Project2 Wiki

> 2021011158 김선희

📌 **Goal: Implement a simple kernel-level thread**

Design

Thread

- 각 thread는 data, text, file을 공유하며 고유의 stack과 register를 갖는다.
- xv6는 process를 기준으로 동작한다.

⇒ 💡 key idea: process를 thread처럼 동작하게 만든다.

- thread 간의 공유하는 것과 독립적으로 가지는 것을 잘 구분하여 구현한다.
  - thread 간의 공유 자원: pagetable, file
  - thread 개별 자원: trapframe, stack

proc 구조체

process를 thread처럼 동작하게 하기 위해 proc.h의 proc 구조체를 수정했다.

```
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;       // Process state
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
  int xstate;                 // Exit status to be returned to parent's wait
  int pid;                    // Process ID

  // wait_lock must be held when using this:
  struct proc *parent;        // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;              // Virtual address of kernel stack
  uint64 sz;                  // Size of process memory (bytes)
  pagetable_t pagetable;      // User page table
  uint64 trapframe_va;        // virtual address of the trapframe
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;     // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;          // Current directory
  char name[16];              // Process name (debugging)

  /* implementation for project02 */
  struct proc *main_thread;
  int tid;
  void *stack;
  uint64 *main_sz; // pointer to sz of main thread
  /* ====== */
};
```

`allocproc()`을 통해 처음 만들어진 thread를 process의 본체인 main thread라 가정한다. 이후 clone을 통해 들어진 process들은 main thread의 pagetable, file 등을 공유한다. 따라서 `main_thread` 변수를 두어 모든 thread가 최초의 thread를 가리키게 했다.

`tid`는 thread의 id이고, `stack`은 user program으로부터 넘겨 받은 user stack을 저장하는 변수이다.

모든 thread는 main thread가 처음 할당 받은 pagetable을 공유하므로, `sz` 또한 main_thread의 값을 가져야 한다. `sbrk()`, `malloc()` 등에 대한 system call 호출을 정상적으로 수행하기 위해 이를 가리키는 포인터 값 `main_sz`를 두어서 변경된 값을 반영할 수 있도록 했다.

API

1. **int clone(void(fcn)(void*, void*), void *arg1, void *arg2, void *stack);**

현재 실행 중인 프로세스의 address space를 공유하는 새로운 kernel-level thread를 만드는 함수이다.

→ 현재 실행 중인 프로세스로부터 새로운 프로세스를 만드는 `fork()`와 이를 동작시키는 `exec()`을 참고하여 구현했다.

- 새로 만들어진 thread는 메모리 공간을 다른 thread들과 공유해야 하므로, main thread의 pagetable을 공유함으로서 이를 관리하도록 한다.
- 새로 만들어지는 thread는 process가 아니기 때문에 `fork()`와 마찬가지로 allocproc을 사용하되,
  - pid는 증가하지 않고,
  - pagetable을 새로 할당 받지 않는다.
- thread가 생성된 후 이를 실행하기 위해 `exec()`을 참고하여 trapframe을 수정한다.

2. **int join(void **stack);**

자식 thread가 종료되기를 기다리는 함수이다.

→ 자식 프로세스가 종료되기를 기다리는 `wait()`을 참고하여 구현했다.

- thread들은 main thread의 pagetable을 공유하므로, main thread만이 pagetable free 권한을 가진다. 따라서 기존의 `wait()`과는 다르게 pagetable을 free하는 과정을 제외한다.

기존 system call 호환성

기존의 system call은 프로세스를 기준으로 동작하기 때문에, 수정된 proc 구조체 (thread)와 호환 가능하도록 수정한다.

1. **fork**

- 새로운 프로세스가 생성될 때, p의 sz가 아닌 p의 main_sz가 가리키는 값을 사용하도록 한다.
- main thread가 process의 본체이므로, 다른 프로세스의 부모가 될 수 있는 것은 오직 main thread이다.

