운영체제 project#2

2018009125 조성우

pmanager

1. 상황 이해 및 디자인:

Pmanager는 새로 만든 시스템콜인 exec2와 setmemorylimit을 이용하여 프로세스들을 관리하는 유저 프로그램입니다. List, kill, execute, memlim exit 총 5개의 명령을 받습니다. list는 프로세스들의 정보를 나타내는 명령어입니다. 현재 실행중인(RUNNING, RUNNABLE, SLEEPING) 프로세스들의 pid, 이름, 스택용 페이지의 개수, 할당받은 메모리의 크기, 메모리의 최대 제한을 출력합니다. Kill은 해당 pid를 가진 프로세스를 kill 시스템콜을 통해 죽이면 됩니다. execute명령어는 path와 stacksize를 인자로 받기 때문에 새로 만든 exec2 시스템콜을 실행시키면 됩니다. 이때 exec2를 실행시켜도 pmanager가 종료되지 않고 살아 있어야 됩니다. Memlim명령어도 마찬가지로 새로 만든 setmemorylimit을 통해 실행시킵니다. exit명령어를 통해 pmanager를 종료시킵니다.

2. 코드 구현:

먼저 exec2 시스템 콜의 구현입니다. Exec2는 본래 정의 되어있던 시스템콜인 exec와 거의 동일합니다. 다른 점이라면 스택용 페이지의 개수를 설정하는 부분입니다. 각각의 process의 스택용 페이지의 개수를 나타내는 stack이라는 변수를 proc.h에 선언했습니다.

```
// 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)
  int memlimit;                // Memory limit
  int stack;                  //The number of the stack page
  struct proc *main;           //Main thread
  int num_thread;             //The number of child thread.
};
```

그리고 pagetable의 사이즈를 결정하는 함수인 allocuvm을 다음과 같이 변경하였습니다.

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

1+stacksize만큼의 사이즈를 pgdir에 할당하였습니다. 1+stacksize에서 1은 가드용 페이지의 개수(사이즈)이며 나머지 stacksize는 스택용 페이지의 개수(사이즈)입니다. 일반적인 exec함수는 스택용 페이지가 1개이므로

```
  oldpgdir = curproc->pgdir;
  curproc->pgdir = pgdir;
  curproc->stack = 1;                    //stacksize = 1
  curproc->sz = sz;
  curproc->tf->eip = elf.entry;  // main
  curproc->tf->esp = sp;
```

다음과 같이 stack=1로 지정해줍니다. Exec2함수는 밑의 사진과 같이 stacksize로 저장해줍니다.

```
curproc->pgdir = pgdir;
curproc->sz = sz;
curproc->stack = stacksize;      //set stacksize
curproc->tf->eip = elf.entry;  // main
curproc->tf->esp = sp;
```

setmemorylimit함수는 메모리의 최대치를 설정하는 시스템콜입니다. 따라서 proc.h에 memlimit이라는 변수를 다음과 같이 만들었습니다,.

```
// 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)
  int memlimit;                // Memory limit
  int stack;                  //The number of the stack page
  struct proc *main;           //Main thread
  int num_thread;            //The number of child thread.
};
```

다음은 함수내용입니다.

```
int
setmemorylimit(int pid, int limit){
  struct proc *p;
  int check = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      check = 1;
      break;
    }
  }
  if(check == 0) return -1;            //cant find process
  if(p->sz >limit || limit <0 ) return -1;  //must limit >=0 and p->sz <= limit
  acquire(&ptable.lock);
  p->memlimit = limit;                 //set memory limit.
  release(&ptable.lock);
  return 0;
}
```

Pid와 limit을 인수로 받습니다. 인수로 받은 pid를 찾는 for문을 만들었고 pid를 찾았다면 check =1로 설정했습니다. 따라서 check==0 이라면 pid에 맞는 process를 찾지 못한 것이므로 return -1을 합니다. 만약 원래 할당되어 있는 메모리인 p->sz보다 limit이 작거나 limit이 음수라면 이 또한 오류이므로 return -1을 합니다. 그리고 오류에 해당하지 않고 정상진행 된다면p->memlimit을 limit으로 설정합니다. 이 memlimit은 growproc에서 다음과 같이 사용됩니다.

```
160  int
161  growproc(int n)
162  {
163    uint sz;
164    struct proc *curproc = myproc();
165
166    sz = curproc->sz;
167    if(curproc->memlimit > 0){
168      if(sz+n > curproc->memlimit)
169        return -1;
170    }
```

