Project03 LWP and Locking.

1. Design.

- LWP에 대한 디자인입니다.
- Pthread에 대한 기본적인 이해와 구조를 설명하고 있습니다.
- 구현 전 설명 단계입니다.

2. Implement.

- 구현 설명입니다.
- 명세에 설명한 API를 어떻게 구현했는지에 대한 설명이 담겨있습니다.
- locking 구현이 담겨 있습니다.

3. Result.

- 결과 단계입니다.
- 테스트 결과 및 실행 방법에 대한 설명을 담고 있습니다.
- locking 결과를 담고 있습니다.

4. Trouble Shooting.

- 결과에 대한 문제 상황에 대한 고찰입니다.
- 어떤게 문제였는지, 어떤 상황 때문에 이런 문제가 생겼는지에 대한 과정을 담아두고 있습니다.

1. Design.

- 디자인 전, 먼저 Pthread에 대해 알아봅시다.
- Pthread는 유닉스 계열의 운영체제에서 병렬적인 프로그램을 작성하는데 쓰는 LWP의 대표적인 구현체 중 하나라고 명세에 설명되어 있습니다.
- 이때 thread는 프로세스 내에서 실행되는 여러 흐름의 단위라고 설명되어 있습니다.
- 일반적으로 프로세스는 서로 독립적으로 실행되고, 자원을 공유하지 않지만, LWP의 경우엔 다른 LWP와 자원과 주소공간을 공유한다고 설명되어 있습니다.
- 그럼 이제, 구현 전 이 Pthread를 어떻게 구현해야할지 생각해봅시다.
- 일단 기본적으로 xv6엔 프로세스가 구현되어 있습니다.
- 그럼 프로세스와 쓰레드는 어떤 차이가 있는지 알아보는게 중요할 것 같습니다.
- 기본적으로 프로세스는 자원을 공유하지 않지만, 쓰레드끼리는 자원을 공유한다고 명세에 설명되어 있었습니다.
- 그럼 먼저 프로세스가 어떻게 실행되는지 코드를 분석할 필요가 있을 것 같습니다.
- 명세에서도 프로세스와 쓰레드는 거의 비슷하다고 하였으니, 프로세스가 실행되는 과정을 좀 더 알아보고, 그 다음 proc 구조체를 조금 손보고, proc 구조체가 어떻게 실행되고 움직이는지를 파악하면 쓰레드 구조를 짜는데 굉장히 도움이 될 것 같습니다.

- 먼저 exec, allowproc, growproc, fork, exit, wait을 차례대로 살펴봅시다.
- 먼저 allocproc 함수부터 살펴보겠습니다.

```
static struct proc*
allocproc(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;
```

```
  // 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;
}
```

- allowproc의 경우 처음에 프로세스를 할당해주는 역할을 하는걸 알 수 있습니다.
- 스택 포인터에 trap frame만큼의 사이즈를 빼고 계속해서 sp가 감소하고 마지막에 memset 함수를 통해 할당됨을 알 수 있습니다.
- 여기서 크게 수정할 점은 없어 보입니다.

- 다만 found부분에서 pid와 state를 변경해주는 점이 있는걸 보니, 여기서 thread에 필요한 부분을 처음에 설정해야함을 알 수 있습니다.

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

- 다음은 exec 함수 입니다.
- 이 함수는 처음 실행될 때 무조건적으로 실행되는, 즉 cmdline을 입력했을 때 실행되게 만들어주는 함수입니다.

```
// 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);
freevm(oldpgdir);
return 0;
```

- 두 개의 페이지를 할당하여 다음 페이지의 바운더리를 설정해주는 부분이 존재합니다.
- 처음에 인자를 집어 넣을 때 ustack을 사용합니다. 이 부분에 인자값을 집어넣고 리턴값을 집어넣는걸 확인할 수 있습니다.

- 다음으론 exit, fork, wait을 함꺼번에 살펴봅시다.

```
// Close all open files.
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);

// Parent might be sleeping in wait().
wakeup1(curproc->parent);

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

// Jump into the scheduler, never to return.
curproc->state = ZOMBIE;
sched();
panic("zombie exit");
}
```

- exit입니다. 최종적으로 조금 변경된 부분이 있지만 원본 부분이 크게 훼손된 부분이 없어 여기서 분석을 하는데 크게 어려움이 없을 듯 합니다.
- 다음은 wait함수 입니다.