2. **exec**

- thread가 exec을 호출하면 모든 thread를 정리하고 완전히 새로운 프로세스를 시작해야 한다. → 나는 이것을 exec을 호출한 thread만 남아 나머지 thread를 정리하고, 이 thread의 정보를 새로운 process로 덮어 씌운다.
  - exec을 실행한 thread가 main thread가 아닐 경우, pagetable에서 clone을 통해 만들어진 thread의 trapframe을 unmap한 이후에 해당 thread를 main thread로 바꾼다.

3. **sbrk**

sbrk는 내부적으로 growproc을 호출하여 메모리 영역의 크기를 grow/shrink한다. thread들은 main thread를 통해 자원을 공유하므로 메모리 영역의 사이즈는 항상 main thread의 sz 값을 사용해야 한다.

4. **kill**

kill을 호출하면 인자로 pid를 받는데, 프로세스 단위의 정상 종료를 위해 main thread를 찾아 그것의 killed를 1로 세팅한다.

5. **exit**

main thread가 exit을 호출하면 해당 thread가 속한 프로세스의 모든 thread가 종료되어야 하고, 일반 thread가 exit을 호출하면 오직 그 thread만 종료되어야 한다.

main_thread가 exit을 호출한 경우에는 같은 pid를 가진 thread를 모두 찾아 종료시킨다.

Implementation

int clone(void(fcn)(void*, void*), void *arg1, void *arg2, void *stack)

새로운 thread 만들기

allocproc_for_clone:

```
static struct proc*
allocproc_for_clone(struct proc *main_thread)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == UNUSED) {
      goto found;
    } else {
      release(&p->lock);
    }
  }
  return 0;

found:
  p->pid = main_thread->pid;
  p->state = USED;
  p->main_thread = main_thread;
  p->tid = nexttid;
  nexttid++;
  p->stack = 0;
  p->main_sz = &(main_thread->sz);

  // Allocate a trapframe page.
  if((p->trapframe = (struct trapframe *)kalloc()) == 0){
    freeproc(p);
    release(&p->lock);
    return 0;
  }

  // Set up new context to start executing at forkret,
  // which returns to user space.
  memset(&p->context, 0, sizeof(p->context));
  p->context.ra = (uint64)forkret;
  p->context.sp = p->kstack + PGSIZE;

  return p;
}
```

clone:

```
    if (p->main_thread){
    main_thread = p->main_thread;
  }
  else{
    main_thread = p;
  }

  if((np = allocproc_for_clone(main_thread)) == 0){
    return -1;
  }

  np->sz = *(np->main_sz);
  np->parent = main_thread->parent;
  np->main_thread = main_thread;
  np->stack = stack;
```

```
  np->pid = main_thread->pid;
 *(np->trapframe) = *(main_thread->trapframe);

  np->pagetable = main_thread->pagetable;
  np->trapframe_va = TRAPFRAME - np->tid * PGSIZE;
```

기존 fork와 다르게 pid는 증가하지 않고, pagetable을 새로 할당 받지 않게 하기 위해 allocproc에서 해당 부분을 변형한 allocproc_for_clone을 새로 정의하여 사용했다.

allocproc_for_clone을 통해 새로 할당 받은 np가 thread로 동작할 수 있게 설정했다. sz, parent, trapframe 등은 main_thread에 기반하여 설정했다. 새 thread는 main thread와 pagetable을 공유하고, tid를 사용하여 thread를 구분한다.

새로운 thread를 start rountine에서 실행하기

```
  if (mappages(np->pagetable, np->trapframe_va, PGSIZE,
          (uint64)(np->trapframe), PTE_R | PTE_W) < 0) {
  uvmunmap(np->pagetable, np->trapframe_va, 1, 0);
  freeproc(np);
  return -1;
 }
```

이 부분의 구현은 project1과 달라진 부분에서 힌트를 얻었다.

project1:

```


# a0: user page table
li a0, TRAPFRAME
```

project2

```
# a0: user page table, a1: trapframe va
mv a0, a1
```

userret에서 trapframe 주소를 고정값(TRAPFRAME) 으로 접근하던 것을, 유동적으로 `trapframe_va`를 전달 받아 설정하도록 변경되었다.

