OS Project02 Wiki

Author: 2020021949 구건모

이 프로젝트는 운영체제(ELE3021-11742)과목 Project03에 관한 문서입니다.
이 문서는 다음 네 가지 정보를 담고 있습니다.

- Desin
- Implement
- Result
- Trouble shooting

1. Design

이 프로젝트에서는 다음 세 가지 feature를 구현하였습니다.

1. 과제 명세에 나온 Kernel Thread 구현
2. Thread를 관리하는 System Call 구현 (thread_create, thread_join 등)
3. 이를 테스트해볼 수 있는 유저 프로그램 작성

Kernel Thread

쓰레드는 Light Weight Process (LWP) 이기때문에 이를 구현하기 위해서는 `proc` 구조체를 적절히 수정하여 쓰레드를 프로세스처럼 관리하도록 구현하면 됩니다.

System Call for Handling Threads

쓰레드를 관리하기 위한 시스템 콜은 다음과 같습니다.

- `thread_create`: 새로운 쓰레드 생성 및 실행
- `thread_exit`: 쓰레드 종료 및 값 반환 (return value)
- `thread_join`: 쓰레드 종료 대기 및 반환 값 수신 및 자원 회수

How to implement?

먼저 쓰레드를 생성하고 관리하기 위해선 각 쓰레드 별 커널 스택과 유저 스택이 필요합니다. 따라서, 쓰레드를 생성할 때, 쓰레드 별로 스택을 할당해주어야 합니다. 이때, 주의할 점은 모든 쓰레드는 같은 주소 공간 내에서 할당되어야 합니다.

또한, sbrk 사용 시, 가상 주소 공간이 선형적으로 증감하기 때문에, 중복 주소 사용을 방지할 수 있어야 합니다.

그 외 내용은 프로세스와 동일하게 작동하기 때문에, 프로세스처럼 `proc` 구조체를 수정하여 쓰레드를 관리하면 됩니다.

만약, `proc` 구조체를 적절히 변형하지 않고, 새로운 Thread 구조체로 관리한다면, 쓰레드와 프로세스에서 공통적으로 사용하는 기능들 (대부분이 프로세스와 유사합니다.)을 모두 수정해야 합니다.

예를 들어, Trap Frame을 관리하는 코드에서는 `myproc()->tf` 처럼 사용되는 부분을 `(myproc()->is_thread ? myproc()->thread->tf : myproc()->tf)` 처럼 수정해야 합니다. 하지만, 이는 xv6의 상당 부분을 수정해야 하고, 코드의 가독성이 떨어지게 됩니다. 따라서, `proc` 구조체를 적절히 변형하여 쓰레드를 관리하는 것이 좋습니다.

Thread 관련 코드를 작성했다면 다음과 같이 유저 테스트 코드를 작성할 수 있습니다. (시스템 콜을 등록하는 부분은 Project01에 나와있으므로 생략했습니다.)

1. 여러 쓰레드를 생성하고 각 쓰레드가 특정 일을 하도록 합니다.
2. 해당 프로세스가 공유 변수를 잘 사용하는지, thread_exit, thread_join으로 종료 및 반환 및 자원 회수가 잘 이루어지는지 확인합니다.
3. 이후 kill, sbrk 등 여러 상황을 만들어 쓰레드가 잘 동작하는지 확인합니다.

2. Implement

쓰레드를 구현하기 이전 기존의 프로세스를 생성하고 변경하는 (fork, exec) 코드를 분석해보겠습니다.