우선 curproc의 memlimit이 0보다 큰지 확인합니다. 0보다 크다는 뜻은 제한이 있다는 것이며 memlimit이 0이라면 제한이 없기 때문에 정상 진행합니다. 다시 memlimit이 0보다 큰 겨우 sz+n즉 원래 할당된 메모리인 sz에 새로운 메모리 n만큼을 더했을 때 이 값이 memlimit보다 큰지 확인합니다. 크다면 이는 memlimit을 넘은 것이기 때문에 return -1로 오류가 발생했음을 알립니다.

pmanager에서 명령어 list를 위한 시스템콜인 list도 작성하였습니다. 다음과 같습니다.

```
562  void
563  list(void){
564    struct proc *p;
565    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
566      if(p->state != RUNNABLE && p->state != RUNNING && p->state !=SLEEPING)      //check RUNNABLE or RUNNIG or SLEEPING then print process informa
567        continue;
568      cprintf("name : %s, pid: %d, stackpagesize: %d, allocated memory: %d, memory limit: %d \n", p->name, p->pid, p->stack, p->sz, p->memlimit);
569    }
570  }
```

for문을 통해 state가 RUNNABLE, RUNNING, SLEEPING인 process를 찾고 이 process의 정보들을 출력합니다.

다음은 pmanager.c 입니다.

```
1   #include "types.h"
2   #include "stat.h"
3   #include "user.h"
4   int
5   main(int argc , char *argv[]){
6     int i = 0;
7     char input[70];
8     char num[20];
9     char lim[20];
10    char id[20];
11    char *arv[50];
12    uint n = 0;
13    int stack;
14    char path[50];
```

함수의 선언 부분입니다. i는 for문에서 사용될 인덱스이며 input은 전체 커맨드를 받기 위한 배열, num은 stacksize를 받기 위한 배열, lim은 memlimit을 받기 위한 배열, id는 pid를 받기 위한 배열, arv는 path를 복사해서 exec2에서 사용될 배열, n은 배열의 자릿수를 확인하기 위한 변수, stack은 stacksize에 인자로 전달될 변수, path는 exec2에서 path로 전달될 배열입니다.

```
while(1){
  n = 0;
  gets(input,70);
  if(input[0]=='l' && input[1] == 'i' && input[2] == 's'&& input[3] =='t'){
      list();             //syscall that list up RUNNING, RUNNALBE, SLEEPING process and it definded in proc.c
  }

  else if(input[0]=='k' && input[1] == 'i' && input[2] == 'l'&& input[3] =='l'){
      strcpy(num,input+5);        //input[4] is spacebar(blank) so copy from input + 5(input[5])
      if(kill(atoi(num))==0){
          printf(1,"success kill\n");
      }
      else printf(1,"failed kill\n");
  }

```

전체적인 뼈대는 while문으로 덮었습니다. 우선 gets을 통해 input에 명령어를 입력받습니다. 만약 명령어가 list라면 proc.c에서 정의된 list 시스템콜을 호출합니다. 명령어가 kill이라면 공백(input[4]) 후에 pid가 들어오므로 strcpy(num, input+5)를 통해 input[5]부터 num으로 복사합니다. 그리고 atoi(num)을 통해 int로 바꾸고 kill을 호출합니다. Kill에 성공하면 성공메시지를, 실패하면 실패메시지를 띄웁니다.

