Project 03

컴퓨터 소프트웨어학부 2022056262 곽주리

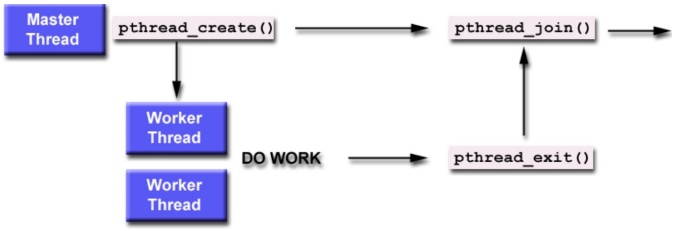
thread 구현과 locking 구현 사이 유기점이 없어, wiki를 구분해서 작성하였습니다.

과제 목표

XV6에서 Light Weight Process(LWP)를 구현하고 Linux에서 pthread간 상호 배제를 보장하기 위한 locking method를 고안

Design

lab 9, 6 ppt를 참고하여 작성하였다.



시작하기에 앞서 lab 6 ppt를 통해 pthread의 작동방식을 살펴보았다. master thread가 일종의 부모 process이고 create, exit, join은 각각 fork, exit, wait(reap) 역할을 한다. master thread는 자식을 만들고 자식이 종료될 때까지 wait하다가 자식이 종료되면 join으로 자식의 반환값을 받고 자원을 정리한다.

proc 구조체와 thread 구조체를 만들어, proc 구조체에서 thread list를 다루게 만들까 했으나 process와 thread의 작동 방식은 매우 유사하기 때문에 별도의 구조체를 만드는 대신 proc 구조체에 변수들을 추가해주었다.

단군할아버지격인 Master thread가 thread를 생성한 process이므로 master thread를 표시. thread id 표시, return value 표시

scheduler 함수를 확인해보니 ptable에 thread의 정보가 잘 저장되어 있고, runnable 상태 역시 잘 변경됐다면 다른 프로세스와 비슷하게 스케줄링될 것이라 생각한다. 따라서 scheduler 함수는 수정할 필요가 없다.

thread 관련 함수

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

process를 새로 만드는 과정과 유사하게 작성하면 될 것 같아 fork 함수를 살펴보았다.

fork, allocproc 함수를 보니 alloproc 함수는 pid 부여, kernel stack 부여, 메모리 할당등의 일을 하고 있으므로 allocproc 함수를 그대로 사용해도 될 것이라 판단하였다.

따라서 fork함수를 참고하여 thread에 불필요한 내용이나 맞지 않은 내용을 수정하여 작성했다.

thread는 공간을 공유하므로 pgdir은 copy없이 같은 곳을 가리키도록 변경했다.

파일, 작업 디렉토리 정보를 공유하는 것은 thread에도 필요하다고 생각해, 그대로 두었다.

다만 arg을 담아 start_routine을 호출하려면 해당 정보를 담을 stack이 필요할텐데 allocproc 함수에서 할당 받은 kernel stack은 kernel mode에만 사용되는 stack이므로 user stack을 추가로 할당해야 했다.

새로운 프로그램이 load되면 exec 함수에서 메모리가 할당되므로 exec 함수에서 user stack을 할당하는 방식을 참고하였다. 해당 방식은 lab ppt 9를 통해 직관적으로 이해할 수 있었다.

```
//EXEC.C 파일 발췌
// 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;
```

해당 부분을 참고하였다. sz는 메모리 크기이므로 thread의 sz는 자신이 속한 프로세스의 sz로 변경해주었다. exec은 argv가 **였으나 thread에서는 *arg라 인자가 하나이므로 for문은 생략하였다.

⇒ sbrk에서 메모리 할당을 받으면 growproc, allocuvm 함수를 호출하는데, 중복 할당을 방지하기 위해 처음엔 allocuvm, sbrk만 수정하였다.

sbrk에서 allocuvm에 접근하기 전에 ptable.lock으로 동기화해주고 allocuvm 안에서 메모리를 할당할 때 이미 할당된 메모리를 요청하면 pagesize만큼 뛰어 넘으며 할당 가능한 메모리를 찾아서 할당해주는 방식으로 구현했다. 그러나 여러 thread가 요청하는 경우 중복 할당이 되는 문제는 없지만 할당이 가능한 page를 찾는 과정에서 계속 잘못된 주소를 가리킨다는 오류가 떠서 구현 방식을 변경하였다.

thread가 여러 개일 경우 master thread의 sz는 여러 thread의 메모리를 다 할당한 후의 전체 크기인 반면 각각의 thread는 본인의 sz를 갖고 있기 때문에 추가 할당을 받을 때 그 이후에 들어온 thread의 page를 건든다고 생각했다.

따라서 create에서 메모리를 할당한 이후, 모든 thread(방금 만들어진 thread 외에 같은 프로세스 안내 모든 thread)의 sz를 master thread의 sz로 설정하여 다른 thread의 메모리에 할당을 요청하는 경우를 사전에 방지하였다.

초기에 구현하였던 allocuvm 코드는 다음과 같다.

```
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;
  pte_t *pte;

  if(newsz >= KERNBASE)
    return 0;
  if(newsz < oldsz)
```

```
    return oldsz;

  uint n = newsz - oldsz;
  n= PGROUNDUP(n);
  uint alloc= 0;
  a = PGROUNDUP(oldsz);
  for(; ; a += PGSIZE){
    // Check if the page table entry already exists and is mapped
    if ((pte = walkpgdir(pgdir, (void*)a, 0)) != 0 && (*pte & PTE_P)) {
      cprintf("allocuvm: address %p already mapped, pte %p\n", (void*)a, (void*)*pte);
      continue;  // Skip already mapped addresses
    }
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
    alloc += PGSIZE;
    if(alloc == n) break;
  }
  return a+PGSIZE;
}
```

추가로 할당 받은 후의 sz가 newsz, 기존 sz가 oldsz이다. 즉 추가 할당 받을 메모리의 크기는 newsz-oldsz이다. 만약 이미 할당된 메모리라면 continue로 넘어가고 할당 가능한 메모리가 있다면 메모리 할당 이후에 지금까지 할당 받은 메모리의 크기에 PGSIZE 만큼 더한다. 더할 때마다 지금까지 할당 받은 메모리와 할당 받아야 하는 메모리를 비교하고 같다면 할당된 후의 sz를 return한다.