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

- 원본 부분이 바뀐 부분이 없어 분석하는데 꽤 도움이 될 듯 합니다.
- 이 함수는 프로세스 테이블을 탐색 후 좀비 프로세스를 리핑해주는 역할을 합니다.

```
198  int
199  fork(void)
200  {
201    int i, pid;
202    struct proc *np;
203    struct proc *curproc = myproc();
204
205    // Allocate process.
206    if((np = allocproc()) == 0){
207      return -1;
208    }
209
210    // Copy process state from proc.
211    if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
212      kfree(np->kstack);
213      np->kstack = 0;
214      np->state = UNUSED;
215      return -1;
216    }
217    np->sz = curproc->sz;
218    np->parent = curproc;
219    *np->tf = *curproc->tf;
220
221    // Clear %eax so that fork returns 0 in the child.
222    np->tf->eax = 0;
223
224    for(i = 0; i < NOFILE; i++)
225      if(curproc->ofile[i])
226        np->ofile[i] = filedup(curproc->ofile[i]);
227    np->cwd = idup(curproc->cwd);
228
229    safestrcpy(np->name, curproc->name, sizeof(curproc->name));
230
231    pid = np->pid;
232
233    acquire(&ptable.lock);
234
235    np->state = RUNNABLE;
236
237    release(&ptable.lock);
238
239    return pid;
240  }
```

- 이 함수들은 추후에 thread_create, thread_join, thread_exit을 만들 때 참고될 것입니다.
- 전체적으로 process에 대해서 살펴보았으니, 이제는 구현해야할 thread에 대해서 알아보겠습니다.
- thread는 proc 구조체를 잘 다듬으면 쓸 수 있다고 명세에 설명되어 있습니다. 또, 프로세스의 페이지 테이블을 스레드 간에 공유하면 주소 공간을 공유할 수 있다고 표시되어 있습니다.
- 그렇다면, 기존 proc 구조체를 thread로 사용하되, thread임을 표시할 수 있는 몇 가지 표시만 달아두면 될 것 같습니다!
- 그럼 먼저 구현해야할 API들을 어떻게 디자인할지 고민해 봅시다.

- 크게 구현해야할 API는 3개 입니다.
  - `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);`
- 다음 3개의 API에 대해 구현을 어떻게 디자인할지 생각해보겠습니다.
- 먼저 `int thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg);` 함수 부터 구현을 생각해봅시다.
  - 이 함수의 경우 쓰레드를 생성하는 함수 입니다.
  - 우리는 쓰레드를 프로세스를 생성하는 것과 동일한 방향으로 생성하기로 했으니, fork, exec 함수를 이용하여 구현해야할 것 같습니다.
  - 이때 생성된 스레드는 같은 자원을 공유하고 있습니다.
  - 이 점을 생각하며 적절히 exec와 fork를 섞어서 구현해야할 듯 합니다.
- 다음으론 `void thread_exit(void* retval);` 함수의 구현입니다.
  - 쓰레드를 exit 시켜주는 함수 입니다. 쓰레드를 종료하고 값을 반환하는 함수입니다.
  - 모든 쓰레드는 반드시 이 함수를 통해 종료된다고 합니다.
  - 그렇다면 기존의 `exit()`함수를 이용하여 구현해야 할 듯 합니다.
- 마지막으로 `int thread_join(thread_t thread, void **retval);` 함수 입니다.
  - 지정한 스레드가 종료되길 기다리고, 스레드가 `thread_exit()`함수를 통해 반환한 값을 받아온다고 합니다.
  - 스레드가 종료된 후, 스레드에 할당된 자원들을 회수하고 정리해야하니, 기존의 `wait()`함수와 매우 닮아 있습니다.
  - wait을 이용하여 구현하면 될 듯 합니다.

