Project3

2018008877 이상원

Project3 Light Weight Process (thread)

xv6의 process는 single thread로 동작하기 때문에 process를 위한 stack 할당, scheduling으로 인한 context switching 등을 process 단위로만 진행하였다. 하지만 single thread로만 동작하게 되면 scheduler가 동작할 때마다 context switching overhead를 감수해야만 하고, 모든 process는 각각의 독립된 메모리 영역을 할당받았기 때문에 process 간의 통신이 어렵고 복잡해진다. 하지만 thread를 사용하게 되면 process의 단점들을 어느 정도 극복할 수 있게 된다. 하나의 process에서 생성된 thread들은 code, data, heap을 공유하고 각각 stack만 다르게 할당받는다. 이로 인해 context switching의 cost가 저렴해지고, thread간의 통신이 간단해 진다. 주의해야 할 점은 data segment를 공유하기 때문에 data synchronization을 해치지 않도록 해야한다는 것이다. 이번 과제는 이 Light Weight Process를 구현하는 것이다.

Design

proc.h

가장 먼저, `proc` 구조체를 변경하고 새롭게 thread를 관리할 수 있는 구조체를 정의해 주어야한다. 기본적으로 각 process 하나 당 32개의 thread를 담을 수 있는 배열을 가지도록 만들 것이고, 처음 프로세스가 만들어질 때는 thread 배열의 0 번째 index에 main thread를 만들려고 한다.

```
struct proc {
  uint sz;
  pde_t *pgdir;
  char *kstack;
  enum procstate state;
  int pid;
  struct proc *parent;
  struct trapframe *tf;
  struct context *context;
  void *chan;
  int killed;
  struct file *ofile[NOFILE];
  struct inode *cwd;
  char name[16];
};
```

기존 `proc` 구조체이다. single thread로 동작하기 때문에 kernel stack의 주소가 담겨있는 `kstack` 과 trap frame에 관한 정보가 담겨져 있는 `tf`, 그리고 context switching 을 위해 필요한 `context` 와 sleep을 위한 `chan`과 같은 정보들이 모두 하나의 process 구조체에 다 담겨있는 것을 확인 할 수 있다. 하지만 이 정보들은 모두 thread 각각 관리해야 하는 것들이다. 따라서 이 정보들은 thread 구조체를 새롭게 정의하여 thread 각각에서 관리할 수 있도록 할 것이다. 또한 process 하나 당 자신의 thread 들을 배열로 갖고 있고, 현재 동작하고 있는 thread가 어떤 thread인 지 알 수 있는 정보를 `proc` 구조체에 새롭게 추가해야한다.

exec.c

현재 process를 path로 들어온 program을 실행하도록 하는 system call이다. program을 memory로 가져오고, page를 할당받아서 user stack을 만들어주고, thread→tf→eip 를 해당 프로그램의 main 함수로 바꾼다. 하지만 이제 thread로 동작하기 때문에 현재 process의 current thread를 main thread로 바꾸어주고, 새로운 프로그램이 실행된 것이기 때문에 다른 모든 thread들은 초기화 해주는 작업이 필요하다.

proc.c

allocproc(void)

`allocproc(void)` 는 기존에 `ptable` 에 UNUSED 상태인 `proc` 을 찾아서 EMBRYO 상태로 만든 후 `kstack`, `tf`, `context` 를 할당해 주고 해당 `proc` 의 주소를 반환해 주는 함수이다. 하지만, 이젠 proc 구조체는 process의 정보와 thread가 공유하는 data와 현재 동작하는 thread의 정보만 유지하면 되므로 main thread를 새롭게 할당하는 것으로 바꾸어 줄 것이다.

growproc(void)

`growproc(int n)` 은 현재 process의 memory를 n bytes 만큼 키우는 함수이다. 여러 thread가 동시에 호출하게 되면 문제가 생길 수 있기 때문에 시작과 끝에 `ptable.lock` 을 획득하고 풀어주는 작업이 필요하다.

fork(void)

`fork(void)` 는 이 system call 을 호출한 process를 parent process로 하는 process를 새롭게 만드는 system call이다. parent process의 모든 data를 복사한다. 이제는 새롭게 만든 process의 main thread에 parent process의 current thread의 trap frame을 복사해 주어야한다.

exit(void)

`exit(void)` 는 현재 process의 상태를 ZOMBIE 로 만들어주는 함수이다. 만약 특정 thread의 작업중 exit system call이 호출된다면 모든 thread를 ZOMBIE 로 만들어주도록

해야 한다.

wait(void)

`wait(void)` 는 해당 process의 child process 중 ZOMBIE 상태인 process가 있으면 UNUSED 상태로 바꾸고 할당되어 있는 것들을 해제하는 함수이다. 이제 exit 함수에서 ZOMBIE 상태로 바뀐 thread들을 UNUSED 상태로 바꾸는 작업이 필요하다.

