Wiki_p2_2020028586

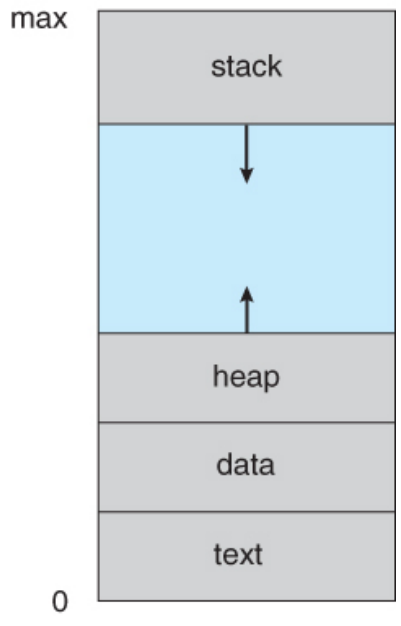
- Design
  - 배경 지식
    - xv6의 프로세스 memory layout
    - threading
  - Process Management in xv6
    - Process with various stack size
    - Process memory limitation
    - Process manager
  - Threading using LWP in xv6
    - 전반적인 디자인
    - Thread Management
    - 기존의 System call과의 호환성
- Implement
  - Process Management
    - Process with various stack size: `exec2`
    - Process memory limitation
    - pmanager
  - threading
    - int thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg);
    - void thread_exit(void *retval);
    - int thread_join(thread_t thread, void **retval);
    - 기존 system call과의 호환
    - threading을 구현하기 위한 부가적인 함수
- Test Result
  - pmanager
  - LWP Threading
    - thread_exec
    - thread_exit
    - thread_kill
    - thread_test
- Trouble Shooting
  - 디자인 관련
    - xv6의 프로세스 구조
    - threading와 proc 구조체 멤버 변수 추가 관련
    - thread를 create할 때 무엇을 그대로 가져오고 무엇을 새로 생성하는지
    - thread create, exit, join이 각각 어떤 기존의 시스템 콜과 유사한지
  - 버그
    - make가 되지 않는 문제
    - pmanager에서 input string을 파싱할 때, 로컬 변수에 이전 입력값이 남아있는 문제
    - xv6가 재부팅되는 문제
    - remap panic
    - acquire panic
    - 각종 panic, trap 13, trap 14, xv6가 멈추는 문제 등

Design

배경 지식

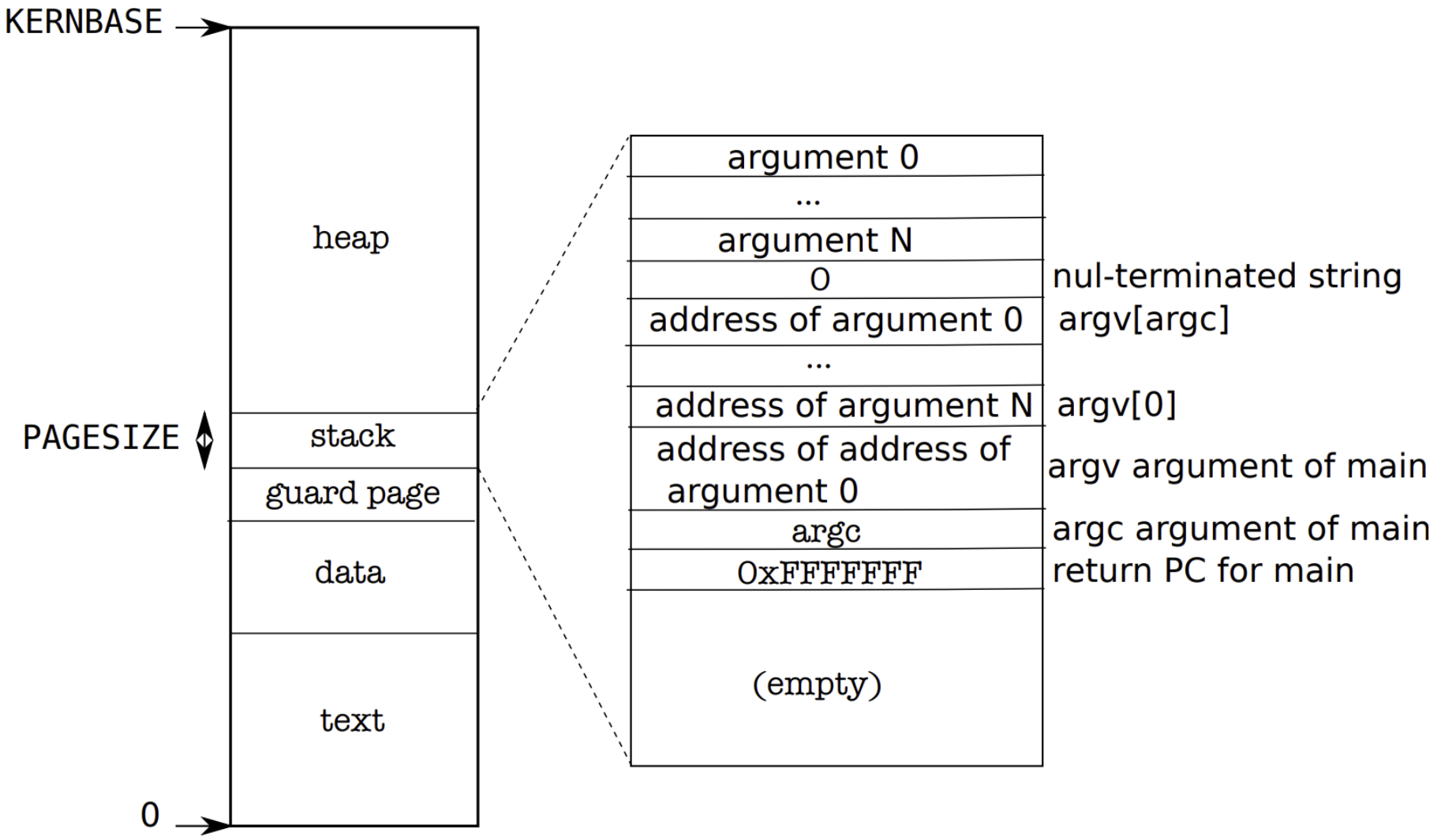
xv6의 프로세스 memory layout

일반적으로 linux 등의 운영체제에서 구현되는 프로세스의 메모리 할당 구조는 다음과 같다.