(Origin) fork

fork 함수는 다음과 같이 구현되어 있습니다.

```
//FILE: proc.c

//PAGEBREAK: 32
// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and initialize
// state required to run in the kernel.
// Otherwise return 0.
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;

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

  release(&ptable.lock);

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
```

```
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

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

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

  return p;
}

// Create a new process copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

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

  // 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);

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

  pid = np->pid;
```

```
  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

먼저 `allocproc`으로 새로운 프로세스 공간을 할당받은 뒤, `trap frame`, `return address (trapret)`, `context` 순으로 커널 스택을 사용합니다. 이때, context->eip는 forkret으로 설정합니다.

이런 call stack을 구성한 이유는, 해당 프로세스가 처음 실행될 때, forkret으로 들어가서 `ptable.lock`을 해제하고 (forkret 함수의 역할), 이후 forkret이 종료되면 return address인 trapret으로 돌아가서 trap frame을 보고, 유저 코드가 실행되어야 하는 위치로 context switching을 하기 위함입니다.

call stack을 구성한 뒤, `copyuvm` 함수를 통해 새로운 프로세스의 페이지 테이블을 복사하고 (별도의 주소 공간을 갖게 됨), `np->tf`에 현재 프로세스의 trap frame을 복사합니다. 이후, `np->tf->eax`를 0으로 설정하고 (자식의 fork 반환 값은 0이기 때문), `np->ofile`, `np->cwd`, `np->name`을 복사합니다.

이후, 프로세스의 상태를 RUNNABLE로 설정해서, 실제 프로세스가 실행될 수 있도록 합니다.

쓰레드를 생성하는데 필요한 대부분 단계를 `fork`와 `allocproc` 함수에서 알 수 있습니다.

다만, `fork`에서는 새로운 프로세스를 생성할 때, `np->pgdir`을 복사하는 `copyuvm`을 사용하는데, 쓰레드를 생성할 때는 같은 주소 공간을 사용하면서 새로운 페이지를 할당받아야 하기 때문에, exec를 보며 관련 코드를 분석해 보겠습니다.

(Origin) exec

exec 함수는 다음과 같이 구현되어 있습니다.

```
//FILE: exec.c

int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off;
  uint argc, sz, sp, ustack[3+MAXARG+1];
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pde_t *pgdir, *oldpgdir;
  struct proc *curproc = myproc();

  begin_op();

  if((ip = namei(path)) == 0){
    end_op();
    cprintf("exec: fail\n");
    return -1;
```

```
  }
  ilock(ip);
  pgdir = 0;

  // Check ELF header
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;

  if((pgdir = setupkvm()) == 0)
    goto bad;

  // Load program into memory.
  sz = 0;
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
      goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
      goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
    if(ph.vaddr % PGSIZE != 0)
      goto bad;
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;
  }
  iunlockput(ip);
  end_op();
  ip = 0;

  // 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;

   bad:
    if(pgdir)
      freevm(pgdir);
    if(ip){
      iunlockput(ip);
      end_op();
    }
    return -1;
  }
```

exec 함수는 다음과 같이 동작합니다.

1. ELF 파일 시스템을 읽어들여 실행할 실행 파일을 불러옵니다.
2. 실행 파일에 사용되는 프로그램 코드를 메모리에 로드합니다. (이때 새로운 page table을 할당받아 사용합니다. 기존 page table은 free됩니다.)
3. 새로운 실행 파일이 실행 될 때, 사용될 유저 스택을 할당받습니다.
4. 이때 두 페이지를 할당받은 뒤, 첫 번째 페이지는 접근 불가능하게 만들고, 두 번째 페이지를 유저 스택으로 사용합니다. (guard page - stack overflow 발생 시, page fault를 발생시키기 위함)
5. 유저 스택에 argument를 push하고 (변경된 main 함수에서 사용), trap frame 등을 설정합니다.
6. 이후 변경된 프로세스로 돌아갑니다.

여기서 집중해야 할 파트는, 유저 스택을 사용하기 위해 새로운 페이지를 할당받는 부분입니다.

```
    sz = PGROUNDUP(sz);
    if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
      goto bad;
    clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
```

`allocuvm` 함수를 사용해서 유저 스택을 할당받을 수 있습니다. 이후 `clearpteu` 함수를 사용해서 첫 번째 페이지를 guard page로 만들어 줍니다.

이렇게 쓰레드를 생성할 준비는 끝났습니다.

(New) struct proc

쓰레드를 생성하기 위해서는 `proc` 구조체를 적절히 수정해야 합니다.

