`int kill_threads(int pid)`

```
int
kill_threads(int pid)
{
  struct proc *p;
  int found = 0;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid && p->tid > 0){
        found = 1;
      clear_threads(p);
    }
  }
  release(&ptable.lock);
  return found ? 0 : -1;  // 스레드를 찾았으면 0, 못 찾았으면 -1 반환
}
```

이 함수의 경우 `exec.c`의 `exec()`에서 호출이 되어 해당 thread를 제외한 나머지 thread들을 전부 kill하는 역할을 합니다.

```
int
kill(int pid)
{
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING)
        p->state = RUNNABLE;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

기존의 `kill()`과 비교를 해보면 `kill()`에서는 pid에 해당하는 process를 찾아서 killed=1로 설정해서 kill 했다는 것을 명시적으로 선언해주고 **SLEEPING** 상태라면 **RUNNABLE** 상태로 바꿔주는 역할을 합니다. 그에 반해 `kill_threads()`는 상태와 상관없이 할당되어 있던 자원을 회수하는 역할을 합니다.

✅ exec.c

```
  if(curproc->tid > 0){
    // Set the current thread to main thread
    curproc->main_thread->tid = curproc->tid;
    curproc->tid = 0;
    curproc->parent = curproc->main_thread->parent;
    curproc->main_thread = 0;
  }
```

```
  kill_threads(curproc->pid); // 나머지 thread를 종료
```

`exec.c`에 위의 code를 추가해서 현재 process의 tid 값이 0보다 큰 경우 이 thread를 main thread로 설정해주고 초기화를 해주었습니다. 그 뒤 `kill_threads()`를 호출해서 해당 thread를 제외한 나머지 thread들을 전부 kill 해주었습니다. 이렇게 하지 않으면 ***"unexpected trap 14 from cpu 0 eip 1010101"*** 오류가 뜨는데, 이는 초기화를 하지 않았을 경우 main_thread와 parent 포인터가 잘못된 주소를 참조하게 되었을 때 발생한 문제라고 판단했습니다.

System call

✅ syscall.h

```
#define SYS_thread_create 22
#define SYS_thread_exit 23
#define SYS_thread_join 24
#define SYS_kill_threads 25
```

✅ syscall.c

```
extern int sys_thread_create(void);
extern int sys_thread_exit(void);
extern int sys_thread_join(void);
extern int sys_kill_threads(void);
...
[SYS_thread_create] sys_thread_create,
[SYS_thread_exit] sys_thread_exit,
[SYS_thread_join] sys_thread_join,
[SYS_kill_threads] sys_kill_threads,
```

✅ user.h

```
// system calls
...
int thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg);
void thread_exit(void *retval);
int thread_join(thread_t thread, void **retval);
int kill_threads(int);
...
```

✅ defs.h

```
//proc.c
...
int             thread_create(thread_t*, void*(*)(void*), void*);
void            thread_exit(void*);
int             thread_join(thread_t, void**);
```

```
int             kill_threads(int);
...
```

✅ sysproc.c

```
int
sys_kill_threads(void)
{
  int pid;

  if(argint(0, &pid)<0)
    return -1;
  return kill_threads(pid);
}
...
int
sys_thread_create(void)
{
  thread_t *thread;
  void *(*start_routine)(void *), *arg;

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

int
sys_thread_exit(void)
{
  void *retval;

  if(argptr(0, (char**)&retval, sizeof(*retval)) < 0)
    return -1;
  thread_exit(retval);
  return 0;
}


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

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

✅ usys.S

```
...
SYSCALL(thread_create)
SYSCALL(thread_exit)
SYSCALL(thread_join)
SYSCALL(kill_threads)
...
```

✅ MakeFile

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _wc\
    _zombie\
    _hello_thread\
    _pthread_lock_linux\
    _thread_exec\
    _thread_exit\
    _thread_kill\
    _thread_test\
```

```
EXTRA=\
    mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
    ln.c ls.c mkdir.c rm.c stressfs.c wc.c zombie.c\
    printf.c umalloc.c hello_thread.c pthread_lock_linux.c thread_test.c thread_exec.c thread_exit.c thread_kill.c
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

Locking

✅ pthread_lock_linux

- **Without lock()/unlock()**

```
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 10000
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 9964
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 9962
```