이 방식 대신, thread_create에서 메모리 할당 이후 ptable을 돌며 같은 프로세스 내의 thread를 모두 찾아 sz를 업데이트 해주는 방식을 사용했다.

따라서 수정한 함수는 growproc뿐이다.

`void thread_exit(void *retval)` (proc.c)

exit 함수를 참고하여 작성하였다. (proc.c)

기존 exit 함수는 열었던 모든 파일을 닫고 자식이 종료되고 결과를 reap하기 위해 wait 중이던 부모를 깨운다. 그 후 ptable에서 종료시킬 process를 부모로 갖고 있는 process들을 모조리 찾고 자식 대신 부모가 먼저 종료되므로 해당 process의 부모를 initproc으로 변경한 후 만약 그 process의 상태가 좀비라면 깨워서 reap해준다.

종료할 process의 상태를 zombie로 변경하고 스케줄러를 호출해서 다른 process가 실행되도록 한다.

만약 스케줄링에 실패하면 panic을 발생한다.

exit 함수는 기존 함수에서 불필요한 부분만 제거하고 다 동일하게 사용하고 추가로 return value 설정만 해주면 될 것 같다.

`int thread_join(thread_t thread, void **retval)` **(proc.c)**

join 함수는 wait 함수와 자식이 종료될 때까지 기다리다가 반환값을 받아주고 자원을 회수하는 역할을 하는 데 유사하기 때문에 wait 함수를 참고하여 작성하였다.

wait 함수는 현재 process의 자식 process를 찾아서 자식 process가 zombie면 kernel stack과 page table 자원을 회수 및 proc의 member 변수들을 초기화한다. 만약 ptable을 다 돌았는데 자식을 못 찾았거나 현재 프로세스가 kill됐다면 -1을 return한다.

만약 자식은 있지만 zombie 상태 즉 종료된 프로세스가 없다면 sleep해서 자식이 종료 후 깨울 때까지 기다린다.

thread_join 함수 역시 ptable을 돌면서 id가 thread고 현재 process의 자식 process라면 자원을 회수하고 정리하고 자식 thread가 종료하면서 return한 값도 저장해준다. 기존의 wait 함수를 조금 수정해서 사용할 것이다.

위 3개 함수 모두 syscall로 선언하여 kernel과 user mode에서 사용할 수 있도록 만들었다. syscall 등록 방식은 lab3 ppt를 참고하였다. project 01에서 get_gpid syscall을 등록하듯 makefile, 각종 header file에 추가하였다.

syscall

fork

thread가 fork 함수를 호출하면, allocproc으로 새로운 process를 할당하고 thread의 주소 공간의 내용을 복사할 수 있어야 한다. 그 과정이 정상적으로 수행되려면 thread의 pgdir, name, 열린 파일 정보, 메모리 크기, trap frame 정보 등이 잘 설정되었어야 한다. 이는 방금 나열한 정보는 thread_create 함수에서 모두 구현해두었으므로 잘 작동할 것이라 생각한다.

exec (syscall은 아니지만 수정한 함수라서 첨부하였다.)

새로운 process를 할당하기 전에 현재 process를 부모로 하는 모든 thread들을 찾아서 종료시킨 후 다시 한 번 방금 종료시킨 thread를 하나씩 thread_join을 통해 회수해준다. 따라서 기존 exec 함수에 해당 과정을 수행하는 코드를 추가한다.

⇒ 이 방식에서 pgdir을 잘못 건드려서인지 계속해서 재부팅되는 오류가 발생하였다. thread_exit, thread_join 함수를 고치다보니 thread_test 파일까지 안 돌아가는 문제가 발생하여 thread api를 사용해서 자원을 회수하는 대신에 기존 exit, wait 함수에서 수행하던 종료, 자원 해제 로직을 갖고 와 추가해주었다.

sbrk

cscope를 이용해서 해당 함수를 살펴보았다. 함수 내부에서 growproc 함수를 호출하고 있어, growproc 함수도 살펴보았다. n에 따라서 메모리를 키우거나 줄였다. 함수 logic 자체를 변경할 필요는 없지만 여러 thread가 동시에 요청해서 문제가 생기는 경우를 방지하기 위해 allocuvm 안에서 lock을 걸어주었다. 중복 할당 문제는 처음엔 allocuvm에서 할당 가능한 page가 나올 때까지 돌면서 찾는 방식으로 해결하려고 하였으나 잘못된 주소를 가리킨다는 오류가 생겨서 logic을 수정하였다. 본인의 sz를 토대로 메모리 할당을 요청하기 때문에 이후에 생성된 sz는 알지 못 해, 다른 thread의 page에 자꾸 접근한다고 생각하여 thread가 생성될 때마다 같은 프로세스 내의 thread끼리 sz를 동기화시켜주었다. 또한 growproc 이후에 update된 sz를 다시 한 번 동기화시켜주었다.

exit

기존 코드에서 만약 종료하는 process가 thread라면 같은 process 내의 모든 thread를 종료시킬 수 있도록 wait, exit 함수에서 사용하는 자원 회수 방식을 넣어주었다.

kill

기존 kill함수를 보면 인자로 받은 pid를 가진 process를 죽이고 상태가 sleep 상태면 실행 가능한 상태로 변경한다. 기존 xv6에 달려있던 주석을 통해 trap.c에서 kill이 된 process를 처리한다고 유추할 수 있었다. 따라서 trap.c에서 kill된 process를 관리하는 방식을 찾아보았다.

trap.c에서 killed가 1인지 (kill이 됐는지) 확인하고 만약 그렇다면 exit 함수를 호출해서 해당 process를 처리했다. trap.c의 logic은 thread와 process 무관하게 올바르게 작동할 것 같아서 kill 함수만 수정하였다. 기존에는 인자로 받은 pid의 process만 kill했지만 해당 thread가 속한 process 내의 모든 thread가 모두 kill되어야 하므로 ptable을 돌면서 kill할 process를 찾는 과정만 수정하였다.

