Project3 WIKI

Design

이번 과제에서는 간단한 pthread를 구현한다. pthread는 다른 lwp와 자원과 주소 공간 등을 공유한다. pthread를 구현하기 위해 thread_create, thread_exit, thread_join 총 3개의 시스템 콜을 새롭게 등록한다.

thread_create

새로운 thread를 만드는 함수이다. 새로운 thread를 생성할 때는 여러 가지를 초기화해야한다.

1. `allocproc`를 통해 새 프로세스 할당
2. 새 thread의 스택을 할당, 초기화한다. 이때 thread는 각자만의 스택 프레임을 가진다.
3. thread의 tid 등을 초기화한다. 이때 proc.h에서 proc의 구조체를 참고한다.
4. thread의 스택을 설정한다. 이때 start_routine을 설정한다.
5. 새로운 thread의 파일 디스크립터와 작업 디렉터리는 메인 thread와 공유한다.
6. 새 thread 상태를 `RUNNABLE`로 변경한다.

create함수는 기존에 있던 fork 함수와 exec 함수를 참고하여 작성한다.

thread_exit

thread를 종료하고 값을 반환한다.

1. thread의 모든 파일 디스크립터와 현재 작업 디렉터리 해제한다.
2. main thread가 종료될 때, 모든 자식 thread를 종료시킨다.
3. main 스레드 깨운다.

4. retval 값 반환하고 thread 종료시켜준다.

thread_exit 함수는 exit 함수를 참고한다.

thread_join

지정한 thread가 종료되기를 기다린다. 만약 thread가 이미 종료되었다면 즉시 반환한다.

thread가 종료된 후, thread에 할당된 자원들 (페이지 테이블, 메모리, 스택 등)을 회수하고 정리해야 한다. 기존의 wait 함수를 참고한다.

thread_join 함수와 thread_exit 함수의 관계를 알아보자.

thread_exit 함수가 호출되면, 현재 thread는 종료되고 zombie 상태로 전환된다.

wakeup1(main)을 호출하여, main thread를 깨우는데, main thread는 thread_join에서 종료된 thread를 기다리고 있을 수 있다.

wakeup1(main)으로 runnable한 상태가 된 main thread는 종료된 thread의 자원을 정리한다. 이때 retval 값을 회수한다.

proc.h의 proc 구조체

thread를 위한 변수들이 새롭게 선언되어야 한다.

그 외 함수들 수정: allocproc, exit, growproc

1. allocproc 함수 수정

   allocproc에서는 새로 생긴 thread를 초기화해준다. 이때 내가 추가한 proc 구조체 내의 변수들을 추가로 초기화해주도록 수정한다.

2. exit 함수 수정

   프로세스가 종료될 때, 모든 자식 thread를 종료하고 자원을 정리해야 한다.

3. growproc 함수 수정

   growproc 함수를 작성할 때, 현재 프로세스가 thread인지 여부에 따라 메모리 크기 변경을 다르게 처리해야 한다.

   - 현재 프로세스가 thread가 아닐 경우
     - 현재 프로세스의 크기(sz)를 사용해야 함
     - 메모리 크기를 업데이트한 후, 현 프로세스의 크기만 업데이트하면 된다.
   - 현재 프로세스가 thread일 경우
     - 현재 thread의 메인 thread의 크기를 사용해야 함
     - 메모리 크기를 업데이트한 후, 메인 thread의 모든 자식 thread의 크기 모두 업데이트해야함

Implement

proc.h

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for thi
  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)

  struct proc *main; // Main thread
  thread_t tid;  // Thread id
  void *retval;
  int thread_num;
  int is_thread;
};
```

마지막 5개의 자원이 thread 구현을 위해 추가한 변수들이다.

각각 main thread, thread의 tid, thread_exit와 thread_join에서 사용되는 retval, thread가 main thread일 때 자식 thread의 개수, thread인 지를 나타내는 is_thread이다.

참고로 thread_t는 types.h 파일에 정의해주었다.

```
typedef int thread_t; //for thread
```

thread_create

```
struct proc *curproc = myproc();
struct proc *new_proc;
struct proc *main = curproc->main;
uint sp, sz; //size of thread
pde_t *pgdir;
```

우선 변수들을 선언한다. curproc은 현재 thread이고, new_proc은 thread_create를 사용해 새로 생성할 thread이다. main은 현재 curproc의 main thread이다.

그 외 sp은 스택 포인터, sz은 size, pgdir은 페이지 디렉토리이다.