다음은 execute명령어입니다.

```
  else if(input[0]=='e' && input[1] == 'x' && input[2] == 'e'&& input[3] =='c' && input[4] =='u' && input[5]== 't'&& input[6]=='e'){
      for(i = 0; input[8+i] != 32 ;i++){
          path[i] = input[8+i];             //copy from input[8].
          n++;                               // n is length of the path.
      }
      path[i] = '\0';
      for(i = 0; input[9+n+i]; i++){
          num[i] = input[9+n+i];             //copy form input[9+n]. atoi(num) is stack size
      }
      stack = atoi(num);
      strcpy(*arv, path);
      if(fork()==0){
          if(fork()==0){                          //like 'back' in sh.c. call fork() and implement exec2
              if(exec2(path, arv, stack)== -1){   //exec2 is syscall in exec.c
                  printf(1,"failed execute!\n");
              }
          }
          exit();
      }                           //child is exited by parent do wait()
  }
```

명령어가 execute인 것이 확인되면 for문을 통해 path내용을 path배열에 받습니다. 마찬가지로 공백이 주어지고 path가 주어지기 때문에 input[8]부터 값이 존재합니다. 따라서 path[i] = input[8+i]를 통해 path배열에 path값을 옮겨담습니다. 그리고 n에 path배열의 길이를 담습니다. 이 n은 다음 stacksize를 받을 때 input의 몇번째 부터 복사를 해야될지를 정해줍니다. 즉 공백두개와 n만큼의 index후 즉 input[9+n]부터 stacksize가 존재하게 됩니다. for문을 통해 stacksize를 num배열에 옮기고 atoi(num)을 통해 int를 stack으로 옮깁니다. *arv에 path도 복사한다음 exec2를 실행하려고 하였지만 바로 exec2를 실행한다면 pmangaer가 종료됩니다. 따라서 sh.c에서 runcmd

함수가 BACK명령어를 처리하는 것처럼 fork()를 통해 자식프로세스를 만들고 이 자식프로세스가 또 fork()를 실행에 자식프로세스를 만듭니다. 이 자식의 자식프로세스가 exec2를 진행하고 그냥 자식프로세스는 exit하게됩니다. 이렇게 되면 원본pmanager는 백그라운드에서 계속실행이 가능한 상태가 됩니다.

```
51      else if(input[0]=='m' && input[1] == 'e' && input[2] == 'm'&& input[3] =='l' && input[4] =='i' && input[5]== 'm'){
52      for(i = 0; input[7+i] != 32; i++){
53          id[i] = input[7+i];                    //copy from input[7]. id is pid
54          n++;                                   //n is length id.
55      }
56      for(i = 0; input[8+n+i]; i++){
57        lim[i] = input[8+n+i];                   //copy from input[8+n]. lim is memory limit.
58      }
59      if(setmemorylimit(atoi(id), atoi(lim))==0 ){    //setmemorylimit is syscall in proc.c.
60        printf(1,"success memlim!\n");
61      }
62      else printf(1,"failed memlim!\n");
63      }
64
65      else if(input[0]=='e' && input[1] == 'x' && input[2] == 'i'&& input[3] =='t'){
66          printf(1, "pmanger exit\n");
67          break;                                         //break while. so meet exit() line 74.
68      }
69      else{                                              //if input is wrong command.
70          printf(1,"wrong command\n");
71      }
72    }
73    exit();
74  }
```

그 다음은 memlim명령어입니다. for문을 통해 pid를 id배열에 옮겨 담습니다. 마찬가지로 lim배열에 limit을 옮겨 담습니다. 그리고 setmemortlimit을 실행하고 성공하면 성공메시지를 실패하면 실패 메시지를 출력합니다. 마지막으로 exit명령어입니다. While 문을 break하게됩니다. 그러면 exit()을 만나 pmanager가 종료됩니다. 만약 제시된 명령어가 아닌 다른 명령어가 입력된다면 wrong command라는 에러메시지를 띄우고 다시 입력 받도록 합니다.

3. 실행 결과

실행하고 list를 입력한 결과입니다.

```
$ pmanager
pmanager start!
list
name : init, pid: 1, stackpagesize: 1, allocated memory: 12288, memory limit: 0
name : sh, pid: 2, stackpagesize: 1, allocated memory: 16384, memory limit: 0
name : pmanager, pid: 3, stackpagesize: 1, allocated memory: 12288, memory limit: 0
```

기본적으로 모든 프로그램은 stack용page가 1개이기 때문에 stackpagesize를 1(개)로 표시하였습니다. 이를 allocproc함수에서 다음과 같이 설정해주었습니다.

```
88    found:
89      p->state = EMBRYO;
90      p->pid = nextpid++;
91      p->num_thread = 0;
92      p->stack = 1;
93      p->memlimit = 0;
94      release(&ptable.lock);
```