⇒ 추후에 이 방식 대신 exit에 이 행위를 수행하도록 변경하였기 때문에 kill 함수는 변경하지 않았다.

sleep

process, thread와 무관하게 작동하는 함수여서 수정하지 않았다. 그냥 process 구조체의 변수만 수정해주기 때문에 thread와는 무관하다.

Implement

`struct proc` **(proc.h)**

```
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 tid;
  struct proc *master;
  void *retval;
};
```

thread 구현을 위해 필요한 변수만 추가해주었다.

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

```
//thread 생성. 프로세스랑 비슷한 방식이므로 fork를 참고
int
thread_create(thread_t *thread, void* (*start_routine)(void *), void *arg)
{
  struct proc *nt;
  struct proc *curproc = myproc();
  int i;
  uint ustack[2];

  // Allocate process.
  if ((nt = allocproc()) == 0) {
    return -1;
  }
  //메모리 할당 이후 thread 관련 변수 초기화
  acquire(&ptable.lock);

  // Master thread 찾기
  struct proc *master = curproc;
  if(master->tid!=0) nt->master=master->master;
  else nt->master = master;
  // thread는 master process의 pid와 동일
```

```
nt->pid = nt->master->pid;
nt->parent = curproc;
nt->tid = nexttid++;
*thread = nt->tid;
// master process와 주소 공간, 메모리 사이즈 공유
nt->pgdir = nt->master->pgdir;

// fork 함수 trap frame 설정
*nt->tf = *curproc->tf;

//exec 함수에서 user stack 할당하는 부분 참고
//고유한 스택 크기 할당 (스레드 스택은 각 스레드에 대해 2 페이지 할당)
nt->master->sz= PGROUNDUP(nt->master->sz);
uint oldsz=nt->master->sz;
//기존 프로세스 크기를 업데이트하여 새로운 스택 공간 할당
//allocuvm의 return값은 새로운 sz. 즉 master thread는 새로운 thread
//할당 이후 sz값 update.
if ((nt->master->sz = allocuvm(nt->master->pgdir,  oldsz, oldsz+2 * PGSIZE)) == 0) {
  cprintf("goto bad!\n");
  goto bad;
}
clearpteu(nt->master->pgdir, (char *)(nt->master->sz-(2 * PGSIZE)));
//할당된 thread의 sz update
nt->sz = nt->master->sz;
uint sp =nt->master->sz;

// 스레드의 사용자 스택에 start_routine 할당
ustack[0] = 0xffffffff; // fake return PC
ustack[1] = (uint)arg;  // start_routine의 argument
sp -=sizeof(ustack);

if (copyout(nt->pgdir, sp, ustack, sizeof(ustack)) < 0) {
  cprintf("copyout failed\n");
  goto bad;
}
//같은 process 내에 있는 thread는 sz 다 동일하게 update해서
//메모리 중복 할당 방지.
for(struct proc* p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
  if(p->pid == curproc->pid)
    p->sz = curproc->sz;
}
//메모리, stack은 다 할당 받았으므로 start routine 실행 위해 trap frame 설정
//stack pointer와 다음에 실행할 instruction의 주소를 start_routine으로 변경
nt->tf->eip =(uint) start_routine;
nt->tf->esp = sp;

release(&ptable.lock);


//파일 디렉토리 정보 전달 fork 함수 참고
for (i = 0; i < NOFILE; i++)
```

```
    if (curproc->ofile[i])
      nt->ofile[i] = filedup(curproc->ofile[i]);
  nt->cwd = idup(curproc->cwd);
  safestrcpy(nt->name, curproc->name, sizeof(curproc->name));

  acquire(&ptable.lock);
  nt->state = RUNNABLE;
  release(&ptable.lock);
  return 0;
bad:
  // In the bad label, add:
  cprintf("nt->pgdir: %p\n", nt->pgdir);
  if (nt->pgdir){
    release(&ptable.lock);
    freevm(nt->pgdir);
  }
  nt->state = UNUSED;
  return -1;
}
```

fork, exec 함수를 참고하여 thread의 kernel stack, user stack을 할당해주고 start_routine 설정을 위해 eip, esp, user stack 정보를 수정한다. thread와 관련된 정보들도 여기서 설정해준다. sz, pid는 master thread와 동일, pgdir 공유 등...

+중복 할당 문제를 방지하기 위해 모든 thread의 sz는 항상 master thread의 sz와 같게 update해주었다.

`void thread_exit(void *retval)` **(proc.c)**

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

  // 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);
    curproc->retval=retval;

  // 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 = curproc->master;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

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

기존 exit 함수를 그대로 사용하되 retval값을 저장해주고 p→parent가 initproc이 아니라 master thread로 설정했다. 구동은 디자인 단계에서 분석한 방식 그대로 진행된다.