scheduler(void)

timer interruption 이 발생하면 다음 process 로 scheduling 되는 함수이다. 이젠 해당 process 의 current thread 다음의 thread가 실행되어야 하고, 해당 process의 thread를 한 바퀴 모두 탐색했다면, 다음 process로 넘어가 같은 작업을 반복하는 작업이 필요하다.

yield(void)

현재 process 의 작업을 포기하고 다음 process 로 scheduling 되도록 하는 system call 이다. current thread의 state를 RUNNABLE 로 바꾸는 작업이 필요하다.

sleep(void *chan, struct spinlock *lk)

process의 상태를 SLEEPING 으로 바꾸는 것이 아닌 current thread의 상태를 SLEEPING 으로 바꾸어 주는 작업이 필요하다.

wakeup1(void *chan)

`chan` 에서 자고 있는 모든 thread 들을 깨워주어야 한다.

kill(int pid)

해당 process 를 kill 하는 system call로 현재 process의 상태가 SLEEPING 이라면 RUNNABLE을 통해 scheduling 될 수 있도록 하고, 이후 trap.c 에서 `exit()` 을 통해 process의 상태가 ZOMBIE로 바뀌게 된다. 이제 해당 process의 thread들 중 어떤 것이라도 scheduling 된다면 해당 process의 상태를 ZOMBIE로 바꿀 수 있도록 해야한다.

thread_create(thread_t *thread, void *start_routine, void *arg)

새롭게 추가 되어야 하는 system call 이다. 현재 process 에서 새롭게 thread를 만든다. `*thread` 에는 새롭게 생성된 thread의 id를 담아주어야 하고, `*start_routine` 은 해당 thread가 실행해야할 routine이다. 해당 routine을 실행하기 위해 필요한 argument는 `*arg` 에 담겨 온다. 전체적인 구조는 `exec()` 과 비슷하게 할 것이지만, 다른 점이 있다면 `thread→tf→eip` 를 `start_routine`의 주소로 바꾸어 주어야 한다는 것이다.

thread_exit(void *retval)

새롭게 추가 되어야 하는 system call 이다. current thread를 ZOMBIE 상태로 만드는 system call 이다.

thread_join(thread_t thread, void **retval)

새롭게 추가 되어야 하는 system call 이다. 해당하는 thread 가 종료될 때 까지 기다려야 하는 system call로 `thread_exit()` 을 통해 이미 종료가 되었다면 thread의 상태를 UNUSED로 만들어 주어야 한다. 전체적인 구조는 `wait()` 와 비슷하게 구현할 것이다.

Implementation

proc.h

`kstack`, `tf`, `context`, `chan` 은 새롭게 추가된 `thd` 구조체에 추가되었다. `thd` 구조체는 하나하나를 thread 한 개라고 생각한다. `tid` 는 자신의 고유한 숫자이고, `state` 는 현재 thread의 상태를 나타낸다. `proc` 구조체에는 새롭게 현재 자신의 thread 들 중 실행중인 thread의 id가 담겨 있는 `tid`, `thd` 구조체의 배열인 `thds`가 추가되었다.

exec.c

```
int
exec(char *path, char **argv)
{
  ...
#if !defined(MULTILEVEL_SCHED) && !defined(MLFQ_SCHED)
  *MAINTHD(curproc) = *CURTHD(curproc);
  if(curproc->tid > 0)
    CURTHD(curproc)->kstack = 0;
  for(i = 1; i != NTHREAD; i++){
    t = THDADDR(curproc, i);
    if (t->kstack)
      kfree(t->kstack);
    t->kstack = 0;
    t->state = UNUSED;
    t->tid = 0;
    t->retval = 0;
  }
  MAINTHD(curproc)->tf->eip = elf.entry;
  MAINTHD(curproc)->tf->esp = sp;
  curproc->tid = 0;
#else
  ...
}
```

1. current thread를 main thread로 바꾸어준다.
2. main thread 를 제외한 모든 thread를 초기화 시켜준다.
3. main thread의 trap frame의 eip를 실행하고자 하는 프로그램의 main 주소로 바꾼다.

proc.c

allocproc(void)

```
#if !defined(MULTILEVEL_SCHED) && !defined(MLFQ_SCHED)
  t = MAINTHD(p);
  if(!(t->kstack = kalloc())){
    ...
    return 0;
  }
  sp = t->kstack + KSTACKSIZE;
  sp -= sizeof *(t->tf);
  t->tf = (struct trapframe *)sp;
  sp -= 4;
  *(uint *)sp = (uint)trapret;
#endif
```

위와 같이 새롭게 만들어진 process의 main thread를 만들어주는 작업으로 변화하였다.

fork(void)

```
#else
  ...
  struct thd *main_thd = MAINTHD(np);
  ...
  *(main_thd->tf) = *(CURTHD(curproc)->tf);
  ...
#endif
...
```

기존의 parent process의 trap frame의 child process의 trap frame으로 복사하는 방법에서 child main thread의 trap frame에 parent process의 current thread의 trap frame을 복사해주는 작업으로 바뀌었다.