일반적인 프로세스의 memory layout (사진 출처: Operating Systems: Processes (uic.edu))

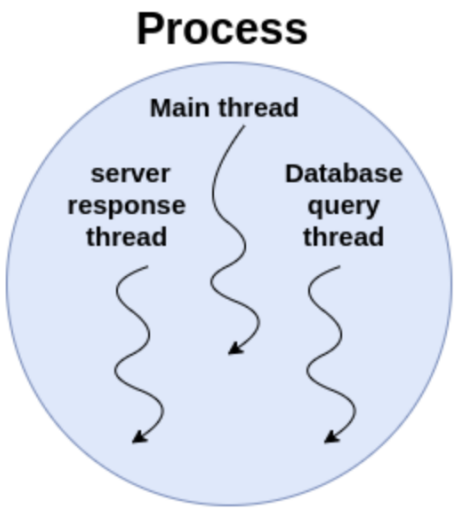
그러나, xv6의 프로세스는 stack이 최상단부터 할당되는 것이 아니라, data 위에 guard page가 존재하고 그 위에 stack이 PAGESIZE만큼 할당되는 형식이다. 그 위로 heap이 할당되고, 최상단(KERNBASE 위)에는 kernel 영역이 존재한다.



Memory layout of a user process with its initial stack. (사진 출처: xv6 - DRAFT as of September 4, 2018 (mit.edu))

따라서, Light weight process(이하 LWP)를 구현함에 있어서는, 기존의 proc 구조체를 재활용하되, threading 구현을 위한 추가적인 멤버변수를 proc 구조체 내부에 선언해줄 필요가 있었다.

threading



main thread가 존재하고, 동시에 다른 작업을 하는 thread가 존재한다.

각 thread는 data, text를 main thread와 공유하며, stack과 register context를 고유하게 갖는다. 이로써 프로세스에 비해 context switch 비용을 줄일 수 있다.

Process Management in xv6

Process with various stack size

기존의 `exec()` 에서는 guard page하나, user stack page 하나를 하나의 프로세스를 위해 할당했다. 그러나, various stack size를 갖는 프로세스를 구현하기 위해(스택을 일정 이상 요구하는 프로세스를 xv6에 구현하기 위해)서는 allocuvm을 해줄 때 PGSIZE에 input stack size를 곱한다(guard용 페이지는 input stack size와 관계없이 무조건 1개를 할당한). 이로써 여러개의 stack size를 할당하여 execution하는 시스템콜 `exec2`를 구현할 수 있다.

Process memory limitation

프로세스가 가질 수 있는 process memory size를 제한할 수 있도록 proc 구조체에 제한값이 얼마인지 확인할 수 있는 멤버 변수를 추가해야한다.

시스템 콜을 통해 프로세스 생성 이후 제한을 설정할 수 있어야하므로, memory limit 제한을 설정하는 시스템콜을 추가해야한다(처음 프로세스가 생성될 때는 제한이 없음)

또한, 유저 프로세스는 추가적인 메모리를 할당받을 때 해당 프로세스의 memory limit을 넘는 경우 할당받지 않아야한다. 즉, memory를 추가적으로 할당받는 `growproc` 함수에서 메모리를 추가할당하려고 하는 경우 해당 프로세스의 memory limit과 비교하여 추가할당하려는 메모리 크기가 limit을 초과하면 비정상적인 동작으로 보는 과정을 추가한다.

또한, thread만들 때 해당 thread를 관리하는 main thread의 memory size가 커짐에 따라 memory limit을 초과하는지 확인해줘야한다.

Process manager

현재 실행중인 프로세스들의 정보를 확인하고 관리하는 유저 프로그램으로, `list`, `kill`, `execute`, `memlim`, `exit`의 명령어를 수행할 수 있다.

- `list`: 현재 실행중인 프로세스들의 정보를 출력하기 위한 시스템 콜을 추가한다. 해당 시스템 콜에서는 ptable을 순회하면서 실행중인 프로세스(RUNNING, RUNNABLE, SLEEPING)를 출력한다. 다만 이때, process가 실행한 thread의 정보를 고려하여 출력해야한다. thread의 경우 출력하지 않지만, thread의 정보를 고려해야하므로, thread의 세부 정보(pid, stack page 개수, 메모리 크기 등)는 출력하지 안되, 프로세스에서 실행중인 스레드가 무엇이 있는지 정도의 정보만 출력하도록 한다. (해당 주황색 하이라이트 부분은 명세에서 조금 더 추가된 사항입니다.)
- `kill`: 기존의 `kill` 시스템 콜을 활용하여 특정 pid를 가진 프로세스를 kill한다.
- `execute <path> <stacksize>`: path의 경로에 위치한 프로그램을 stacksize 개수만큼의 스택용 페이지와 함께 실행한다. 넘겨주는 인자는 `path` 하나이지만 기존의 `exec` 시스템콜을 활용하기 위해(즉, 넘겨주는 인자가 char** 타입), 인자의 0번째 항목은 `path`로, 1번째 항목은 null을 넘겨준다.
  - 실행한 프로그램의 종료를 기다리지 않고 이어서 실행되어야하므로 기존의 xv6 shell과는 달리 main에서 `wait()`을 하는 부분이 빠지게 된다. 이로 인해 execute 명령어를 실행하게 되면 순서가 꼬이면서 pmanager shell의 input을 유저가 보기 편하게 해주는 `>>` 접두어가 보이지 않을 수도 있다. 그럴 때는 엔터를 한번 입력해보거나, 그냥 무시하고 입력하면 된다. 또한, pmanager shell을 종료할 때 initproc에 의해 zombie! 가 출력될 수 있다.
  - 실패시에만 필요한 경우 기존 `exec` 시스템 콜의 exec 실패 문구와, execute ~~ failed를 출력한다.
- `memlim <pid> <limit>`: 특정 pid의 프로세스 메모리 제한을 limit으로 설정한다. setmemorylimit 시스템콜을 활용하여 해당 반환값을 바탕으로 성공했는지 실패했는지 확인할 수 있다.
- `exit`: pmanager를 종료한다. `exit` 시스템 콜을 활용한다.

이 외의 input에 대해서는 invalid input으로 처리되며, 실행되지 않는다.

Threading using LWP in xv6

Design-배경 지식에서 살펴본 바와 같이, xv6는 text / data / guard page / stack / heap / 커널 영역 순서의 memory layout을 가진다.

이때, LWP를 통한 threading을 구현하기 위해서, LWP는 다른 LWP와 자원과 주소 공간 등을 공유해야하며 유저 레벨에서는 멀티태스킹이 가능하게 된다. 또한, LWP도 프로세스이므로, scheduler에 의해 순차적으로 scheduling 되어야한다.

전반적인 디자인

proc 구조체에 threading을 위한 변수를 추가했다(tid, mthread, retval). thread를 생성하면, ptable 중 하나를 차지하여 기존의 xv6와 스케쥴링이나 시스템 콜 작업 등에 쉽게 호환되도록 한다.