Execute forktest 100을 입력한 결과입니다.

```
execute forktest 100
fork test
fork test OK
zombie!
list
name : init, pid: 1, stackpagesize: 1, allocated memory: 12288, memory limit: 0
name : sh, pid: 2, stackpagesize: 1, allocated memory: 16384, memory limit: 0
name : pmanager, pid: 3, stackpagesize: 1, allocated memory: 12288, memory limit: 15000
```

Pmanager가 종료되지 않고 계속 실행됨을 알 수 있습니다.

```
memlim 3 20000
success memlim!
list
name : init, pid: 1, stackpagesize: 1, allocated memory: 12288, memory limit: 0
name : sh, pid: 2, stackpagesize: 1, allocated memory: 16384, memory limit: 0
name : pmanager, pid: 3, stackpagesize: 1, allocated memory: 12288, memory limit: 20000
```

Memlim 3 20000을 입력한 결과입니다. 본래 할당된 memory가 12288이었기 때문에 limit을 20000으로 설정해주면 성공하는 것을 볼 수 있습니다.

```
list
name : init, pid: 1, stackpagesize: 1, allocated memory: 12288, memory limit: 0
name : sh, pid: 2, stackpagesize: 1, allocated memory: 16384, memory limit: 0
name : pmanager, pid: 3, stackpagesize: 1, allocated memory: 12288, memory limit: 0
memlim 3 10000
failed memlim!
```

12288보다 낮게 설정하면 실패하는 것을 볼 수 있습니다.

```
execute pmanager 100
pmanager start!
list
name : init, pid: 1, stackpagesize: 1, allocated memory: 12288, memory limit: 0
name : sh, pid: 2, stackpagesize: 1, allocated memory: 16384, memory limit: 0
name : pmanager, pid: 3, stackpagesize: 1, allocated memory: 12288, memory limit: 0
name : pmanager, pid: 5, stackpagesize: 100, allocated memory: 417792, memory limit: 0
```

다음은 execute pmanager 100을 한 결과입니다. Stackpagesize가 정상적으로 100(개)이 되었음을 알 수 있습니다.

```
pmanager start!
list
name : init, pid: 1, stackpagesize: 1, allocated memory: 12288, memory limit: 0
name : sh, pid: 2, stackpagesize: 1, allocated memory: 16384, memory limit: 0
name : pmanager, pid: 3, stackpagesize: 1, allocated memory: 12288, memory limit: 0
exit
pmanger exit
$
```

Exit을 입력하면 바로 종료됩니다.

4. 구현에 어려웠던 점

가장 어려웠던 점은 pmanager에서 execute를 실행할 때 였습니다. Exec2를 실행하게 되면 pmanager가 바로 종료되었습니다. 다행스럽게도 과제 명세에 sh.c에 runcmd함수를 참고라하는 힌트가 있어서 fork로 해결할 방법을 찾았던 것 같습니다. 감사합니다.

LWP

1.상황이해 및 디자인

Xv6에서는 기본적으로 thread라는 개념이 없기 때문에 우선 process로 만들고 pagetable을 공유하는 식으로 설계를 하였습니다. 메인이 되는 process가 있고 이 밑에 메인 프로세스의 pid와 pagetable을 공유하는 프로세스를 thread라고 설정하였습니다. 결국 xv6에서는 스레드도 프로세스의 일종이기 때문에 process를 생성하는 과정과 비슷하게 진행하였습니다. thread_create에는 fork와 exec의 로직을 사용하였고, thread_exit에는 exit의 로직, thread_join은 wait의 로직을 사용하였습니다.

2. 코드구현

우선 proc.h struct proc안에 thread와 관련된 여러 변수를 선언하였습니다. 다음과 같습니다.

```
52    struct proc *main_thread;   //Main thread
53    int memlimit;          // Memory limit
54    int stack;           //The number of the stack page
55    int is_thread;       //check_thread
56    int thread_id;       //Thread_id
57    void *ret;            //Using for exit and join
58  };
```

Main_thread는 이 process의 메인스레드를 가리키게 됩니다. Is_thread는 이 프로세스가 thread인지를 확인하는 것이며 thread이면 1 아니면 0이 됩니다. Thread_id는 pid처럼 thread_id를 설정한 것입니다. 마지막으로 ret은 exit과 join함수에서 쓰이게 됩니다.