`int thread_join(thread_t thread, void **retval)` **(proc.c)**

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

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->tid !=thread)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
          if (retval != NULL)
         *retval = p->retval;
        // Found one.
        kfree(p->kstack);
        p->kstack = 0;
        p->pid = 0;
        p->tid = 0;
        p->master = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        release(&ptable.lock);
        return 0;
      }
```

```
    }

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

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
  release(&ptable.lock);
}
```

기존의 wait 함수를 그대로 사용하되 종료된 자식 process, 지정한 thread가 있다면 return value를 저장하는 과정, tid, master 변수 초기화 과정을 추가하였다. 구동은 디자인 단계에서 분석한 방식 그대로 진행된다.

기존의 wait 함수에서는 freevm을 해주었는데 thread의 경우 모든 thread가 pgdit을 공유하기 때문에 free를 하면 문제가 생길 수 있어, 해당 코드만 제거해주었다.

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

```
int exec(char *path, char **argv)
{
    //기존 함수와 동일. 보고서 길이상 생략합니다.
  //만약 exec을 실행한 것이 thread라면,
 acquire(&ptable.lock);

  if(curproc->tid!=0){
    curproc->parent = curproc->master->parent;
    curproc->master = 0;
    curproc->tid = 0;
  }
  for(p = ptable.proc; p < &ptable.proc[NPROC]; ++p) {
    if(p->pid == curproc->pid && p != curproc) {
      for(int fd = 0; fd < NOFILE; fd++){
        if(p->ofile[fd]){
          fileclose(p->ofile[fd]);
          p->ofile[fd] = 0;
        }
      }
      kfree(p->kstack);
      p->kstack = 0;
      p->pid = 0;
      p->tid = 0;
      p->master = 0;
      p->parent = 0;
      p->name[0] = 0;
      p->killed = 0;
      p->state = UNUSED;
    }
  }
```

```
  release(&ptable.lock);

}
```

원래 구조는 자식 process가 exit으로 종료, wait에서 기다리던 부모가 자식 자원 해제이기 때문에 먼저 exit에서 필요한 로직을 추가하고 그 후에 wait의 로직을 추가해주었다.

exec의 경우 exec을 호출한 worker thread만 남기고 모든 thread를 종료하고, exec을 호출한 thread가 process가 되기 때문에 기존 exit 로직을 조금 수정해야 했다.

원래 자식 process를 찾아서 부모를 initproc으로 바꿔주는데, exec에서는 본인의 부모를 master thread의 부모로 변경하고 본인이 process가 되기 때문에 master, tid를 0으로 초기화주어야 한다. 그 후 열었던 파일을 다 닫아주는 로직을 그대로 추가하면 된다.

wait 함수에서 자원을 해제해줄 때는 ptable을 돌면서 zombie인 process를 찾아 해제해주었는데, exec의 경우 같은 process 내의 모든 thread를 제거해주어야 해서 ptable을 돌 때 조건문만 수정해주었다.

`void exit(void)`

```
void exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;
 //기존 함수와 동일. 보고서 길이상 생략합니다.

 if(curproc->tid!=0) curproc->parent = curproc->master->parent;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; ++p) {
    if(p->pid == curproc->pid && p != curproc) {
      kfree(p->kstack);
      p->kstack = 0;
      p->pid = 0;
      p->parent = 0;
      p->name[0] = 0;
      p->killed = 0;
      p->state = UNUSED;
      p->master = 0;
      p->tid = 0;
    }
  }
  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  wakeup1(curproc->parent);
  sched();
  panic("zombie exit");
}
```

기존 exit 함수에서는 파일, 디렉토리 정보 정리 후 본인을 부모로 하는 process를 찾아서 부모를 initproc으로 바꿔주고 만약 그 process의 상태가 zombie라면 initproc을 깨워주었다.

(자식을 처리하고 가지 않으면 zombie가 발생할 수 있기 때문에 자식이 있다면 자식을 수확할 수 있도록 부모를 변경해주고 만약 이미 자식이 종료됐다면 바꿔준 부모를 깨워서 바로 수확을 해준다.)

여기까지 수행한 후 exec을 수행하고 있는 주체가 thread인지 확인한다. 만약 thread라면 이 process의 모든 thread가 종료되기 때문에 parent를 master thread의 parent로 변경해주어야 한다.

그 후에는 wait에서 해주는 자원 해제 로직만 추가해주면 된다.

`int growproc(int n)`

```
int growproc(int n)
{
  uint sz;
  struct proc *curproc = myproc();
  acquire(&ptable.lock);
  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;
  switchuvm(curproc);
  for(struct proc* p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->pid == curproc->pid)
      p->sz = curproc->sz;
  }
  release(&ptable.lock);
  return 0;
}
```

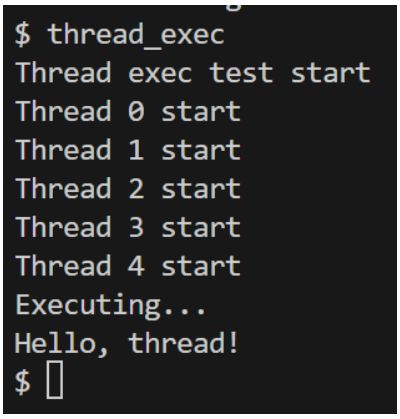
기존 함수와 동일하고 lock 과정만 추가해주었다.

+처음에 추가로 메모리를 할당 받은 후 모든 thread의 sz를 updata해주지 않아 문제가 발생해, 할당 이후 sz를 updata해주는 logic도 추가했다.

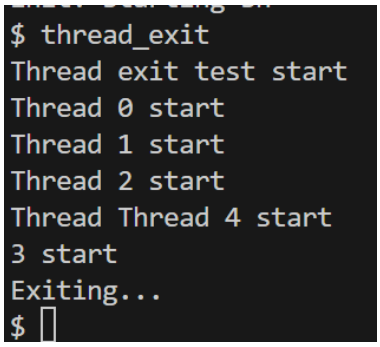
Result

```
$ 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 startChild of thread 2 start
Child of thread 3 star
Child of thread 0 start
Child of thread 1 start
t
Child of thread 4 start
Child of thread 2 end
Thread 2 end
Child of thread 0 end
Child of thread 1 end
Child of thread 3 end
Child Thread 0 end
Thread 1 end
of thread 4 end
Thread 3 end
Thread 4 end
Test 2 passed