- 다음으로 locking 입니다.
  - locking의 경우, 기존의 구현된 함수가 아닌 직접 구현인데, n개의 쓰레드에 대해 race condition을 막아야 합니다.
  - 이를 위해 조사해보니, n개의 쓰레드의 상호배제를 위한 Lamport's bakery algorithm이 있는 것을 알 수 있었습니다.
  - 쓰레드가 들어가기 위한 번호표를 부여하고, 번호표를 받은 순서대로 들어갈 수 있는 n개의 상호배제에 대한 알고리즘이 바로 램포트의 빵집 알고리즘입니다.
  - 이 구현을 우리의 명세에 맞게 고치기 위해선 먼저 기존 코드를 뜯어볼 필요가 있습니다.
  - 기본적인 Lamport bakery algorithm은 다음과 같습니다.

```
while(1) {              // 프로세스i의 진입 영역
  choosing[i] = ture;   // 번호표 받을 준비
  // (번호표 부여중 선점이 되어 같은 번호를 부여 받는 프로세스가 발생할 수 있음)
  number[i] = max(number[0], number[1], ..., number[n-1]) + 1;  // 번호표 부여
  chossing[i] = false;  // 번호표를 받음

  for (j = 0; j < n; j++) { // 모드 프로세스와 번호표를 비교함.
    while (choosing[j]);    // 프로세스j가 번호표를 받을 때까지 대기

    // 프로세스 j가 프로세스 i보다 번호표가 작거나(우선순위가 높고)
    // 또는 번호표가 같을 경우 j 가 i 보다 작다면
    // 프로세스 j가 임계구역에서 나올 때까지 대기.
    while ((number[j] != 0) &&
          ((number[j] < number[i])
          || (number[j] == number[i] && j < i)));
  }


  // Critical Section
  number[i] = 0;  // 임계구역 사용완료를 알림.
}
```

  - 이 구현을 우리의 lock과 unlock에 맞게 고치기 위해선 기존 함수의 추가와 더불어 수정이 필요합니다.

- 기존 함수에서 lock과 unlock을 수정해주어야 합니다.
- 빵집 알고리즘에서 쓰이는 choosing, number배열이 필요할 듯 합니다.
- 또, 번호표를 뽑아야하니, thread의 id 값이 필요합니다.
- 그에 맞게 수정하면 결과가 나올 듯 합니다.

- 이제 어느정도 디자인을 생각해 두었으니, 실제로 구현을 해보겠습니다.

2. Implement.

- `thread_create(thread_t *thread, void* *(start_routine)(void *), void* arg)` 구현입니다.
- `exec()`, `fork()`함수를 이용하여 구현해 봅시다.
- 먼저 `proc.h`를 수정해 줍시다.
  - 현재 이 프로세스가 thread인지 process인지 나타내주는 숫자가 필요합니다.
  - 쓰레드의 id를 나타내는 변수가 필요합니다.
  - retval이라는 주소값을 받아오는 변수가 필요합니다. `thread_join()`사용할 것 입니다.
  - 이 쓰레드를 생성한 creator라는 프로세스가 필요합니다.

```
struct proc {
  ...
  //thread
  typedef uint thread_t;
  thread_t tid;
  thread_t nexttid;
  int is_thread;
  void * retval;
  struct proc* creator;
}
```

- 다음으로는 `thread_create()` 함수 구현입니다.
  - 저희는 프로세스를 쓰레드로 사용할 예정이기 때문에(is_thread라는 변수를 사용해서) 먼저 프로세스를 할당시켜주어야 합니다.
  - 쓰레드의 pgdir(페이지 디렉토리)는 공유되어야 하기 때문에 현재 프로세스의 pgdir로 할당시켜 줍니다.
  - 프로세스를 할당 후, 페이지 바운더리를 할당시켜주어야 합니다.
  - 그 부분은 `exec()`에서 가져올 수 있습니다.

```
// from exec.c
// Allocate two pages at the next page boundary.
// Make the first inaccessible.  Use the second as the user stack.

acquire(&ptable.lock);

//share pgdir.
np->pgdir = curproc->pgdir;

if((curproc->sz = allocuvm(np->pgdir, curproc->sz, curproc->sz + 2*PGSIZE)) == 0)
  return -1;

clearpteu(np->pgdir, (char*)(curproc->sz - 2*PGSIZE));
```

- 그 다음 메모리 사이즈와 쓰레드의 초기 설정을 해줍니다.

```
//allocate memory size to thread
np->sz = curproc->sz;
sp = np->sz;

//new process -> now it is thread.
np->is_thread = 1;
np->parent = curproc->parent;
np->creator = curproc;
```

- `(uint)thread_t`에 쓰레드의 tid를 넣어주고, tf엔 현재 프로세스의 tf를 할당해줍니다.

```
//np(thread) creator is curproc.
//thread - process is not parent-child
//but have relationship
//beacause np->parent = curproc->parent.
np->tid = curproc->nexttid++;
*thread = np->tid;
*np->tf = *curproc->tf;
```

- 다음으로 스택 설정입니다.
- 쓰레드 별로 userstack이 달라야 하기 때문에 stack값을 설정해주어야만 합니다.