Thread Management

- create thread
  - 기존의 `fork`와 `exec`의 일부를 조금씩 떼어다가 합쳐놓은 것과 유사하다.
  - 현재 실행중인 프로세스 혹은 스레드로부터 새로운 스레드를 만든다.
    - 프로세스로부터 스레드를 만드는 경우 해당 프로세스를 새롭게 만들어진 스레드의 main thread로 설정한다.
    - 스레드로부터 스레드를 만드는 경우 thread_create를 호출한 스레드의 main thread를 새로운 thread의 main thread로 설정해준다.
  - `thread_create`를 호출하면, 프로세스를 할당할 때처럼 allocproc을 통해 ptable 배열의 proc 하나를 차지하되, thread임을 알 수 있는 변수(tid와 main thread 등)를 할당한다.
  - 메모리 공간을 공유하기 위해 해당 sub thread를 관리하는 main process의 pgdir을 공유한다. 이로써 text나 data에 대한 정보를 공유할 수 있다.
  - 이외의 유저 stack이나 kernel 스택, trapframe 등은 LWP가 자신만의 것을 가질 수 있도록 한다. exec 함수처럼 allocuvm을 통해 stack을 할당해준다.
  - 기존의 main thread위에 thread(guard 하나, user stack 하나)의 stack이 차곡차곡 쌓이는 형태이므로, 최상단 stack을 가리키고 있는 main thread의 stack 위에 새롭게 thread를 쌓아준다.
  - thread create가 완료되면, trapframe에서 수정해야할 부분을 수정해주는데, thread가 실행할 명령 주소를 start_routine으로 설정하고, esp를 stack의 가장 아래 부분으로 설정한다.
  - 또한, function argument인 `thread`가 유저 차원에서 스레드의 id를 지정한다고 명세에 쓰여져있는데, testcode를 살펴보면 thread_create시 빈 포인터 배열을 넘겨주는 것을 볼 수 있다. 즉, 유저 차원에서 스레드의 id를 지정해서 관리할 수 있도록 커널 측에서 설정한 thread id를 유저가 볼 수 있도록 설정한다.
  - 모든 설정이 완료되면 스레드의 state를 RUNNABLE로 설정해서 기존의 프로세스와 비슷하게 scheduler가 스케줄링할 수 있도록 설정해준다. 성공한 경우 0, 이외에 도중 실패한 경우는 -1을 반환한다.
- exit thread
  - 기존의 `exit`과 비슷하다.
    - 단, thread의 parent가 아닌, main thread가 thread_exit을 호출한 스레드를 정리하도록 한다. 해당 main thread를 wakeup하여 스레드를 정리해주기를 요청한다. 또한, thread의 자식을 정리하는 부분은 필요 없으므로 삭제하였다.
  - 현재 실행중이던 스레드를 종료한다. 리턴값을 thread 멤버 변수의 retval에 저장하고, 본인은 ZOMBIE가 되어, main thread가 자신의 리턴값과 함께 자신을 정리해주기를(thread_join 해주기를) 기다린다.
- join thread
  - 기존의 `wait`와 비슷하다.
    - 단, wait과는 다르게, 자식을 찾는것이 아니라, 스레드가 있는지 찾는다.
    - 또한, 다른 sub thread에서 pgdir을 여전히 이용하고 있을 수 있기 때문에 기존의 wait와는 다르게 pgdir을 정리하는 과정이 제거되어있다.
  - 주어진 thread id가 종료될 때까지 기다리다가, 종료된게 있게 되면(retval을 설정하고 ZOMBIE가 된 sub thread가 존재한다면) 해당 thread의 변수 및 자원을 정리해준다.

기존의 System call과의 호환성

- fork: thread는 기존의 proc 구조체를 재활용하였기 때문에, 프로세스와 유사하게 작동한다. 하지만, thread에서 fork하여 새로운 process(main thread)를 만들게 되므로 thread를 부모로 설정하지 않도록 main thread를 찾는 과정이 추가되어 있다.
- exec: 마찬가지로 크게 달라지는 점은 없으나, 스레드에서 exec를 실행하는 경우 자신 외의 다른 스레드를 모두 정리하고 sub thread에서 main thread(프로세스)가 되면서 새로운 실행 코드로 자신의 내용을 재구성하게 된다. 이를 위해 cleanOtherThreadsForExec를 호출하여 자신 외의 다른 스레드를 모두 정리하고, makeMainThread에서 자신을 main thread로 설정하여 이후의 exec을 실행한다.
- sbrk: sbrk는 프로세스에게 메모리를 할당하는 시스템 콜로, 여러 스레드가 동시에 메모리 할당을 요청하더라도 할당하는 공간이 겹치면 안되기 때문에 mthread를 활용하여 growproc을 해주게 된다. 단, 이때 memory limit을 초과했는지 검사한다.
- kill: 스레드가 kill되면 해당 스레드가 속한 프로세스 내 모든 스레드가 종료되어야 하는데, trap.c에서 kill이 프로세스 혹은 스레드에서는 exit을 호출한다. 따라서, exit에서 마무리 작업(다른 스레드 종료 및 자원 회수)을 하게 한다.
- sleep, pipe: 스레드 역시 ptable에 속하여 process처럼 스케줄링되므로, 별도로 수정할 내용이 없다.
- (optional) exit
  - 명세 pdf에는 명시되어있지 않으나, 실습시간에 구두로 말씀하신 내용이며 테스트코드에 있는 사항이기도 하고, kill의 처리를 용이하게 하기 위해 구현하였습니다.
  - 스레드가 exit되면, 해당 스레드가 속한 프로세스의 다른 스레드의 자원을 정리하고, 자신은 ZOMBIE가 되어, process의 parent를 깨워 자신을 정리하도록 한다.

Implement

Process Management

Process with various stack size: `exec2`

```
int exec2(char *path, char **argv, int stacksize);
```

기존의 `exec`가 무조건 하나의 user stack과 하나의 guard page를 할당했던 것과는 달리, stack size를 input으로 받아서 여러 user stack을 할당할 수 있게 된다.

```
// stack size만큼 page를 할당(+1 for guard page)
if ((sz = allocuvm(pgdir, sz, sz + (stacksize + 1) * PGSIZE)) == 0)
  goto bad;
clearpteu(pgdir, (char *)(sz - (stacksize + 1) * PGSIZE)); // guard용 page 1개 설정
```

exec로부터 exec2가 수정된 부분은 위와 같다. allocuvm의 newsz의 인자로 stacksize만큼의 user stack과, 1개의 guard page 만큼의 memory size를 넣어줘서 various stack size를 할당할 수 있도록 수정하였다.

그리고 몇개의 page가 할당되었는지 확인하기 위해 proc 구조체에 `stackpages`라는 count 변수를 추가하였다.