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

All tests passed!
$
```

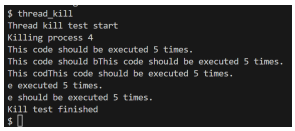
thread_test 실행 후 test 1, 2, 3 모두 통과하였다.



thread_exec 실행 후 hello, thread!를 출력하고 shell로 스케줄링된 것을 확인할 수 있다.



thread_exit 실행 후 exiting 출력 후 shell로 바로 빠져나온 모습을 볼 수 있다.



thread_kill test도 통과하였다.

Trouble shooting

처음엔 프로세스 1, 프로세스 2에서 각각 thread를 만들면 이 두 thread의 tid는 둘 다 1이 되어야 한다고 생각했다.

그러나 wait 함수를 작성할 때 ptable에서 tid가 thread인 process를 찾아서 상태 확인을 해주어야 하므로 thread의 id는 프로세스가 달라도 시스템 전체에서 고유해야 한다는 사실을 알 수 있었다.

exec.c에서 ptable에 접근하는데 ptable에 대해 선언이 안 되어 있다는 오류가 떴다

특정 헤더파일을 추가하지 않아서 생기는 오류라고 생각했으나 파일 상단에 extern struct PTABLE ptable을 선언해주니 해결되었다.

make: * 'fs.img'에서 필요한 '_thread_test' 타겟을 만들 규칙이 없습니다. 멈춤.

make fs.img 명령어 실행 시 계속 이런 error가 발생해서 makefile을 계속 수정해보았는데 p3_test 파일을 폴더에 넣은 상태로 make 하려고 해서 발생한 문제였다. 폴더에서 파일을 다 빼서 xv6 폴더 안에 넣고 다시 make해보니 문제가 해결되었다.

kernel에서 thread_create 함수를 만들기 위한 규칙을 찾을 수 없다는 에러

thread 관련 함수를 따로 파일로 만들지 않고 proc.c 파일 안에 선언했는데 Makefile의 OBJS에 thread_create.o 파일을 추가해서 생긴 문제였다. proc.o는 이미 선언되어 있었기 때문에 OBJS 수정 없이 make하니 해결되었다.

잘못된 주소 접근 error

test 1에서 user stack, page table, kernel stack 모두 잘 만들어진 것 같은데 스케줄링 과정에서 계속 잘못된 주소 접근 trap이 발생해서 어셈블리 코드 파일을 만들어서 살펴보았다. 매번 같은 명령어에서 trap이 발생하는 게 아니라서 정확한 원인은 파악하지 못 했으나 thread_create에서 master thread를 찾는 logic이 잘못되어, master thread 기반으로 만들어진 것들에서 오류가 생기는 것이었다. master thread 찾는 방식을 수정하였더니 에러가 해결되었다.

thread_exit test 수행 중 오류

exec 함수에서 hello 출력 후 exit 함수를 호출하면 정상 종료 후 shell로 나오는 것을 통해 exit 함수의 구조가 올바르다고 생각하였으나 thread_exit 소스 코드를 실행하니 스케줄러에서 무한 루프가 발생하였다. exit 함수에서 스케줄러로 들어가기 전에 ptable에 있는 모든 process의 상태를 출력해보았다.

```
Hsched: scheduling process 3
ello, thread!
sched: scheduling process 3
exit: process 3 exiting
sleeping: 1
runnable: 2
running: 3
exit: setting process 3 to ZOMBIE state
sched: scheduling process 3
sched: scheduling process 2
$ sched: scheduling process 2
```

exec의 경우 2번 process 즉 shell이 runnable한 상태였기 때문에 스케줄러에서 shell이 실행되었지만

```
Exiting...
exit: process 3 exiting
sleeping: 1
sleeping: 2
running: 3
exit: setting process 3 to ZOMBIE state
sched: scheduling process 3
```

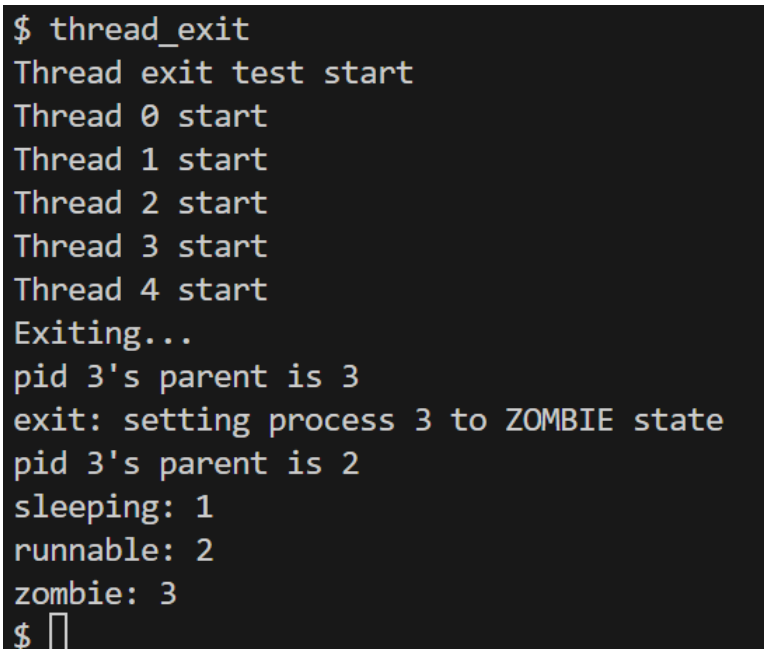
exit의 경우 2번 process가 sleeping상태여서 스케줄러에서 무한 루프가 걸리는 것이었다.

pid 3 process는 2번 process가 fork를 통해 만든 자식 process이므로 3번의 종료를 기다리고 있는 상태다. 즉 exit 함수에서 부모를 wakeup하는 함수가 제대로 작동이 되지 않았다는 의미이다.

디버깅을 위해 자원 정리 전 후로 pid 3의 부모 pid를 출력해보았다.

```
Exiting...
pid 3's parent is 3
sleeping: 1
sleeping: 2
running: 3
exit: setting process 3 to ZOMBIE state
pid 3's parent is 2
```

자원 정리 후엔 부모가 2로 잘 나와 있지만 자원 정리 전에는 부모가 3인 상태여서 wakeup이 되지 않아, 생기는 문제였다.



자원 정리 후, parent를 wakeup해서 runnable한 상태로 변경해주니 해결되었다.

Locking-wiki

과제 목표: 멀티 스레딩 환경에서 race condition을 방지하기 위해 동기화 구현하기. 단, c 라이브러리에서 제공하는 pthread_mutex와 같은 API 사용 없이 lock과 unlock 함수를 구현하여야 한다.

Design

우선 동기화를 위해 가장 많이 쓰는 aquire, release 함수를 먼저 살펴보았다.