```
//FILE: 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)
  uint tid;                    // thread ID If 0, then it is a process
  struct proc* tparent;        // Parent process of thread
  void* retval;                // Return value of thread
};
```

`tid`는 쓰레드의 ID를 나타내며, 0이면 프로세스를 나타냅니다. 0이 아닌 수가 존재한다면, 해당 프로세스는 쓰레드로 구분됩니다. 또한, 쓰레드를 구분하기 위해서도 사용됩니다.

`tparent`는 쓰레드의 부모 프로세스를 나타냅니다. 어떤 프로세스가 쓰레드를 생성했는지 기록하기 위해 사용됩니다.

`retval`는 쓰레드가 종료될 때, 반환할 값을 저장합니다. 이는 `thread_exit` 함수를 통해 저장됩니다. 이후 `thread_join` 함수를 통해 반환 값을 받을 수 있습니다.

그 외엔 프로세스와 동일하게 사용합니다.

(New) thread_t

`thread_t` 타입은 `uint` 타입을 typedef한 것입니다. thread_create에서 할당 받은 쓰레드의 번호를 저장하고, thread_join에서 어떤 쓰레드를 기다릴지 결정하기 위해 사용됩니다.

(New) thread_create

쓰레드를 생성하기 위해 다음 함수를 구현했습니다.

```
//FILE: proc.c

// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and initialize
// state required to run in the kernel.
// Otherwise return 0.
static struct proc*
allocthread(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);

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

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->retval = 0;
  p->pid = -1;
  p->tid = nexttid++;

  release(&ptable.lock);

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

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

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

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

  return p;
```

```
}

// Allocation User Stack
static int
alloctstack(struct proc *p)
{
  acquire(&ptable.lock);
  p->tparent->sz = PGROUNDUP(p->tparent->sz);
  if ((p->tparent->sz = allocuvm(p->tparent->pgdir, p->tparent->sz, p-
>tparent->sz + 2 * PGSIZE)) == 0) {
    release(&ptable.lock);
    return -1;
  }
  clearpteu(p->tparent->pgdir, (char*)(p->tparent->sz - 2 * PGSIZE));

  p->sz = p->tparent->sz;
  release(&ptable.lock);

  return 0;
}

// Create a new thread copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
int
thread_create(thread_t* thread, void* (*start_routine)(void*), void* arg)
{
  int i, sp;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate thread.
  if((np = allocthread()) == 0){
    return -1;
  }

  // Copy process state from proc.
  np->pgdir = curproc->pgdir;
  np->tparent = curproc->tid ? curproc->tparent : curproc;

  if(alloctstack(np) < 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }

  acquire(&ptable.lock);

  np->parent = curproc->parent;
  np->pid = curproc->pid;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
```

```
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

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

  sp = np->sz;
  sp -= sizeof(uint);
  *(uint*)sp = (uint)arg;
  sp -= sizeof(uint);
  *(uint*)sp = (uint)0xFFFFFFFF; // fake return PC
  *np->tf = *curproc->tf;
  np->tf->esp = sp;
  np->tf->eip = (uint)start_routine;

  *thread = np->tid;
  np->state = RUNNABLE;

  release(&ptable.lock);

  return 0;
}
// When creating a new thread & scheduling it for the first time
// we need to release the ptable.lock
void
threadret(void) {
  release(&ptable.lock);

  // return to caller, actually trapret (see allocthread)
}
```

`allocthread`는 ptable에 새로운 쓰레드를 할당받고, 커널 스택을 할당받는 함수입니다. 이후 커널 스택에 call stack을 구성하고, 쓰레드를 생성할 준비를 합니다. `allocproc`과 거의 유사합니다. 다만, `pid` 대신 `tid`를 할당하고, `forkret`이 아닌 `threadret`으로 설정합니다.

`threadret`은 `forkret`과 유사하게 동작하며, 처음 쓰레드가 실행될 때, `ptable.lock`을 해제하기 위해 사용됩니다.

`alloctstack`은 쓰레드를 생성할 때, 유저 스택을 할당받는 함수입니다. 이때, 쓰레드는 부모 프로세스의 주소 공간을 이용하여 유저 스택을 할당받습니다.

할당받은 유저 스택에 argument를 push하고, return address를 설정합니다. 다만, 쓰레드는 return하지 않기 때문에 디버깅에 용이한 값을 넣어줍니다. (0xFFFFFFFF를 넣어주면 실제 return 발생 시 page fault를 발생시키게 되므로, 의도치 않은 다른 코드가 실행되는 것을 막아줍니다.)

이후 필요한 프로세스 정보들을 복사한 뒤, trap frame을 조작하여 넘겨받은 함수를 실행할 수 있도록 합니다.

> 새로운 스택을 할당받기 위해서 `tparent->sz`를 사용하는 이유는, 메모리 주소 관리를 무조건 쓰레드의 부모가 하기로 결정했기 때문입니다. 추후 sbrk에서도 쓰레드의 부모의 sz를 통해 메모리 주소를 관리할 것입니다. 만약 사용하고 있는 최댓값인 sz를 알 수 없다면, 이미 할당받은 주소 공간을 다시 할당받게 되는 문제가 있습니다. 따라서, 쓰레드의 부모만 최대 sz를 관리하도록 결정했습니다.

(New) thread_exit

쓰레드를 종료하기 위해 다음 함수를 구현했습니다.