또한, sys_exec2에 stacksize가 1이상 100 이하의 정수일 수 있도록 예외처리하는 구문을 추가하였다.

```
// t1. stack용 page 개수는 1이상 100이하의 정수
if (stacksize < 1 || stacksize > 100)
{
  return -1;
}
```

Process memory limitation

특정 프로세스에 대해 할당받을 수 있는 메모리의 최대치를 제한하되, 처음 프로세스가 생성될 때는 제한이 없도록 해야한다. 추후 시스템 콜로 설정할 수 있게 해야한다.

따라서, proc 구조체에 `mlimit`라는 memory limit(bytes 단위)를 확인할 수 있는 변수를 추가하였다.

```
int setmemorylimit(int pid, int limit);
```

위 함수를 통해 해당되는 pid를 가진 process의 memory limit을 설정해줄 수 있다. 단, 존재하지 않는 pid이거나, limit이 현재 프로세스가 가진 메모리 크기보다 작은 경우, 음수인 경우 등의 예외 상황에서는 -1을 반환하도록 했다.

setmemorylimit 자체는 pid에 해당하는 proc을 ptable에서 찾고, `mlimit` 멤버 변수를 input mlimit으로 설정해주는 시스템콜의 역할만 수행한다. 다만, 메모리를 추가적으로 할당해주는 growproc이나 thread를 만들 때 메모리가 추가적으로 사용되는 부분 등에서 이 memory limit을 확인하여 그 역할을 수행할 수 있도록 한다.

처음 프로세스가 생성될 때(allocproc)는 memory limit이 0으로 설정되는데, 0은 memory limit이 없다는 뜻이다.

pmanager

xv6의 쉘 코드인 sh.c를 참고해서 만들었다. 단, 명령어 수가 다양하지 않으므로 명령어를 main에서 파싱하도록 간략화하였다(sh.c에서 `cd`를 파싱하는 것처럼 buf 배열을 조사한다).

기존 쉘과 구분하기 위해 `>>` 접두어를 사용한다.

- list: `showProcessList`시스템콜을 proc.c에 추가하였고, `list`라는 명령어를 받으면 해당 함수를 호출한다. 해당 함수 내에서는 실행중인 프로세스의 proc 구조체의 정보를 출력한다. 단, thread가 있는 경우를 고려하여, sz는 메인 프로세스 내부의 sub-스레드가 차지하는 메모리의 합을 출력한다. 또한, 스레드의 상세한 정보는 출력하지 않되, thread의 정보 자체는 고려해야하므로 (optional) 해당 process가 실행중인 thread도 출력하도록 한다.

```
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
{
  // TODO: 실행중인 것의 정의? RUNNABLE이면 스케줄러에 의해 실행되므로 실행중이라고 볼
  // TODO: 쓰레드 고려하여 출력(쓰레드는 출력하지 않지만, 테스트를 위해 TID를 포함하여

  // 만약 현재 process가 thread인 경우, ptable을 한번 더 돌면서 mthread를 찾아줌
  if (p->state == RUNNABLE || p->state == RUNNING || p->state == SLEEPING)
  {
    if (p->tid == 0 || p->mthread == 0)
    {
      cprintf("name: %s | ", p->name);
      cprintf("pid: %d | ", p->pid);
      cprintf("stack pages: %d | ", p->stackpages);
      cprintf("memory: %d | ", p->sz);
      cprintf("memlim: %d | ", p->mlimit);
      cprintf("\n");
      struct proc *q;
      for (q = ptable.proc; q < &ptable.proc[NPROC]; q++)
      {
        if (q->pid == p->pid && q != p && q->mthread == p)
        {
          cprintf("   thread for process %d with thread id: %d\n", p->pid, q->tid);
        }
      }
    }
  }
```

- kill: 마찬가지로 buf를 파싱해서 kill 하려는 pid를 받아온다. 기존의 `kill` 시스템콜을 활용하여, 성공 여부를 출력한다.

```
// kill process
if (kill(pid) == 0)
{
    printf(1, "kill [success]\n");
}
else
{
    printf(1, "kill [failed]\n");
}
```

- execute: sh.c에서 exec 시스템 콜을 호출하는 과정을 참고하였다. path와 stack size를 파싱하고, pmanager로부터 새롭게 fork한 자식 프로세스가 해당 path에 해당하는 프로세스를 stacksize 개수대로 실행하도록 한다. 이때, 종료를 기다리지 않으므로 별다르게 wait을 하지는 않는다.

```
if (fork1() == 0)
{
    // TODO: fork를 한번 더 해서, wait을 pmanager shel
    // 그런데 이렇게 하더라도 결국 pmanager가 처음 fork
    // piazza 참고(좀비가 되는게 맞음)

    exec2(strPath, argv, stacksize);
    printf(2, "execute %s failed\n", strPath);

    exit();
}
```

- memlim: setmemorylimit 시스템 콜을 호출하여 특정 pid의 memory limit을 설정해준다.

  단, 프로세스의 메모리는 thread의 메모리를 고려해야하는데, 이때 sz를 비교하는 구문이 setmemorylimit 내부에 있으므로 설정하려는 memory limit이 전체 process의 메모리 사이즈보다 작은지 확인할 수 있다.

성공 여부를 출력한다.

```
if (setmemorylimit(pid, mlimit) == 0)
{
    printf(1, "memlim [success]\n");
}
else
{
    printf(1, "memlim [failed]: system call failed\n");
}
```

threading

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

💡
- 새 스레드를 생성하고 시작한다.
- thread : 스레드의 id를 지정한다.
- start_routine : 스레드가 시작할 함수를 지정한다.
  새로운 스레드는 start_routine이 가리키는 함수에서 시작하게 된다.
- arg : 스레드의 start_routine에 전달할 인자다.
- return : 스레드가 성공적으로 만들어졌으면 0, 에러가 있다면 -1을 반환한다.

thread_create는 기존의 fork와 exec의 일부를 합쳐놓은 로직을 가진다. 우선, fork의 첫부분처럼 thread_create를 호출하게 되면, 내부적으로 allocproc을 통해 새로운 kernel stack을 할당하거나 trapframe을 호출하고(즉 thread만의 것을 갖는다), 기타 proc 멤버 변수를 초기화해주는 등의 과정을 거친다. 단, process를 할당하는게 아니라 thread를 생성하는 것이므로 pid를 조정해준다.

```
// Allocate process.
// 새로운 LWP를 ptable에 생성한다.
// - ptable의 UNUSED 상태의 entry를 찾아서 멤버 변수를 초기화해준다.
// - 생성될 thread만의 kernel stack을 할당하고, trap frame과 context를 셋업해준다.
if ((np = allocproc()) == 0)
{
  return -1;
}
--nextpid; // allocproc에서 nextpid가 증가하지만, thread를 만들 때는 pid가 증가하면 안됨
```