- 이 부분은 exec.c에 있습니다. 참고하여 사용하겠습니다.
- ustack의 4칸을 사용하여 0번째엔 return주소, 3번째엔 arg값을 할당해줍니다. 1번째에 3번째 값의 주소를 넣어주고, 2번째 값엔 0을 넣어줍니다.
- 2,3번째 배열을 0,1에서 0xffffff, arg로 넣어도 무방합니다.
- 다만 exec에서 `ustack[3+argc] = 0;`부분이 있어 이에 맞추어 4칸을 맞추어 사용하겠습니다.
- vm.c에 있는 `copyout()`을 이용하여 stack을 복사하여 줍시다.

```
//stack init
ustack[3] = (uint)arg;
ustack[2] = 0;
ustack[1] = ustack[3];
ustack[0] = 0xffffffff;

//stack pointer down.
//4Byte * 4 = 16
sp -= 16;

// Copy len bytes from p to user address va in page table pgdir.
// Most useful when pgdir is not the current page table.
// uva2ka ensures this only works for PTE_U pages.
// fatal copyout error and copyout stack.
if(copyout(np->pgdir, sp, ustack, 16) < 0)
  return -1;
```

- 레지스터 설정 입니다.
- 쓰레드가 시작할 진입점을 eip에 할당함으로써 쓰레드가 올바르게 진입하여 시작할 수 있도록 설정해줍니다.

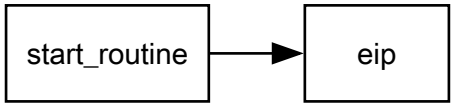


Figure 1: stack init

```
//register setting.
np->tf->eax = 0;
np->tf->eip = (uint)start_routine;
np->tf->esp = sp;
```

- 다음으로, 쓰레드는 메모리 사이즈를 공유합니다.

```
//share sz value for np.
struct proc* p;

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->parent->pid == np->parent->pid){
    p->sz = np->sz;
  }
}
```

- 그 후, 전체적인 동작사항은 `fork()`시스템 콜을 따릅니다.

```
for(int 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));

// switchuvm-like
pushcli();
lcr3(V2P(np->pgdir));
popcli();

np->state = RUNNABLE;

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

- `thread_exit()`구현입니다.
- 이 함수는 `exit()`함수와 유사하기 때문에, 기존 함수를 이용하여 구현하였습니다.

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

  // it is not process
  // if(curproc == initproc)
  //   panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++) {
    if(curproc->ofile[fd]) {
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd]=0;
    }
  }

  ...



  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  curproc->retval = retval;
  sched();
  panic("zombie exit");
}
```

- `thread_join()`구현입니다.
- 이 함수는 `wait()`함수와 매우 유사하기 때문에 전체적인 동작 구조를 그대로 따라가면 됩니다.
- 프로세스 테이블을 탐방하면서 현재 지정한 쓰레드가 아니면 실행하지 않습니다.
- 이때 명세에 언급된 `retval`만 `ZOMBIE`프로세스를 발견했을 때, 초기화 시킨 후 `p->retval`에 넣어주면 됩니다.
- 이때 쓰레드는 `pgidr`을 공유하므로 `freevm(p->pgdir)`부분은 주석 처리 해줍니다.

```
...
if(p->tid != thread)
    continue;
if(p->state == ZOMBIE){
  // Found one.
  kfree(p->kstack);
  p->kstack = 0;
  p->pid = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->killed = 0;
  p->nexttid = 0;
  p->is_thread = 0;
  p->tid = -1;
  p->state = UNUSED;

  //share pgdir.
  //so don't freevm
  //freevm(p->pgdir);

  //add retval.
  *retval = p->retval;

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

- 명세에 나온 3가지 함수에 대한 구현은 끝났습니다.
- 하지만 xv6와의 상호작용을 위해선 기존 시스템콜 또한 변경해야 합니다.
- 변경해야할 시스템콜은 4가지가 있습니다.
  - `exec()`
  - `exit()`
  - `kill()`
  - `growproc()`
- 먼저 `exit()`함수부터 수정해봅시다.
  - exit의 경우, 기존 부분은 거의 동일하나, thread가 `exit()`을 호출했을 시, 다른 thread와 자기 자신을 죽이도록 해야합니다.
  - 이때, thread의 자식이 프로세스인 경우, swap을 통해 자리를 바꾼 후 kill을 진행합니다.

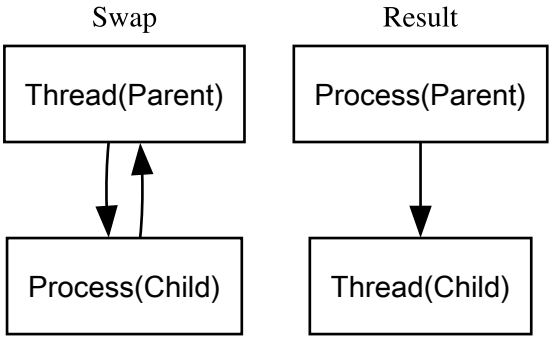


Figure 2: swap thread and process

  - 간단하게 그림으로 표현하면 이렇게 표현 할 수 있습니다.
  - 이 같은 경우에 함부로 exit을 하게 되면 부모를 제대로 찾지 못한 채 계속해서 shell이 재시작하게 되는 경우가 발생하게 됩니다. 이 경우를 발생시키지 않게 하기 위해 swap과정을 해주어야만 합니다.
  - 따로 함수로 만들어주면 필요한 부분이 있을 때마다 함수 호출만 하면 되기 때문에 편합니다. 따라서 따로 함수로 만들어 주겠습니다.
  - 이때 ptable.lock을 걸어주어야 합니다. 아니면 쓰레드가 꼬일 가능성이 생깁니다.

- 코드로 구현해보면 이렇게 됩니다.
- 이전에 관계를 위해 만들어놓았던 creator를 이용하면 됩니다.