NUM_ITER과 **NUM_THREADS**를 각각 10으로 설정한 뒤, `lock()`과 `unlock()`에 주석처리를 하고 이 파일들을 실행시키게 되면 거의 대부분 100이 출력되지만, 평균적으로 10번에 1번꼴로 100이 아닌 94, 96 같은 값들이 출력되는 것을 확인할 수 있었습니다. **NUM_ITER**과 **NUM_THREADS** 값을 훨씬 키워서 두 값 모두 100으로 설정하게 되면 10번에 4번 꼴로 위의 이미지처럼 10000이 아닌 값이 출력되는 경우가 존재합니다. 그 이유는 동기화가 보장되지 않아 여러 thread가 동시에 공유 변수인 `shared_resource`에 접근했기 때문이고 이러한 현상을 더 많은 thread가 존재할 수록 확률이 높아진다는 것을 확인할 수 있습니다. 따라서 여러 thread들이 동시에 공유 변수에 접근할 수 없도록 해야 합니다.

- **struct**

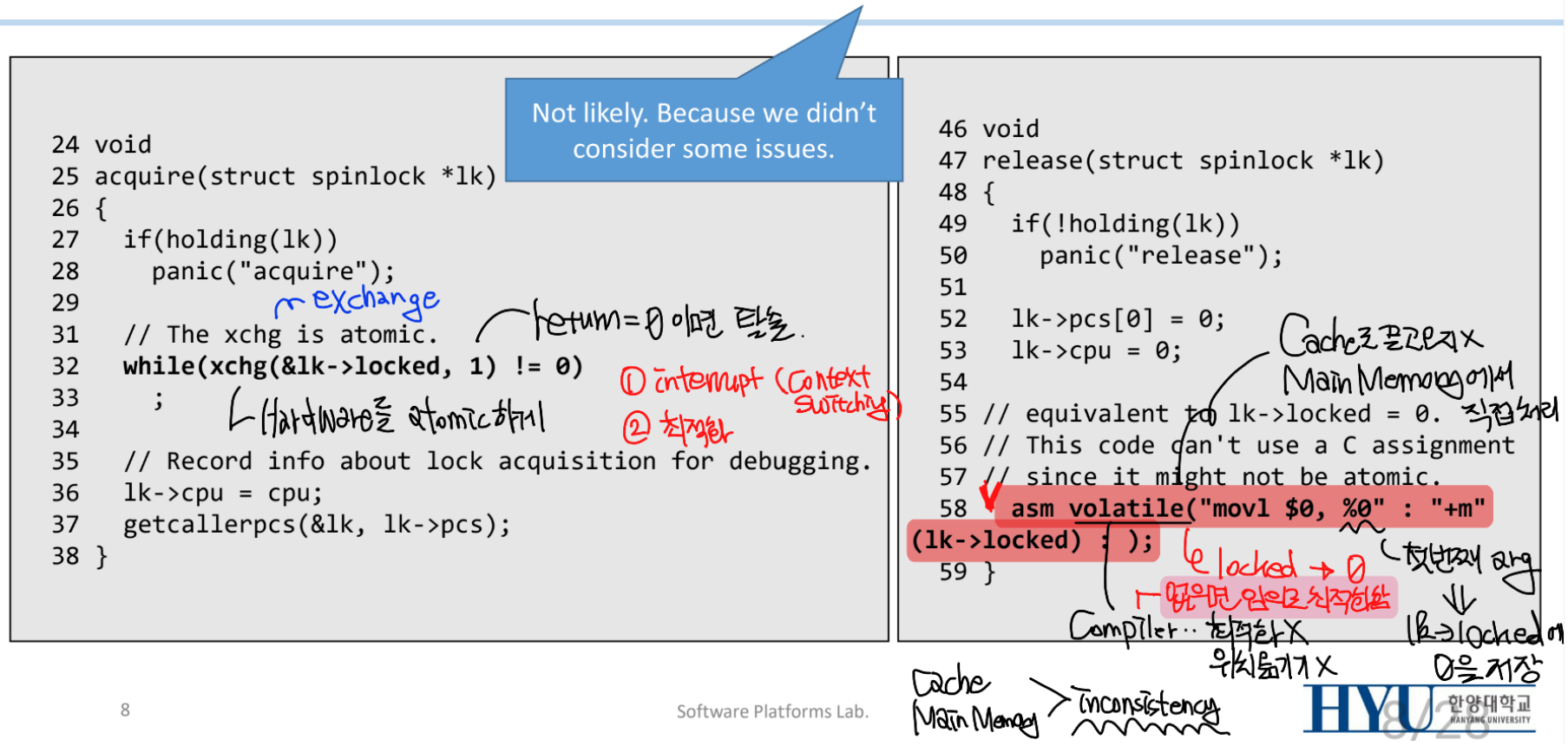
```
typedef struct {
    volatile bool enter[NUM_THREADS];
    volatile int number[NUM_THREADS];
```

```
} lock_t;

lock_t mutex;
```