```
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  //만약 이미 lock을 잡고 있다면 error
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
```

```
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

cscope를 통해 xchg 함수도 살펴보았다.

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
                              // addr, newval swap
               "+m" (*addr), "=a" (result) :
               // addr 값을 읽고 쓸 수 있고 result 변수를 eax에 저장
               "1" (newval) :
               //newval을 두 번째 피연산자로 사용
               "cc");
  return result;
}
```

우선 inline함수로 선언되어 있는 것을 보고 그 이유를 찾아보았다.

inline 함수로 정의할 경우 함수 호출 시 overhead를 줄일 수 있다고 한다. 한 번에 한 process만 호출할 수 있기 때문에 다른 process의 대기 시간을 조금이라도 줄이기 위함이라고 생각했다.

addr이 volatile type인 이유는 컴파일러가 이 변수를 최적화하지 않게 하기 위함이라고 한다.

body는 어셈블리 코드로 구현되어 있다. lock은 xchgl(두 피연산자의 값을 swap한다) 명령어를 atomic하게 수행할 수 있게 해준다.

+m: 읽기 쓰기가 가능한 메모리

=a: eax에 결과값 저장

cc: 조건 코드

어셈블리 코드 분석은 주석을 통해 작성하였다.

즉 xchg 함수는 두 값을 원자적으로 교환하는 기능을 수행한다.

수업시간에 배웠던 하드웨어 atomic swap과 유사한 것 같다.

xchg을 통해 lock과 1을 swap한 후 swap 전의 lock 값을 받아온다.

그 값이 1이라면 while문에서 계속 대기한다. (다른 process가 이미 lock을 잡고 있는 상황)

그렇지 않다면 while을 탈출해, 다음 명령어를 실행한다.

__sync_synchroniz()은 gcc 함수로, 컴파일러에서 제공하는 atomic 함수이다.

이 함수의 역할은 메모리 참조가 lock이 획득되어야 수행할 수 있도록 보장하는 것이다.

```
void
__sync_synchronize (void)
{
#if defined(arm revisions supporting dmb)
  asm volatile("dmb" : : : "memory");
```

```
#else
  asm volatile("" : : : "memory");
#endif
}
```

lock을 획득한 cpu를 기록한 후

getcallerpcs 함수를 호출한다.

```
// Record the current call stack in pcs[] by following the %ebp chain.
void
getcallerpcs(void *v, uint pcs[])
{
  uint *ebp;
  int i;

  ebp = (uint*)v - 2;
  for(i = 0; i < 10; i++){
    if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
      break;
    pcs[i] = ebp[1];     // saved %eip
    ebp = (uint*)ebp[0]; // saved %ebp
  }
  for(; i < 10; i++)
    pcs[i] = 0;
}
```

이 함수에서는 현재 call한 stack의 pc들을 기록한다.

aquire에서 중요한 과정은 interrupt 무시→ lock 중복 방지 → atomic한 xchg을 통해 swap → lock 획득 가능할 때까지 while문에서 대기이다.

```
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other cores before the lock is released.
  // Both the C compiler and the hardware may re-order loads and
  // stores; __sync_synchronize() tells them both not to.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code can't use a C assignment, since it might
```

```
  // not be atomic. A real OS would use C atomics here.
  asm volatile("movl $0, %0" : "+m" (lk->locked) : );

  popcli();
}
```

acquire과 같이 release 중복을 방지한 후 acquire에서 기록한 pc와 cpu 기록을 초기화한다.

__sync_synchronize()를 통해 메모리 접근이 lock이 해제되기 전에 완료되도록 보장한다.

어셈블리 코드로 atomic하게 lock을 0으로 바꿔서 해제해준다.

그 후 acquire에서 비활성화했던 interrupt를 다시 활성화한다.

acquire, release 방식을 그대로 사용하면 될 것이라 생각했다.

lk 사용을 위해 spinlock 구조체를 확인해보았다.

```
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

lock을 잡고 있는지만 확인하면 되기 때문에 uint locked만 사용하였다.

lock에서는 기존 이 thread가 lock을 잡은 상태에서 또 lock을 잡으려고 하는지 확인하고 그게 아니라면 xchg로 lock을 잡을 수 있을 때까지 busy waiting 하면서 기다리다가 잡을 수 있으면 잡도록 구현할 계획이다.

기존 함수에서는 holding으로 lock 중복을 확인하였다. holding 대신 thread id를 통해서 lock의 주인이 누구인지 작성해두려고 한다. pthread_create에서 tid를 넘겨주고 있으므로 lock 함수에 lock과 함께 본인의 tid까지 넘겨주면 중복 확인을 가능할것이다. 마찬가지로 unlock에서는 lock 해제후 lock의 주인을 초기화해주어야 한다. 다른 thread의 id가 아니라 아예 초기화를 해주어야 swap turn 문제가 발생하지 않을 것이다.

ex) tid 0이 lock (owner=0) → tid 1이 lock (대기) → tid 0이 unlock (owner =-1) → tid 1이 lock(owner =1) → tid 1이 unlock (owner=-1) →tid 1은 끝나서 종료. → tid 0이 lock을 요청해도 owner은 -1이라서 swap turn 문제 안 생김!