Thread_create입니다.

```
620  int thread_create(thread_t *thread, void *(*start_routine)(void*), void *arg){
621    struct proc *curproc = myproc();
622    struct proc *np;
623    pde_t* pgdir;
624    int i;
625    uint sz,sp, ustack[2];
626    if((np = allocproc()) == 0){
627      return -1;
628    }
629    pgdir = curproc->pgdir;
630    if((sz = allocuvm(pgdir, curproc->sz, curproc->sz + 2*PGSIZE)) == 0)
631      goto bad;
632    clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
633
634    sp = sz;
635
636    ustack[0] = 0xffffffff;  // fake return PC
637    ustack[1] = (uint)arg;
638
639    sp -= 8;
640    if(copyout(pgdir, sp, ustack, 8) < 0)
641      goto bad;
642    np->pgdir = pgdir;
643    curproc->sz = sz;
644    np->sz = sz;
645    np->stack = 1;                    //thread stacksize always 1
646    np->memlimit = curproc->memlimit;   //copy mainthread memlimit
647    np->parent = curproc;
648    np->main_thread = curproc;          //np->main_thread is curproc;
649    *np->tf = *curproc->tf;
650    //np->tf->eax = 0;
651    np->tf->eip = (uint)start_routine;               //start function
652    np->tf->esp = sp;
653    np->pid = curproc->pid;
654    np->is_thread = 1;
655    np->thread_id = nexttid;
656    nexttid++;                                       //increase nexttid for next thread
657    for(i = 0; i < NOFILE; i++)
658      if(curproc->ofile[i])
659        np->ofile[i] = filedup(curproc->ofile[i]);
660    np->cwd = idup(curproc->cwd);
```

```
660     np->cwd = idup(curproc->cwd);
661     safestrcpy(np->name, curproc->name, sizeof(curproc->name));
662
663     *thread = (thread_t)np->thread_id;
664     acquire(&ptable.lock);
665
666     np->state = RUNNABLE;
667
668     release(&ptable.lock);
669
670     return 0;
671
672     bad:
673     if(pgdir)
674       freevm(pgdir);
675     return -1;
676   }
```

Fork와 비슷하게 시작합니다. 우선 thread도 프로세스의 일종이므로 allocproc을 통해 process배정을 받게 됩니다. 그 다음이 fork랑 다른데, fork는 새로운 pagetable을 배정받지만 thread는 새로운 pagetable이 아니라 메인스레드의 pagetable을 받기 때문에 새로 할당받지 않고 기존의 메인스레드의 pagetable사이즈를 늘리고 이 부분을 사용하게 하였습니다. 이 부분의 아이디어는 exec에서 얻은 것으로 사이즈가 늘어난 pagetable을 자식thread들도 사용할 수 있도록 ustack을 통해 fakepc와 thread_create함수에서 받은 인자를 넣었고 이를 새로할당받은 pagetable부분에 넣었습니다. 648에서 메인스레드를 설정해주었습니다. 기본적으로 allocproc에서 main_thread는 자신의 process를 가리키게 했습니다. 이 함수를 호출한 프로세스가 메인 스레드이므로 현제 프로세스를 새로생성된 스레드의 메인스레드로 생성하였습니다. 쓰레드가 쓰레드를 호출하는 경우는 배제하였고 이 경우 fork를 사용한다고 가정하였습니다. 651줄에서 시작함수를 start_routine으로 설정하였고, 663줄에서 함수 인자로 받았던 thread에 thread_id를 입력하였습니다. Thread_t는 types.h에 int로 설정하였습니다. 이를 제외한 나머지 부분들은 fork함수랑 비슷합니다.

```
722 //its logic is same as exit but save retval in thread->ret
723 void
724 thread_exit(void *retval)
725 {
726   struct proc *p;
727   struct proc *curproc = myproc();
728   int fd;
729
730
731   // Close all open files.
732   for(fd = 0; fd < NOFILE; fd++){
733     if(curproc->ofile[fd]){
734       fileclose(curproc->ofile[fd]);
735       curproc->ofile[fd] = 0;
736     }
737   }
738
739   begin_op();
740   iput(curproc->cwd);
741   end_op();
742   curproc->cwd = 0;
743
744   acquire(&ptable.lock);
745
746   curproc->ret = retval;
747   // mainth might be sleeping in wait().
748   wakeup1(curproc->main_thread);
749
750   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
751     if(p->parent == curproc){
752       p->parent = initproc;
753       if(p->state == ZOMBIE)
754         wakeup1(initproc);
755     }
756   }
757
758   // Jump into the scheduler, never to return.
759   curproc->state = ZOMBIE;
760   sched();
761   panic("zombie exit");
762 }
```