가장 먼저 구조체를 선언해주었습니다. Peterson algorithm을 기반으로 구현하였기 때문에 flag와 turn 같은 변수가 필요한데 저는 조금 더 직관적으로 알아볼 수 있도록 flag 대신 `enter`와 turn 대신 `number`로 변수명을 정하였습니다. 여기서 **volatile**을 선언한 이유는 밑에서 설명할 것이고, 두 변수 모두 **NUM_THREAD** 크기의 배열로 설정해서 macro 변수인 thread의 개수만큼 배열에 저장해서 한두개의 thread가 아니라 여러개의 thread에 대해서도 작업을 할 수 있도록 해주었습니다.
구조체의 이름은
lock_t(lock_thread)로 하였고 전역변수로 **mutex**라고 정의해서 mutex 변수를 통해 구조체에 접근할 수 있도록 하였습니다.

- **Volatile**



4/19일 실습시간에 배운 locking 중에 **volatile**에 대한 내용이 있었습니다. 그 중에서 **volatile**의 경우 compiler가 임의로 최적화하는 것을 방지해 cache와 main memory 간의 inconsistency를 방지하는 역할을 한다고 배웠습니다.

```
typedef struct {
    volatile bool enter[NUM_THREADS];
    volatile int number[NUM_THREADS];
} lock_t;

lock_t mutex;
```

구조체 변수 `enter`와 `number`에 **volatile**이라는 keyword를 붙인 이유는 위에서 설명한 것 처럼 sychronization과 consistency를 보장하기 위해서 입니다.

만약 **volatile**을 선언하지 않게 되면 `lock()`과 `unlock()`을 주석처리하고 컴파일 했을 때 출력되는 결과처럼 중간에 100이 아닌 값이 출력됩니다. Peterson algorithm의 경우 code의 순서가 결과에 큰 영향을 미치기 때문에 LOAD, STORE 하는 과정 역시 동일한 순서대로 진행되어야 하고 이것을 보장하기 위해 **volatile**이 필요하다고 생각했습니다.

```
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 100
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 94
```

그래서 구조체의 변수는 전부 **volatile** 선언을 해주었고, boolean type을 쓰기 위해 `#include <stdbool.h>` 를 추가하였습니다. boolean type을 써야겠다고 생각한 이유는 number가 "번호"를 결정한다면 enter는 critical section에 들어갈지 말지를 결정하기 때문에 **true/false**로 결정해 주는 것이 조금 더 명확하게 구분을 할 수 있을 것 같아서 **int** type이 아닌 **boolean** type을 썼습니다.

- **init_lock()**

```
void init_lock(lock_t *m){
    for(int i=0; i<NUM_THREADS; i++){
        m->enter[i] = false;
        m->number[i] = 0;
    }
}
```

그 다음 `init_lock()`을 만들어서 모든 thread의 enter, number 정보를 false와 0으로 초기화해주었습니다.

- **lock()/unlock()**

```
void lock(int tid)
{
    mutex.enter[tid] = true; // 자신의 차례를 알림

    int max = 0;
    for(int i=0; i<NUM_THREADS; i++){
        if(max < mutex.number[i]){
            max = mutex.number[i];
        }
    }

    mutex.number[tid] = max + 1; // 현재 thread의 번호
    mutex.enter[tid] = false; // 아직 자신의 차례가 아님

    for(int i=0; i<NUM_THREADS; i++){
        while(mutex.enter[i]);
        while(mutex.number[i] != 0 &&
            (mutex.number[i] < mutex.number[tid] ||
                 (mutex.number[i] == mutex.number[tid] && i < tid)));
    }
}
```

`lock()`은 critical section에 들어가려고 하는 thread들의 순서를 정해서 한 번에 하나의 thread만 critical section을 들어가도록 lock을 걸어주는 역할을 하는 함수입니다.

```
void unlock(int tid)
{
    mutex.number[tid] = 0;
}
```

`unlock()`은 호출되었을 때 critical section을 빠져나온 thread의 number를 초기 상태인 0으로 초기화해줍니다.

