Project 3 Wiki

Design

스레드 구현 방식

- linux의 pthread에는 두 가지의 thread 생성 방식이 존재합니다. 그것은 PCS(Process Contention Scope), SCS(System Contention Scope)인데 해당 구현은 SCS 방식으로 전체스레드가 스케쥴링을 하기 위해서 경쟁하는 형태를 취할 것입니다.

수정/변경 사항

- **proc 구조체**
  - 해당 프로젝트의 thread 구현은 기존 xv6의 process 구조체를 거의 그대로 사용하되 스레드 마다의 구분이 필요하기 때문에 스레드 아이디가 추가될 것이며 각각의 stack 영역을 가지면서 모든 스레드가 해당 stack 영역을 확인할 수 있어야하므로 pgdir에서 자신의 user stack을 가리키는 stack 포인터를 추가할 것입니다. 그리고 thread 결과를 저장하기 위한 변수가 하나 추가될 것입니다.
- **proc 구조체간의 상호작용**
  - system call에 의해서 하나의 thread가 영향을 받게 되면 ptable(=thread table)을 탐색해가면서 같은 pid를 가지는 thread들을 찾아서 명세에 맞게 작동할 수 있도록 구현할 것입니다.

Implement

Structure

- 기존 xv6의 ptable 구조체를 그대로 이용하며 스레드 구분을 위한 tid, 스레드 각각의 ustack, thread 결과를 저장하기 위한 retval이 존재합니다. 그리고 process를 모두 하나의 thread로 보고 ptable에 넣기 때문에 기존의 64개 제한은 부족하다고 생각하여 최대 동시에 256개의 thread가 작동할 수 있도록 수정하였습니다.

```
//proc.h
struct proc {
    ...
    thread_t tid;               //thread_t는 types.h에 정의되어 있으며 이는 int와 같습니다.
    ...
  uint *ustack;               // End of user stack for this thread
    ...
  void* retval;               // thread return
};

struct ptable_t{
  struct spinlock lock;
  struct proc thread[NPROC];   // 64-> 256
} ptable;

extern struct ptable_t ptable;
```

Thread API

- **int thread_create(thread_t* thread, void*(*start_routine)(void), void* arg)**
  - thread create는 크게 kernel stack 할당, user stack 할당, parent와 같은 내용을 복사, arg와 return 설정, thread 시작전 설정으로 이루어져 있습니다.
    - user stack 할당
      - 만약 이전에 할당된 공간이 있다면 해당 공간을 pgdir로 사용하고 아니면 새로 PGSIZE만큼만 늘려서 stack으로 할당합니다.
    - arg와 return 설정
      - 모든 스레드는 thread_exit을 통해서만 종료가 가능하므로 thread가 실행할 함수를 모두 마치면 thread_wait이라는 함수로 return하여 ZOMBIE 상태로 변경합니다.

```
//proc.c
void thread_wait(){
  struct proc* curthread=myproc();
  acquire(&ptable.lock);

  wakeup1((void*)curthread->tid);
  curthread->state=ZOMBIE;
  sched();
  panic("zombie thread");
}

int thread_create(thread_t* thread, void*(*start_routine)(void*), void *arg){
  struct proc *t;
  char *sp;
  acquire(&ptable.lock);
  struct proc* curthread=myproc();
  for(t = ptable.thread; t < &ptable.thread[NPROC]; t++)
    if(t->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return -1;

found:
  // Allocate kernel stack.
  if((t->kstack = kalloc()) == 0){
    t->state = UNUSED;
    release(&ptable.lock);
    return -1;
  }

// Allocate User stack
  uint sz=curthread->sz;
  int flag=0;
  for(struct proc* thread=ptable.thread; thread<&ptable.thread[NPROC]; thread++){
    if(thread->pid==curthread->pid && thread->state==UNUSED){
      t->ustack=thread->ustack;
      flag=1;
      break;
    }
  }
  if(flag==0){
    sz=PGROUNDUP(sz);
    if((sz=allocuvm(curthread->pgdir, sz, sz+PGSIZE))==0){
      kfree(t->kstack);
      t->kstack=0;
      t->state=UNUSED;
      release(&ptable.lock);
      return -1;
    }
    for(struct proc* thread=ptable.thread; thread<&ptable.thread[NPROC]; thread++){
      if(thread->pid==curthread->pid && thread->state!=UNUSED){
        thread->sz=sz;
      }
    }
    t->ustack=(uint*)sz;
  }
```

```
  t->pgdir=curthread->pgdir;
  t->sz=sz;  //Copy same part of parent process
  t->parent=curthread->parent;
  for(int i=0; i<NOFILE; i++)
    if(curthread->ofile[i])
      t->ofile[i]=filedup(curthread->ofile[i]);
  t->cwd=idup(curthread->cwd);
  safestrcpy(t->name, curthread->name, sizeof(curthread->name));

  // Set up kernel part of stack.
  sp = t->kstack + KSTACKSIZE;

  // Leave room for trap frame. & Copy
  sp -= sizeof *t->tf;
  t->tf = (struct trapframe*)sp;
  *t->tf = *curthread->tf;

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

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

  //Set up user part of stack
  //push argument(syscall의 fetchint와 syscall 참조)
  sp=(char*)sz;
  sp-=4;
  (*(uint*)sp)=(uint)arg;
  sp-=4;
  (*(uint*)sp)=(uint)thread_wait; //항상 함수 호출전에 return address가 먼저 stack에 들어간다.

  //push return address
  t->tf->eip=(uint)start_routine;
  t->tf->esp=(uint)sp;

  // Set up thread
  t->state = RUNNABLE;
  t->pid = curthread->pid;
  t->tid= nexttid++;
  t->retval=0;

  (*thread)=t->tid;
  release(&ptable.lock);
  return 0;
}
```