```
//FILE: proc.c

// exit the current thread
void thread_exit(void* retval) {
  struct proc *curproc = myproc();
  int fd;

  if(curproc->tid == 0)
    panic("non-thread exiting");

  // 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);

  // Thread parent might be sleeping in thread_wait().
  wakeup1(curproc->tparent);

  curproc->retval = retval;

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

이 함수에서는 공유자원이 아닌 쓰레드 자신의 자원을 해제하고, 부모 프로세스가 기다리고 있다면 깨웁니다. 이후, 반환 값을 저장하고, `ZOMBIE` 상태로 변경한 뒤, 스케쥴러를 호출합니다. `exit` 함수와 거의 유사합니다.

(New) thread_join

쓰레드가 종료될 때까지 기다리기 위해 다음 함수를 구현했습니다.

```
//FILE: proc.c

// wait for a thread to thread_exit and return
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 threads.
      havekids = 0;
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->tparent != curproc || p->tid != thread)
          continue;
        havekids = 1;
        if(p->state == ZOMBIE){
          // Found one.
          *retval = p->retval;
          kfree(p->kstack);
          p->kstack = 0;
          p->pid = 0;
          p->tid = 0;
          p->parent = 0;
          p->tparent = 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 threads.
      if(!havekids || curproc->killed){
        release(&ptable.lock);
        return -1;
      }

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

이 함수에서는 쓰레드 ID가 thread인 쓰레드가 종료될 때까지 기다리며, 쓰레드가 종료되면 반환 값을 받아옵니다. 이후, 쓰레드의 자원을 해제하고, 반환 값을 반환합니다. `wait` 함수와 거의 유사합니다.

이렇게 위 세 함수를 구현하면, 기본적인 쓰레드 생성이 가능합니다.

하지만, 현재 쓰레드를 프로세스처럼 취급하고 사용하고 있기 때문에, 프로세으와 쓰레드와 차이가 생기는 부분들 (exit, wait, exec, sbrk 등)을 수정해야 합니다.

예시를 들면, 기존 프로세스에서 exit을 호출하면, 해당 프로세스만 종료됐지만, 현재는 같은 프로세스의 모든 쓰레드가 종료되어야 합니다. 따라서 차이가 생기는 위 부분들을 수정해야 합니다.

또한, 기존엔 멀티쓰레딩을 지원하지 않았기 때문에, race condition에 대한 고려가 필요하지 않았습니다.

하지만, 쓰레드가 생김에 따라 race condition이 발생할 수 있습니다. (sbrk 등) 따라서 이 문제도 해결해야 합니다.

(New) fork

fork는 다음과 같이 변경됐습니다.