Implement

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
int shared_resource = 0;

#define NUM_ITERS 500
#define NUM_THREADS 100
int lk=0;
int owner=-1;
```

```
void lock(int *lk, int tid);
void unlock(int *lk, int tid);

static inline int xchg(volatile int *addr, int newval)
{
  int result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}

void lock(int *lk, int tid)
{
    if(owner==tid){
        printf("panic: acquire. tid:%d\n",tid);
        exit(EXIT_FAILURE);
    }
    while(xchg(lk, 1) != 0)
        ;
    owner=tid;
}

void unlock(int *lk, int tid)
{
    if (owner!= tid) {
        printf("panic: release lock from another thread tid:%d\n",tid);
        exit(EXIT_FAILURE);
    }
    if(owner==-1){
        printf("panic: release tid:%d\n",tid);
        exit(EXIT_FAILURE);
    }
    owner = -1;
    *lk=0;

}

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

    lock(&lk,tid);
    lock(&lk,tid);
        for(int i = 0; i < NUM_ITERS; i++)    shared_resource++;

    unlock(&lk,tid);
    pthread_exit(NULL);
}
```

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

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

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

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

    return 0;
}
```

uint를 지원하지 않아서 별도의 헤더파일을 추가해야 해서 그냥 int로 선언해주었다.

⇒ unsigned int로 선언하면 된다는 것을 추후에 알았으나 lk이 음수가 될 일도 없고 int 범위 밖으로 나갈 일도 없을 것 같아서 그냥 int로 두었다.

acquire 함수를 위해 xchg 함수를 추가해주었다. lock의 매개변수로 lock과 현재 thread의 id를 넘겨준다.

원래 lock은 spinlock 구조체에 있는 변수인데, 처음엔 spinlock 구조체를 그대로 사용하고 member 변수가 하나인 상태로 사용하였으나 그냥 int로 선언해도 문제 없을 거라 판단하여 int로 선언하였다.

lock에서는 현재 lock을 갖고 있는 상태에서 또 잡으려고 하는 지 확인하고 그렇다면 에러 문구 출력 후 강제 종료를 시켜주었다. xchg 함수를 통해 atomic 하게 lock을 잡을 수 있는지 확인하고 (기존 lock값 반환) 잡을 수 있을 때까지 대기한다.

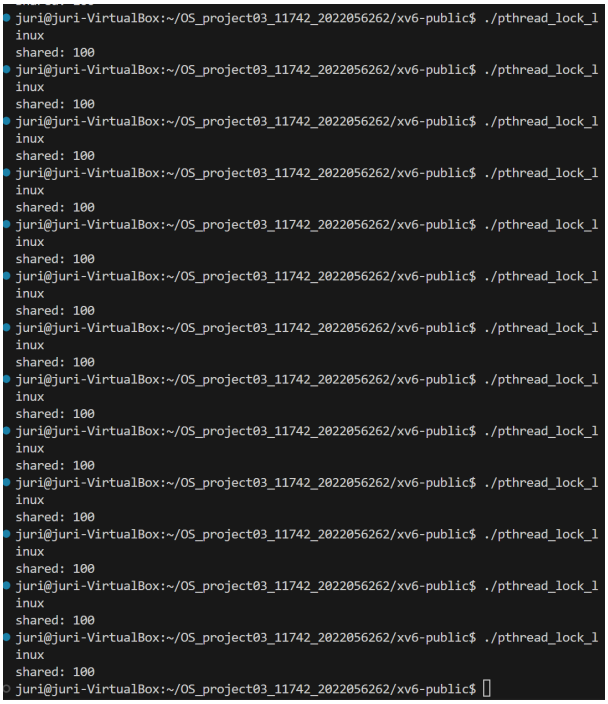
잡았다면 lock의 주인을 현재 thread로 설정해서 이 thread가 다시 lock을 잡을 때 에러가 뜨도록 만들어주었다.

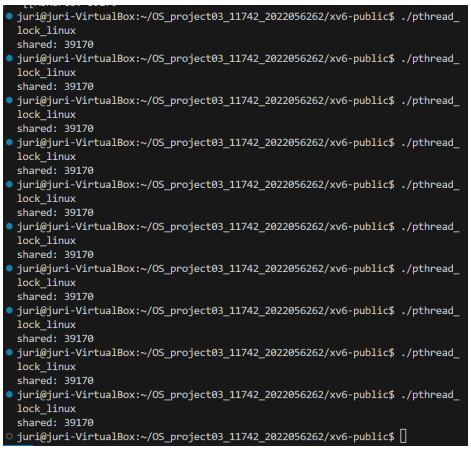
unlock에서도 lock과 tid를 받아서 중복 release를 하지 않는 지 확인하였다. release를 이미 수행해서 owner가 -1이 되었다면 panic: release 출력, lock을 잡고 있지 않은 thread가 release 수행 시에도 에러 문구를 출력해주었다. exit 함수 사용을 위해 #include <stdlib.h> 헤더파일을 추가해주었다.

Result

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

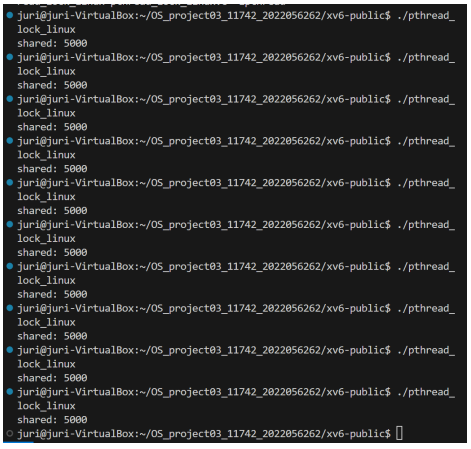
```
#define NUM_ITERS 10
#define NUM_THREADS 3917
```