- **void thread_exit(void * retval)**
  - join에서 retval을 받아가기 위해서 구초체의 retval에 저장하고 ZOMBIE 상태 변경됩니다.

```
//proc.c
void thread_exit(void* retval){
  struct proc* curthread=myproc();
  curthread->retval=retval;
```

```
  thread_wait();
}
```

- **int thread_join(thread_t thread, void** retval)**
  - 종료할 스레드가 ZOMBIE가 아니면 기다리고 thread_wait에서 깨워주는 것을 기다립니다. 이후 깨어나면 target 스레드에 할당되었던 데이터들을 해제합니다. 이때 thread_create에서 이전의 할당된 메모리를 확인하기 위한 최소한의 정보만을 남겨둡니다.

```
//proc.c
int thread_join(thread_t thread, void** retval){
  struct proc* target;
  acquire(&ptable.lock);
  for(target=ptable.thread;target<&ptable.thread[NPROC];target++){
    if(target->tid==thread)
      goto found;
  }
  release(&ptable.lock);
  return -1;

found:
  if(target->state!=ZOMBIE)
    sleep((void*)thread, &ptable.lock);
  *retval=target->retval;
  kfree(target->kstack);
   target->kstack=0;
  target->state=UNUSED;
  target->sz=0;
  target->parent=0;
  target->tid=0;
  target->name[0]=0;
  target->killed=0;
  release(&ptable.lock);
  release(&ptable.lock);
  return 0;
}
```

System Call

process를 thread로 바꾸면서 생겨난 system call 들의 변경점들은 아래와 같습니다.

- static struct proc* allocproc(void)

```
//proc.c
static struct proc*
allocproc(void){
...
p->tid= nexttid++;
p->ustack=0;
p->pgdir = 0;
...
}
```

- void userinit(void)

```
//proc.c
void
userinit(void)
{
    ...
```

```
    p->ustack=(uint*)PGSIZE;
    ...
}
```

- int growproc(int n)
  - 전체 테이블을 탐색하면서 모든 thread의 sz를 늘려주는 코드가 추가되었습니다.

```
//proc.c
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){
      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;
  for(struct proc* thread=ptable.thread; thread<&ptable.thread[NPROC]; thread++){
    if(thread->pid==curproc->pid && thread->state!=UNUSED){
      thread->sz=sz;
    }
  }
  release(&ptable.lock);
  switchuvm(curproc);
  return 0;
}
```

- void exit(void)
  - 다른 스레드들도 zombie 처리하는 코드가 추가되었습니다.

```
//proc.c
void
exit(void)
{
    ...
    // Jump into the scheduler, never to return.
  for(p = ptable.thread; p < &ptable.thread[NPROC]; p++){
    if(p->pid==curproc->pid && p->state!=UNUSED){
      p->state=ZOMBIE;
    }
  }
  ...
}
```

- int wait(void)
  - 모든 스레드가 ZOMBIE, 즉 프로세스가 ZOMBIE 상태일때만 pgdir을 해제하기 위해서 ptable을 탐색하는 코드가 추가되었습니다.

```
//proc.c
...
 pid=0;
for(p = ptable.thread; p < &ptable.thread[NPROC]; p++){
  if(p->parent != curproc)
    continue;
  havekids = 1;
  if(p->state == ZOMBIE){
    // Found one.
    pid = p->pid;
    break;
  }
}
int flag=1;
for(p=ptable.thread;p<&ptable.thread[NPROC];p++){
  if(p->parent!=curproc) continue;
  flag=flag && (p->state==ZOMBIE);
}
if(flag && pid!=0 && havekids){ //zombie process
  int free_pd=0;
  for(p=ptable.thread;p<&ptable.thread[NPROC];p++){
    if(p->parent!=curproc) continue;
    if(p->state==ZOMBIE){
      kfree(p->kstack);
      p->kstack = 0;
      p->ustack=0;
      if(!free_pd){
        freevm(p->pgdir);
        free_pd=1;
      }
      p->sz=0;
      p->pid = 0;
      p->parent = 0;
      p->name[0] = 0;
      p->killed = 0;
      p->state = UNUSED;
    }
  }
  release(&ptable.lock);
    return pid;
  }
  ...
```