```
//FILE: proc.c

// Create a new process copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
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;
    release(&ptable.lock);
    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);

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

  pid = np->pid;

  np->state = RUNNABLE;

  release(&ptable.lock);
```

```
  return pid;
}
```

ptable.lock을 잡는 위치가 변경됐습니다.

이렇게 변경한 이유는 다음 문제를 해결하기 위함입니다.

1. curproc->sz를 읽어와서 `copyuvm`을 시작함
2. context-switching이 발생해서 다른 쓰레드에서 sbrk를 호출함 (curproc->sz가 커짐)
3. context-switching이 발생해서 기존 `fork`를 진행하던 쓰레드로 돌아옴
4. copyuvm이 끝나고, np->sz = curproc->sz를 실행함
5. 이후 해당 프로세스에서 다시 `fork`를 호출하거나 `mappages`처럼 모든 페이지를 접근해야 하는 일이 생긴다면, 생성하지 않은 페이지에 대해 접근해 복사를 시도할 수 있습니다. 혹은 2번 과정에서 curproc->sz가 작아졌다면, 이미 생성된 `virtual address`에 다시 페이지를 할당하려는 시도를 할 수도 있습니다.

이를 방지하기 위해, copyuvm을 시작할 때, ptable.lock을 잡고, copyuvm이 끝난 뒤, ptable.lock을 해제하도록 변경했습니다.

(New) exit

exit는 다음과 같이 변경됐습니다.

```
//FILE: proc.c

// Exit the current process.  Does not return.
// An exited process remains in the zombie state
// until its parent calls wait() to find out it exited.
void
exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  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;
    }
  }

  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);
    }
  }

  // delete all thread of the process
  for (p = ptable.proc; p < &ptable.proc[NPROC]; ++p) {
    if (p != curproc && p->pid == curproc->pid) {
      kfree(p->kstack);
      p->kstack = 0;
      p->pid = 0;
      p->tid = 0;
      p->parent = 0;
      p->tparent = 0;
      p->name[0] = 0;
      p->killed = 0;
      p->state = UNUSED;
    }
  }

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  curproc->tid = 0;
  sched();
  panic("zombie exit");
}
```

쓰레드를 포함한 메인 프로세스이거나 쓰레드에서 `exit`을 호출하면, 모든 쓰레드가 종료되어야 합니다. 따라서 기존 `exit` 코드에서 자신이 아닌, 같은 pid를 가진 모든 쓰레드를 종료하도록 변경했습니다. 이때, 한 번에 모든 자원도 회수합니다.

> exit에서 curproc->tid = 0으로 바꿔주는 이유는, 종료된 프로세스를 부모로부터 reaping을 시키기 위함입니다. 현재 wait에서는 프로세스만 reaping하고있기 때문에, 만약 쓰레드에서 exit을 호출하면, 부모 프로세스가 쓰레드를 reaping하지 않습니다. 따라서, 쓰레드가 종료됐다고 해도, 프로세스로 인식되어야 합니다.

(New) wait

wait은 다음과 같이 변경됐습니다.

```
//FILE: proc.c

// Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
```

```
int
wait(void)
{
  struct proc *p;
  int havekids, pid;
  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->parent != curproc || p->tid != 0)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->tid = 0;
        p->parent = 0;
        p->tparent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        release(&ptable.lock);
        return pid;
      }
    }

    // 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
  }
}
```

`proc` 구조체에 새로 할당된 변수들을 초기화 하는 코드를 추가했습니다.

또, 종료된 `proc`에 포함되어 있지만, 쓰레드인 경우 자식으로 판단하지 않고, 무시했습니다.

> 이렇게 처리한 이유는, 쓰레드는 프로세스의 일부이기 때문에, 쓰레드가 종료됐다고 해서 프로세스가 종료된 것이 아니기 때문입니다. 따라서, 쓰레드 종료 여부는 전체 프로세스의 자원을 회수하는 데 영향을 주지 않습니다. 쓰레드를 무시하지 않는다면, 쓰레드가 종료될 때, 해당 프로세스가 모두 종료될 수 있는 문제가 발생합니다.

(New) exec

exec은 다음과 같이 변경됐습니다.

