OS project2 - Implementing a simple kernel-level thread

2021007138 서교빈

Design

💡 Outline of implementation approach

이번 과제에서 구현해야 할 점은 기존 xv6의 single thread에서 kernel-level thread를 구현해 multi thread가 가능하게 하는 것이다. kernel-level threads는 복수의 context를 동시에 가능하게 하고 multi process의 single thread와 다른 점은 process의 address space를 공유하며, 고유의 register와 stack을 가지는 것이다.

address space는 process의 pagetable로써 공유하며, 특이점은 thread마다 고유 file descriptor table을 갖는다. 기존에 존재하는 PCB(struct proc) 구조체를 활용하여 thread 구현을 위한 member만 추가하여 개별 process들을 main thread와 sub thread로 구분하도록 디자인한다. 더불어 기존의 각 system call별로 sub thread와 main thread 경우를 세분화하여 구현한다.

thread 관리를 위한 system call은 clone()과 join()을 추가한다. clone()에서는 process의 pagetable을 공유 받는 kernel-level thread를 생성하도록 하고, join()에서는 main thread에서 sub thread가 종료될 때까지 대기하다가 pid를 반환하도록 한다.

또한 각 system call을 User level API로 사용하도록 thread_create()와 thread_join()을 구현한다. thread_create에서는 user stack의 공간을 할당 후 clone()의 인자로 넘겨준다. thread_join에서는 join()을 호출해 thread가 종료될 때까지 기다린 후 넘겨 받은 stack의 주소를 통해 공간을 해제한다. user stack의 공간은 user level에서만 관리하므로 stack의 공간 관리 또한 user program에서 이루어진다. 따라서 각각의 함수에서만 stack을 할당하고 해제한다.

Details

💡 Details of approach in code

우선 thread control block은 기존의 process control block으로 사용할 수 있다. thread id는 pid와 동일하기 때문에 process의 thread 여부, process의 main thread, thread의 user stack(tstack)주소만 추가해준다.

```
struct proc {
  ...

  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)

  int isthread;              //thread == 1
  void* tstack;              //userstack pointer
  struct proc* main;         //main thread(process thread leader)
};
```

만약 fork() 등으로 생성된 process p라면 p→isthread ==0; p→main = p; 처럼 초기화한다.

fork()를 통해 부모 process의 자식 process를 생성한다면, clone은 main thread의 sub thread를 생성한다. 처음 process를 생성할 경우, 그 자체가 main thread 역할을 갖는다. 따라서 모든 sub thread의 parent는 그 main thread의 parent와 동일, 또 다른 process내의 main thread를 가리킨다. 단, process와 thread의 계층 구조는 다른 양상을 갖는데 process는 자식을 가질 때마다 수직적인 트리 구조를 갖지만 thread는 결국 어떤 sub thread이든 main thread 아래에 수평적인 구조를 갖는다.

process(main thread)와 sub thread 생성의 차이는 allocthread 함수를 통해 구현했다. allocproc에서는 proc_pagetable 함수를 통해 새로운 pagetable을 생성하고 mapping하는 과정이 포함된다. 반면 thread의 핵심은 process 내에서 pagetable을 공유해야 한다. 따라서

main thread의 pagetable과 동일하게 해주면 된다. 단, 각 thread의 register는 고유한 공간을 갖기 때문에 trapframe의 주소 또한 각 공간으로 mapping해줘야 한다. 이는 proc 구조체의 trapframe_va에 가상주소를 저장해 한 페이지씩 차이를 두어 접근하도록 한다.

process의 wait역할과 thread의 join system call도 유사한 역할을 갖는다. wait 호출 시 자식 process가 ZOMBIE가 되어 자원을 해제할 때까지 sleep을 유지하고 깨어난다. join에서는 main thread가 sub thread의 ZOMBIE를 발견할 때 까지 sleep을 유지하고 깨어난다. 차이점은 join에서는 thread의 자원만 해제하고 process의 자원은 유지해야 한다. 따라서 wait에서의 freeproc 호출 대신 join에서는 freethread를 통해 구현했다. freeproc시 proc_freepagetable을 통해 pagetable의 자원을 해제한다. thread에서는 process의 공유 자원은 유지해야 하므로 trapframe_va의 mapping 해제, trapframe해제, file descriptor 해제를 적용한다.

