project03 wiki

2017030055 윤상현

design&implement:

lwp(light-weight-process)는 서로 독립적으로 실행되고 자원을 공유하지 않으며, 각각 개별적인 주소공간과 파일스크립터를 가지는 xv6에서의 프로세스들과 다르게 다른 lwp들과 자원, 주소공간 등을 공유하며 유저레벨의 멀티태스킹을 가능하게 해주는 개념입니다. 이러한 개념을 통해 얻는 이점은 여러가지가 있지만 대표적으로 각각의 thread가 자원을 공유함으로써 메모리 공간을 절약할 수 있다는 점이 있습니다. 또한 각 thread들은 state를 공유하지 않습니다. 예를 들어 하나의 thread가 작업중 io를 wait한다고 같은 프로세스에 속해있는 모든 thread들이 해당 io를 wait하지 않아도 되기에 response역시 좋아지게 됩니다. 이와 같은 특징과 장점을 가지는 lwp를 xv6에서 구현해보도록 하겠습니다.

과제명세의 xv6에서 thread가 프로세스와 비슷하게 취급되어야 한다는 항목을 고려하여, thread의 구조체를 별도로 만들어 구현하는 것이 아닌, 기존의 proc 구조체를 그대로 사용하되, process(master)/thread 를 구분해주는 방식으로 구현하였습니다. process는 속해있는 thread들이 개별적인 process 처럼 동작하는 것이 아닌, 하나의 덩어리처럼 동작할 수 있게 control 해주는 역할을 수행하게 됩니다. 모든 자신에게 속해있는 thread들을 관리하는 process(master)는 여느 프로세스들 처럼 pid를 가지고, 이 pid를 자신에게 속해있는 thread들과 공유하게 됩니다. 이에 따라 '같은 프로세스에 속해있는가? ' 에 대한 물음은 thread들이 가지고 있는 pid를 통해 확인 할 수 있게됩니다. 또한 process는 기존 parent- child 프로세스 관계에서 parent가 하는 역할중 일부 역시 수행하게 됩니다. parent와 유사하게, process는 thread 들이 종료될때까지 wait하고, thread들의 뒤처리를 담당하게 됩니다.

또한 xv6에서 메모리 영역에 관한 구현에 어려움이 있어 cscope를 통해 찾던 중, kalloc.c와 같은 가상화된 메모리에 관련된 함수가 존재하는 파일들을 찾아 스레드의 메모리 영역 구현에 이용하였습니다. 또한 thread들은 기존의 heap과 stack 처럼 상황에 따라 dynamic하게 메모리 영역을 할당 받게 됩니다.

아래 사진은 proc.h입니다.

```
struct empty_mem{
    int size;
    uint data[NPROC];
};

// 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)
//project2 added
  uint ticks;
  int level;//queue level
  int priority;
  int last_out;
  int out;


//project03 add
  int  tid;//thread id
  struct proc* master;// upper process that control thread
  void* tmp_retval;//return val of thread
  uint vir_add;//virtual address
  struct empty_mem empty_mem;// empty memory space


};
```

추가된 인스턴스들은 기존 process 구조체에 thread의 특성을 더해주고, 이러한 thread들을 묶어서 관리하는 process(master)와 구분하여 관리되도록 만들어주었습니다.

tid는 기존 pid와 동일한 역할을 하는 thread id 이고, vir_add는 virtual memory의 주소를 저장하는 인스턴스입니다.

또한 empty_mem 구조체는 비어있는 memory space를 나타내는 구조체로, 비어있는 메모리 공간을 표시합니다. empty_mem은 비어있는 메모리를 모아둔 자료구조로,

thread를 생성할 때 만약 empty_mem이 비어있다면, 크기를 증가시키고 새로 생성된 top을 thread에게 할당하여 메모리를 virtual하게 할당해줍니다(push).

만약 비어있지않다면, 빈 공간을 할당해줌과 동시에 empty_mem에서 해당 공간을 제거해줍니다 (pop)

지금부터는 thread가 올바르게 동작하기 위해 구현한, 혹은 구현하라고 명시 되어있는 함수들에 대하여 설명하겠습니다. 우선 proc.c에 구현한 함수들에 대하여 설명하겠습니다.