- int exec(char* path, char** argv)
  - exec시 하나의 스레드 말고 다른 스레드를 전부 종료시키기 위해서 초기화를 진행하는 코드가 아래와 같이 추가되었습니다.

```
//exec.c
...
 acquire(&ptable.lock);
  //terminate the other threads
  for(i=0; i<NPROC; i++){
    if(ptable.thread[i].state != UNUSED && ptable.thread[i].tid != curproc->tid && ptable.thr
      kfree(ptable.thread[i].kstack);
      ptable.thread[i].kstack = 0;
      ptable.thread[i].ustack=0;
      ptable.thread[i].state = UNUSED;
```

```
      ptable.thread[i].pid = 0;
      ptable.thread[i].tid = 0;
      ptable.thread[i].chan = 0;
      ptable.thread[i].retval = 0;
      ptable.thread[i].parent = 0;
      ptable.thread[i].cwd = 0;
      ptable.thread[i].sz = 0;
      ptable.thread[i].parent=0;
      ptable.thread[i].killed=0;
      ptable.thread[i].name[0]=0;

    }
  }
  release(&ptable.lock);
  ...
```

Result

Given Test

- thread_test.c

- thread_exec.c

```
$ thread_exec
Thread exec test start
Thread 0 start
Thread 1 sThread 2 start
Thread 3 start
Thread 4 start
tart
Executing...
Hello, thread!
```

- 실행되지 않아야할 코드가 실행되지 않고 정상적으로 종료됨을 알 수 있습니다.

```
$ thread_test
Test 1: Basic test
Thread 0 start
Thread 0 end
ThThread 2 start
Thread 2 end
read 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
Child of thread 0 start
Child of thread 2 start
Thread 4 starChild of thread 1 start
Chit
ld ofChild of thread 4 start
 thread 3 start
Child of thread 0 end
Thread 0 end
Child of thread 2 end
Thread 2 end
Child of thread 1 end
Thread 1 end
Child of thread 3 end
Child of thread 4 end
Thread 4 end
Thread 3 end
Test 2 passed

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

All tests passed!
```

- 모두 이상없이 테스트를 통과함을 알 수 있습니다.

- thread_kill.c

```
$ thread_kill
Thread kill test start
Killing process 11
ThThis code shoulThis code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 times.
is code should be execud be executed 5 times.
ted 5 times.
Kill test finished
```

- code가 5번 실행이 된 것으로 보아 테스트를 통과함을 알 수 있습니다.

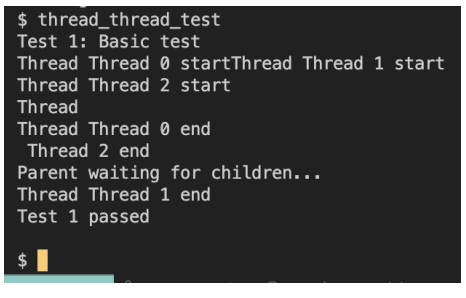
- thread_exit.c

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

- 실행되지 않아야할 코드가 실행되지 않고 정상적으로 종료됨을 알 수 있습니다.

Personal Test

- thread_thread_test.c
  - thread_test의 1번 테스트를 변형한 것으로 스레드가 스레드를 생성할 수 있음을 테스트합니다.



- 이상없이 테스트가 수행된 것을 알 수 있습니다.

Trouble Shooting

- thread_test에서 테스트를 모두 통과하였지만 종료가 되지 않던 문제
  - 원인
    - 기존의 fork에서는 pid를 먼저 process 구조체에 넣어두고 밑에 작업을 진행하는데 thread_create에서는 맨 밑에 thread에 대한 값을 넣어주는데 이로 인해서 위에 있던 pid로 필터링되는 loop문이 현재 생성중인 스레드에게 영향을 주지 않아서 생겼던 문제였습니다.
  - 해결방법
    - loop문을 점검하여 앞뒤로 업데이트해야될 변수들을 바로바로 업데이트 해주는 방식으로 코드를 작성하여 해결하였습니다.