Operating Systems Project 3 Wiki

2018008904 이성진

Contents

1. Design

- xv6는 기본적으로 프로세스 단위로 작동됩니다. 각각의 프로세스는 현재 실행중인 프로그램을 나타내는 것이라고 할 수 있으며, 해당 프로그램의 코드와 데이터 및 현재 사용중인 스택이나 힙 등이 포함됩니다.

- 스레드란 프로세스와 유사하게 현재 실행중인 프로그램이라고 할 수 있지만, 프로세스에 속해 있습니다. 스레드는 프로세스와 다르게 코드, 데이터, 파일 등 일부분의 데이터를 각각 할당받지 않고 공유하여 사용하는 LWP라고 할 수 있습니다. 각각의 프로세스는 여러 스레드들을 포함할 수 있습니다.

- 각각의 프로세스는 fork, exit, wait 등의 system call을 통해 유저 레벨에서 새로운 스레드를 생성하고, 제거하고, 자식 스레드의 리턴값을 받아 사용할 수 있습니다. 이와 유사하게, 유저 레벨에서 API들을 사용하여 스레드를 생성, 제거하는 등의 실행 흐름을 제어할 수 있고, 이를 통해 유저 프로그램에서 스레드를 사용할 수 있습니다.

- 유저 레벨 스레드 API는 새로운 스레드를 생성하는 thread_create, 현재 스레드의 실행을 종료하고 값을 반환하는 thread_exit, 그리고 이 스레드가 반환한 값을 받아올 때 까지 기다도록 하는 thread_join가 있습니다.

- 스레드를 실제로 구현하기 위해, xv6에서 프로세스를 관리하는 proc 구조체에 새로운 속성을 추가할 수 있습니다. 스레드는 xv6에서 프로세스와 유사하게 스케줄링되고, 메모리를 추가로 할당받거나 새로운 프로세스 또는 스레드를 fork할 수 있어야 하기 때문에 이러한 방식으로 구현하는 것이 더 효율적이라고 생각했습니다.

- 스레드가 생성되고 실행하는 과정에서 기존 프로세스의 실행 흐름을 유사하게 따라가지만, 일부 다른 점도 존재합니다. 이를 위해서 fork, exit, wait, kill 등의 시스템 콜이 기존 프로세스 실행 흐름 뿐만 아니라 새로운 스레드 흐름도 원활하게 제어할 수 있도록 수정할 필요성이 생겼습니다.

2. Implement

Per-thread struct implementation

- 아래 나와있는 모습처럼 proc 구조체에 새로운 멤버 변수를 추가하여 스레드 단위로 동작 가능한 시스템을 구현했습니다.

- tid(thread ID)는 fork()로 생성된 프로세스의 메인 스레드인지 아니면 thread_create ()로 생성된 스레드인지를 구분하고 각각의 스레드를 구별하기 위해 사용됩니다.

- fork()를 통해 만들어진 프로세스 구조체는 해당 프로세스를 나타내면서 동시에 해당 프로세스의 메인 스레드를 나타내게 되고 다른 스레드는 이 구조체의 많은 데이터(코드, 데이터 등)를 공유하게 됩니다. 각각의 스레드가 메인 스레드에 접근하기 위한 포인터 main을 정의했습니다.

- 각각의 스레드는 return하지 않고 thread_exit()과 thread_join()을 통해 값을 전달합니다. 실행 과정에서 thread_exit()이 호출된 이후 값을 저장하기 위한 retval을 정의했습니다.

- sbase는 하나의 프로세스가 사용하는 메모리 공간 위에 서로 다른 스레드가 사용하는 스택 영역의 위치를 저장하기 위해 사용됩니다.

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

  thread_t tid;                // Thread ID (0 if main thread)
  struct proc* main;           // Main thread of current process (0 if main thread)
  void* retval;                // Temporary save return value
  uint sbase;                  // Base address for the stack of a new thread