```
//FILE: exec.c

extern struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;

int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off, plock;
  uint argc, sz, sp, ustack[3+MAXARG+1];
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pde_t *pgdir, *oldpgdir;
  struct proc *p, *curproc = myproc();

  plock = 0;

  begin_op();

  if((ip = namei(path)) == 0){
    end_op();
    cprintf("exec: fail\n");
    return -1;
  }
  ilock(ip);
  pgdir = 0;

  // Check ELF header
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;

  if((pgdir = setupkvm()) == 0)
    goto bad;

  // Load program into memory.
  sz = 0;
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
```

```
      goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
      goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
    if(ph.vaddr % PGSIZE != 0)
      goto bad;
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;
  }
  iunlockput(ip);
  end_op();
  ip = 0;

  acquire(&ptable.lock);
  plock = 1;

  // 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;

  // remove all other threads
  for (p = ptable.proc; p < &ptable.proc[NPROC]; ++p) {
    if (p != curproc && p->pid == curproc->pid) {
      kfree(p->kstack);
      p->kstack = 0;
      p->pid = 0;
      p->tid = 0;
      p->parent = 0;
      p->tparent = 0;
      p->name[0] = 0;
      p->killed = 0;
      p->state = UNUSED;
    }
  }

  release(&ptable.lock);

  switchuvm(curproc);
  freevm(oldpgdir);
  return 0;

 bad:
  if (plock) release(&ptable.lock);
  if(pgdir)
    freevm(pgdir);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;
}
```

exit과 마찬가지로, exec에서도 존재하는 모든 쓰레드를 종료해야 합니다. 따라서, exec을 호출할 때, 같은 pid를 가진 모든 쓰레드를 종료하도록 변경했습니다. 이때 exec를 호출한 프로세스(쓰레드)의 경우 exec에서 실행하는 새로운 프로그램을 실행해야 하므로, 프로세스(쓰레드)를 종료하지 않습니다.

또, fork와 같은 맥락으로 exec을 호출할 때, ptable.lock을 잡는 위치가 변경됐습니다.

(New) sbrk

sbrk는 다음과 같이 변경됐습니다.

```
//FILE: sysproc.c

void sbrk_init(void) {
  initlock(&sbrk_lock, "sbrk");
}
int
```

```
sys_sbrk(void)
{
  int addr;
  int n;
  struct proc* p = myproc();

  if(argint(0, &n) < 0)
    return -1;

  acquire(&sbrk_lock);

  if (p->tid) p = p->tparent;

  addr = p->sz;

  if(growproc(n) < 0) {
    release(&sbrk_lock);
    return -1;
  }
  release(&sbrk_lock);
  return addr;
}
```

sbrk는 프로세스의 메모리를 할당받는 함수입니다. 이때, 쓰레드의 경우, 부모 프로세스의 sz를 사용하여 메모리를 할당받아야 합니다. 따라서, 쓰레드인 경우, 부모 프로세스의 sz를 사용하도록 변경했습니다.

부모 프로세스의 virtual address 중 빈 공간을 반환해준 뒤, 부모의 sz를 조정해줍니다.

이때, 여러 쓰레드가 p->sz 값에 대해 동시에 접근할 수 있기 때문에, sbrk_lock을 사용하여 동기화를 해줍니다.

이를 사용하지 않는다면 race condition이 발생하여 다른 쓰레드가 sbrk를 동시에 호출했을 때 중복된 주소공간을 할당받을 수도 있습니다.

> 부모 프로세스의 sz를 사용하는 이유는 thread_create 파트에서도 언급했듯이, 메모리 주소를 부모 프로세스가 관리하기로 결정했기 때문입니다. 현재 사용하고 있는 virtual address의 최댓값을 부모 프로세스에서 관리하고 있기 때문에, 쓰레드도 부모 프로세스의 sz를 사용하여 메모리를 할당받아야 합니다.

```
//FILE: proc.c

// Grow current process's memory by n bytes.
// Return 0 on success, -1 on failure.
int
growproc(int n)
{
  uint sz;
  struct proc *curproc, *origin_proc = myproc();

  acquire(&ptable.lock);

  if (origin_proc->tid) curproc = origin_proc->tparent;
  else curproc = origin_proc;
```

```
  sz = curproc->sz;
  if(n > 0){
    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0) {
      release(&ptable.lock);
      return -1;
    }
  } else if(n < 0){
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0) {
      release(&ptable.lock);
      return -1;
    }
  }
  curproc->sz = sz;

  release(&ptable.lock);

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

sbrk 함수에서 사용하는 growproc 함수입니다. 메모리를 늘려주거나 줄여주는 역할을 수행합니다. 이때, 쓰레드가 요청을 보낸 경우 부모 프로세스의 sz를 사용하여 메모리를 할당받습니다. 이후 부모 프로세스의 sz를 조정해줍니다.