```
void thread_swap(struct proc* curproc) {
...
if(curproc->is_thread) {
  curproc->parent = curproc->creator->parent;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p == curproc)
      continue;

    if(p->creator == curproc->creator || p == curproc->creator) {
      p->creator = curproc;
      p->is_thread = 1;
    }
  }
  curproc->creator = 0;
  curproc->is_thread = 0;
  }
}
```

- 그 다음으론 모든 thread를 kill 해주어야 합니다.
- 이 과정은 exit이 호출 되었을 때, 그리고 exec가 실행 될 때 마다 계속해서 실행될 것 입니다.

```
void killallthread(struct proc* curproc) {

  ...

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == curproc->pid)
      continue;

    if(p->is_thread) {
      //swaping
      if(p->parent == curproc->parent) {
        if(p->kstack){
          kfree(p->kstack);
        }
        p->kstack = 0;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        p->is_thread = 0;
        p->tid = -1;
        p->creator = 0;
        p->sz = 0;
      }
    }
  }
}
```

- 자, 이제 `exit()`에 이 과정을 추가해줍니다.

```
void exit(void) {
  ...
  thread_swap(curproc);
  killallthread(curproc);
  ...
}
```

- 나머지는 동일 합니다.
- 이번엔 `kill()`을 고쳐보겠습니다.
- `kill()`의 경우 Process가 kill을 당하면 그 프로세스에 속해있는 모든 프로세스를 종료해줘야 합니다. 그 부분만 추가하면 끝입니다.

```
int kill(int pid) {
  ...
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    //p->creator->pid is same set of thread.
    //if kill process -> same set of thread killed.
    if(p->creator->pid == pid){
      p->killed = 1;
      if(p->state == SLEEPING)
        p->state = RUNNABLE;
    }
  }
  ...
}
```

- wait, fork의 경우 수정이 없어도 실행됩니다. 프로세스와 쓰레드 간의 차이 때문에 수정없이 진행해도 문제가 없는 것으로 파악됩니다.
- `allocproc()`, `growproc()`도 약간의 수정이 필요합니다.
  - 전자의 경우, 초기에 프로세스 혹은 쓰레드를 할당할 때, `tid, is_thread, nexttid` 값을 설정해주어야 합니다.
  - 후자의 경우, 스레드의 주소공간이 늘어났다면, 같은 프로세스 군에 속해있는 모든 thread가 이 변경된 sz의 값을 알아야합니다. 그렇지 않으면 page_fault가 발생합니다. 이때 후자는 process가 아닌 thread일 때만 진행합니다.

- 간단하게 바로 수정해보도록 하겠습니다.

```
static struct proc* allocproc(void) {
  ...
  p->tid = -1;
  p->is_thread = 0;
  p->nexttid = 0;
}
```

```
int growproc(int n) {
  ...
  //sz must be share
  struct proc* p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    //same set of Thread.
    if(p->is_thread &&
    (p->creator == curproc->creator || p == curproc->creator)){
      p->sz = curproc->sz;
    }
  }
  ...
}
```

- 이제 마지막으로 이 시스템 콜을 감쌀 Wrapper Function만 만들면 되겠습니다. `sysproc.c`파일에 추가해줍니다.