이후에는 thread에서 특별히 더 해주어야하는 main thread 설정과, tid등을 설정해준다.

```
struct proc *mthread; // main thread를 가리키는 포인터
if (curproc->mthread) // 스레드를 새로 만드려는 lwp에 이미 ma
{
  mthread = curproc->mthread; // 해당 mthread 포인터를 찾아준
}
else
{
  mthread = curproc; // 없으면 현재 스레드를 생성하려는 프로세
}

// Copy process state from main thread
np->parent = mthread->parent;
np->pid = mthread->pid; // main process와 동일한 pid
np->tid = nexttid++;    // thread 간의 구별을 위한 tid 할당(
*np->tf = *mthread->tf;
np->mthread = mthread;

// Clear %eax so that fork returns 0 in the child.
np->tf->eax = 0;

// copy opened file
for (int i = 0; i < NOFILE; i++)
  if (mthread->ofile[i])
    np->ofile[i] = filedup(mthread->ofile[i]);

// copy current directory
np->cwd = idup(mthread->cwd);

// copy the name of process
safestrcpy(np->name, mthread->name, sizeof(mthread->name));
```

이제 exec 파트로 넘어온다. thread는 다른 thread들과 리소스(text, data)를 공유하게 되는데, 즉 pgdir을 main thread의 것을 그대로 사용함으로써 이를 구현할 수 있다.

```
/* exec part */
pde_t *pgdir = mthread->pgdir;
```

그러나 stack은 thread만의 stack을 사용해야한다. 따라서, stack을 copyalloc하지 않고 allocuvm을 통해 새롭게 할당해준다. (guard page도 하나 할당해준다.)

```
if ((sz = allocuvm(pgdir, sz, sz + 2 * PGSIZE)) == 0)
{
  goto bad;
}
clearpteu(pgdir, (char *)(sz - 2 * PGSIZE)); // guard pa
```

이후에는 thread를 실행하기 위해 argument를 설정하고, 기존의 exec과 유사하게 copyout을 검사하는 등의 과정을 거친다. 추가적으로 memory limit을 검사한다.

```
arguments[0] = 0xFFFFFFFF; // return address
arguments[1] = (uint)arg;  // arguments
sp -= 8;                   // 2 * 4

if (copyout(pgdir, sp, arguments, 8) < 0)
{
  deallocuvm(pgdir, sz, sz + 2 * PGSIZE); // copy에 실패하면
  goto bad;
}

// sz가 memory limit을 초과했는지 확인
if (mthread->mlimit != 0 && sz > mthread->mlimit)
{
  deallocuvm(pgdir, sz, sz + 2 * PGSIZE); // copy에 실패하면
  goto bad;
}
```

```
mthread->sz = sz; // mthread에서 할당된 s

// commit to user image
np->sz = sz;
np->pgdir = pgdir;

np->tf->eip = (uint)start_routine; // 실
np->tf->esp = sp;                  // st
```

main thread(main process)의 stack에 스레드들이 쓰는 stack을 차곡차곡 쌓는다. 또한, 실행할 명령 주소인 start_routine을 thread trap frame의 eip에 설정해준다. 또한, esp에는 sp를 저장해준다. 이로써 thread create이 완료된다.

모든 작업을 완료하면, user program에서 thread를 관리할 수 있도록 함수 인자로 넘어온 thread_t * thread에 thread id를 포인터를 통해 넘겨준다. 이제 thread의 상태를 RUNNABLE로 바꿔서 스케줄러에 의해 스케줄되도록 한다. 이후 정상적으로 생성되었기 때문에 0을 반환한다.

도중 에러가 나면 state를 UNUSED로 바꾸고 -1을 리턴한다.

void thread_exit(void *retval);

- 스레드를 종료하고 값을 반환합니다.
- 모든 스레드는 반드시 이 함수를 통해 종료하고,
- 시작 함수의 끝에 도달하여 종료하는 경우는 고려하지 않습니다.

thread를 종료하고 retval을 설정한다. 기존의 `exit`과 유사하다. 스레드가 종료될 때 호출되는 함수로, 해당 스레드 proc 구조체의 멤버 변수인 retval을 함수 인자인 retval로 설정해준다. 나머지는 exit과 비슷하게 open되었던 file을 닫고, main thread를 깨우고, 자신은 ZOMBIE가 되어 main thread에서 자신의 자원을 회수해주기를 기다린다.

```
curproc->retval = retval;
```

```
// Parent or main thread might be sleep
if (curproc->mthread)
{
  wakeup1(curproc->mthread);
}
else
{
  wakeup1(curproc->parent);
}
```

int thread_join(thread_t thread, void **retval);

• 해당 스레드의 종료를 기다리고, 스레드가 thread_exit을 통해 반환한 값을 반환합니다.
• 스레드가 이미 종료되었다면 즉시 반환합니다.
• 스레드가 종료된 후, 스레드에 할당된 자원들을 회수하고 정리해야 합니다.
(페이지 테이블, 메모리, 스택 등)
• thread : join할 스레드의 id입니다.
• retval : 스레드가 반환한 값을 저장해 줍니다.
• return : 정상적으로 join했다면 0을, 그렇지 않다면 -1을 반환합니다.

기존의 `wait`과 유사하다. 단, 자식 프로세스가 있는지 찾는게 아니라 thread가 있는지 찾는다. thread_exit되어 retval을 저장해둔채 ZOMBIE가 되어 join되기를 기다리고 있는 sub thread가 있다면 해당 retval을 저장해두고 thread는 cleanThread를 통해 정리한다. 이때, pgdir은 할당해제하지 않았는데, 그 이유는 다른 thread에서 아직 해당 pgdir을 사용하고 있을 수 있기 때문이다.

무한 루프 안에서 thread exit한 thread가 있는지 찾는 과정은 다음과 같다.

```
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
{
  if (p->tid != thread)
    continue;
  haveThread = 1;
  if (p->state == ZOMBIE)
  {
    // Found one.
    *retval = p->retval;
    cleanThread(p);

    release(&ptable.lock);
    return 0;
  }
}
```

thread를 정리해주는 `cleanThread`의 구현은 다음과 같다.

```
void cleanThread(struct proc *p)
{
  p->state = UNUSED;
  kfree(p->kstack);
  p->kstack = 0;
  p->pid = 0;
  p->parent = 0;
  p->killed = 0;
  p->name[0] = 0;
  p->tid = 0;
  p->mthread = 0;
}
```

만약 exit된 스레드가 없다면 sleep을 통해 기다린다.