clone과 join 모두 stack의 인자를 받는다. 이는 user program에서 user stack의 공간을 동적할당하기 때문이다. TCB내에 userstack의 주소를 유지한 후 join 호출 시 그 stack의 주소를 통해 user level에서 해제한다. 이는 각각 thread_create와 thread_join으로 구현된다.

growproc시 sz를 새로 지정하기 때문에 process 내 모든 thread의 sz도 동일한 크기를 유지해야 한다. 이는 process배열을 순회하며 같은 process내 thread의 sz 크기를 다시 지정해주면 된다.

exit호출은 예외 사항이 있다. 만약 main thread가 호출 시, process내 전체 thread가 종료되어야 하지만, sub thread에서 호출 시 해당 sub thread만 종료되어야 한다.

기존 exit의 경우 parent를 wakeup 하고 state를 ZOMBIE로 바꾸며 스케줄링을 받게 된다.

main thread가 호출 시 기존 로직과 동일하지만 그 이전에 proc배열을 순회하며 같은 process 내의 thread를 종료해야 한다. 이는 기존에 구현한 freethread를 통한다. 이후 ZOMBIE state가 되어 parent의 wait를 통한다.

만약 sub thread가 호출 시에는 기존 exit 로직과 차이가 없지만 parent의 wait를 통하는 대신, main thread의 join을 통해 해제되도록 구현했다.

exec 또한 thread의 경우를 고려해야 한다. thread에서 exec할 경우 thread가 process의 main thread가 되어 기존 process의 모든 thread를 종료시킨다. 기존 exec의 로직의 새로운 pagetable을 생성하는 과정을 그대로 유지한다. 단 thread일 경우 기존의 trapframe_va를 unmapping 하고 새로운 pagetable을 새로운 main thread에 지정한다. 그 외 기존 thread는 기존 main thread에 kill을 호출해 killed flag를 통해 기존 main thread가 exit이 실행되도록 구현했다.