```
int sys_thread_create(void) {
  thread_t* thread;
  void* (*start_rotine)(void *);
  void* arg;
  if(argptr(0, (void*)&thread, sizeof(thread)) < 0
  || argptr(1, (void*)&start_rotine, sizeof(start_rotine)) < 0
  || argptr(2, (void*)&arg, sizeof(arg)) < 0)
    return -1;
  return thread_create(thread, start_rotine, arg);
}

int sys_thread_exit(void) {
  void* ret_val;
  if (argptr(0, (void*)&ret_val, sizeof(ret_val)) < 0)
    return -1;
  thread_exit(ret_val);
  return 0;
}

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

- 다음으로 Locking 입니다.
- Locking의 경우 먼저 Lamport의 알고리즘을 적용해봅시다.
- 이를 명세에 맞게 고쳐봅시다.
  - 먼저 choosing과 number가 필요합니다.

```
int choosing[NUM_THREADS]; //번호표를 받을 준비.
int number[NUM_THREADS]; //번호표 부여.
```

  - 다음으로 번호표를 부여할 thread의 id인 tid를 추가해 봅시다.

```
int tids[NUM_THREADS];
for (int i = 0; i < NUM_THREADS; i++) {
  tids[i] = i;
  ...
}
```

  - 번호표를 부여했다면, 이제 Lock과 unlock만 구현하면 됩니다.
  - 특정 쓰레드가 번호표를 받을 준비를 하고, 번호표 중 가장 높은 값을 미리 선점해줍니다.
  - 이때, 번호가 낮을수록 우선순위는 높아집니다.

```
void lock(int tid) {
  choosing[tid] = 1; //특정 쓰레드가 번호표 받을 준비중.
  ...
  number[tid] = max + 1;
  choosing[tid] = 0;
}
```

  - 이제 알고리즘을 그대로 적용하면 됩니다.

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

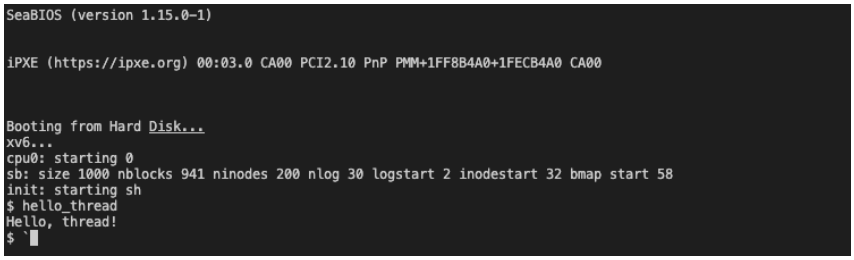
- unlock의 경우 이제 부여한 번호를 초기화 시킵니다.

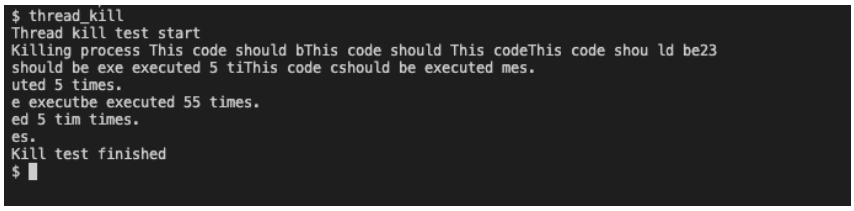
```
void unlock(int tid) {
  number[tid] = 0;
}
```

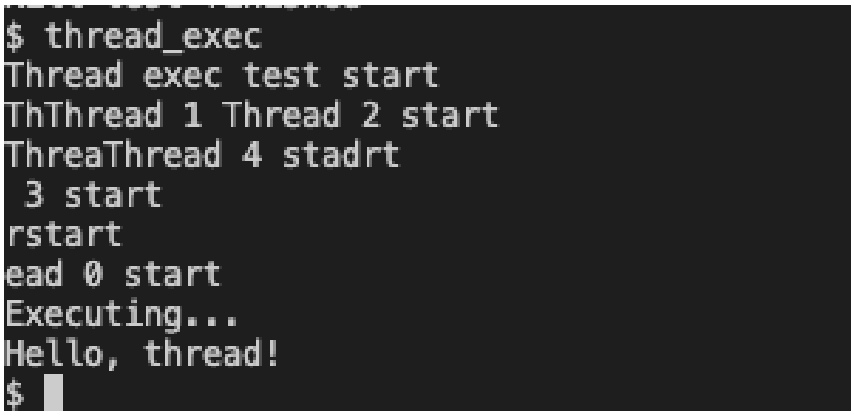
- 이때, busy waiting이 발생할 수 있습니다.
- 또 thread의 수가 많아지면 급격히 느려집니다.