기존 system call과의 호환

- fork: 스레드에서 fork가 호출될 때, 기존의 fork 루틴을 실행할 수 있어야하는데, 해당 스레드의 주소 공간을 복사하고 새로운 프로세스를 시작해야한다. 따라서, fork를 호출한 thread의 pgdir을 복사하는 등의 과정은 그대로 거친다. 다만, thread가 아닌 main thread가 부모가 되어야하므로 해당 예외처리 과정을 추가했다.

```
if (np->tid == 0)
{
  np->parent = curproc;
}
else
{
  np->parent = curproc->mthread;
}
```

- exec: exec가 실행되면 기존 프로세스의 모든 스레드들이 정리되어야 하고, 하나의 스레드에서 새로운 프로세스가 시작하고 나머지 스레드는 종료되어야 한다. 따라서, exec 및 exec2 함수 내부에 다른 thread를 모두 정리하는 과정과, 자신이 sub thread인 경우 main process가 되는 과정을 추가하였다.

```
cleanOtherThreadsForExec(curproc->pid, curproc->tid);
if (curproc->mthread != 0 || curproc->tid != 0)
{
  makeMainThread(curproc);
}
```

- sbrk: sbrk는, 이미 할당된 곳에 할당되는 것을 막기 위해(main process의 stack에 thread의 stack이 차곡차곡 쌓이는데, 스레드에서 sbrk를 호출하면 자기 자신의 윗 stack이 이미 할당된걸 모를 수도 있음) main thread의 sz를 이용하도록 한다. sbrk 시스템 콜 내부적으로 호출되는 growproc에서도 mthread의 sz를 이용하는 부분을 추가하였다. 즉, main thread를 통하여 추가적인 메모리를 할당하게 된다. (혹은 dealloc)
  - sbrk 수정 사항

```
if (p->tid == 0 || !(p->mthread)) // su
{
  addr = p->sz;
}
else
{
  addr = p->mthread->sz; // 본인이 sub t
}
```

  - growproc 수정 사항

```
int growproc(int n)
{
  uint sz;
  struct proc *curproc = myproc();
  struct proc *mthread = curproc->mthread;

  if (!(mthread) || curproc->tid == 0) // mthread가 없으면, 본인이 mthread라는 뜻이므로
    mthread = curproc;                 // mthread를 current proc으로 설정해준다.
      You, 5 hours ago • implement project2: process management & LWP thre…
  sz = mthread->sz;
```

```
    if ((sz = allocuvm(mthread->pgdir, sz, sz + n)) == 0)
      return -1;
  }
  else if (n < 0)
  {
    if ((sz = deallocuvm(mthread->pgdir, sz, sz + n)) == 0)
      return -1;
  }
```

- kill: kill된 프로세스는 추후 trap.c에서 exit을 호출하게 되므로, exit에서 하나 이상의 스레드가 kill 되면 프로세스 내의 모든 스레드가 종료되어야 하는 것과, kill 및 종료된 스레드의 자원들은 모두 정리되고 회수되어야 하는 것을 구현하였다.
- sleep, pipe: 변경 사항 없음
- exit: 스레드가 exit을 호출하면 프로세스 내의 모든 스레드가 종료되어야 하는 것과, 종료된 스레드의 자원들은 모두 정리되고 회수되어야 하는 것을 구현하였다.

  즉, ptable을 순회하면서 우선 다른 스레드를 모두 정리한다.

```
// clean other threads
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
{
  if (p->pid == curproc->pid && p->tid != curproc->tid) // p
  {
    cleanThread(p);
  }
}
```

이후, parent를 깨워서 ZOMBIE가 된 자신을 정리해주도록 요청한다.

threading을 구현하기 위한 부가적인 함수

```
void      cleanThread(struct proc* p);
void      cleanOtherThreadsForExec(int pid, int tid);
void      makeMainThread(struct proc *p);
```

- cleanThread: 앞서 설명한 바와 같이, thread가 추후 main process에 의해 회수될 때 혹은 exec을 호출하면서 다른 스레드를 정리해야 할 때 등의 상황에서 호출된다.
- cleanOtherThreadsForExec: exec을 할 때, exec을 호출한 자신을 제외한 다른 스레드를 모두 정리하기 위해 사용한다.
- makeMainThread: exec을 할 때, 자신이 이제는 스레드가 아니라 mainProcess가 되기 위해 호출한다.

Test Result

pmanager

- list

```
-----------------------------------------------------------------
name: init | pid: 1 | stack pages: 1 | memory: 12288 | memlim: 0 |
name: sh | pid: 2 | stack pages: 1 | memory: 16384 | memlim: 0 |
name: thread_exec | pid: 3 | stack pages: 1 | memory: 53248 | memlim: 0 |
   thread for process 3 with thread id: 1
   thread for process 3 with thread id: 2
   thread for process 3 with thread id: 3
   thread for process 3 with thread id: 4
   thread for process 3 with thread id: 5
-----------------------------------------------------------------
```

LWP thread는 디자인에서 언급한 바와 같이 별도로 출력된다

- kill
- execute

```
$ pmanager
>> list
-----------------------------------------------------------------
name: init | pid: 1 | stack pages: 1 | memory: 12288 | memlim: 0 |
name: sh | pid: 2 | stack pages: 1 | memory: 16384 | memlim: 0 |
name: pmanager | pid: 3 | stack pages: 1 | memory: 16384 | memlim: 0 |
-----------------------------------------------------------------
>> memlim 3 10000
memlim [failed]: system call failed
>> memlim 3 10
memlim [failed]: system call failed
>> memlim 3 50000
memlim [success]
>> list
-----------------------------------------------------------------
name: init | pid: 1 | stack pages: 1 | memory: 12288 | memlim: 0 |
name: sh | pid: 2 | stack pages: 1 | memory: 16384 | memlim: 0 |
name: pmanager | pid: 3 | stack pages: 1 | memory: 16384 | memlim: 50000 |
-----------------------------------------------------------------
>> execute thread_exec
execute [failed]: invalid stacksize input
>> execute thread_exec 1
>> Thread exec test start
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Executing...
Hello, thread!
```

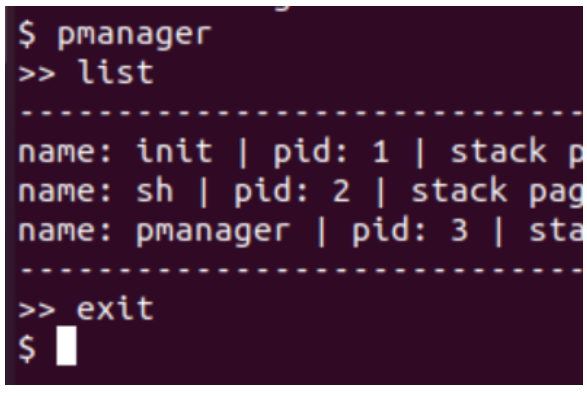
execute <path> <stacksize>를 통해 thread_exec를 실행한 모습이다. 단, wait을 해주지 않으므로(동시에 실행되어야해서) 이후 pmanager를 exit할 때 ZOMBIE! 가 호출된다.

