🎨 Design

1. LWP 구현

스레드는 프로세스와 비슷하게 취급되며, cpu 스케줄링의 단위가 되므로 동일한 PCB(proc 구조체)에 스레드 / 프로세스(메인 스레드) 여부를 구분하는 변수를 추가하는 방법으로 구현하였습니다.
LWP와 관련하여 구현해야 할 함수는 thread_create, thread_exit, thread_join 세 개인데, 각각 아래와 같이 프로세스를 다루는 함수를 참고하여 구현하였습니다.

Thread	**Process**
thread_create	fork, exec
thread_exit	exit
thread_join	wait

(1) thread_create

- 스레드를 프로세스와 동일하게 ptable 배열의 요소로 사용하기 위해 allocproc 함수를 사용합니다.
- 스레드는 프로세스와 code, data, heap 영역을 공유하고, 독립적인 stack 영역을 갖습니다. 새로운 스택 영역을 할당하는 exec 코드를 참고하여 메모리 공간을 할당합니다.
- sz는 프로세스가 점유하는 메모리 공간을 의미하는데, stack 위에 heap 영역이 할당되는 xv6의 특성 상 같은 pid를 공유하는 모든 프로세스의 sz가 동일하게 유지되어야 heap 메모리의 데이터를 공유할 수 있습니다. 따라서 thread_create과정에서 pid가 같은 모든 스레드(프로세스)의 sz를 갱신 해 주어야 합니다.
- 스레드는 프로세스의 파일을 공유하므로 fork의 파일 복사와 동일한 방식으로 프로세스에도 ofile 정보를 초기화 해 줍니다.
- 스레드 생성 후 start_routine으로 전달된 함수를 호출하므로, 스레드 trapframe의 eip를 start_routine의 주소값으로 설정하고, user stack 공간에 arg를 저장합니다.
- start_routine은 thread_exit, thread_join으로 종료 되므로 return PC 값에 들어가는 값은 신경 쓰지 않아도 됩니다. exec 함수의 구현을 따라 0xffffffff 값을 저장합니다.

(2) thread_exit

- 프로세스를 종료하는 exit 함수와 동일하게 작성하되, wakeup1으로 부모 프로세스가 아니라 thread_join을 호출하여 해당 스레드를 기다리고 있는 프로세스(스레드)를 깨웁니다.

(3) thread_join

- wait 함수와 동일하게 동작하되, 기다리는 스레드를 특정할 수 있습니다.
- 프로세스가 모든 스레드의 종료를 기다리지 않고 exit 하는 경우, 해당 pid의 스레드를 모두 정리하도록 구현하였습니다. 스레드를 정리하지 않는 경우 exit 안에서 할당 해제된 page table에 스레드가 비정상적으로 접근하면서 xv6가 reboot 되는 문제가 있습니다.

2. pmanager 및 관련 시스템 콜

pmanager의 기본적인 CLI 구조는 xv6의 sh 프로그램의 구조를 참고하였습니다. execute 명령어는

sh 프로그램의 백그라운드 실행과 동일하게 동작하도록 디자인하였습니다.
exec2는 exec 함수에서 user stack을 할당하는 부분을 수정하여 구현하였고, setmemorylimit은 프로세스마다 메모리 limit이 정의되어야 하므로 PCB(proc 구조체)에 새로운 변수를 추가하는 것으로 디자인하였습니다.