또한, sz값에 관련해서 race condition이 발생할 수도 있기 때문에 ptable.lock을 사용하여 동기화를 해줍니다.

> 이후 switchuvm을 호출하는데 curproc가 아닌 origin_proc를 사용하는 이유는, 부모 프로세스의 페이지가 아닌, sbrk 함수를 호출한 쓰레드의 페이지를 사용해야 하기 때문입니다.

```
//FILE: main.c

extern void sbrk_init(void);

// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  picinit();       // disable pic
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  sbrk_init();     // sbrk lock
  pinit();         // process table
  tvinit();        // trap vectors
```

```
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}
```

sbrk_lock을 초기화 하는 코드입니다. main 함수에서 sbrk_lock을 초기화해줍니다.

(New) kill

kill 함수는 변경된 부분이 없습니다.

kill함수가 호출되면, 해당 pid를 가진 프로세스가 죽고, 해당 프로세스가 죽으면 trap에 의해 exit함수가 호출되기 때문에, exit 함수에서 존재하는 모든 쓰레드가 종료되는 것을 보장할 수 있습니다.

(New) sleep

sleep 함수는 변경된 부분이 없습니다.

sleep 함수가 호출되면, ptable.proc에 존재하는 해당 쓰레드의 state만 SLEEPING으로 변경됩니다.

따라서, 별도 처리를 하지 않아도, 모든 쓰레드가 SLEEPING 상태가 되지 않고, 해당 쓰레드만 SLEEPING 상태가 됩니다.

(New) pipe

pipe 함수는 변경된 부분이 없습니다.

명세에서 요구한대로 pipe는 화면에 출력하는 데에만 문제가 없도록 하면 됩니다. 이를 위해서 별도 처리는 필요하지 않으므로 변경된 부분이 없습니다.

user app for testing threading

쓰레드를 잘 만들고, 쓰레드와 프로세스의 간극에서 오는 문제가 없는지 확인하기 위해 다음과 같은 테스트 프로그램을 작성할 수 있습니다.

코드는 별도 첨부하지 않았지만, `rn_test.c` 파일에서 확인할 수 있습니다.

해당 코드는 명세에서 준 테스트를 참조하여 작성되었습니다.

테스트를 통해, 쓰레드가 잘 작동하는지, 공유 변수를 사용하며 생기는 문제는 없는지 등을 테스트할 수 있습니다.

test 1

1. 쓰레드를 두 개 생성해서, 쓰레드 하나에서 공유변수에 값을 덮어쓰도록 합니다.
2. 두 쓰레드가 끝날때까지 thread_join으로 대기한 후, 값을 출력합니다.
3. 공유변수를 잘 활용해서 값을 잘 썼는지, thread_exit에서 넘겨준 값이 올바른지 확인합니다.
4. 또한, thread_join이 프로그램이 끝날 때까지 제대로 대기했는지도 확인합니다.

기본적인 쓸레드 구현이 됐는지 확인할 수 있습니다.

test 2

1. 쓰레드 {NUM_THREAD} 개를 생성해서, 각각 fork를 진행하도록 합니다.
2. 부모에서 값을 쓰고, 자식에서도 값을 각각 다른 값으로 씁니다. (자식이 더 늦게 값을 씁니다.)
3. 이때 fork에서 각각 서로 다른 주소 공간을 사용하는지, 또, fork가 제대로 이루어지는지 확인합니다.
4. 이후 모든 쓰레드가 종료된 후, 부모에서 쓴 값만 기록되어있는지 확인합니다.

이 테스트를 통해, 쓰레드 내에서 fork가 제대로 동작하는지 (자식을 잃어버리거나 잘못된 코드 실행 등) 확인할 수 있습니다.

test 3

1. 쓰레드 {NUM_THREAD} 개를 생성해서, 메모리 할당 해제를 반복하도록 합니다.
2. 이후 각각 별도 공간을 할당 받았는지, 중복된 메모리 공간을 잘 사용할 수 있는지 등을 테스트합니다.

