Process Management & LWP

xv6의 프로세스들을 관리해주는 manager 기능을 추가하고, Thread(Light-weight process, LWP)를 구현해본다. 파일은 git의 proj2 브랜치에 업로드되어있다.

Process Management

exec2

현재 exec 함수에서 유저 스택을 할당하는 방법은 다음과 같다. 우선 sz(프로세스별로 할당된 크기 변수)를 올림해 준 후, allocuvm함수로 페이지 2개를 추가로 할당받는다. 이 중 한 페이지를 사용하는데, ustack 변수에 유저스택에 기본적으로 들어갈 변수들(프로그램 실행 시 같이 넣어주는 변수들 포함)을 할당해주고 allockuvm으로 할당된 페이지의 윗부분에 쌓는다. 이 때 기본적으로 할당된 유저스택은 한 페이지인데 allocuvm에서 페이지크기를 추가적으로 더 할당해 주면 스택용 페이지를 여러개로 사용할 수 있게 된다.

```
// exec.c

// ...
    curproc->stacksize = stacksize;
    sz = PGROUNDUP(sz);
    if((sz = allocuvm(pgdir, sz, sz + (stacksize + 1)*PGSIZE)
      goto bad;
    clearpteu(pgdir, (char*)(sz - (stacksize + 1)*PGSIZE));
    curproc->ustack[curproc->tidx] = sz;
    sp = sz;

    // Push argument strings, prepare rest of stack in ustack
    for(argc = 0; argv[argc]; argc++) {
      if(argc >= MAXARG)
        goto bad;
      sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
      if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 
        goto bad;
      ustack[3+argc] = sp;
    }
    ustack[3+argc] = 0;
```

```
    ustack[0] = 0xffffffff;  // fake return PC
    ustack[1] = argc;
    ustack[2] = sp - (argc+1)*4;  // argv pointer

    sp -= (3+argc+1) * 4;
    if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
      goto bad;
// ...
```

setmemorylimit

xv6에서는 setupkvm 함수나 allocuvm 함수를 호출 시 메모리를 할당받고 sz변수가 늘어나게 되는데 따라서 동적 할당 또는 스레드를 생성 시 allocuvm이 호출되어 sz값이 늘어나게 된다. setmemorylimit함수는 pid와 limit을 인자로 받아 해당 프로세스의 sz에 limit을 정해두는 역할을 한다. proc 구조체에 memlimit변수를 선언하고 이것을 적절히 사용하면 구현할 수 있다. setupkvm은 처음에만 실행되므로 allocuvm 전에 memlim을 확인한다.

```
// proc.h

// ...
    int memlimit;                // Memory limit of the proce
// ...
```

```
// proc.c

int
setmemorylimit(int pid, int limit)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      if (limit == 0) {
        p->memlimit = limit;
        return 0;
      } else if (limit < 0) {
```

```
        cprintf("memory limit must be positive\n");
        return -1;
      }
      else if (limit < p->sz) {
        cprintf("memory limit can't be smaller than allocated
        return -1;
      }
      p->memlimit = limit;
      return 0;
    }
  }
  return -1;
}
```

```
// proc.c growproc()

// ...
    sz = curproc->sz;
    if (curproc->memlimit != 0 && curproc->sz + n > curproc->m
      cprintf("memory limit exceeded\n");
      return -1;
    }
// ...
```

pmanager

해당 프로그램에는 5가지 기능이 있다.

list는 단순히 현재 프로세스들을 확인한 후 출력하는 proclist 시스템콜을 만들었고, kill, execute, memlim, exit 도 이미 존재하거나 앞서 만든 시스템콜들을 호출하는 식으로 하였다.

```
// pmanager.c

// ...
if (strcmp(com, "list") == 0){
```

```
    proclist();
}else if (strcmp(com, "kill") == 0){
    if (kill(atoi(arg[0])) == 0) printf(1, "kill succeed\n");
    else printf(1, "kill failed\n");
}else if (strcmp(com, "execute") == 0){
    if (fork() == 0) {
        if (execute(arg[0], atoi(arg[1])) == -1){
            printf(1, "execution failed\n");
            exit();
        }
    }
}else if (strcmp(com, "memlim") == 0){
    if (setmemorylimit(atoi(arg[0]), atoi(arg[1])) == 0) prin
    else printf(1, "setting failed\n");
}else if (strcmp(com, "exit") == 0){
    exit();
}
// ...
```