🛠 Implement

(1) thread_create

독립적인 메모리 공간을 가지는 새로운 프로세스가 생성되는 fork 함수에는 아래와 같이 페이지 테이블을 복사하는 코드가 있습니다.

```
207     // Copy process state from proc.
208     if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
```

새로 만든 메모리 공간에 페이지 테이블을 매핑하는 exec 함수에는 아래와 같은 코드가 있습니다.

```
 98     // Commit to the user image.
 99     oldpgdir = curproc->pgdir;
100     curproc->pgdir = pgdir;
```

따라서 thread_create에서 새로 생성되는 스레드의 pgdir을 메인 스레드(프로세스)와 동일하게 초기화 해 주면 메모리 영역을 공유하게 됩니다.
이제 stack 영역을 새로 할당 해 주어야 하므로, allocuvm을 사용하여 user stack을 할당 해 줍니다. 이 때 스택용 페이지의 개수는 thread_create를 호출한 스레드의 스택용 페이지 개수와 동일하게 생성되도록 하였습니다. 스택용 페이지를 할당하는 코드는 exec 함수를 참고하였습니다.

```
609     // allocate user stack
610     sz = nt->sz;
611     if((sz = allocuvm(nt->pgdir, sz, sz + (curproc->stacksz + 1)*PGSIZE)) == 0)
612         return -1;
613     clearpteu(nt->pgdir, (char*)(sz - (curproc->stacksz + 1)*PGSIZE));
614     sp = sz;
615     nt->sz = sz;
616     acquire(&ptable.lock);
617     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
618       if(p->pid == curproc->pid)
619         p->sz = sz;
620     }
621     release(&ptable.lock);
```

스택을 할당하여 메모리 공간이 증가한 만큼 같은 pid를 공유하는 스레드의 sz도 갱신 해 주어야 합니다. 이는 sbrk로 힙 메모리를 할당 할 때 어떤 스레드에서 할당하더라도 다른 스레드의 스택 영역을 침범하지 않도록 하기 위함입니다. 같은 pid를 공유하는 모든 스레드는 항상 sz 값이 동일합니다.

(2) thread_exit

thread를 종료하기 위한 함수이며 exit 함수와 wakeup1 부분에서만 차이가 있습니다.

```
673     // Parent might be sleeping in thread_join().
674     wakeup1(curthread->main); // parent가 아니라 메인 스레드를 깨워야 함
```

부모 프로세스가 아니라 join을 호출한 메인 스레드를 깨웁니다.

(3) thread_join

파라미터로 받은 스레드의 종료를 기다리고 종료되었을 때 자원을 회수합니다. 프로세스의 종료를 기다리는 wait 함수와 기본적으로 동일하나 유저 메모리를 해제하지 않습니다.

```
694   // Equivalent to wait(void) for process
695   int thread_join(thread_t thread, void **retval)
696   {
697     struct proc *p;
698     int havekids;
699     struct proc *curthread = myproc();
700
701     acquire(&ptable.lock);
702     for(;;){
703       // Scan through table looking for exited children.
704       havekids = 0;
705       for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
706         if(p->main != curthread)
707           continue;
708         havekids = 1;
709         if(p->tid == thread && p->state == ZOMBIE){
710           // Found one.
711           *retval = p->retval;
712           kfree(p->kstack);
713           p->kstack = 0;
714           // freevm(p->pgdir);
715           p->pid = 0;
716           p->parent = 0;
717           p->name[0] = 0;
718           p->killed = 0;
719           p->mlimit = 0;
720           p->isThread = 0;
721           p->main = 0;
722           p->retval = 0;
723           p->tid = 0;
724           p->state = UNUSED;
725           release(&ptable.lock);
726           return 0;
727         }
728       }
```

스레드에 개별적으로 할당된 커널 스택 페이지는 해제하되, freevm(p->pgdir)는 주석 처리하였습니다. 유저 메모리에 대한 정리는 프로세스 자원이 회수되는 wait 함수 안에서 이루어집니다.

추가로 thread_join은 기다릴 스레드 id를 지정할 수 있는데, thread_create를 호출한 스레드와 thread_join을 호출한 스레드가 다를 경우 join을 호출한 스레드가 wakeup1 신호를 받지 못할 수 있습니다. 다만 저는 이 케이스가 유저 프로그래머의 실수라고 생각하여 따로 처리 해 주지 않았습니다. wakeup1 신호를 받지 못하고 지속적인 대기 상태에 있는 스레드가 있다면 프로세스가 종료될 때 정리됩니다.