```
 if((new_proc = allocproc()) == 0) {
   cprintf("Failed : allocproc");
       return -1;
 }

 nextpid--;
```

새로 생성할 thread를 allocproc 함수를 사용해 생성한다. 만약 실패하면 오류를 출력하고 -1을 리턴한다.

그 후 nextpid를 1 감소시킨다. 그 이유는 allocproc에서 새로운 프로세스에게 pid를 할당시켜줄 때 겹치지 않도록 하기 위해서이다.

```
acquire(&ptable.lock);

 pgdir = main->pgdir;
 sz = main->sz;
 main->sz += 2*PGSIZE;


 // allocate user stack!!!
 sz = PGROUNDUP(sz);
 if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0) {
       kfree(new_proc->kstack);
               new_proc->kstack = 0;
               new_proc->state = UNUSED;
               cprintf("Failed : allocate stack to thread\n");
               release(&ptable.lock);
               return -1;
 }
 clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
 main->sz = sz; // change main thread's size
 //allocate user stack end!
```

```
  sp = main->sz;
  release(&ptable.lock);
```

acquire(&ptable.lock)을 사용해 락을 걸어준다. 그 후 pgdir과 sz를 메인 thread의 것으로 초기화해주고 main thread의 사이즈를 PGSIZE의 2배만큼 확장시켜 새로운 thread의 스택을 위한 메모리 공간을 만들어준다.

이제 새로운 thread에게 user stack 공간을 할당해준다. pgdir 페이지 디렉터리에서 sz 주소부터 sz + 2*PGSIZE까지의 메모리를 할당해준다. if문 내부는 할당이 실패했을 때이다.

clearpteu(pgdir, (char*)(sz - 2*PGSIZE)); 은 스택 오버플로우를 방지해준다.

할당이 성공하면 main thread의 sz를 이에 맞게 변경해주고 sp도 새롭게 초기화해준다.

마지막으로 락을 풀어준다.

```
// New thread initialize
 // new thread share lots of things with main thread
 new_proc->pgdir = main->pgdir;

 new_proc->tid = main->thread_num;
 main->thread_num++;
 *thread = new_proc->tid;

 new_proc->sz = main->sz;
 new_proc->main = curproc->main;
 new_proc->parent = curproc->parent;
 new_proc->pid = main->pid;
 *new_proc->tf = *main->tf;
 new_proc->is_thread = 1;
```

이제 new_proc의 자원들을 초기화해준다.

new_proc은 main thread와 pgdir, sz, trap frame 등을 공유한다. tid는 고유한 번호를 가지지만 pid는 main thread의 pid를 가진다. new_proc이 thread라는 것을 표시하기 위해 is_thread를 set해준다.

참고로 여기서 thread_create의 인자인 thread에 new_proc의 tid를 저장한다.

```
// set stack
uint ustack[2];
ustack[0] = 0xffffffff;
ustack[1] = (uint)arg;
sp -= 2*sizeof(uint);

if (copyout(pgdir, sp, ustack, 2*sizeof(uint)) < 0) {
    cprintf("Failed : copyout\n");
    main->sz -= 2*PGSIZE;
    return -1;
}

new_proc->tf->eax = 0;
new_proc->tf->eip = (uint)start_routine;
new_proc->tf->esp = sp;
```

이제 new_proc의 stack을 구체적으로 설정하고 초기화해준다.

ustack[0] = 0xffffffff; 이 부분은 함수가 종료될 때의 값이고, ustack[1]에는 thread_create의 인자로 받은 arg 값을 넣어준다. 그러면 start_routine 함수에 해당 인자가 전달될 것이다.

스택에 해당 두 값이 들어갔으므로 그만큼 스택 포인터를 옮겨준다. 그리고 copyout(pgdir, sp, ustack, 2*sizeof(uint))를 사용해 `ustack` 배열의 내용을 `sp` 위치에 복사한다.
그 다음에는 new_proc의 trap frame의 레지스터들을 초기화해준다.

eax에는 0을 넣어줘서 자식 thread임을 밝히고, eip에는 인자로 받은 start_routine을, esp는 값이 바뀐 sp를 넣어준다.

```
// share thread size with all threads
struct proc *p;
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if (p->pid == main->pid) { // child threads
              p->sz = sz;
```

```
                switchuvm(p);
        }
}
```

main thread의 사이즈가 바뀌었으므로 자식 thread들의 사이즈도 모두 바꿔줘야 한다.