Light-weight process(LWP)

기존 xv6에 존재하지 않는 Thread 기능을 하는 LWP를 구현하도록 한다.

basic structure

```
// proc.h

struct thread {
  char *kstack;                // Bottom of kernel stack for
  enum procstate state;        // Process state
  thread_t tid;                // Thread ID
  struct trapframe *tf;        // Trap frame for current sysc
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on ch
};
```

```
// Per-process state
struct proc {
  //..

    struct thread t[NTHRD];      // Thread array
  uint ustack[NTHRD];           // user stack space already us
  uint emptystack[NTHRD];       // user stack space cleaned by
  int tidx;                     // Running thread index
  int waiting[NTHRD];           // Need for thread join
};
```

thread는 process와 달리 데이터 영역, 파일 디스크립터 등을 공유하므로 별도의 프로세스 구조체로 하지 않고 한 프로세스 안에 배열로 여러 스레드(최대 16)를 사용할 수 있도록 했다. 또한 tidx변수를 선언하여 현재 무슨 스레드를 실행중인지 알게 했다. 명세의 xv6환경과 달리 CPU가 늘어나면 tidx를 배열로 저장하는 것으로 해결할 수 있다.

ustack과 emptystack은 각각 여러 스레드가 어떤 페이지를 user stack으로 사용중인지, 할당받았지만 사용중이 아닌 페이지는 어디인지 표시해주는 역할을 한다.

프로세스 구조체를 바꿨으므로 p→kstack등의 코드들은 전부 올바른 인덱스의 스레드 데이터를 가리키도록 수정했다.

thread management

thread_create

allocproc()과 exec() 함수와 유사하게 만들었다. 우선 memlim을 확인하고, 공간이 있으면 allocthread()를 호출에 커널 스택, 트랩 프레임 등을 정의한다. allocthread()는 allocproc()에서 커널스택을 할당받는 부분을 빼와서 작성했다.

```
// proc.c

// need to acquire lock before call this function.
static struct thread*
allocthread(struct proc *p, thread_t tid){
  struct thread *t;
  char *sp;

  for(t = p->t; t < &p->t[NTHRD]; t++){
```

```
    if(t->state != UNUSED && t->tid == tid){
      cprintf("thread allocation error: pid %d tid %d duplica
      return 0;
    }
  }

  for(t = p->t; t < &p->t[NTHRD]; t++)
    if(t->state == UNUSED)
      goto found;

  return 0;

found:
  t->state = EMBRYO;
  t->tid = tid;

  // ...
  return t;
}
```

커널 스택을 할당받은 후 유저 스택용 페이지를 할당받는다. 이전에 스레드가 할당되었다가 해제된 경우 해당 영역은 pgdir에 빈 부분으로 남게 된다. 이는 사이즈 계산에 오차를 발생시키고 해당 상황이 지속될 경우 sz가 계속 올라가 가상 주소를 넘어 더 이상 스레드를 만들 수 없게 될 수 있다. 이를 방지하기 위해 그러한 영역이 있는지부터 확인하고 있을경우 해당 페이지를 유저 스택으로 사용한다.

```
// check if the empty user stack space already exists.
  for (i = 0; i < NTHRD; i++)
    if (p->emptystack[i] != 0) break;
  if (p->emptystack[nt - p->t] != 0) i = p->emptystack[nt - p

  if (i != NTHRD){
    // write ustack location.
    p->ustack[nt - p->t] = p->emptystack[i];
    p->emptystack[i] = 0;
  }else {
    // test if the curproc will exceed the limit.
```

```
    if (p->memlimit != 0 && PGROUNDUP(p->sz) + 2*(p->stacksiz
      cprintf("memory limit exceeded.\n");
      goto bad;
    }

    // Allocate user stack.
    p->sz = PGROUNDUP(p->sz);
    if (((p->sz = allocuvm(p->pgdir, p->sz, p->sz + (p->stack
      goto bad;

    p->ustack[nt - p->t] = p->sz;
  }

  sp = p->ustack[nt - p->t];
```