(4) exec2

기존 exec 함수에 스택용 페이지를 할당하는 부분만 수정하여 구현하였습니다.

```
178     if((sz = allocuvm(pgdir, sz, sz + (stacksize + 1)*PGSIZE)) == 0)
179       goto bad;
180     clearpteu(pgdir, (char*)(sz - (stacksize + 1)*PGSIZE));
181     sp = sz;
```

(5) setmemorylimit

PCB(proc 구조체)에 mlimit 변수를 추가하여 limit을 설정할 수 있도록 하였고, 명세에 맞게 해당 변수를 설정하는 시스템 콜을 추가하였습니다.

```
557   int
558   setmemorylimit(int pid, int limit)
559   {
560     struct proc *p;
561
562     if(limit < 0)
563       return -1;
564     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
565       if(p->pid == pid){
566         if(limit && p->sz > limit)
567           return -1;
568         else{
569           p->mlimit = limit;
570           return 0;
571         }
572       }
573     }
574     return -1; // pid not found
575   }
```

(6) pmanager

sh.c 를 참고하여 커맨드 라인 인터페이스를 구성하고 각 명령어마다 fork1로 새로운 프로세스를 생성하여 실행하도록 구현하였습니다.

- list

현재 실행 중인 프로세스 (메모리에 올라와 있는 프로세스, RUNNING, RUNNABLE, SLEEPING) 정보를 ps 구조체 변수로 담아 출력합니다.

process_status 시스템 콜을 이용합니다.

```
67  // process status
68  struct ps {
69    int active;         // # of running & runnable processes
70    struct info {       // information of process
71      uint sz;
72      int pid;
73      int stacksz;
74      int mlimit;
75      char name[16];
76    } info[NPROC];
77  };
```

```
793  void
794  process_status(struct ps *s)
795  {
796    struct proc *p;
797    int i;
798
799    acquire(&ptable.lock);
800    for(p = ptable.proc, i = 0; p < &ptable.proc[NPROC]; p++){
801      if((p->state == RUNNABLE || p->state == RUNNING || p->state == SLEEPING) && !p->isThread){
802        s->info[i].sz = p->sz;
803        s->info[i].pid = p->pid;
804        s->info[i].stacksz = p->stacksz;
805        s->info[i].mlimit = p->mlimit;
806        safestrcpy(s->info[i].name, p->name, sizeof(s->info[i].name));
807        i++;
808      }
809    }
810    s->active = i;
811    release(&ptable.lock);
812  }
```

- kill

kill 시스템 콜을 이용하여 간단히 구현하였습니다.

```
89   void
90   runkill(char *strpid)
91   {
92       int pid;
93
94       pid = atoi(strpid);
95       if (kill(pid) < 0)
96           panic("kill error!");
97       printf(1, "process %d successfully killed\n", pid);
98   }
```

- execute

백그라운드 실행을 위해 fork를 한번 더 수행하고 wait 없이 pmanager 안의 "execute" 프로세스를 종료합니다. 백그라운드 프로세스가 정상적으로 종료되고 나면 init 프로세스에 의해 회수되어 zombie! 메시지가 출력됩니다.

```
100  void
101  runexec(char *path, char *strstacksize)
102  {
103      int stacksize;
104      char *argv[2];
105
106      argv[0] = path;
107      argv[1] = 0;
108      stacksize = atoi(strstacksize);
109      if(fork1() == 0){
110          if(path == 0 || exec2(path, argv, stacksize) < 0){
111              printf(2, "execute %s failed\n", path);
112              exit();
113          }
114      }
115  }
```

- memlim

```
117  void
118  runmemlim(char *strpid, char *strlimit)
119  {
120      int pid;
121      int limit;
122
123      pid = atoi(strpid);
124      limit = atoi(strlimit);
125      if(setmemorylimit(pid, limit) < 0)
126          panic("memlim error!");
127      printf(1, "memory limit of process %d successfully set to %d\n", pid, limit);
128  }
```

setmemorylimit을 이용하여 구현하였습니다. setmemorylimit 내부적으로 sz 변수를 사용하고 있으므로 프로세스의 메모리는 thread의 메모리를 고려합니다.