- Ex)

  위의 내용을 간단한 예시로 설명해보겠습니다.

```
/*main()*/
...
    for (int i = 0; i < NUM_THREADS; i++) {
        tids[i] = i;
        pthread_create(&threads[i], NULL, thread_func, &tids[i]);
```

```
    }
...
```

그 전에 `pthread_lock_linux.c`의 구조를 먼저 확인해보면, `main()`에서 `pthread_create()`가 `threads[i]`에 있는 thread들을 불러와서 `tid[i]`를 인자로 받는 `thread_func()`를 통과시킵니다.

```
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);
}
```

그 뒤 `thread_func()`에서 arg(=tid)를 인자로 받아서 shared_resource 값을 증가시키는데 이 때 lock/unlock으로 동기화를 보장하는 과정으로 이뤄집니다. main에서 **NUM_THREADS** 만큼 반복문을 돌면서 `tid[i]`를 arg로 thread_func에 넘겨서 arg를 `lock()`에 넘기는 순서로 진행됩니다.

간단히
NUM_THREAD=5라고 가정할 때, thread는 0~4까지이고 0번부터 `lock()`에 들어가 번호를 부여 받습니다. 그 전에 `init_lock()`에 의해 0번부터 4번까지의 thread는 `enter=false`, `number=0`으로 초기화 된 상태로 `lock()`에 진입하게 됩니다. 먼저 0번 thread가 진입을 해서 `mutex.enter[tid]=true` 에 의해 true로 변경되고 조건문 `max<mutex.number[i]`는 현재 number가 0인 상태이므로 조건문을 통과하지 못해서 number는 그대로 0이 됩니다. 그 뒤, `mutex.number[tid] = max+1`에 의해 0번 thread는 1번 번호를 부여받게 됩니다. 그 뒤, enter를 다시 false로 바꿔서 while loop에서 **NUM_THREAD** 만큼의 thread가 전부 번호를 부여받을 때까지 대기하게 됩니다.

여기까지가 첫 번째 thread에 대한 예시이고, 그 이후 thread들은 다음과 같이 진행되게 됩니다. tid=1인 thread가 `lock()`에 들어와서 자신의 enter를 true로 바꾸고 반복문으로 들어가는데 현재 number를 부여받은 0번의 number가 1이기 때문에 반복문에서 1번 thread는 max값 1을 받고 밖으로 나오게 됩니다. 그 뒤 `mutex.number[tid] = max+1` 에 의해 1번 thread는 2번 번호를 부여받고 마찬가지로 `enter=false`로 바껴서 0번 thread와 함께 첫 번째 while loop에서 대기하게 됩니다. 이 과정을 **NUM_THREAD** 만큼 반복을 해서 tid=0부터 4까지 각각 1번~5번의 번호를 부여받았다면 번호 순서대로 critical section에 들어가서 작업을 수행하고 빠져 나오면서 `unlock()`에 의해 자신의 번호를 0으로 초기화하게 됩니다.

이를 통해 더 작은 숫자를 가진 thread가 먼저 critical section에 진입하는 것을 보장할 수 있고 **Mutual Exclusion**, **Progress**, **Bounded waiting** 3가지 조건을 만족할 수 있습니다. 그러나, 여전히 while loop에서 busy waiting이 존재한다는 단점이 있습니다.

3. Result

LWP (Light Weight Process)

- Booting

```
./makexv6.sh
./bootxv6.sh
```

- Executing
  - thread_test

```
$ 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
```

```
Test 2: Fork test
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Child of thread 0 start
Child of thread 1 start
Child of thread 2 start
Child of thread 4 start
Child of thread 3 start
Child of thread 1 end
Child of thread 2 end
Thread 1 end
Child of thread 0 end
Child of thread 3 end
Child of thread 4 end
Thread 0 end
Thread 2 end
Thread 3 end
Thread 4 end
Test 2 passed
```

```
Test 3: Sbrk test
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Test 3 passed

All tests passed!
```

- `thread_exec`

```
$ thread_exec
Thread exec test start
Thread 0 start
Thread 1 start
TThread 3 start
Thread 4 start
hread 2 start
Executing...
Hello, thread!
```