#ifdef MLFQ_K
  int priority;                // Priority of each process
  int level;                   // MLFQ queue level
  int extime;                  // Time passed during execution
#endif
};
```

Thread creation(thread_create)

- 유저 레벨 API thread_create를 호출하면 현재 프로세스의 데이터를 일부 공유하는 새로운 스레드가 만들어지고, 파라미터로 사용한 start_routine의 위치에서 시작하여 thread_exit가 호출될 때 까지 실행하게 됩니다.

- thread_create은 fork 함수와 작동 방식이 유사하여 fork 함수의 일부를 수정하여 만들었습니다.

- fork와 다른 점으로는 새로운 프로세스가 생성되는 것이 아니므로 allocproc에서 증가된 pid를 다시 1 감소시키고, 현재 프로세스의 pid와 같게 해준 뒤 pid 할당 방식과 유사하게 새로운 스레드를 위한 tid를 할당해 준 뒤 메인 스레드와 연결합니다.

- 또한 기존 페이지 디렉토리를 복사한 새로운 페이지 디렉토리를 사용하는 것이 아니라 메인 스레드가 사용하는 페이지 디렉토리를 그대로 사용하여 메모리 공간을 공유하게 됩니다.

- 각각의 스레드는 프로세스와 유저 스택은 공유하지 않기 때문에, exec 함수에서 나온 부분과 유사하게 추가로 메모리를 할당받아 새로운 스레드를 위한 스택을 정의하고 주소를 sbase에 저장해둡니다. 그리고 그 안에 thread_create의 인자로 들어온 arg를 위치하여 start_routine에서 사용할 수 있게 합니다.

- thread_create가 끝나고 난 뒤 start_routine을 실행하고 새로운 유저 스택을 사용할 수 있도록, trapframe의 eip와 esp에 해당하는 값을 적절하게 저장해 줍니다.

- 이후 만들어진 스레드는 스레드 번호를 주어진 위치에 저장하여 join을 대비하고 기존의 프로세스와 유사하게 작동하게 실행되고, 스케줄됩니다.

Thread terminating(thread_exit)

- 유저 레벨 API thread_exit를 호출하면 현재 실행중이던 스레드가 종료된다는 것을 의미합니다. 이 함수가 호출되면 현재 스레드는 반환할 값을 proc 구조체 안에 저장한 뒤 좀비 상태가 되어 thread_join을 통해 수거되기를 기다리게 됩니다.

- thread_exit은 exit 함수와 작동 방식이 유사하여 exit 함수의 일부를 수정하여 만들었습니다. exit와는 다르게 자식 프로세스가 없으므로 따로 관리해주지 않고, 부모 프로세스 대신 메인 스레드를 wakeup하여 thead_join을 통해 반환값을 받아갈 수 있게 했습니다.

Thread waiting(thread_join)

- 유저 레벨 API thread_join를 호출하면 주어진 스레드 id에 해당하는 스레드가 종료될 때까지 기다린 후, 종료되면 해당 스레드가 보관하던 retval을 정해진 위치에 저장합니다.

- thread_join은 wait 함수와 작동 방식이 유사하여 wait 함수의 일부를 수정하여 만들었습니다. wait과는 다르게 스레드만 정리하는 것이기 때문에 pgdir 전체를 free해서는 안됩니다. 다른 스레드에서 사용중일 수 있기 때문입니다. 스레드의 자원 회수를 위한 함수는 다른 곳에서 사용되어서 따로 thread_clear라는 이름으로 정의했습니다.

```
// Clear given thread.
// Free kstack and change state into UNUSED
// Needed ptable lock before calling
void thread_clear(struct proc* p){
  // Do not free pgdir because it is shared with other threads.
  // freevm(p->pgdir);
  kfree(p->kstack);
  p->kstack = 0;
  p->pid = 0;
  p->tid = 0;
  p->main = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->killed = 0;
  p->state = UNUSED;
}
```

System call compatibility

- 새로 정의한 스레드는 기존 프로세스 방식과 매우 유사하게 실행되고 스케줄링되며 정상적으로 실행됩니다. 프로세스와는 다르게 pid가 같은 스레드 구조체들은 전체 메모리 공간을 공유하면서 자신만의 유저 스택 영역을 가지고 있게 됩니다.
- 프로세스의 실행 흐름을 조절하는 다른 시스템 콜과의 호환성을 위해 일부를 수정했습니다.
- fork: 각각의 스레드가 구조체로 구현되어 있기 때문에, 기존의 fork 함수를 그대로 사용해도 정상적으로 동작합니다. 메인 스레드가 아닌 스레드가 다른 프로세스의 부모가 되는 것을 막기 위한 과정을 추가했습니다.
- sbrk: 시스템 콜 sbrk를 호출하면 내부적으로 growproc이 호출되어 메모리 사용 영역의 크기를 늘려주게 됩니다. 메인 스레드의 페이지 테이블을 공유하면서 사용하므로, 프로세스의 메모리 영역의 사이즈는 항상 메인 스레드의 것을 사용하도록 합니다.
- exit: 하나 이상의 스레드가 exit을 호출하게 되면 해당 스레드가 속한 모든 스레드가 종료되어야 하기 때문에, exit가 호출되면 메인 스레드인 경우 clear_subthreads 함수를 호출하여 모든 서브스레드의 자원을 회수한 뒤 프로세스의 자원을 정리합니다. 메인 스레드가 아닌 경우 일반 프로세스와 유사하게 자신의 상태를 좀비로 만들고 메인 스레드를 깨우게 됩니다.
- kill: kill 시스템 콜을 실행하면 killed를 1로 설정하게 되고 이후 exit 함수가 실행되므로 하나의 스레드에서 kill을 호출해도 모든 스레드가 종료되게 됩니다. 사실상 kill 시스템 콜은 수정할 부분이 없습니다.
- sleep: sleep 시스템 콜 역시 기존 프로세스 단위로 스케줄링되는 것은 변하지 않았기 때문에, 특별히 수정할 부분 없이 그대로 사용할 수 있습니다.
- exec: exec 시스템 콜을 실행하면 해당 시스템 콜을 실행한 스레드를 제외한 모든 스레드가 종료된 뒤 해당 스레드의 정보가 새로운 프로세스로 덮어씌워지게 됩니다. 이를 위해서 현재 스레드를 제외한 나머지 모든 스레드를 종료하고 자원을 회수하는 kill_threads_except 함수를 만들어 사용했습니다. exec을 실행한 스레드가 메인 스레드가 아닐 경우, 메인 스레드의 parent를 상속하고 tid를 0으로 설정하여 스스로가 메인 스레드가 됩니다.

3. Result

Test result

- proc 구조체를 수정한 것을 기반으로 하여 시스템 콜과의 호환성도 갖추었기 때문에, 모든 테스트를 성공적으로 완료하였습니다.

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