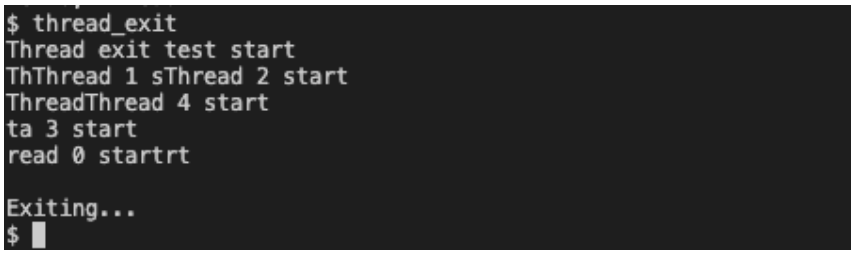
3. Result.

- 차례대로 실행해줍시다.
- hello_thread의 경우 exit()가 정상적으로 실행됨을 알 수 있습니다.
- thread_test의 경우 thread와 관련된 명세가 모두 올바르게 작동함을 알 수 있습니다.
- thread_kill, thread_exec, thread_exit의 경우 기존 시스템콜과 thread간의 상호작용이 올바른지 확인할 수 있습니다.









```
$ thread_test
Test 1: Basic test
Thread Thread 1 st0 start
Thread 0 end
art
Parent waiting for children...
Thread 1 end
Test 1 passed

Test 2: Fork test
ThrThread 1 start
Thread Thread 3 startThread 4 staead 0 start
2 start

rt
Child of thread Child of thread Child of threChild of thread 0Child of thread 1 star3 start
ad 4 start
2 startt

start
Child of thread 0 end
Child of thChild of thread Child of threChild of tTread 1 end
3 end
hread 2 end
hread 4 end
ad 0 endThread
1 Thread 2 end
Thread 3 eThread end
nd
4 end
Test 2 passed

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

All tests passed!
$
```

- 다음으론 Locking 결과입니다. 적절히 NUM_ITERS와 NUM_THREADS의 수를 변형시켜 결과를 나타내었습니다.

```
root@b3f7fea370c5:/OS/xv6-public# ./pthread_lock_linux
shared: 1000000
```

- 상호배제가 잘 되었는지 중간에 printf문을 넣어서도 확인하였습니다.

```
root@b3f7fea370c5:/OS/xv6-public# ./pthread_lock_linux
lock 0  unlock 0
lock 1  unlock 1
lock 2  unlock 2
lock 3  unlock 3
lock 4  unlock 4
lock 5  unlock 5
lock 6  unlock 6
lock 7  unlock 7
lock 8  unlock 8
lock 9  unlock 9
shared: 1000
```

4. Trouble Shooting.

1. 구현애서의 막막함.
   - 가장 큰 문제점은 이 명세를 처음 접근 했을 때, 어떻게 구현하느냐 였습니다. 접근 자체가 굉장히 힘든 막막한 상황이었습니다. 해결한 방안은 명세였습니다. 명세에서 기존 코드를 분석하면 많은 힌트를 얻을 수 있다, 프로세스와 거의 동일한 형태이니 proc 구조체를 조금만 손보면 된다는 말이 가장 큰 힌트가 되었습니다.
   - proc구조체를 쓰레드 구조체로 사용하고 구분을 두자라는 접근이 가장 좋은 접근법 이었습니다. 이 접근법을 통해 기존 process가 어떻게 실행되는지만 알면 구현에서의 어려움은 거의 없었습니다. 따라서 기존 process가 어떻게 진행되는지 그 과정을 분석했습니다.