이 테스트를 통해 sbrk가 잘 할당 되는지, 중복된 공간을 할당하진 않는지 등을 확인할 수 있습니다.

test 4

1. 쓰레드 두 종류를 각각 {NUM_THREAD} 개 생성합니다.
2. 한 쓰레드는 일정 시간이 지나면 rn_test_fail()을 호출합니다.
3. 다른 쓰레드는 rn_test_fail()을 호출하는 쓰레드를 kill합니다. (프로세스 자체를 kill 합니다.)
4. 이후, rn_test_fail()이 호출되지 않았는지 확인합니다. 또한, 나머지 쓰레드는 정상적으로 작동했는지 확인합니다.

이 테스트를 통해 kill이 모든 쓰레드를 잘 종료하는지, 확인할 수 있습니다.

test 5

1. fork를 통해 자식 프로세스에서는 쓰레드를 {NUM_THREAD} 개 생성합니다.
2. 자식 프로세스는 쓰레드 내부에서 exit()을 호출하도록 되어있습니다. 이후 쓰레드 밖에서는 무한 루프를 돌며 대기합니다.
3. 부모 프로세스에서는 자식 프로세스가 정상적으로 종료 및 reaping되는지 확인합니다.

이 테스트를 통해 exit이 모든 쓰레드를 잘 종료하는지, 이후 reaping까지 잘 이루어지는지 확인할 수 있습니다.

test 6

1. fork를 통해 자식 프로세스에서는 쓰레드를 {NUM_THREAD} 개 생성합니다.
2. 자식 프로세스는 쓰레드 내부에서 exec()를 호출하도록 되어있습니다. 이후 쓰레드 밖에서는 무한 루프를 돌며 대기합니다.
3. 부모 프로세스에서는 자식 프로세스가 정상적으로 종료 및 reaping되는지 확인합니다.

이 테스트를 통해 exec이 모든 쓰레드를 잘 동료하는지, 이후 reaping까지 잘 이루어지는지 확인할 수 있습니다.

3. Result

스레드 테스트 앱을 구현하고 테스트한 결과, 정상적으로 동작하는 것을 확인할 수 있었습니다.

```
$ rn_test
rn_test1 [Basic]
T 0 S
```

```
T 1 S
T 0 E
T 1 E
rn_test1 Passed

rn_test2 [Fork]
T 0 S
T 1 S
T 2 S
T 3 S
T 4 S
C 1 S
C 2 S
C 4 S
C 0 S
C 3 S
C 1 E
C 2 E
C 4 E
T 2 E
C 0 E
C 3 E
T 1 E
T 3 E
T 4 E
T 0 E
rn_test2 Passed

rn_test3 [Sbrk]
T 0 ST 1 S
T 2 T 3
S
ST
 4 S
rn_test3 Passed

rn_test4 [Thread Kill]
T 10029 S
Kill 29
TT 2 S
T 3 S
T 4 S
 1 S
rn_test4 Passed

rn_test5 [Thread Exit]
T 0 T 1 S
T 2 S
T 3 S
T 4 S
S
Exit
rn_test5 Passed
```

```
rn_test6 [Thread Exec]
T 0 S
T 1 S
T 2 S
T 3 S
T 4 S
Exec
'exec echo'
rn_test6 Passed

rn_test finish
```

직접 컴파일 및 실행하려면 다음 명령어를 사용하면 됩니다. 이후 xv6 내부에서 `rn_test`를 실행하면 위와 비슷한 결과를 확인할 수 있습니다.

```
$ sh boot.sh
```

4. Trouble shooting

이번 과제는 자원 관리 방법을 알아내는게 어려웠습니다. `mmu` 관련 코드를 확인해보면 `virtual address -> physical address`로 변환하는 과정이 복잡하게 이루어지는데, 이를 이해하는데 시간이 걸렸습니다. 그래도, 기존 `fork`와 `exec`에서 사용하는 예제를 확인해보고, 어떤 함수를 통해 자원을 할당 및 관리하는지 확인할 수 있어서 수월하게 진행됐습니다.

그 외엔 코드를 수정할 부분이 거의 없어서 큰 문제 없이 진행됐습니다.