Code Implementation

💡 Key code modification and purpose

```
static struct proc*
allocthread(struct proc *main, void *tstack)
{
  struct proc *t;
  int idx = -1;

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

found:
  t->pid = allocpid();
  t->state = USED;
  t->isthread = 1;
  t->main = main;        // 그룹 리더(메인 스레드) 지정
  t->tstack = tstack;    // 사용자 스택(유저가 malloc한 주소)
  t->parent = main->parent; // fork/clone 규칙에 맞게 부모 설정

  // trapframe, 커널 스택, context 등 스레드별 자원 할당
  if((t->trapframe = (struct trapframe*)kalloc()) == 0) {
    freethread(t);
    release(&t->lock);
    return 0;
  }
  memset(t->trapframe, 0, PGSIZE);
```

```
  // 커널 스택
  t->kstack = KSTACK(idx);
  // context 초기화
  memset(&t->context, 0, sizeof(t->context));
  t->context.ra = (uint64)forkret;
  t->context.sp = t->kstack + PGSIZE;

  // 주소 공간, 파일 디스크립터, cwd 등은 main에서 공유/복사
  // trappage_va는 thread별로 mapping
  t->pagetable = main->pagetable; //shared
  if(mappages(main->pagetable, t->trapframe_va = TRAPFRAME - idx * PGSIZE, PGSIZE, (uint64)t->trapframe, PTE_R | PTE_W) < 0) {
    return 0;
  }

  t->sz = main->sz; //복사 grow예외
  for(int i = 0; i < NOFILE; i++)
    if(main->ofile[i])
      t->ofile[i] = filedup(main->ofile[i]);
  t->cwd = idup(main->cwd);

  safestrcpy(t->name, main->name, sizeof(t->name));

  return t;
}

// 스레드별 자원만 해제 (커널 스택, trapframe 등)
// t->lock을 잡은 상태에서 호출해야 함
static void
freethread(struct proc *t)
{
  //va mapping 해제
  if(t->pagetable && t->trapframe_va)
    uvmunmap(t->pagetable, t->trapframe_va, 1, 0);
  // trapframe 해제
  if(t->trapframe) {
    kfree((void*)t->trapframe);
    t->trapframe = 0;
    t->trapframe_va = 0;
  }

  // 커널 스택 해제
  // 사용자 스택(t->tstack)은 해제X
  // join에서 user space에서 free로 해제

  //fd 해제
  for(int fd = 0; fd < NOFILE; fd++) {
    if(t->ofile[fd]) {
      fileclose(t->ofile[fd]);
      t->ofile[fd] = 0;
    }
  }
  if(t->cwd) {
    begin_op();
    iput(t->cwd);
    end_op();
    t->cwd = 0;
  }

  // 기타 초기화
  t->state = UNUSED;
  t->pid = 0;
  t->parent = 0;
  t->main = 0;
  t->isthread = 0;
  t->tstack = 0;
  t->chan = 0;
  t->killed = 0;
  t->xstate = 0;
  t->name[0] = 0;
  t->pagetable = 0;
  // pagetable, sz 등은 main thread가 관리
}

int
growproc(int n)
{
  uint64 sz;
  struct proc *p = myproc();

  sz = p->sz;
  if(n > 0){
    if((sz = uvmalloc(p->pagetable, sz, sz + n, PTE_W)) == 0) {
      return -1;
    }
  } else if(n < 0){
    sz = uvmdealloc(p->pagetable, sz, sz + n);
  }
  for(struct proc *t = proc; t < &proc[NPROC]; t++) {
    if(t->pagetable == p->pagetable) {
      t->sz = sz;
    }
  }
  return 0;
}

int
clone(void(*fcn)(void*, void*), void* arg1, void* arg2, void* stack) {
  struct proc *np;
```

```
  struct proc *p = myproc();

  //allocthread으로 thread 할당(~= fork)
  if((np = allocthread(p->isthread ? p->main : p, stack)) == 0)
    return -1;

  //trapframe copy
  *(np->trapframe) = *(p->trapframe);
  //
  np->trapframe->epc = (uint64)fcn;
  // - a0, a1: 함수 인자
  np->trapframe->a0 = (uint64)arg1;
  np->trapframe->a1 = (uint64)arg2;
  // - sp: 새 스택의 최상단
  np->trapframe->sp = (uint64)stack + PGSIZE;

  //scheduling
  int pid = np->pid;
  np->state = RUNNABLE;
  release(&np->lock);

  return pid;
}

void
exit(int status)
{
  struct proc *p = myproc();
  if(p == initproc)
    panic("init exiting");

// 메인 스레드가 exit()를 호출한 경우: 프로세스 전체 종료
if(p->isthread == 0) {
  // 같은 주소 공간을 공유하는 모든 스레드(자신 제외) 순회
  for(struct proc *t = proc; t < &proc[NPROC]; t++) {
    if(t->pagetable == p->pagetable && t->state != ZOMBIE && t != p) {
      begin_op();
      iput(t->cwd);
      end_op();
      acquire(&t->lock);
      t->cwd = 0;

      // t->tstack (user stack)는 해제하지 않음! (join 없이 종료)
      freethread(t); // 커널 자원 해제

      release(&t->lock);
    }
  }
  // 메인 스레드가 마지막에 parent를 깨움
  // 열린 파일, cwd 등 자원 해제
      for(int fd = 0; fd < NOFILE; fd++){
        if(p->ofile[fd]){
          struct file *f = p->ofile[fd];
          fileclose(f);
          p->ofile[fd] = 0;
        }
      }
  acquire(&wait_lock);
  reparent(p);
  wakeup(p->parent);
  acquire(&p->lock);
  p->xstate = status;
  p->state = ZOMBIE;
  release(&wait_lock);
  sched();
  panic("zombie exit");
}

  // 서브 스레드가 exit()를 호출한 경우: 자신만 종료(ZOMBIE로 남김)
  for(int fd = 0; fd < NOFILE; fd++){
    if(p->ofile[fd]){
      struct file *f = p->ofile[fd];
      fileclose(f);
      p->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(p->cwd);
  end_op();
  p->cwd = 0;

  acquire(&wait_lock);
  // main 깨우기 (join에서 회수)
  wakeup(p->main);

  acquire(&p->lock);

  p->xstate = status;
  p->state = ZOMBIE;

  release(&wait_lock);

  sched();
  panic("zombie exit");
}

int
```