그 후 arg 인자를 exec와 유사하게 집어넣으면 유저 스택이 완성된. 마지막으로 트랩 프레임을 복사한 뒤 eip, esp만 각각 start_routine, 새로운 sp로 바꿔주면 새로운 스레드가 완성된다.

```
// proc.c thread_create()

// ...
// Allocate user stack.
  p->sz = PGROUNDUP(p->sz);
  if (((p->sz = allocuvm(p->pgdir, p->sz, p->sz + (p->stacksi
    goto bad;
  sp = p->sz;

  // fetch argument and stack
  ustack[3] = (uint)arg;

  ustack[0] = 0xffffffff; // fake return PC
  ustack[1] = (uint)arg;
  ustack[2] = sp - 8;

  sp = sp - (3+1+1) * 4;
  if(copyout(p->pgdir, sp, ustack, (3+1+1)*4) < 0)
```

```
    goto bad;

  // copy trap frame
  *nt->tf = *p->t[p->tidx].tf;

  nt->tf->eip = (uint)start_routine;
  nt->tf->esp = sp;
// ...
```

thread_exit

exit() 함수와 유사하게 만들었다. 우선 proc의 ret배열의 tidx번째에 리턴값을 저장한다.

exit을 할 때 waiting의 자신 tidx를 확인하면 자신을 join하려는 스레드가 현재 있는지 알 수 있다. 이 값이 1일 때만 wakeup시키도록 하여 join도 없는데 상관 없는 스레드가 깨어나는 현상을 방지하고자 했다.

그 후 스레드를 ZOMBIE 상태로 바꾸는데, 이 때 해당 프로세스가 전부 ZOMBIE가 되어 어느 스레드도 자원을 회수하지 못할 상황을 대비해 현재 살아있는 스레드가 없을 경우 exit()을 호출해주도록 하여 자원의 낭비를 줄이고자 했다.

```
// proc.c thread_exit()

void thread_exit(void *retval){
  struct proc *curproc = myproc();
  struct thread *t = &(curproc->t[curproc->tidx]);
  int cnt;

  ret[curproc->tidx] = (uint)retval;

  acquire(&ptable.lock);

  if (curproc->waiting[curproc->tidx] != 0){
    wakeup1(curproc);
  }

  t->state = ZOMBIE;
  curproc->state = RUNNABLE;
```

```
  // if no thread has left on the process, call exit().
  cnt = 0;
  for(t = curproc->t; t < &(curproc->t[NTHRD]); t++){
    if (t->state != UNUSED && t->state != ZOMBIE) cnt++;
  }
  if (cnt == 0) {
    release(&ptable.lock);
    exit();
  }

  sched();
  panic("zombie exit");
}
```

thread_join

wait()함수와 유사하게 만들었다. 우선 자신의 tid를 join하는 경우 -1을 리턴한다. 스레드 배열을 한번 순회하고, 자신이 join하려는 tid가 없으면 -1을 리턴하고, 이미 종료되었으면 바로 자원 회수하고 스레드 배열의 해당 영역을 비워놓고, 리턴값을 받아온다. 실행 중이면 waiting을 1로 하고 sleep한다.

```
// proc.c thread_join()

// ...
curproc->waiting[tidx] = 1;
sleep(curproc, &ptable.lock);  //DOC: wait-sleep
curproc->waiting[tidx] = 0;
// ...
```