- thread_exit

```
$ thread_exit
Thread exit test start
Thread 0 start
Thread 1 start
ThThread 3 start
Thread 4 start
read 2 start
Exiting...
```

- thread_kill

```
$ thread_kill
Thread kill test start
Killing process 19
This code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 times.
TThis code should be executed 5 times.
his code should be executed 5 times.
Kill test finished
```

Locking

- Compile

```
gcc -o pthread_lock_linux pthread_lock_linux.c -lpthread
./pthread_lock_linux
```

- Result

```
NUM_THREADS = 10, NUM_ITER = 10
```

```
root@43976b6db020:/OS/xv6-public# gcc -o pthread_lock_linux pthread_lock_linux.c -lpthread
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 100
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 100
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 100
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 100
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 100
```

`NUM_THREADS = 200, NUM_ITER = 200`

```
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 40000
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 40000
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 40000
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 40000
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 40000
```

`NUM_THREADS = 500, NUM_ITER = 500`

```
root@43976b6db020:/OS/xv6-public# gcc -o pthread_lock_linux pthread_lock_linux.c -lpthread
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 250000
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 250000
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 250000
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 250000
root@43976b6db020:/OS/xv6-public# ./pthread_lock_linux
shared: 250000
```

4. Trouble Shooting

LWP (Light Weight Process)

```c
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
enum threadstate { T_UNUSED, T_EMBRYO, T_SLEEPING, T_RUNNABLE, T_RUNNING, T_ZOMBIE };

struct thread {
  uint sz;                     // Size of thread memory (bytes)
  pde_t* pgdir;                // Page table - 다른 LWP와 공유

  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run thread
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  enum threadstate state;      // Thread state

  thread_t tid;                // Thread ID
  void *retval;                // Return value of the thread
};
```

처음에는 thread와 process는 비슷한 측면이 많지만 thread 만의 구조체가 필요할 것이라고 생각했고 thread 구조체를 만들어서 proc 구조체에서 thread들의 list를 만들어서 thread들을 관리하는 형태로 구현하였습니다. 그러나 구현을 해보면서 느꼈지만, 이 부분에서 더 진전이 없다는 것을 느꼈고 thread도 proc 구조체를 이용하여 구현하는 것이 더 나을 것 같다고 생각해서 process 처럼 구현을 하게 되었습니다.

```
init: starting sh
$ thread_exec
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x1240 addr 0x4004--kill proc
$ thread_test
pid 4 sh: trap 14 err 6 on cpu 0 eip 0x1240 addr 0x4004--kill proc
$ thread_kill
pid 5 sh: trap 14 err 6 on cpu 0 eip 0x1240 addr 0x4004--kill proc
$ thread_exit
pid 6 sh: trap 14 err 6 on cpu 0 eip 0x1240 addr 0x4004--kill proc
```

어느 정도 구현을 끝마친 뒤, xv6를 booting 했는데 trap에 걸려서 process가 kill되었다는 오류가 떴습니다.

Part One: Eliminate allocation from sbrk()

Your first task is to delete page allocation from the sbrk(n) system call implementation, which is the function sys_sbrk() in sysproc.c. The sbrk(n) system call grows the process's memory size by n bytes, and then returns the start of the newly allocated region (i.e., the old size). Your new sbrk(n) should just increment the process's size (proc->sz) by n and return the old size. It should not allocate memory -- so you should delete the call to growproc() (but you still need to increase the process's size!).

Try to guess what the result of this modification will be: what will break?

Make this modification, boot xv6, and type `echo hi` to the shell. You should see something like this:

```
init: starting sh
$ echo hi
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x12f1 addr 0x4004--kill proc
$
```

The "pid 3 sh: trap..." message is from the kernel trap handler in trap.c; it has caught a page fault (trap 14, or T_PGFLT), which the xv6 kernel does not know how to handle. Make sure you understand why this page fault occurs. The "addr 0x4004" indicates that the virtual address that caused the page fault is 0x4004.

Part Two: Lazy allocation

Modify the code in trap.c to respond to a page fault from user space by mapping a newly-allocated page of physical memory at the faulting address, and then returning back to user space to let the process continue executing. You should add your code just before the `cprintf` call that produced the "pid 3 sh: trap 14" message. Your code is not required to cover all corner cases and error situations; it just needs to be good enough to let sh run simple commands like echo and ls.

Hint: look at the cprintf arguments to see how to find the virtual address that caused the page fault.