- memlim

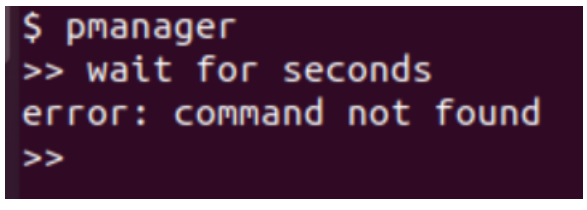
```
$ pmanager
>> list
-----------------------------------------------------------------
name: init | pid: 1 | stack pages: 1 | memory: 12288 | memlim: 0 |
name: sh | pid: 2 | stack pages: 1 | memory: 16384 | memlim: 0 |
name: pmanager | pid: 3 | stack pages: 1 | memory: 16384 | memlim: 0 |
-----------------------------------------------------------------
>> memlim 3 20000
memlim [success]
>> list
-----------------------------------------------------------------
name: init | pid: 1 | stack pages: 1 | memory: 12288 | memlim: 0 |
name: sh | pid: 2 | stack pages: 1 | memory: 16384 | memlim: 0 |
name: pmanager | pid: 3 | stack pages: 1 | memory: 16384 | memlim: 20000 |
-----------------------------------------------------------------
>> memlim 3 10
memlim [failed]: system call failed
>>
```

메모리 리밋이 현재 메모리 할당된 값보다 크면 잘 제한이 되고, 작으면 에러가 발생한다.

- exit

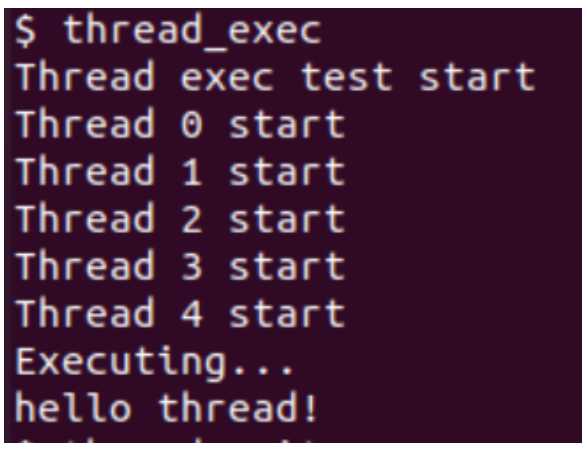


- 그외의 input



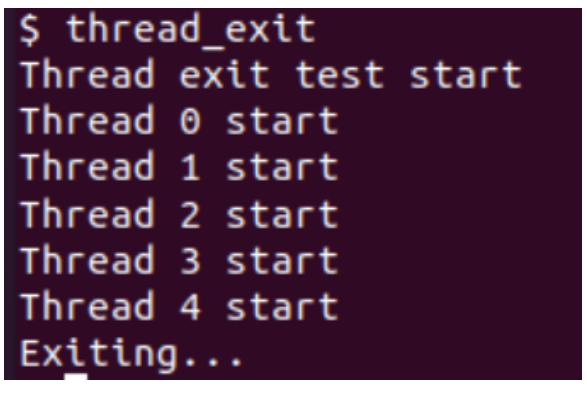
LWP Threading

thread_exec



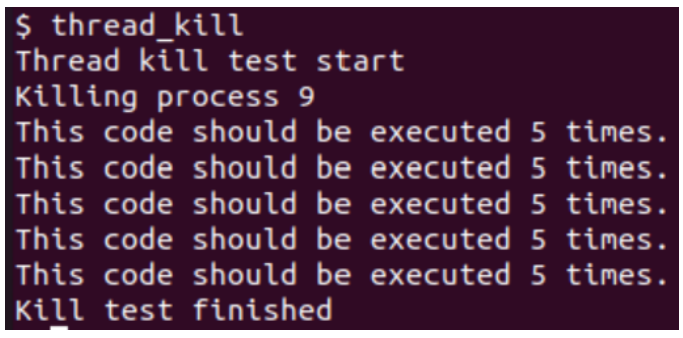
NUM_THREAD만큼의 스레드를 만들고 해당 스레드에서 thread_main을 start_routine으로 받아 작동하는 모습이다. 이때, arg가 0인 스레드는 0번째 스레드로 hello_thread가 실행된다.

thread_exit



스레드와 exit()가 잘 호환되는지 확인한다. 스레드를 시작한 뒤 exit를 호출하게 되면, 다른 스레드도 함께 정리되는 모습을 볼 수 있다.

thread_kill



thread_test

```
$ thread_test
Test 1: Basic test
Thread 0 start
Thread 1 start
Thread 0 end
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 1 start
Child of thread 2 start
Child of thread 4 start
Child of thread 3 start
Child of thread 0 end
Thread 0 end
Child of thread 1 end
Child of thread 2 end
Child of thread 4 end
Thread 1 end
Thread 2 end
Thread 4 end
Child of thread 3 end
Thread 3 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!
$ QEMU: Terminated
```

1. 스레드 API의 기본 기능(create, exit, join)과 스레드 사이에서 메모리가 잘 공유되는지 확인할 수 있다.
2. fork와의 호환성을 확인하는 테스트를 했을 때 올바른 동작을 확인할 수 있었다. 즉, 자식 프로세스가 부모 프로세스와 주소 공간을 공유하지 않음을 확인할 수 있다.
3. sbrk와의 호환성을 확인하는 테스트를 했을 때, 여러 스레드가 중복된(겹치는) 주소에 할당되지 않았는지, 동시에 할당받고 해제했을 때 문제가 없는지 확인해볼 수 있다.

Trouble Shooting

디자인 관련

xv6의 프로세스 구조

- 처음에는 디자인과 관련하여 감이 쉽게 오지 않아서, xv6 book에서 process 파트를 자세히 읽어보았다. thread와 process에 대한 이해 및 xv6에서 내부적으로 사용되는 함수에 대한 이해가 부족한 것이 원인이었다.
    - xv6에서는 data, text 위로 stack이 고정된 page size만큼 할당되게 되므로, thread는 같은 data와 text를 공유하면서 thread 고유의 stack을 위에 차곡차곡 쌓게 하자는 결론에 이를 수 있었다.

threading와 proc 구조체 멤버 변수 추가 관련

크게 두가지 구현 방법을 생각했었다.