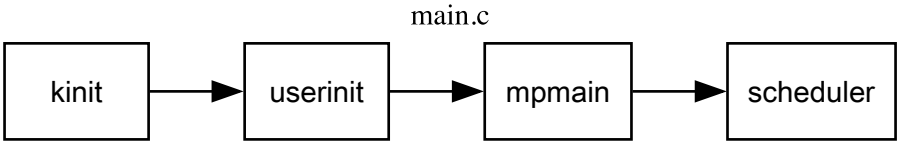


Figure 3: Process progress

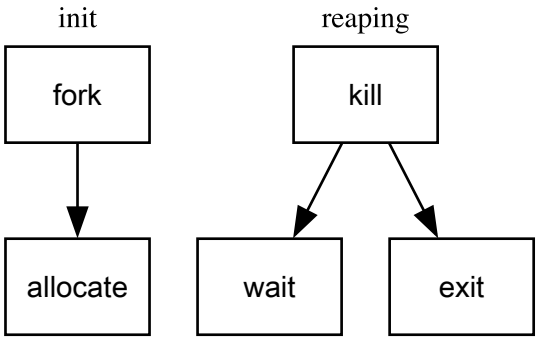


Figure 4: Process allocate

   - 제가 이해한 프로세스의 흐름과 실행과정 이었습니다. 이 프로세스가 만들어지고 kill되고 reaping되는 과정이 동일하게 쓰레드에게서 일어난다고 생각하고 구현하니, 구현에 방향성을 잡을 수 있었습니다.
   - 구현의 방향을 잡고 난 후 구현을 진행했습니다.
2. 구현과정에서의 문제점.
   - 먼저 처음 쉘을 실행하고 테스트를 진행한 후, 그 다음 테스트를 진행하면 계속해서 page not present 트랩에 걸리는 상황이었습니다.
   - 어떤 문제인지 계속 고민해보니, page트랩이 걸리는건 page사이즈를 쓰레드에 맞게 공유해주고 조절해주는 함수에서 발생하는 문제임을 알 수 있었습니다. 그 페이지 사이즈를 공유해주는건 `growproc()`함수임을 알 수 있었습니다.

- 따라서 growproc에서 sz를 공유해주는 부분에서 문제점을 발견했습니다.
- 기존 조건 문은 프로세스일때도 무조건적으로 sz를 공유해줍니다. (is_thread일 때를 사용하지 않았기 때문에) 하지만 우리는 프로세스가 쓰레드일때만 sz를 공유하므로 조건문에 조건을 추가해주니 문제가 해결되었습니다.

```
if(p->is_thread && (p->creator == curproc->creator || p == curproc->creator))
```

- 다른 부분은 exit, exec, kill 호출 시 쉘이 다시 시작되는 문제점이었습니다. 이건 쓰레드를 없애는 과정에서 문제가 생겼음을 의미합니다. 모든 쓰레드가 아닌 모든 프로세스를 죽여버려 다시 처음부터 initproc이 실행될 때 그렇게 됨이라고 추측했습니다.
- 따라서 `exit()`함수를 고쳐야 겠다고 분석했습니다.
- 하지만 고칠 부분 중, `exec()`와 함께 사용하는 부분이 있었는데, 바로 `killallthread()`, `thread_swap()`부분이었습니다.
- 이 부분에서 문제가 생겼음을 알 수 있었습니다. 따라서 이 부분을 수정해야하는데, 잘못된 부분을 발견할 수 있었습니다. 바로 swap한 후 killallthread를 실행시킬 때의 조건이었습니다. 기존 조건은 다음과 같았습니다.

```
if(p->is_thread && (p->creator == curproc->creator || p == curproc->creator))
```

- p가 쓰레드면서 현재 내 쓰레드를 만든 creator가 프로세스와 같다면 이 쓰레드를 만든 creator를 의미하니 p가 쓰레드임을 알 수 있고 그럴 경우에 이 쓰레드를 전부 없앤다는 방식이었습니다. 하지만, 이미 swap과정에서 이 조건을 썼으므로 조건이 바뀌어야 했습니다.
- 따라서 현재 p가 쓰레드이면서 p의 부모가 현재 내 프로세스의 부모와 같다면 전 과정에서 진행한 thread_swap에 의해 부모가 바뀌었을 테니 지금 내가 없애야하는 쓰레드임을 알 수 있습니다.
- 이 조건문을 수정하고 나니 문제가 해결 되었습니다.

```
if(p->is_thread && (p->parent == curproc->parent))
```

- Lock의 경우 Thread가 비정상적으로 많아지면 segfault가 뜨며 shared_resource값이 제대로 찍히지 않는 오류가 발생했습니다.
- 너무 많은 수의 쓰레드 때문이라고 추측됩니다.