Hint: steal code from allocuvm() in vm.c, which is what sbrk() calls (via growproc()).

Hint: use PGROUNDDOWN(va) to round the faulting virtual address down to a page boundary.

Hint: `break` or `return` in order to avoid the `cprintf` and the `proc->killed = 1`.

Hint: you'll need to call mappages(). In order to do this you'll need to delete the `static` in the declaration of mappages() in vm.c, and you'll need to declare mappages() in trap.c. Add this declaration to trap.c before any call to `mappages()`:

```
int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
```

Hint: you can check whether a fault is a page fault by checking if `tf->trapno` is equal to `T_PGFLT` in trap().

If all goes well, your lazy allocation code should result in `echo hi` working. You should get at least one page fault (and thus lazy allocation) in the shell, and perhaps two.

By the way, this is not a fully correct implementation. See the challenges below for a list of problems we're aware of.

Optional challenges: Handle negative sbrk() arguments. Handle error cases such as sbrk() arguments that are too large. Verify that fork() and exit() work even if some sbrk()'d address have no memory allocated for them. Correctly handle faults on the invalid page below the stack. Make sure that kernel use of not-yet-allocated user addresses works -- for example, if a program passes an sbrk()-allocated address to read().

이 오류를 Google에 검색했더니 위와 같이 대처 방법에 대한 hint를 얻을 수 있었고 이를 기준으로 error를 handling하기 위해 여러 방법을 사용해 보았습니다.

```
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
Error creating thread 0
Test failed!
$ thread_exec
Thread exec test start
This code shouldn't be executed!!
$ thread_exit
Thread exit test start
This code shouldn't be executed!!
$ thread_kill
Thread kill test start
This code shouldn't be executed!!
Kill test finished
$
```

위의 오류를 해결하였으나, 이번엔 모든 test program을 실행시켰을 때 전부 다 error일 때 출력되는 문구들이 출력되는 것을 확인할 수 있었습니다.

```
$ thread_exec
Thread exec test start
No available thread slot.
No available thread slot.
No available thread slot.
No available thread slot.
No available thread slot.
This code shouldn't be executed!!
$ thread_exit
exec: fail
```

debugging 문구를 넣어서 실행해본 결과 thread slot을 할당하지 못해서 생긴 문제라는 것을 확인할 수 있었고, 이를 중심적으로 code를 수정해보았습니다.

```
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
Thread state: UNUSED
lapicid 0: panic: remap
 801070d7 80107469 8010458e 80105f29 801050d9 8010624d 80105fe4 0 0 0
```

Thread state를 출력해보니 **UNKNOWN**이라고 떠서 thread state를 mapping해주고 초기화하는 함수를 만들어서 thread를 **T_EMBRYO** 상태로 만들기 전에 조건문(`if (curproc->threads[i]->state == T_UNUSED)`)을 통과시키게 했는데, thread의 state는 UNUSED가 되었지만 remap panic이 발생한 것을 확인할 수 있었습니다.
이 부분을 찾으면
`vm.c`에서 `static int mappages` 함수에서 찾을 수 있었습니다.

```
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P; // physical address와 permission을 설정
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
```

```
  return 0;
}
```

```
  /*growproc*/
  ...
  curproc->main_thread->sz = sz;
  curproc->sz = sz;
  ...
```

growproc에서 현재 process의 thread의 size를 할당한 size 값으로 선언해줬는데 이렇게 하고 xv6를 booting한 뒤 thread_test 파일을 실행시키면 다음과 같은 오류가 발생합니다.

```
Test 3: Sbrk test
ThrThread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
ead 0 start
lapicid 0: panic: remap
 80107097 80107429 80103baf 80105d8b 80105099 8010620d 80105fa4 0 0 0
```

remap 오류가 발생하는 이유는 sz 값이 update되지 않아서 메모리 접근할 때 문제가 생겼거나 잘못된 주소로 접근을 했기 때문이라고 생각합니다. 따라서 특정한 process의 sz를 update하는 것이 아닌 ptable에 있는 모든 process의 sz를 update 해줘야 consistency가 보장될 것이라고 생각했고 다음과 같이 수정했습니다.

```
/*growproc*/
...
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->pid == curproc->pid)
  p->sz = sz;
}
...
```

```
Test 3: Sbrk test
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Test 3 passed

All tests passed!
$
```

혹시나 하고 수정하였더니 생각지도 못하게 원하는 출력을 얻을 수 있었습니다.