- exit

```
154      while(getcmd(buf, sizeof(buf)) >= 0){
155          buf[strlen(buf)-1] = 0; // chop \n
156          if(!strcmp(buf, "exit")) {
157              break;
158          }
```

main 의 입력 버퍼에 exit가 들어오면 바로 프로그램을 종료합니다.

(7) exec

exec 실행 시 해당 프로세스의 모든 스레드를 정리하기 위해 thread_clear 함수를 추가하였습니다. 스레드에서 exec 가 실행 된 경우 해당 스레드를 메인 스레드로 바꾸고 나머지 스레드를 모두 정리합니다.

```
741  void
742  thread_clear(struct proc *curproc)
743  {
744    struct proc *p;
745    int fd;
746
747    // 스레드에서 exec가 호출된 경우
748    // 스레드보다 process가 먼저 exit 되는 경우
749    if(curproc->isThread) {
750      curproc->isThread = 0;
751      // make original main thread (process) to thread
752      curproc->main->isThread = 1;
753      curproc->main = 0;
754    }
```

스레드의 메인 스레드 여부를 확인하고 아닌 경우 메인 스레드로 바꿔 주는 부분입니다.

```
755     // clear all threads
756     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
757       if(p->pid == curproc->pid && p->isThread && p->state != UNUSED) {
758         if(p == initproc)
759           panic("init exiting");
760         if(p->state != ZOMBIE) {
761           // Close all open files.
762           for(fd = 0; fd < NOFILE; fd++){
763             if(p->ofile[fd]){
764               fileclose(p->ofile[fd]);
765               p->ofile[fd] = 0;
766             }
767           }
768           begin_op();
769           iput(p->cwd);
770           end_op();
771           p->cwd = 0;
772           p->state = ZOMBIE;
773         }
774         if(p->state == ZOMBIE) {
775           kfree(p->kstack);
776           p->kstack = 0;
777           p->pid = 0;
778           p->parent = 0;
779           p->name[0] = 0;
780           p->killed = 0;
781           p->mlimit = 0;
782           p->isThread = 0;
783           p->main = 0;
784           p->retval = 0;
785           p->tid = 0;
786           p->state = UNUSED;
787         }
788         // sz의 경우 exec 안에서 재설정 되므로 갱신 필요 없음
789       }
790     }
791   }
```

exec 로 새로운 프로세스를 실행하는 메인 스레드 (프로세스)를 제외한 모든 스레드 자원을 회수합니다. 이 때 가능한 스레드의 state는 RUNNING, RUNNABLE, ZOMBIE인데, ZOMBIE의 경우 해당 스레드에 대해 thread_exit가 호출되었음을 의미합니다. 따라서 thread_exit에서의 작업을 건너뛰고 thread_join의 작업을 수행하여 PCB(proc 구조체)의 변수들을 초기화합니다.

(8) growproc

프로세스에 메모리를 할당하는 데 사용되는 sbrk 시스템콜은 growproc 함수를 사용하므로, growproc 함수를 수정하였습니다.

```
158  // Grow current process's memory by n bytes.
159  // Return 0 on success, -1 on failure.
160  int
161  growproc(int n)
162  {
163    uint sz;
164    struct proc *curproc = myproc();
165    struct proc *p;
166
167    sz = curproc->sz;
168    if(n > 0){
169      // sz: oldsz, sz + n: newsz
170      if (curproc->mlimit != 0 && sz + n > curproc->mlimit) // memory limitation
171        return -1;
172      if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
173        return -1;
174    } else if(n < 0){
175      if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
176        return -1;
177    }
178    curproc->sz = sz;
179    // pid가 동일한 스레드 간 heap 메모리 영역을 공유할 수 있게 함
180    // 새로 생성되는 스레드에 대한 스택 영역이 힙 영역을 침범하지 않도록 함
181    acquire(&ptable.lock);
182    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
183      if(p->pid == curproc->pid)
184        p->sz = curproc->sz;
185    }
186    release(&ptable.lock);
187    switchuvm(curproc);
188    return 0;
189  }
```