```
$ thread_kill
Thread kill test start
Killing process 10
This code should be executed 5 times.
This code should be executed 5 times.
This codThis code should be executed 5 times.
This code should be executed 5 times.
e should be executed 5 times.
Kill test finished
```

```
$ 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...
$
```

4. Troubleshooting

- 기존의 프로세스 기반 운영체제를 스레드 기반 시스템으로 변경하기 위한 개념을 적용하는 과정에서 아래와 같은 문제 상황들과 그 해결법이 있었습니다.

Clearing all sub threads @exit

- exit 시스템 콜은 kill과도 밀접하게 연관되어 있으며, exit 시스템 콜은 하나의 스레드에 대해서만 호출되는데 모든 스레드를 종료해야 했기 때문에 exit 함수 중간에 새로운 작업을 할 필요가 있었습니다.
- 이를 해결하기 위해서 메인 스레드가 아닌 스레드가 종료되면 본인을 ZOMBIE 상태로 바꾸기 전 메인 스레드에 연결된 포인터를 활용해 메인 스레드의 killed 값을 1로 설정하여 해당 스레드 뿐만이 아니라 메인 스레드가 exit을 호출하게 되고 clear_subthreads 함수를 호출하여 나머지 스레드 또한 모두 정리할 수 있었습니다.

```
// Wait for threads except current thread to be exited before
// Used similar structure with wait()
void clear_subthreads(struct proc* curproc){
  struct proc* p;
  int havethreads;
  if(curproc->tid == 0){
    acquire(&ptable.lock);
    for(;;){
      // Scan through table looking for sub threads to be exited.
      havethreads = 0;
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid != curproc->pid || p == curproc){
          continue;
        }
        if(p->state == ZOMBIE){
          // Found one.
          thread_clear(p);
        }
        else{
          havethreads++;
          p->killed = 1;
          wakeup1(p);
        }
      }

      // No point waiting if we don't have any threads to exit.
      if(havethreads == 0){
        release(&ptable.lock);
        break;
      }

      // Wait for children to exit.  (See wakeup1 call in proc_exit.)
      sleep(curproc, &ptable.lock);  //DOC: wait-sleep
    }
  }
  else{
    //cprintf("Not main thread.\n");
  }
}
```

Clearing other threads and inheriting main @exec

- exec 시스템 콜을 호출하는 스레드는 메인 스레드일 수도 있고 아닐 수도 있습니다. 어떤 스레드에서 호출되었든지 간에 해당 스레드만 남기고 모두 종료해야 했기 때문에 exit와 유사하면서도 다른 점이 있었습니다.
- 처음에는 어떤 스레드에서 exec이 호출되었든지간에 항상 메인 스레드를 제외한 나머지를 종료한 뒤 메인 스레드를 새로운 프로그램의 데이터로 덮어씌우려고 하였으나, exec 함수가 호출된 스레드에서 메인 스레드로 실행 흐름을 넘기는 과정에 어려움이 있었습니다.
- 그래서 메인 스레드가 아닌 스레드에서 exec이 호출되면 해당 스레드가 메인 스레드의 여러 특성(parent 프로세스가 누구인지, tid가 0인 점 등)을 상속하여 새로운 메인 스레드가 되게 하고, 기존 메인 스레드를 포함한 나머지 모든 스레드를 정리하고 자원을 반납하는 kill_threads_except 함수를 정의하여 사용했습니다.

```
void kill_threads_except(int pid, struct proc* cp){
  struct proc* p, * q;
  int fd;

  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid != pid || p == cp){
      continue;
    }

    for(q = ptable.proc; q < &ptable.proc[NPROC]; q++){
      if(q->parent == p){
        q->parent = initproc;
        if(q->state == ZOMBIE)
        wakeup1(initproc);
      }
    }
     // Close all open files.
    for(fd = 0; fd < NOFILE; fd++){
      if(p->ofile[fd]){
        fileclose(p->ofile[fd]);
        p->ofile[fd] = 0;
      }
    }
    release(&ptable.lock);

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

    acquire(&ptable.lock);

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

감사합니다. -END-