다음은 thread_exit입니다. 746줄에 함수인자를 받아주었고, 748줄의 wakeup1에 curproc->parent대신 curproc->main_thread를 넣었습니다. 나머지는 원래의 exit과 동일합니다.

```
806  int
807  thread_join(thread_t thread, void **retval){
808    struct proc *p;
809    struct proc *curproc = myproc();
810    int have_thread;
811    acquire(&ptable.lock);
812    for(;;){
813      have_thread = 0;
814      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
815        if(p->thread_id != thread) continue;
816        have_thread = 1;
817        if(p->state == ZOMBIE){
818          *retval = p->ret;
819          kfree(p->kstack);
820          p->kstack = 0;
821          p->pid = 0;
822          p->memlimit = 0;
823          p->stack = 0;
824          p->is_thread = 0;
825          p->thread_id = 0;
826          p->parent = 0;
827          p->name[0] = 0;
828          p->killed = 0;
829          p->state = UNUSED;
830          nexttid--;
831          p->main_thread->sz = deallocuvm(p->main_thread->pgdir, p->main_thread->sz, p->main_thread->sz - 2*PGSIZE );
832          p->sz = 0;
833          release(&ptable.lock);
834          return 0;
835
836        }
837
838      }
839      if(have_thread==0){
840        release(&ptable.lock);
841        return -1;
842      }
843
844      // Wait for child thread to exit.  (See wakeup1 call in thread_exit.)
845      sleep(curproc, &ptable.lock);  //DOC: wait-sleep
846
847    }
848  }
849
```

다음은 thread_join입니다. wait함수과 거의 비슷합니다. 인자로 받은thread와 p->thread_id를 비교하여 같으면 ZOMBIE인지를 확인하여 할당된 자원들을 모두 회수하고 종료합니다. 만약 찾았는데 ZOMBIE가 아니라면 sleep 함수를 설정하여 자식thread가 끝날 때까지 기다립니다. 자식thread가 thread_exit의 wakeup1을 만나면 sleep이 해제되어 다시 for문을 실행하고 zombie스레드를 종료시킵니다. 하지만 문제가 있었습니다. 실제로 testcode를 돌려본결과 sleep부분에서 알 수 없는 오류가 발생했습니다. 꽤 오랜시간 왜 오류가 발생하였는지를 살펴보았지만 찾을 수 없었습니다.

3. 실행결과 및 trouble shooting

주어진 thread_test.c로 test를 해보았습니다.

```
Test 1: Basic test
Thread 0 start
Thread 0 end
Thread 1 start
Parent waiting for children...
pid 3 thread_test: trap 14 err 4 on cpu 0 eip 0x7b9 addr 0x7fd8--kill proc
Thread 1 returned 0, but expected 1
Test failed!
$
```

Thread 0은 잘 실행되고 종료까지 됩니다만, thread 1이 start가 되었지만 end되지않고 오류가 발생합니다.

```
void *thread_basic(void *arg)
{
  int val = (int)arg;
  printf(1, "Thread %d start\n", val);
  if (val == 1) {
    sleep(200);
    status = 1;
  }
  printf(1, "Thread %d end\n", val);
  thread_exit(arg);
  return 0;
}
```

Thread1 start가 된 후

If (val==1){

Sleep(200);