xv6에서 heap 영역은 code, data, stack 영역을 차례로 올린 후 그 위에 존재합니다. 따라서 growproc에서는 프로세스의 sz 변수를 n만큼 증가시켜 heap 메모리를 할당하고 있습니다. 이 공간은 모든 스레드에 의해 접근되어야 하므로, 기존의 코드에 sz를 갱신하는 부분을 추가하여 모든 스레드에서 할당된 메모리 공간을 공유하고 새롭게 할당하는 데에 오류가 없도록 하였습니다.

(9) fork, kill, sleep, pipe

스레드를 프로세스와 동일한 PCB(proc 구조체)로 관리하고 있기 때문에, 별 다른 코드 수정 없이 기존과 동일한 기능을 수행할 수 있습니다.

(10) exit

명세에 명확히 표시된 구현 사항은 아니지만, thread_join이 호출되지 않은 경우에 유저 메모리 공간이 임의로 해제되면서 (freevm(p->pgdir)) 남은 스레드가 할당되지 않은 페이지 공간을 참조하는 것은 올바른 커널 구현이 아니라고 생각하여 exit 함수에 thread_clear로 남아 있는 스레드를 모두 정리하는 코드를 추가하였습니다. "zombie!" 메시지를 띄우며 자식 프로세스의 비동기적 종료를 알리는 wait 함수와 달리, exit가 호출된 프로세스 pid의 스레드는 그 즉시 종료되며 출력되는 메시지가 없습니다. 따라서 제가 디자인한 xv6에서 스레드 간 작업 순서를 보장하는 것은 유저 프로그래머의 역할입니다.

(11) allocproc

프로세스 공간을 할당하기 위한 static 함수로 정의되어 있는데, 다른 부분은 스레드 할당에 그대로 사용해도 무방하지만 호출 될 때마다 전역변수로 선언된 pid가 증가한다는 문제가 있었습니다. 따라서 pid를 초기화하는 부분을 삭제하고 fork, create_thread 함수에서 초기화하도록 구현하였습니다.

```
201     // Allocate process.
202     if((np = allocproc()) == 0){
203       return -1;
204     }
205     np->pid = nextpid++;
```

```
586     // Allocate thread
587     if((nt = allocproc()) == 0){
588       return -1;
589     }
590     nt->pid = curproc->pid;
```

🌈 Result (Test)

컴파일, 실행 과정에서 특별히 고려해야 할 사항은 없습니다.

make clean; make; make fs.img 로 컴파일합니다.

qemu-system-i386 -nographic -serial mon:stdio -hdb fs.img xv6.img -smp 1 -m 512 로 xv6를 실행합니다.

테스트 결과는 아래와 같습니다.

(1) thread_test

```
$ thread_test
Test 1: Basic test
ThrThread 1ead 0 star start
t
Thread 0 end
Parent waiting for children...
Thread 1 end
Test 1 passed

Test 2: Fork test
Thread 1 start
Thread 2 start
Thread 3 sThread 4 startThread 0 startart

Child oft
 thread 1 start
CChild of thhilChild of thread 3 d of thread read 4 stChild of threstart
2 startart

ad 0 start
Child of thread 1 end
Thread 1 end
Child of thread 3Child of thread 2 end
Child of threChild of thre end
ad 4 end
ad 0 end
Thread 2Thread 3 end
 end
Thread 4 end
Thread 0 end
Test 2 passed
```

```
Test 3: Sbrk test
Thread 0 start
ThreThread 2 start
Thread 3 starThread 4 starad 1t
t
 start
Test 3 passed

All tests passed!
$
```

Test 1 basic test, Test 2 fork test, Test 3 sbrk test 모두 정상 동작합니다.