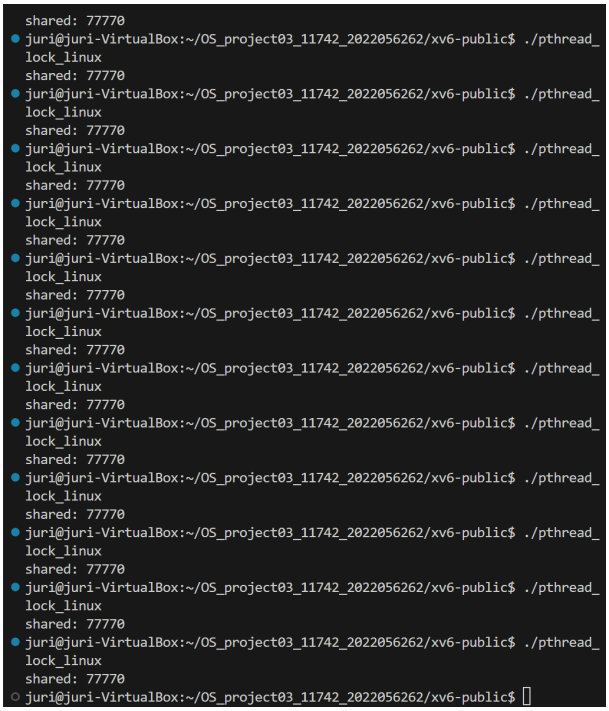





```
#define NUM_ITERS 500
#define NUM_THREADS 10
```

```
#define NUM_ITERS 7777
#define NUM_THREADS 10
```





```
#define NUM_ITERS 500
#define NUM_THREADS 700
```

```
lock_linux
shared: 350000
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 350000
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 350000
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 350000
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 350000
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 350000
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 350000
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 350000
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 350000
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 350000
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 350000
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 350000
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$
```

```
#define NUM_ITERS 500
#define NUM_THREADS 3917
```

```
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 1958500
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 1958500
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 1958500
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 1958500
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 1958500
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 1958500
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 1958500
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 1958500
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 1958500
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 1958500
```

반복 횟수가 많고 thread 수가 적을 때, 반복 횟수가 적고 thread 수가 많을 때, 반복 횟수가 많고 thread 수가 많을 때 모두 race condition이 발생하지 않는 것을 볼 수 있다.

lock, unlock 때문에 race condition이 발생하지 않는 것인지 정확히 확인하기 위해, lock과 unlock을 호출하지 않은 코드도 실행시켜보았다.

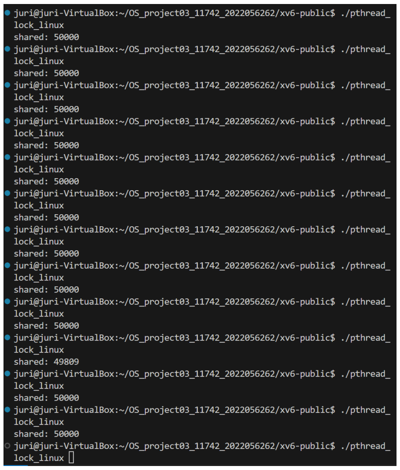
```
#define NUM_ITERS 500
#define NUM_THREADS 100
```

```
#define NUM_ITERS 10
#define NUM_THREADS 3917
```

```
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 39170
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 39160
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 39170
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 39160
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 39120
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 39170
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$
```

```
#define NUM_ITERS 500
#define NUM_THREADS 3917
```

```
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ gcc -o pth
read_lock_linux pthread_lock_linux.c -lpthread
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 1953805
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 1954500
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 1937933
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 1956339
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$ ./pthread_
lock_linux
shared: 1958173
juri@juri-VirtualBox:~/OS_project03_11742_2022056262/xv6-public$
```

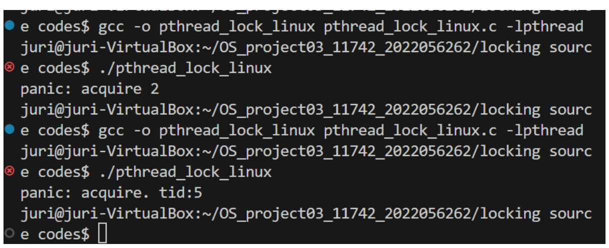


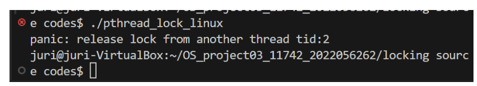
50000으로 race condition이 발생하지 않은 값이 출력되길래 lock이 없어도 race condition이 일어나지 않는 게 보장되는 건가 생각했었다.

그러나 race condition은 항상 발생하는 게 아니라 아주 우연히 절묘하게 타이밍이 맞을 때만 발생하는 것이기 때문에 thread 갯수와 반복 횟수를 크게 해서 확인해보았다.

그 결과 가끔씩 race condition이 발생해, shared 값에 이상이 생긴 것을 확인할 수 있었다.

또한, thread의 갯수가 크게 늘어날 수록 race condition 확률이 증가하는 것을 알 수 있었다.





lock 잡은 뒤 한 번 더 lock 호출, unlock이후에 한 번 더 unlock 호출 두 경우 모두 올바르게 에러 메세지 출력 후 종료되었다.

Trouble shooting

In file included from pthread_lock_linux.c:1:
/usr/include/stdio.h:27:10: fatal error: bits/libc-header-start.h: 그런 파일이나 디렉터리가 없습니다
27 | #include <bits/libc-header-start.h>
| ^~~~~~~~~~~~~~~~~~~~~~~~~~
compilation terminated.
make: * [<내장>: pthread_lock_linux.o] 오류 1**

sudo apt-get install gcc-multilib g++-multilib

이 명령어로 해당 오류는 없앨 수 있었다.

그런데 이 파일은 make가 아니라 gcc 명령어로 컴파일해야 하므로 필요 없는 과정임을 추후에 알게 되었다.

pthread_lock_linux.c:36:5: error: invalid lvalue in asm output 0
36 | asm volatile("movl $0, %0" : "+m" (lk->locked) :);

| ^~~

pthread_lock_linux.c:36:5: error: memory input 1 is not directly addressable

이 방식 대신 atomic하게 lock을 해제할 수 있는 방식을 찾다가 __sync_synchroniz()같은 gcc 내장 함수에 atomic하게 lock을 해제하는 함수가 있는 것을 알게 되었다.

`__sync_lock_release(int *lk)`

다만 이 함수가 동기화 api이기 때문에 사용하지 못 해서 그냥 *lock=0;으로 lock을 해제해주었다.