thread는 `trapframe` 주소가 다르다. 같은 process 내의 thread끼리는 같은 pagetable을 공유하므로 project1의 방식처럼 TRAPFRAME에 하나만 매핑할 수 없고, 스레드마다 다른 `trapframe_va`를 매핑해야 한다.

```
  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);

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

  np->trapframe->epc = (uint64)fcn;
  np->trapframe->sp = (uint64)stack + PGSIZE;
  np->trapframe->a0 = (uint64)arg1;
  np->trapframe->a1 = (uint64)arg2;

  np->state = RUNNABLE;

  release(&np->lock);

  return np->tid;
```

trapframe의 변수를 변경하여 `fcn`에서 `arg1`, `arg2`를 가지고 새로운 thread가 실행될 수 있도록 했다. stack은 user program에서 할당하여 넘긴 stack을 사용하도록 했다.

이 과정을 통해 usertrapret을 거쳐 user program으로 돌아가 원하는 `fcn`을 수행할 수 있게 된다.

int join(void **stack);

```
int
join(void **stack)
{
  struct proc *p;
  struct proc *curproc = myproc();
  int havekids;
  int pid;

  acquire(&wait_lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;

    if(curproc->tid == 0){
      pid = curproc->pid;

      for(p = proc; p < &proc[NPROC]; p++){
        acquire(&p->lock);
        if(p->main_thread == curproc && p != curproc){
          havekids = 1;
          if(p->state == ZOMBIE){

            if (copyout(curproc->pagetable, (uint64)stack, (char *)&p->stack, sizeof(uint64)) < 0) {
              release(&wait_lock);
              return -1;
            }

            // // Clean up thread
            if(p->trapframe)
              kfree((void*)p->trapframe);
            p->trapframe = 0;
            p->trapframe_va = 0;
            p->sz = 0;
            p->pid = 0;
            p->parent = 0;
            p->name[0] = 0;
            p->chan = 0;
            p->killed = 0;
            p->xstate = 0;
            p->state = UNUSED;
            p->tid = 0;
            p->main_thread = 0;
            p->stack = 0;

            release(&p->lock);
            release(&wait_lock);

            return pid;
          }

        }
        release(&p->lock);
      }
    }

    if(!havekids || killed(curproc)){
      release(&wait_lock);
      return -1;
    }

    // Wait for a child to exit.
    sleep(curproc, &wait_lock);  //DOC: wait-sleep
```

```
  }
}
```

기존의 wait과 동작 방식은 매우 유사하다.

각 thread에 대한 wait이기 때문에 freeproc을 참고하여 작성한 free 과정에서 main thread만 가능한 pagetable free 부분은 제외했다.

또한 clone 시 user program으로부터 받았던 stack을 인자에 담아 전달하기 위해 copyout을 사용하였다.

호환성을 위해 수정/추가한 부분

allocproc

```
  p->main_thread = p;
  p->tid = 0;
  p->stack = 0;
  p->main_sz = &(p->sz);
```

thread 구현을 위해 새로 정의한 변수에 대한 초기화 부분을 추가하였다.

growproc

```
uint64 sz = *(p->main_sz);
...
*(p->main_sz) = sz;
```

thread들은 main thread를 통해 자원을 공유하므로 기존의 p->sz를 main thread의 sz를 가리키는 포인터로 수정하였다.

fork

```
np->sz = *(p->main_sz);
...
if (np->tid == 0){
  np->parent = p;
}
else{
  np->parent = p->main_thread;
}
```

새로운 프로세스가 생성될 때, p의 sz가 아닌 p의 main_sz가 가리키는 값을 사용하도록 했다.

main thread가 process의 본체이므로, 다른 프로세스의 부모가 될 수 있는 것은 오직 main thread이다.