1) thread_create

아래 사진은 thread_creat함수의 일부분입니다.

```
// Create thread with process
int
thread_create(thread_t* thread, void* (*start_routine)(void *), void* arg)
{
  pde_t *pgdir;

  int i;
  uint sz, sp, vir_add;


  struct proc *curproc = myproc();
  struct proc *np;

 // struct proc *master= curproc->master ? curproc->master : curproc;
 struct proc *master;
 if(!curproc->master)
     master=curproc;
 else
     master=curproc->master;



  // alloc process.
  if((np = allocproc()) == 0){
      return -1;
  }

  --nextpid;

  // set
  np->tid = nexttid++;
  np->master = master;
  np->pid = master->pid;


  acquire(&ptable.lock);
  pgdir = master->pgdir;


  // if no empty memeory, grow memory and put new thread located at the top of mem

  if(!(master->empty_mem.size)){
    vir_add = master->sz;
    master->sz += 2*PGSIZE;
  }


  else{//get form top of stack(pop)
    vir_add = master->empty_mem.data[--master->empty_mem.size];

  }

  // Allocate two pages for current thread.
  // Make the first inaccessible. Use the second as the user stack for new thread
  if((sz = allocuvm(pgdir, vir_add, vir_add + 2*PGSIZE)) == 0){
    np->state = UNUSED;
    return -1;
  }
  //clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
```

이 함수는 기존 xv6의 fork() 함수와 마찬가지로 thread(process)를 생성해주는 함수입니다.

구현은 상당부분을 기존 xv6의 fork함수를 참고하여 구현하였습니다.

우선 thread는 같은 프로세스 내부의 thread간의 자원공유가 가능하여야 합니다.

이에 따라 같은 process 내의 thread들은 pid를 공유하되, 고유한 tid를 가지게 함으로써 같은 process 내부에 위치한 thread들을 표시하면서 각 thread들을 구분 지었습니다.

만약 새로운 thread를 생성할 프로세스에 master process가 존재하지 않는다면, 새롭게 생성되는 thread를 master process로 임명합니다.

thread의 주소를 process(master) 의 메모리에 추가해주게 됩니다. 이전에 설명하였듯이, dynamic하게 해당 주소를 할당한 뒤에, process의 sz를 argument로 넣어줍니다.

2) thread_exit

```
void
thread_exit(void* ret_val)
{
  struct proc *curproc = myproc();
  int fd;

  // close every  opened file
  for(fd = 0; fd < NOFILE; fd++){
    if(!(curproc->ofile[fd]))continue;
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;

  }

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

  acquire(&ptable.lock);

  // Save return value
  curproc->tmp_retval = ret_val;

  // masster maybe sleep, so wake
  wakeup1(curproc->master);


  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

해당 함수는 thread의 eixt()을 구현한 함수로, 이 역시 기존 exit() 함수를 참조하여 작성하였

습니다.

저의 구현 방식에선 thread들을 총괄하는 process가 parent의 역할을 대신하기 때문에

기존 exit() 함수가 자신을 생성한 parent를 깨우면 종료되는 것에 반하여, 해당 함수는 thread에서 exit이 발생하면, 속해있는 process(master)를 wake합니다.

3) thread_join

```
thread_join(thread_t thread, void** retval)
{
  struct proc *p;
  struct proc *curproc = myproc();

  // exception. (lower thread cannot join thread.
  if(curproc->master != 0){
    return -1;
  }

  acquire(&ptable.lock);
  for(;;){
    //master  clean zombie thread
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->tid == thread&&p->state==ZOMBIE){

        *retval = p->tmp_retval;
        thread_clear(p);

        release(&ptable.lock);
        return 0;