또한 자원을 회수할 때 자신의 인덱스 부분에 비워지는 유저스택 정보를 집어넣어 thread_create 호출시 사용할 수 있게 했다. 비어있는 스택정보가 사용되기 전에 덮어씌워지는 것을 막기 위해 새로운 스레드가 생성될 때 emptystack 배열의 자신 인덱스부터 사용하도록 설정했다.

```
// proc.c thread_join()
```

```
// ...
if(t->state == ZOMBIE){
  // Found one.
  // cprintf("%d %d exit\n", curproc->pid, t->tid);
  kfree(t->kstack);
  t->kstack = 0;
  t->state = UNUSED;
  t->tid = 0;
  curproc->emptystack[t - curproc->t] = curproc->ustack[t - c
  curproc->ustack[t - curproc->t] = 0;
  release(&ptable.lock);
  *retval = (void *)ret[t - curproc->t];
  return 0;
}
// ...
```

existing code modification

allocproc

스레드 영역에서 공간을 할당해 주어야 하는 부분이 생겼으므로 해당 부분 allocthread() 함수에서 할당해 주어 첫 번째 인덱스 스레드에 넣어주도록 했다.

```
// proc.c allocproc()

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

  if ((t = allocthread(p, 1)) == 0){
    p->state = UNUSED;
    p->pid = 0;
    p = 0;
  };

  p->tidx = t - p->t;
```

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

fork

allocproc()을 한 후 현재 프로세스의 스레드 내용을 덮어쓴다. 또 ustack과 emptystack 상태를 가져와 나중에 새 프로세스에서 thread_create()시 사용할 수 있도록 만든다.

```
// proc.c fork()

// ...
// Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0)
    np->state = UNUSED;
    kfree(np->t[np->tidx].kstack);
    np->t[np->tidx].kstack = 0;
    np->t[np->tidx].state = UNUSED;
    np->t[np->tidx].tid = 0;
    return -1;
  }

  np->sz = curproc->sz;
  np->stacksize = curproc->stacksize;
  np->memlimit = curproc->memlimit;
  np->parent = curproc;
  for (i = 0; i < NTHRD; i++){
    if (i == np->tidx) np->ustack[i] = curproc->ustack[curpro
    else if (i == curproc->tidx){
      if (curproc->ustack[np->tidx] != 0) np->emptystack[i] =
      else np->emptystack[i] = curproc->emptystack[np->tidx];
    }
    else {
      if (curproc->ustack[i] != 0 && i != curproc->tidx) np->
      else np->emptystack[i] = curproc->emptystack[i];
    }
  }
```

```
  *np->t[np->tidx].tf = *curproc->t[curproc->tidx].tf;

  // Clear %eax so that fork returns 0 in the child.
  np->t[np->tidx].tf->eax = 0;
// ...
```

scheduler

RUNNABLE한 process를 선택하면 그 안에서 RUNNABLE한 스레드를 찾아 실행하는 이중 for문으로 구성했다. thread가 선택되면 tidx를 그에 맞게 바꿔주고 process와 thread의 상태를 둘다 RUNNING으로 바꿔준다. 스레드의 작업에 의해 process의 상태가 바뀔 수 있으므로 process의 상태가 RUNNABLE에서 바뀌면 (ex. SLEEP, ZOMBIE) 스레드 탐색을 종료하도록 했다.

```
// proc.c scheduler()

// ...
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      for(t = p->t; t < &p->t[NTHRD] && p->state == RUNNABLE;
        if(t->state != RUNNABLE)
          continue;
                // Select one of the runnable thread and copy
        p->tidx = t - p->t;

        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;
        p->t[p->tidx].state = RUNNING;

        swtch(&(c->scheduler), p->t[p->tidx].context);
        switchkvm();
      }

      c->proc = 0;
```

```
    }
// ...
```

exit, kill, wait, sleep, setmemorylimit

exit, wait, kill, setmemorylimit은 프로세스 단위로, sleep은 스레드 단위로 실행되도록 했다. 단 wakeup은 프로세스 단위로 실행되도록 했는데, 이렇게 해야 자고 있는 스레드들을 한 번에 깨울 수 있어 더 효율적이라고 생각했다.