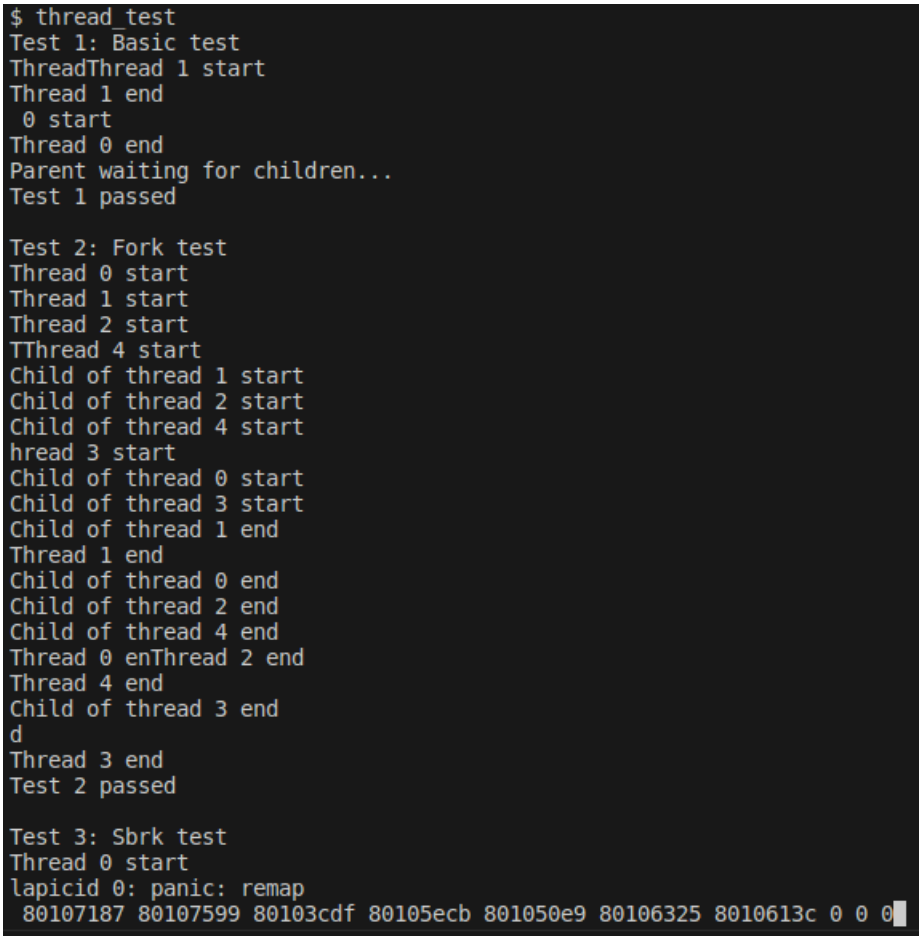
Status =1;

}

이 부분에서 Sleep(200)이 진행이 되는데 Sleep(200)을 할 동안 main thread는 join_all을 실행시킵니다. 이 때 thread_exit보다 thread_join이 먼저 시작됩니다. 따라서 메인theread(thread_test)는 thread1이 thread_exit을 실행해 wakeup1을 실행할 때 까지 Sleep을 하고 있어야 되지만, 어째서인지 이 sleep이 작동하지 않았습니다. sleep은 따로 건든 것이 전혀 없었기 때문에 제가 설계한 부분에서 문제가 발생한 것 같았지만 여전히 찾을 수가 없었습니다. 만약 thread_basic에서 sleep(200)이 없었을 때 즉 다음과 같았을 때

```
17  void *thread_basic(void *arg)
18  {
19    int val = (int)arg;
20    printf(1, "Thread %d start\n", val);
21    if (val == 1) {
22      //sleep(200);
23      status = 1;
24    }
25    printf(1, "Thread %d end\n", val);
26    thread_exit(arg);
27    return 0;
28  }
```

의 결과입니다.



Test2까지 통과는 하였지만 test3는 구현을 하지 못하였기 때문에. remap 오류가 발생하는 모습입니다. 사실 test2도 통과하지 못하는 케이스가 존재합니다. 그만큼 sleep의 구현이 너무 아쉽습니다. 해결방법을 모르겠습니다.

지금까지 긴 글 읽어 주셔서 감사합니다.