추가적으로 Test 1 에서 thread_join을 주석 처리하여 의도한 대로 동작하는지 테스트하였습니다.

```
134     printf(1, "Test 1: Basic test\n");
135     create_all(2, thread_basic);
136     sleep(100);
137     printf(1, "Parent waiting for children...\n");
138     // join_all(2);
139     if (status != 1) {
140       printf(1, "Join returned before thread exit,
141       failed();
142     }
143     printf(1, "Test 1 passed\n\n");
```

메인 스레드가 join으로 나머지 스레드의 종료를 기다리지 않습니다.

```
$ thread_test
Test 1: Basic test
Thread 0Thread 1 s start
Thread 0 entart
d
Parent waiting for children...
Join returned before thread exit, or the address space is not properly shared
Test failed!
$
```

메인 스레드가 exit를 통해 종료되고 스레드도 함께 종료되면서 자원이 회수됩니다.

(2) thread_exec

```
$ thread_exec
Thread exec test start
Thread 0 startThread 1Thread 2 start
Thread 3 start
Thread 4 start

 start
Executing...
Hello, thread!
$
```

(3) thread_exit

```
$ thread_exit
Thread exit test start
Thread Thread 1Thread 2Thread Thread 4 0 start
 st start
art3 start

start
Exiting...
$
```

(4) thread_kill

```
$ thread_kill
Thread kill test start
Killing prThis codThis code This code should beThis code shouocess 12e should beshoulld be executed 5 time
 executed 5 times.
d be exe executed 5 tis.
This code cuted 5 timmes.
should be executed 5 tes.
imes.
Kill test finished
$
```

🤯 Trouble shooting

(1) 프로세스 메모리 구조 관련

이번 과제는 프로세스의 메모리 구조를 파악하는 부분에서 어려움을 겪었습니다. 프로세스 메모리 공간이 낮은 주소부터 code -> data -> heap -> stack으로 할당되는 것으로 오해하고 디자인하기 시작하여, stack 공간을 어떻게 할당해야 할지 파악하는 데 시간이 오래 걸렸습니다. 기존 growproc, exec 코드를 분석하며 xv6의 메모리 구조가 code -> data -> stack -> heap 순서로 구성되어 있음을 알게 되었고, pid가 같은 스레드끼리 sz를 공유하며 sz 위쪽으로 stack이나 heap을 쌓는 방식으로 구현하였습니다.

(2) 스레드 종료 시 리부팅 문제

thread_join의 구현 중 xv6가 스레드를 실행시킨 후에 정상 종료되며 쉘이 실행되지 않고, 리부팅되는 문제가 있었습니다. 이 부분은 스레드가 page table을 할당 해제하여 발생한 문제였고, thread_join의 해당 부분을 삭제하여 해결하였습니다.

(3) panic: kfree 로 비정상 종료

스레드 테스트 코드가 종료되어 쉘까지 실행 된 후 일정 시간이 지나서 아래와 같은 에러 메시지와 함께 비정상 종료 되는 문제가 있었습니다.

```
$
lapicid 0: panic: kfree
 801028c5 80107803 8010436e 80105319 8010649d 80106234 0 0 0 0
```

디버깅 했을 때 wait 함수가 프로세스 뿐 아니라 스레드까지 detect 하여 정리하기 때문에 발생한 문제였고, wait 안에 아래처럼 스레드 여부를 체크하는 코드를 추가하여 해결하였습니다.

```
305         if(p->isThread) // wait thread in thread_join
306           continue;
```