      }
    }

    if(curproc->killed){
      release(&ptable.lock);
      return -1;
    }
    // waiting for lower thread to be terminate
    sleep(curproc, &ptable.lock);
  }
}
```

argument로 지정한 thread가 종료되기를 대기한 뒤, 반환값을 받아옵니다. 또한 스레드가 종료된 뒤, 스레드에 할당된 자원들을 회수하고 정리합니다.

기존의 wait함수와 비슷한 동작을 하기에 wait 함수를 참조하여 구현하였습니다. 우선 join은 모든 thread들을 총괄하는 process만이 요청할 수 있기에 thread에서 요청이 들어왔다면 바로 return을 하여 예외처리를 해주었습니다. process는 join에서 zombie가 된 process 들을 정리해주고, sleep 하여 모든 하위 thread들의 종료를 대기합니다.

4) thread_wakeup

exec() 함수에 의해 호출되고, pid 와 proc 구조체 하나를 인자로 받습니다.

이때 proc구조체로 주어지는 것은 thread들을 관리하는 process입니다.

해당 함수가 실행된 후에 ptable을 돌며 pid를 통해 해당 process에 있는 thread들을 runnable하게 만들어줍니다.

5) thread_kill

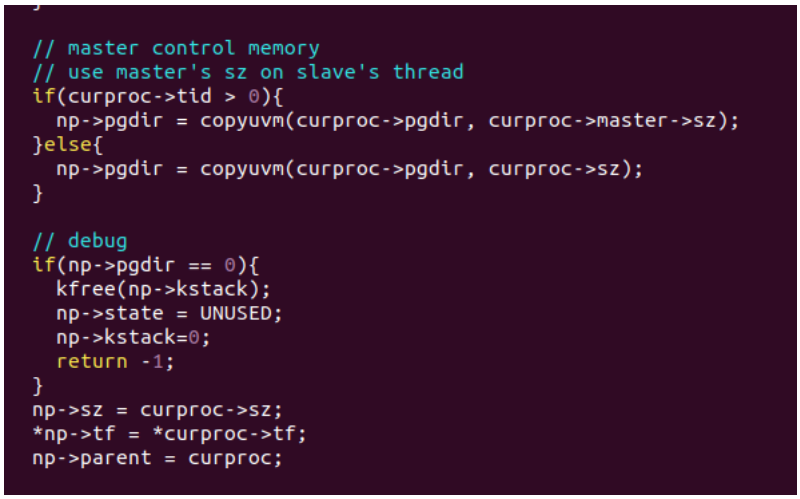
thread의 경우, 만약 1개의 thread가 kill이 된다면, 해당 스레드가 속해있는 process에 속한 모든 thread들 역시 kill 되어야합니다.

하지만 이는 exit() 함수 내부에 구현하였기에, 해당 함수에선 process(master)가 ptable을 돌며 pid를 통하여 자신에게 속해있는 thread들을 재운뒤 killed flag를 표시해준다.

fork:

fork()함수 또한 몇가지 수정사항이 존재합니다.

thread를 구현함에있어 기존의 proc 구조체를 사용하여 구현하였기에, fork()함수 역시 간단하게 수정해주었습니다. 메모리 영역의 관리를 제외하곤 각각의 thread가 proc구조체를 통하여 개별적인 process처럼 동작하기에 thread들을 총괄하는 process가 보유한 정보를 가지고 동작하게끔 변경해주었습니다.



```
// master control memory
// use master's sz on slave's thread
if(curproc->tid > 0){
  np->pgdir = copyuvm(curproc->pgdir, curproc->master->sz);
}else{
  np->pgdir = copyuvm(curproc->pgdir, curproc->sz);
}

// debug
if(np->pgdir == 0){
  kfree(np->kstack);
  np->state = UNUSED;
  np->kstack=0;
  return -1;
}
np->sz = curproc->sz;
*np->tf = *curproc->tf;
np->parent = curproc;
```

위 사진과 같이 fork함수가 master가 가지고있는 size 정보를 가지고 동작하도록 수정하였습니다.

growproc:

growproc역시 변경점이 존재합니다. 메모리 영역의 관리는 master가 가지고 있는 정보로 동작하도록 아래 사진과 같이 수정하였습니다.

```
//project03
int
growproc(int n)
{
  uint prev_sz,sz;
  struct proc *curproc = myproc();
  struct proc *p;

  acquire(&ptable.lock);

  // if curporc==thread size up master

  if(!curproc->master)
      p=curproc;
  else
      p=curproc->master;


  sz = p->sz;
  prev_sz = sz;
  if(n > 0){
    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
      goto bad;
  } else if(n < 0){
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
      goto bad;
  }
  p->sz = sz;
  release(&ptable.lock);

  switchuvm(curproc);
  return prev_sz;

bad:
  release(&ptable.lock);
  return -1;
}
```

exit:

exit함수에서의 변경점에 대해 설명하겠습니다.

```
if(curproc->tid == 0){
  acquire(&ptable.lock);

  for(;;){
    thread_cnt = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->master == curproc){
        // If lower  thread is already zombie, clean-up it
        // Else, kill lower thread and wait for it
        if(p->state == ZOMBIE){
          thread_clear(p);

        }else{
          thread_cnt++;
          p->killed = 1;
          wakeup1(p);
        }
      }
    }
    if(thread_cnt == 0){
      release(&ptable.lock);
      break;
    }
    // Wait for lower thread to exit.
    sleep(curproc, &ptable.lock);
  }
}
```

exit함수에선 이전에 구현하지 않고 넘어갔던 master가 속한 thread들을 모두 제거하는 역할을 추가적으로 수행합니다.

이 작업은 process(master)가 총괄하여 진행하며, ptable을 돌며 zombie state인 thread 들은 clean 해주고, 그외 thread들은 하나하나 kill 해줍니다.

이후 만약 kill 이 한번도 진행되지 않았다면 break로 곧바로 나가주고, 그 외엔 thread들의 exit을 기다려야하기에 sleep 하여 대기해줍니다.

result:

1) thread_test