exit(void)

```
...
#if !defined(MULTILEVEL_SCHED) && !defined(MLFQ_SCHED)
  for(t = MAINTHD(curproc); t != THDADDR(curproc, NTHREAD); t++){
    if(t->state != UNUSED)
      t->state = ZOMBIE;
  }
#endif
...
```

모든 thread들의 상태를 ZOMBIE 로 바꾸어주는 부분이 추가되었다. 이렇게 되면 나중에 `wait()` system call에서 자원이 모두 초기화 된다.

wait(void)

```
...
for(;;){
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
#if !defined(MULTILEVEL_SCHED) && !defined(MLFQ_SCHED)
        for(i = 0; i < NTHREAD; i++){
          t = THDADDR(p, i);
          t->tid = 0;
          t->state = UNUSED;
          if(t->kstack) {
            kfree(t->kstack);
            t->kstack = 0;
          }
        }
#else
    ...
    }
}
...
```

thread의 자원을 해제해 주는 과정이 추가되었다.

scheduler(void)

```
#else
    struct thd *t;
    int infinity;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      infinity = 0;
      if(p->state != RUNNABLE)
        continue;
      for(t = CURTHD(p); ; t++){
        if(t == THDADDR(p, NTHREAD))
          t = MAINTHD(p);
        if(t->state == RUNNABLE){
          p->tid = t - p->thds;
          c->proc = p;
          switchuvm(p);
          t->state = RUNNING;
          swtch(&(c->scheduler), t->context);
          switchkvm();
          c->proc = 0;
        }
        if (infinity && t == CURTHD(p))
          break;
        infinity = 1;
      }
    }
#endif
```

Round Robin 방식을 사용했다. 다만, process의 모든 thread들이 한번씩 scheduling 되었다는 것을 infinity 변수를 통해 확인한 뒤에 다음 process로 넘어가는 방법으로 바뀌었다.

sleep(void *chan, struct spinlock *lk)

```
...
#if !defined(MULTILEVEL_SCHED) && !defined(MLFQ_SCHED)
  struct thd *t = CURTHD(p);
  t->chan = chan;
  t->state = SLEEPING;
  sched();
  t->chan = 0;
#else
...
```

process 전체가 아닌 current thread만 SLEEPING 상태로 바뀐다. 따라서 해당 thread가 자더라도 다른 thread들은 마저 일을 진행할 수 있게 된다.

wakeup1(void *chan)

해당하는 `chan`을 가진 thread가 자고있다면 RUNNABLE 상태로 바꾸어주는 작업으로 바뀌었다.

kill(int pid)

`process→killed`를 1로 바꾼 후에 해당 process의 thread들 중 하나라도 scheduling 될 수 있도록 바꾸어서 *trap.c* 에서 `exit()` 을 호출하도록 유도하였다.

thread_create(thread_t *thread, void *start_routine, void *arg)

전체적인 구조는 exec과 유사하게 하였고, 다른 점은 design 한 것과 같이 `thread→tf→eip`를 `start_routine`으로 설정한 것이다.

thread_exit(void *retval)

현재 process의 current thread의 상태를 ZOMBIE 로 바꾸어준다.

thread_join(thread_t thread, void **retval)

해당하는 id를 가진 thread를 찾는다면, 해당 thread의 상태가 ZOMBIE 가 아닐 땐, sleep() 을 호출하고, 그렇지 않다면 thread가 종료 되었다는 뜻이기 때문에 return value를 저장한 후 thread를 UNUSED 상태로 바꾸어준다.

Result

thread_exec.c && thread_exit.c && thread_kill.c

```
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 test start
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Exiting...
```

```
$ 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_exec 은 정상적으로 Hello, thread! 를 출력하는 것을 확인할 수 있고, thread_exit 도 문제없이 종료되는 것을 확인할 수 있다. 또한 thread_kill 도 실행되어야 할 코드가 5번 출력되는 것을 확인할 수 있다.

thread_test_.c

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

      All tests passed!
```

```
Test 2: Fork test
Thread 0 start
Child of thread 0 start
Thread 1 start
Child of thread 1 start
Thread 2 start
Child of thread 2 start
Thread 3 start
Child of thread 3 start
Thread 4 start
Child of thread 4 start
Child of thread 0 end
Thread 4 end
Child of thread 1 end
Thread 3 end
Child of thread 2 end
Thread 2 end
Child of thread 3 end
Thread 1 end
Child of thread 4 end
Thread 0 end
Test 2 passed
```

모든 test가 정상적으로 종료되 것을 확인할 수 있다.

Trouble Shooting

처음 구상을 떠올리는 것이 가장 어려웠다. `thread_create`, `thread_join`, `thread_exit` system call 을 새롭게 구현해야 한다는 것이 가장 어려웠지만 기존 xv6 함수들을 참고하면서 만들 수 있었다.