Code testing

pmanager

```
$ pmanager
list
Process Name     pid    numofstackpage  memsize  memmax
=====================================================
init             1      1               12288    0
sh               2      1               16384    0
pmanager         3      1               12288    0
execute ls 1
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15652
echo           2 4 14528
forktest       2 5 8964
grep           2 6 18488
init           2 7 15152
kill           2 8 14616
ln             2 9 14512
ls             2 10 17084
mkdir          2 11 14636
rm             2 12 14616
```

```
sh              2 13 28668
stressfs        2 14 15548
wc              2 15 16064
zombie          2 16 14188
pmanager        2 17 15924
hello_thread    2 18 14336
thread_exec     2 19 15724
thread_exit     2 20 15568
thread_kill     2 21 16608
thread_test     2 22 19264
console         3 23 0
execute pmanager
stacksize must be in 1~100: 0
execution failed
execute pmanager 3
list
Process Name     pid    numofstackpage  memsize  memmax
=========================================================
init             1      1               12288    0
sh               2      1               16384    0
pmanager         3      1               12288    0
pmanager         6      3               20480    0
memlim 6 21000
setting succeed
list
Process Name     pid    numofstackpage  memsize  memmax
=========================================================
init             1      1               12288    0
sh               2      1               16384    0
pmanager         3      1               12288    0
pmanager         6      3               20480    21000
kill 6
kill succeed
list
Process Name     pid    numofstackpage  memsize  memmax
=========================================================
init             1      1               12288    0
sh               2      1               16384    0
```

```
pmanager         3      1                12288    0
exit
zombie!
zombie!
$ zombie!
```

thread_exec, thread_exit, thread_kill

```
$ 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 This code should be executed 5 times.
TThis code should be executed 5This code shobe executed 5 tim
his code shoul times.
uld be executed 5 times.
d be executed 5 times.
Kill test finished
$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 starThread 2 start
Thread 3 start
Thread 4 start
Child of thread 0 start
Child of threadChild of thread 3 start
Child of thread 4 start
t
 2 start
Child of thread 1 start
Child of thread 0 end
ThreaChild of thread 3 end
d 0 end
Thread 1 end
Child of thread 4 end
Thread 2 end
Child of thread 2 eChild of thread 1 end
nd
Thread 3 end
Thread 4 end
Test 2 passed

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

```
All tests passed!
$
```

thread_fork, thread_manyt

thread_manyt.c

많은 스레드 생성과 제거시 제대로 생성되는지를 시험하기 위해 만들었다. 스레드 100개를 만든다. 생성된 스레드 개수가 12개째일때부터 이전 스레드를 join으로 하나씩 지워나간다. 메모리 공간 크기가 (12288) + (4096 * 2) * 100 = 831488이 아닌 (12288) + (4096 * 2) * 12 = 110592로 유지되는 것을 볼 수 있다.

```
$ thread_manyt
Thread many create test start
100 thread will be created. '+' should be written 100 times
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Process Name     pid    numofstackpage  memsize  memmax
=============================================================
init             1      1               12288    0
sh               2      1               16384    0
thread_manyt     9      1               110592   0
$
```

thread_fork.c

스레드 생성과 fork() 호출의 안정성을 시험하기 위해 만들었다. main process가 5개의 스레드를 만들고, 5개의 스레드는 각각 fork()를 실행한 후, 이 때 나온 자식 프로세스는 또 각각 5개의 스레드를 만든다. wait()은 따로 관리하지 않는다.

pid 3은 스레드 5개를 생성해서 memsize가 늘어난 모습이다.

pid 4 5 6 7 8은 pid 3의 tid 2 3 4 5 6 스레드로부터 만들어진다. 그 후 각각 자식 스레드를 5개씩 만들게 되는데, 부모 프로세스와 달리 현재 본인의 스레드는 1개이므로 fork()시 이전에 pid 3에서 만들었던 스레드의 빈공간 5개를 가지고 있다. 이곳에 할당받아 thread 공간을 만든다. 따라서 memsize가 추가로 늘어나지 않고 유지되는 모습이다.

```
$ thread_fork
Process Name     pid    numofstackpage  memsize  memmax
=====================================================
init             1      1               12288    0
sh               2      1               16384    0
thread_fork      3      1               53248    0
thread_fork      4      1               53248    0
thread_fork      5      1               53248    0
thread_fork      6      1               53248    0
thread_fork      7      1               53248    0
thread_fork      8      1               53248    0
zombie!
zombie!
zombie!
zombie!
zombie!
```