```
$ thread_test
aaTest 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
ThrThread 3 start
Thread 4 start
Child of thread 1 start
Child of thread 3 start
ead 2 start
Child of thread 0 start
Child of thread 4 start
Child of thread 2 start
Child of thread 1 end
Child of thread 3 end
Thread 1 end
Thread 3 end
Child of thread 0 end
Thread 0 end
Child of thread 4 end
Child of thread 2 end
Thread 4 end
Thread 2 end
Test 2 passed

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

All tests passed!
```

test1의 경우 create, exit, join 등의 기본적인 기능과 스레드 간의 메모리 공유를 확인합니다. 스레드 0은 곧바로 종료되고, 스레드 1은 일정시간동안 wait한 뒤 종료합니다.

test2는 thread 내의 fork에 대해 확인합니다. test 결과 자식 프로세스의 메모리가 부모프로세스에게 아무런 영향을 미치지 않았고, 반대역시 그러하였습니다. 이를 통해 자식-부모간은 별도로 thread처럼 메모리 공유가 발생하지 않는다는 것을 확인할 수 있었습니다.

test3는 thread가 메모리를 할당받은 뒤 다른 스레드들이 접근하는 것에 문제가 없는지 확인합니다. 각 스레드들이 메모리를 할당받을 때 문제가 발생하지 않음이 확인되었습니다.

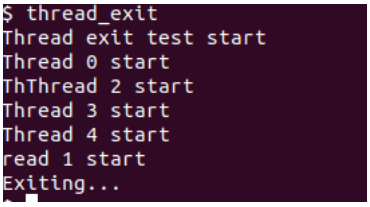
2) thread_exec

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

thread에서 exec의 동작을 확인합니다. 생성된 thread가 올바르게 hello_thread 프로그램을 실행한 것을 확인할 수 있습니다.

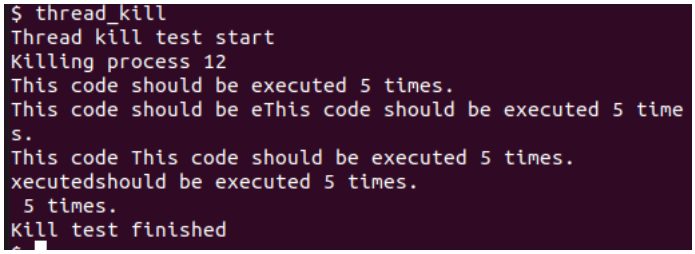
또한 exec가 실행되는 순간, 다른 스레드들이 모두 종료되어 에러메세지가 발생하지 않았습니다

3) thread_exit



thread에서 exit의 동작을 체크하였습니다. thread에서 exit이 호출되면서 해당 thread가 속해있는 프로세스의 다른 thread들 역시 모두 종료됩니다. exiting 출력이후 곧바로 쉘로 빠져나가집니다.

4) thread_kill



process가 kill 되었을 때, 그 프로세서 내의 모든 스레드가 종료되는지 확인합니다.

명시된 process 의 thread가 모두 kill되었고, 올바르지 못한 kill이 없어 zombie process 역시 발생하지 않아 에러메세지가 발생하지 않았습니다.

trouble shooting:

제가 이번 과제를 구현하며 겪었던 가장 큰 문제점은 바로 기존 xv6의 시스템 자체를 건드려야 했다는 부분입니다. 내부적인 구현을 건드릴수록 생각없이 하는 작은 수정이 예기치 못하게 큰 오류를 발생시켜 디버깅에서 어려움이 존재하였습니다. 우선 제가 겪었던 큰 문제들 중하나는 우선 'thread' 의 구현이었습니다. 첫 시도는 아예 독립적인 thread 구조체를 생성하고, process 내부에 이 새로운 thread 구조체들을 저장해주는, 제가 배운 직관적인 내용대로의 설계였습니다.

하지만 구현을 하면 할 수 록 기존 xv6가 구현했던 틀을 건드려야했고, 이로 인하여 수많은 에러들이 발생하였습니다.

xv6 내에서 처음 건드려보는 파트이기도 하였고, 제가 지난과제에서 건드리지 않았던 기본 파일들(kalloc 등등) 에 접근을 하기도 했으며, proc 구조체가 정확하게 어디서 어떻게 돌아가는지에 대한 지식이 부족하여 골머리를 앓던 중, 과제 명세에서, thread의 특징 몇가지를 제외하면 기존의 process처럼 동작한다는 설명을 보고 난 뒤, 방향을 새로 틀어 proc 구조체에 색을 더함으로써 thread구조를 표현하는 것으로 방향을 틀어 기존 코드의 수정을 최대한 피하며 구현하여 문제를 해결하였습니다.