exec

```
// terminate all threads
  for(thread = proc; thread < &proc[NPROC]; thread++){
    if(thread->pid == p->pid && thread != p->main_thread){
      // unmapping
      if(thread->trapframe){
        uvmunmap(thread->pagetable, thread->trapframe_va, 1, 0);
      }
    }
  }

  if(p->tid != 0){
    p->main_thread->tid = p->tid;
    p->tid = 0;
    p->parent = p->main_thread->parent;
    p->main_thread = p;
    p->main_sz = &(p->sz);
```

```
  }


  for(thread = proc; thread < &proc[NPROC]; thread++){
    if(thread->pid == p->pid && thread != p){
      acquire(&thread->lock);

      if(thread->trapframe){
        kfree((void*)thread->trapframe);
      }
      thread->trapframe = 0;
      thread->trapframe_va = 0;
      thread->sz = 0;
      thread->pid = 0;
      thread->parent = 0;
      thread->name[0] = 0;
      thread->chan = 0;
      thread->killed = 0;
      thread->xstate = 0;
      thread->state = UNUSED;
      thread->tid = 0;
      thread->main_thread = 0;
      thread->stack = 0;

      release(&thread->lock);
    }
  }
```

기존의 exec 코드의 상단에 모든 thread를 정리하는 부분을 추가하였다.

그리고 exec을 실행한 thread가 main thread가 아닐 경우, pagetable에서 clone을 통해 만들어진 thread의 trapframe을 unmap한 이후에 해당 thread를 main thread로 바꾸었다.

sbrk

```
uint64
sys_sbrk(void)
{
  uint64 addr;
  int n;

  argint(0, &n);

  /* implementation for project02 */
  struct proc *p = myproc();

  //addr = myproc()->sz;

  if(p->tid == 0 || !(p->main_thread)){
    addr = p->sz;
  }
  else{
    addr = p->main_thread->sz;
  }
  /* ===== */
  if(growproc(n) < 0)
    return -1;

  return addr;
}
```

메모리 사이즈를 main의 것을 사용할 수 있도록 이를 가리키는 포인터의 값으로 변경하였다.

kill

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

  for(p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);
    if(p->pid == pid && p->tid == 0){
      p->killed = 1;
      ...
```

kill system call의 목적은 인자로 주어진 pid를 가지는 프로세스를 종료하는 것이다. 따라서 프로세스의 본체 격인 main thread의 killed를 1로 세팅하여, main thread가 프로세스 종료의 주체가 되는 exit이 제대로 동작할 수 있게 했다.

exit

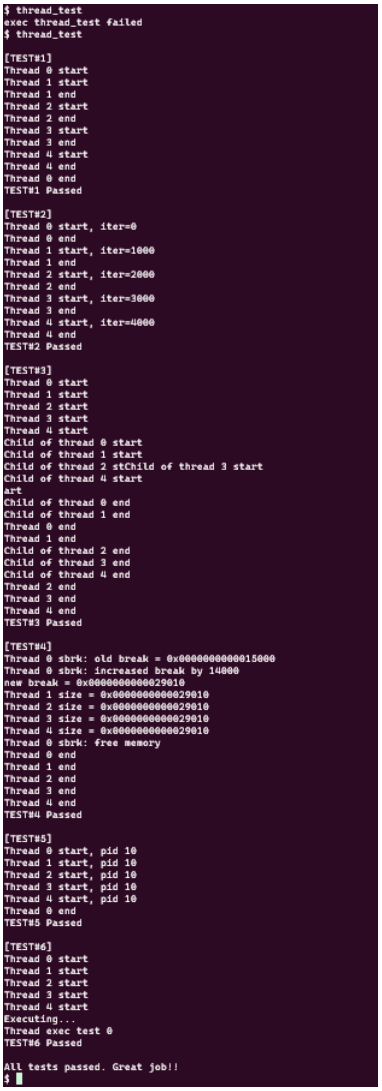
```
struct proc *thread;
  if(p->tid == 0){
    for(thread = proc; thread < &proc[NPROC]; thread++){
      acquire(&thread->lock);
      if(thread->pid == p->pid && thread->tid != 0){
        //freeproc(thread);
        // Clean up thread
        if(thread->trapframe){
          uvmunmap(thread->pagetable, thread->trapframe_va, 1, 0);
          kfree((void*)thread->trapframe);
        }
        thread->trapframe = 0;
        thread->trapframe_va = 0;
        thread->sz = 0;
        thread->pid = 0;
        thread->parent = 0;
        thread->name[0] = 0;
        thread->chan = 0;
        thread->killed = 0;
        thread->xstate = 0;
        thread->state = UNUSED;
        thread->tid = 0;
        thread->main_thread = 0;
        thread->stack = 0;
      }
      release(&thread->lock);
    }
  }
```

main thread가 exit을 호출한 경우에는 해당 thread가 속한 프로세스의 모든 thread가 종료되어야 하므로, freeproc을 참고하여 process table을 순회해 모든 thread를 정리하는 부분을 추가하였다.