```
// copy file descriptor
for (int i = 0; i < NOFILE; i++) {
        if (main->ofile[i])
            new_proc->ofile[i] = filedup(main->ofile[i]);
}
new_proc->cwd = idup(main->cwd);
```

new_proc의 file descriptor는 main thread의 것과 공유한다.

작업 디렉토리도 마찬가지이다.

```
safestrcpy(new_proc->name, main->name, sizeof(main->name));
switchuvm(curproc);


acquire(&ptable.lock);
new_proc->state = RUNNABLE;
release(&ptable.lock);

return 0;
```

마지막으로 new_proc의 이름을 main thread의 이름과 똑같게 설정하고 switchuvm을 사용해 사용자모드로 전환한다. 마지막으로 동기화를 한 상태에서 new_proc의 state을 runnable로 바꿔주고 끝낸다 .

thread_exit

```
struct proc *curproc = myproc();
int fd; //file descriptor
struct proc *main = curproc->main;
struct proc *p;

if (curproc == initproc) // prevent initproc from terminating.
    panic("init is exiting");
```

우선 curproc, main 프로세스를 초기화해준다. 그 외 fd는 후에 file descriptor를, p는 ptable 내의 프로세스를 가리킨다.

curproc이 initproc이면 종료시키면 안되므로 panic을 사용해 이를 방지한다.

```
// clean fd of curproc
for (fd = 0; fd < NOFILE; fd++) {
    if(curproc->ofile[fd]) {
        fileclose(curproc->ofile[fd]);
        curproc->ofile[fd] = 0;
    }
}

// clean cwd of curproc
begin_op();
iput(curproc->cwd);
end_op();
curproc->cwd = 0;
```

curproc의 file descriptor와 작업 디렉토리를 정리해준다.

```
// if curproc is main thread, terminate all child threads
  if (curproc == main) {
```

```
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if (p->main == curproc && p != curproc) { // child threads
        // clean fd of child threads
        for (fd = 0; fd < NOFILE; fd++) {
          if (p->ofile[fd]) {
            fileclose(p->ofile[fd]);
            p->ofile[fd] = 0;
          }
        }

        // clean cwd of child threads
        begin_op();
        iput(p->cwd);
        end_op();
        p->cwd = 0;


        p->state = ZOMBIE;
      }
    }
  }
```

curproc이 만약 main thread라면, curproc을 종료할 때 curproc의 자식 thread들도 모두 종료시켜줘야 한다. 그러므로 그때는 자식 thread들의 file descriptor와 작업 디렉토리도 정리해준다.

```
if (curproc != main) // if curproc is not main thread
    main->thread_num--;

  wakeup1(main);
```

그 후 curproc이 main thread가 아니면 main thread의 thread_num 을 1만큼 감소시켜주고, main thread를 wakeup1 함수를 사용해 깨워준다.

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

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

마지막으로 고아가 된 자식 프로세스들을 처리해준 후 curproc의 state는 zombie로, curproc의 retval에는 인자로 받은 retval을 저장해주고 스케쥴러로 넘긴다.

thread_join

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

acquire(&ptable.lock);
```

curproc과 p 두 프로세스를 초기화해주고 acquire(&ptable.lock)을 사용해 락을 획득한다.

```
for (;;) {
    int found = 0;
```

```
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->tid != thread)
            continue;

        found = 1;

        if (p->state == ZOMBIE) { // 자원 정리하기

            if (retval)
              *retval = p->retval;

            kfree(p->kstack);
        p->kstack = 0;

        p->pid = 0;
        p->tid = -1;
        p->parent = 0;
        p->name[0] = 0;
            p->sz = 0;
        p->killed = 0;
        p->state = UNUSED;
            p->main = 0;
            p->is_thread = 0;

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

    if(!found || curproc->killed) {
        release(&ptable.lock);
        return -1;
    }
```

```
    sleep(curproc, &ptable.lock);
  }
```

thread가 종료될 때까지 무한루프를 도는 데, 그 안에서 ptable을 순회하며 종료시켜야 할 thread를 찾는다. thread를 찾았다면 found 값을 1로 set해준 후, 해당 thread의 자원을 모두 정리해준다.

allocproc

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

  p->main = p; // make main thread
  p->thread_num = 1;
  p->tid = 0; // main thread's tid = 0
  p->is_thread = 0;
```

allocproc 내에서 수정한 부분이다. 처음 thread가 만들어질 때 자원들을 초기화해준다. allocproc으로 만들어지는 thread는 무조건 main thread이므로 p→main = p이고, main thread의 tid는 0으로 만들어주었다. is_thread는 thread_create 함수에서 allocproc이 호출되면 1로 set해준다.

growproc

이 함수 내에서는 thread일 때랑 프로세스일 때 두 가지 경우를 나눠서 하였다.