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

  acquire(&wait_lock);
  for(;;) {
    havekids = 0;
    for(t = proc; t < &proc[NPROC]; t++) {
      // 같은 main_thread(스레드 그룹) && 스레드(메인 제외)만 대상
      if(t->main == p->main && t->isthread) {
        acquire(&t->lock);
        havekids = 1;
        if(t->state == ZOMBIE) {
          // 스택 주소 복사 (유저 공간)
          if(stack != 0 &&
             copyout(p->pagetable, (uint64)stack, (char *)&t->tstack, sizeof(void *)) < 0) {
            release(&t->lock);
            release(&wait_lock);
            return -1;
          }
          int pid = t->pid;
          freethread(t); // 자원 해제
          release(&t->lock);
          release(&wait_lock);
          return pid;
        }
        release(&t->lock);
      }
    }
    // 스레드 그룹 내 자식 스레드가 없거나, 자신이 죽었으면 -1
    if(!havekids || killed(p)) {
      release(&wait_lock);
      return -1;
    }
    // 종료된 스레드가 없으면 대기
    sleep(p, &wait_lock);
  }
}

int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off;
  uint64 argc, sz = 0, sp, ustack[MAXARG], stackbase;
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pagetable_t pagetable = 0, oldpagetable;
  struct proc *p = myproc();
  struct proc *oldmain = p->main;
  int wasthread = p->isthread;

  if(wasthread) {
      uvmunmap(p->pagetable, p->trapframe_va, 1, 0);
  }
  ...
  if(wasthread) {
    p->pagetable = pagetable;
    p->sz = sz;
    p->trapframe->epc = elf.entry;  // initial program counter = main
    p->trapframe->sp = sp; // initial stack pointer
    p->isthread = 0;
    p->main = p;
    kill(oldmain->pid);
    return argc; // this ends up in a0, the first argument to main(argc, argv)
  }
  oldpagetable = p->pagetable;
  p->pagetable = pagetable;
  p->sz = sz;
  p->trapframe->epc = elf.entry;  // initial program counter = main
  p->trapframe->sp = sp; // initial stack pointer

  proc_freepagetable(oldpagetable, oldsz);

  return argc; // this ends up in a0, the first argument to main(argc, argv)

bad:
  if(pagetable)
    proc_freepagetable(pagetable, sz);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;
}
```

Results

💡 Compliation process, program flow and working code

```
$ thread_test

[TEST#1]
Thread 0 start
Thread 1 start
Thread 1 end
Thread 2 start
Thread 2 end
Thread 3 start
Thread 3 end
Thread 4 start
Thread 4 end
Thread 0 end
TEST#1 Passed

[TEST#2]
Thread 0 start, iter=0
Thread 0 end
Thread 1 start, iter=1000
Thread 1 end
Thread 2 start, iter=2000
Thread 2 end
Thread 3 start, iter=3000
Thread 3 end
Thread 4 start, iter=4000
Thread 4 end
TEST#2 Passed
```