```
if(p->tid == 0){
    //printf("main called exit\n");
    // Give any children to init.
    reparent(p);
    // Parent might be sleeping in wait().
    wakeup(p->parent);
  } else{
    //printf("thread called exit\n");
    wakeup(p->main_thread);
  }
```

기존 exit은 process를 기준으로 하기 때문에 자식의 parent를 재설정하고, wait 호출을 위해 자신의 parent를 깨운다. 그러나 지금은 thread 기준이기 때문에 main thread의 경우 원래대로 동작하고, 일반 thread의 경우 main thread를 깨우는 것으로 바꾸어 동작한다.

Results



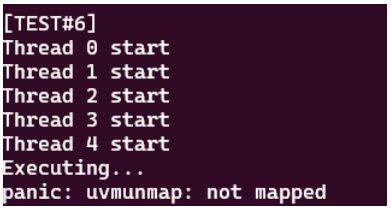
Troubleshooting

exec: pagetable free & unmap 문제

pagetable free를 위해서는 mapping했던 trapframe 정보를 모두 unmap해야 정상적으로 free를 수행할 수 있었다.

```
for(thread = proc; thread < &proc[NPROC]; thread++){
   if(thread->pid == p->pid && thread != p->main_thread){
     // unmapping
     if(thread->trapframe){
       uvmunmap(thread->pagetable, thread->trapframe_va, 1, 0);
     }
   }
 }
```

따라서, exec 상단에 각 thread의 trapframe을 unmap하는 부분을 추가하였다.



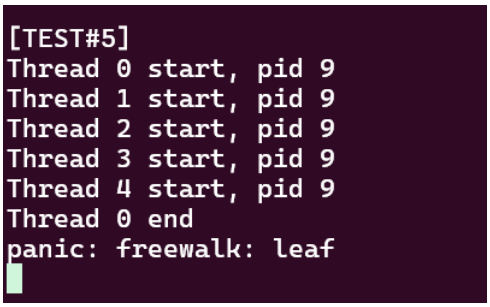
exec을 실행한 thread가 main thread가 아닐 경우 pagetable에서 clone을 통해 만들어진 thread의 trapframe을 unmap한 이후에 해당 thread를 main thread로 바꾸는데, 이를 unmap 이전에 실행하면 mapping의 대상이 아니었던 main thread가 unmap되어 panic이 일어난다.

따라서 trapframe unmap 이후 실행 중인 thread를 main thread로 변경한다.

exit: 자원 해제 시 freewalk 문제

```
if(p->tid == 0){
   for(thread = proc; thread < &proc[NPROC]; thread++){
     acquire(&thread->lock);
     if(thread->pid == p->pid && thread->tid != 0){
       // Clean up thread
       if(thread->trapframe) {
         kfree((void*)thread->trapframe);
       }
       thread->trapframe = 0;
       thread->trapframe_va = 0;
       thread->sz = 0;
       thread->pid = 0;
       thread->parent = 0;
       thread->name[0] = 0;
       thread->chan = 0;
       thread->killed = 0;
       thread->xstate = 0;
       thread->state = UNUSED;
       thread->tid = 0;
       thread->main_thread = 0;
       thread->stack = 0;
     }
     release(&thread->lock);
   }
 }
```

trap에서 main thread의 kill이 1으로 세팅됨을 확인하고 exit을 진행할 때, 위의 코드로 진행하면 freewalk panic이 발생하였다.



clone하면서 임의로 trapframe을 pagetable에 매칭했었기 때문에, 이걸 다시 unmap해야 안전하게 자원을 회수할 수 있게 된다.

→ solution:

```
...
if(thread->trapframe) {
  uvmunmap(thread->pagetable, thread->trapframe_va, 1, 0);
  kfree((void*)thread->trapframe);
```

```
}
...
```