```
  uint sz;
  struct proc *curproc = myproc();
  struct proc *main = curproc->main;
  struct proc *p;
```

```
  acquire(&ptable.lock);
```

우선은 curproc, curproc의 main thread, p 총 세 개를 선언하였다. 그리고 동기화를 위해 lock을 걸어주었다.

```
if (curproc->is_thread) sz = main->sz;
else sz = curproc->sz;
```

curproc이 thread일 때는 size를 main thread의 size를 가지고 진행해야 하고, thread가 아닐 때는 curproc의 size로 진행하면 된다.

```
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;
                }
  }
```

이 부분은 기존 growproc에 있던 코드에서 안전한 동기화를 위해서 `release(&ptable.lock);` 만 추가해주었다.

```
  if (curproc->is_thread) { // when curproc is thread
        main->sz = sz;
        for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if (p->main == main)
                p->sz = main->sz;
        }
```

```
  }
  else curproc->sz = sz; // curproc is not thread
```

이제 curproc이 thread라면 아까 확장(축소)시킨 main thread의 size를 모두 반영해주어야 한다. 즉 자식 thread의 size도 모두 main thread의 size와 동일하게 만들어준다.

thread가 아니면 그냥 curproc의 size만 바꾸면 된다.

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

마지막으로 락을 풀어주고 curproc에 변경사항을 반영해주면 된다.

exit

```
// Exit all threads in the process
  acquire(&ptable.lock);
  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->tid = 0;
      p->parent = 0;
      p->name[0] = 0;
      p->killed = 0;
      p->state = UNUSED;
    }
  }
  release(&ptable.lock);
```

exit 함수 내에서 thread를 정리해주는 코드를 추가하였다.

프로세스가 종료될 때, ptable을 순회하며 curproc과 pid가 같은 모든 thread들의 자원을 정리하고 종료시켜준다. 그래야 정상적으로 해당 프로세스가 제거되게 된다.

```
if (curproc->main != curproc) {
        curproc->main->thread_num--;
 }
```

System Call 등록

thread_create, thread_exit, thread_join 총 3개의 시스템 콜을 등록해주었다.

또한 test 코드들 (thread_exec, thread_exit, thread_kill, thread_test)는 user program으로 등록해주었다.

Result

- build command

  make clean; && make; && make fs.img; && ./bootxv6.sh

  ```
  qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv
  ```

thread_exec

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ thread_exec
Thread exec test start
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Executing...
Hello, thread!
```

thread_exit

```
$ thread_exit
Thread exit test start
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Exiting...
```

thread_kill

```
$ thread_kill
Thread kill test start
Killing process 6
This code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 times.
Kill test finished
```

thread_test

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

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

All tests passed!
$
```

Trouble Shooting

1. thread_exit 문제

   thread_exit 을 실행했을 때 정상적으로 끝나지 않고 다시 재부팅되는 현상이 발생했었다. 이는 exit 함수에서 해당 프로세스와 같은 pid를 가진 thread들을 모두 종료시켜주니까 해결됐다.

2. thread_kill 문제

   thread_kill을 실행했을 때 acquire 문제가 자꾸만 발생했었다. 그래서 kill 함수에

```
int
kill(int pid)
{
  struct proc *p;
  int found = 0; //if there is thread

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
      found = 1;
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING)
        p->state = RUNNABLE;
      //release(&ptable.lock);
      //return 0;
    }
  }
  release(&ptable.lock);
  return found ? 0 : -1;
}
```

   이런 식으로 found 변수를 추가해주었더니 문제가 해결되었다. found 변수를 사용하면 조금 더 확실하게 동기화를 해줄 수 있어서 인 것 같다.

3. sbrk test trap 14 문제

   thread_test에서 마지막 sbrk에서 자꾸만 문제가 발생했었다. trap 14가 발생했다가 remap이 발생하는 등 다양한 오류가 있었는데, growproc에서 thread일 때와 아닐 때를 명확히 구별해서 코드를 작성하였더니 문제가 해결되었다.