```
[TEST#3]
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Child of thread 0 start
Child of thread 1 start
Child of thread 2 start
Child of thread 3 start
Child of thread 4 start
Child of thread 0 end
Child of thread 1 end
Child of thread 2 end
Child of thread 3 end
Child of thread 4 end
Thread 0 end
Thread 1 end
Thread 2 end
Thread 3 end
Thread 4 end
TEST#3 Passed

[TEST#4]
Thread 0 sbrk: old break = 0x0000000000015000
Thread 0 sbrk: increased break by 14000
new break = 0x0000000000029010
Thread 1 size = 0x0000000000029010
Thread 2 size = 0x0000000000029010
Thread 3 size = 0x0000000000029010
Thread 4 size = 0x0000000000029010
Thread 0 sbrk: free memory
Thread 0 end
Thread 1 end
Thread 2 end
Thread 3 end
Thread 4 end
TEST#4 Passed
```

```
[TEST#5]
Thread 0 start, pid 29
Thread 1 start, pid 29
Thread 2 start, pid 29
Thread 3 start, pid 29
Thread 4 start, pid 29
Thread 0 end
TEST#5 Passed

[TEST#6]
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Executing...
Thread exec test 0
TEST#6 Passed

All tests passed. Great job!!
$
```

[TEST#1] 여러 thread를 생성해 자원 공유 상태를 확인한다.

[TEST#2] thread의 인자와 공유 자원을 확인한다.

[TEST#3] thread 내에서 fork() 호출 과정을 확인한다.

[TEST#4] 각 thread의 sbrk 동작을 확인한다.

[TEST#5] thread에서 kill할 경우 종료됨을 확인한다.

[TEST#6] thread 환경에서 exec호출 시 모든 기존 thread가 종료되고, 새로운 프로그램이 정상적으로 실행되는지 확인한다.

TroubleShooting

💡 Encountered problem, applied solutions

❗ **표준 라이브러리 함수 충돌**

thread.c 코드에서 유저 스택의 공간을 malloc으로 할당하여 사용한다. 코드 작성 당시 malloc을 사용하기 위해 습관적으로 stdlib.h 헤더파일을 읽어왔다. 이 경우 umalloc.c에 정의된 malloc 함수와 코드 실행 중 발생하는 exit함수가 C언어 표준 라이브러리 stdlib.h과 중복 선언되어 에러가 발생했다. user.h에 xv6 내의 malloc과 exit이 정의되기 때문에 stdlib.h 헤더 파일을 지워 해결했다.

❗ **이중 lock panic 발생(panic: acquire)**

멀티 스레드 구현에서 중요한 과정 중 하나는 locking이다. 여러 스레드에서 공유 자원에 접근해 race condition이 발생할 수 있기 때문에 적절한 시기에 locking이 이루어져야 한다. exit() 함수 내에서 스레드를 종료할 때 자원을 해제하기 위해 현재 디렉토리의 참조를 해제하고 freethread()를 호출하도록 작성하며, 이때 안전한 해제를 위해 lock으로 감싸 보호한다. 디렉토리를 해제하는 과정은 로그시스템에서 트랜잭션의 경계를 명확히 하기 위해 begin_op()와 end_op()로 구분한다. 초기 코드에서는 exit()을 호출한 메인 스레드의 서브 스레드를

종료하기 위해, 모든 스레드를 순회하며 해당 스레드일시 스레드lock 이후 참조 디렉토리 해제, freethread()호출, 이후 lock을 풀도록 구상했다. 이 과정에서 스레드 lock 이후 end_op() 호출 시 스레드에 lock이 중복되는 현상이 있었고, end_op() 내부에서 wakeup() 함수 호출을 확인했다. wakeup() 함수는 모든 스레드를 순회하며 lock을 얻기 때문에 반드시 호출 전 어떠한 스레드lock도 존재해서는 안된다. 따라서 acquire(&t→lock); 의 순서를 freethread() 호출 직전으로 변경해 해결했다.