1. 기존의 proc 구조체에 곧바로 thread와 관련된 변수를 추가해서 프로세스이면서 thread인 것처럼 작동하도록 하는 방법

2. threading을 위한 변수를 가진 구조체를 만들고, proc 구조체 안에 thread 구조체 배열을 추가해서 main process에서 thread 배열을 가지는 것처럼 보이는 방법

2번의 경우 스케줄링 시 고려해줘야할 점이 많았다. 이를테면, pid가 10인 프로세스에서 각각 tid가 1, 2, 3인 스레드를 가지고 있을 때, 멀티태스킹을 위해서는 각기 다른 프로세스로 취급되어(LWP도 프로세스이므로, 프로세스처럼 스케줄링 되어야한다는 답변에 의거하여) 스케줄링이 되어야하는데, 특정 프로세스 내부의 배열로 스레드들이 존재하기 때문에 ptable을 다 돈뒤 ttable을 가진 ptable을 내부적으로 순회하도록 했었다. 하지만 그러면 process의 ttable을 다 돌아야만 다름 프로세스로 넘어갈 수 있는 이론적인 방법이었고, 여러 에러가 다수 발생하여 1번의 방법으로 구현하였다.

thread를 create할 때 무엇을 그대로 가져오고 무엇을 새로 생성하는지

이론적으로 thread는 text, data 등의 리소스를 공유하고, 별도의 stack을 가진다고 하였는데, 코드적으로는 무엇을 main thread로부터 그대로 가져오면 되고 무엇을 새로 할당해야하는지 초반에 알기 어려웠었다. xv6 book이나 이론 수업, piazza를 통해 좀더 명세를 이해할 수 있었다.

- pgdir을 공유하는 것과 관련하여, pgdir가 곧 page table로, 코드와 데이터를 가리킨다고 이해하였다. 그래서 pgdir을 main thread의 것을 그대로 공유하여 사용한다.
- copyuvm은 메모리 공간을 복사하여 가져오지만, thread는 아예 새로운 내용으로 채워주면 되기 때문에 기존의 fork와 달리 exec처럼 allocuvm을 해주어야한다.
- 이론 수업을 들으면서 여러 함수가 호출되는 이유(예: copyuvm과 allocuvm을 따로 두는 차이)를 좀더 명쾌하게 이해할 수 있었다.

thread create, exit, join이 각각 어떤 기존의 시스템 콜과 유사한지

thread 역시 process와 유사하게 스케줄링 되는 하나의 작업 단위이므로 기존의 시스템 콜 코드와 비슷하게 작성하면 될 것 같다고 생각했었다. 그러나 각 시스템 콜에서 어떤 부분을 수정해서 가져와야하는지, 또한 어떤 시스템 콜과 유사하게 작동해야하는지 정확하게 이해하지 못했었다.

전반적으로는 linux의 pthread와, LWP에 관해서 좀더 공부해보기도 하고, 이전에 객체지향 프로그래밍에서 Java의 thread에 관해 배웠던 자료를 다시 복습해보면서 main thread(main process)가 존재하며 main thread에서 DB, 서버 응답 같은 작업을 하기 위해 thread를 활용하며, 해당 작업을 위해 thread를 만들고, thread를 exit하고, thread를 join하는 등의 실행 흐름을 통해 각각의 thread_create, thread_exit, thread_join이 각각 fork+exec, exit, wait과 유사함을 알 수 있었다.

버그

굉장히 많은 버그가 발생했었으나 초반에 기록했던 버그 및 전반적인 형태를 작성합니다.

make가 되지 않는 문제

```
make: execvp: ./sign.pl: Permission denied
make: *** [Makefile:109: bootblock] Error 127
```

linux - Why do I get permission denied when I try use "make" to install something? - Stack Overflow

solved by:

```
chmod 777 -R xv6-public/
```

---

pmanager에서 input string을 파싱할 때, 로컬 변수에 이전 입력값이 남아있는 문제

해당 문제로 인해 `execute`, `memlim`을 할 때 의도하지 않은 값이 변수에 넘어가는 문제가 발생했었다. 이전 입력 문자열보다 더 긴 문자열을 넣으면 덮어씌워지지만, 짧은 input을 넣을 경우 `str-` 접두어를 가진 변수 배열에 이전 입력값이 남아서 의도하지 않은 입력이 이루어지게 되었다. 따라서, 명시적으로 초기화하는 과정을 추가해주었다.

다음의 버그는 한가지 원인에 의해 발생한다고 단정짓기는 어려웠으며 여러가지 원인에 의해 발생하거나, 각기 다른 코드에서도 같은 문제로 인해 자주 발생했었던 버그입니다. 에러 이름으로 구글링을 해보거나, gdb를 사용해보거나(다만 user code는 gdb의 브레이크 포인트로 설정이 안되어서 gdb를 scheduler 구현 때처럼 많이 사용하지는 못하였습니다), 단순하게는 printf로 흐름을 추적해보기도 하였습니다.

xv6가 재부팅되는 문제

여러 원인에 의해 발생하는 것으로 추정되는데, 그중에서도 특히 init을 재호출할 때 발생할 수 있다는 구글링 결과를 보았다. 혹은 커널 영역을 침범할 때도 재부팅될 수 있다는 검색 결과를 볼 수 있었다. 다만 아직 정확한 원인은 알지 못하는 상태이며, 코드를 전반적으로 다시 작성했다.

remap panic

```
Test 3: Sbrk test
Thread 0 start
lapicid 0: panic: remap
 80107757 80107b89 80103f47 80106499 8010565d 801069f1 80106734 0 0 0
```

remap은 이미 할당된 메모리에 재할당할 때 생기는 trap임에 중점을 두고 원인을 찾아보았다. sbrk와 growproc에서, sub thread 자체는 자신의 상단 stack에 또다른 thread의 stack이 할당되어있는지 아닌지 알 수 없으며, main thread가 이를 관리한다. 따라서, sbrk 및 growproc을 하는 경우 main thread를 통해서 메모리를 할당해주어야 한다.

acquire panic

lock을 잘못 잡아서 발생하는 panic으로, 굉장히 많은 상황에서 발생했었다. 코드의 실행 흐름을 하나하나 따라가보면서 lock이 중복적으로 잡히거나, 잡은적이 없는데 release하는 등의 코드가 있는지 일일히 살펴보았다. 자원을 해제해주면서 lock을 이미 잡았는데 내부적으로 또 lock을 잡는 함수를 호출했을 때도 발생했었다.

각종 panic, trap 13, trap 14, xv6가 멈추는 문제 등

여러가지 원인에 의해 발생하는 trap이었기 때문에, 원인을 정확하게 판별해내지는 못했으며, proc 구조체 내부에 thread 배열을 멤버 변수로 두었던 첫번째 방법에서 thread 역시 proc 구조체 전체를 소유하고 ptable에 allocated 되도록 코드를 전반적으로 수정하였다.