OS_project03_13065_주원웅_2019060546

Design

1. Thread_create 함수 구현

Thread_create 함수를 구현하기 위해서 xv6의 기본 system call인 fork() 함수와 exec() 함수에서 필요한 부분들을 사용할 것 입니다. 우선 fork를 통하여 기존 process를 생성하는 것 과 같이 부모 프로세스 대신 main thread에서 필요한 부분들을 카피해올 것입니다. 이후 exec를 활용하여 생성한 thread의 stack영역을 만들어 주고 thread의 시작 지점을 매개변수로 받아와 할당할 것 입니다. 또한 main thread와 페이지 영역을 공유해야 하기 때문에 이 부분을 처리해줄 것 입니다.

2. Thread_exit 함수 구현

기존 xv6의 syscall인 exit() 함수를 참고하여 thread를 종료하는 함수를 만들 것 입니다. 다른 점은 thread는 자식 프로세스를 가질 수 없으므로 자식 프로세스를 관리하는 영역이 없다는 것 만 다릅니다.

3. Thread_join 함수 구현

기존의 wait 함수를 변형하여 만들 것 입니다. 매개변수 thread에 해당하는 thread가 종료되기를 기다리며 해당 thread가 종료될 시 자원회수 역할을 담당합니다. Wait함수와 다른 점은 페이지 영역을 다루는 부분이 다릅니다. Thread에서는 다른 thread와 페이지 영역을 공유하기 때문에 join에서는 페이지영역을 회수하지 않습니다.

4. clreaThreads 함수 구현

한 thread가 종료될 시 해당 thread가 실행중인 process영역의 모든 thread를 종료해야 하기 때문에 해당 작업을 구현할 함수입니다. Exit함수(본인 종료)와 wait함수(다른 thread 종료)를 참고하여 디자인 할 것입니다. 마찬가지로 페이지 영역에 대해서는 건들지 않습니다.

5. Lock unlock 함수 구현

수업시간에 배운 피터슨 알고리즘을 사용하여 Lock Unlock 을 구현하려 했지만, 피터슨은 두개의 프로세스만 존재하는 상황에서 유효한 것이기에 한계가 있었습니다. 이후 찾아보던 중 이미 멀티 스레드 환경에서 락 언락을 구현하는 알고리즘인 램포트의 베이커리 알고리즘이 존재하는 것을 알게 되었고, 해당 알고리즘을 통해 Lock Unlock 을 구현하려고 합니다.

Implement

1. LWP 구현

```
549
550 int
551 thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg)
552 {
553   int i;
554   struct proc *np;
555   struct proc *curr = myproc();
556   struct proc *mthread;
557   uint sz, sp, ustack[2];
558   pde_t *pgdir;
559
560   // modify fork
561   if ((np = allocproc()) == 0) {  // allocate space for new thread
562     return -1;                    // no allocate, exit
563   }
564
565   --nextpid;      // increasing in alloc function but thread not increasing
566
567   if (curr->mthread) mthread = curr->mthread;
568   else mthread = curr;
569   pgdir = curr->pgdir;
570   np->parent = mthread->parent; // parent is main thread's parent(process)
571   *np->tf = *mthread->tf;
572
573   // clear %eax so that fork returns 0 in the child
574   np->tf->eax = 0;
575
576   for (i = 0; i < NOFILE; i++) {
577     if (mthread->ofile[i])
578       np->ofile[i] = filedup(mthread->ofile[i]);
579   } np->cwd = idup(mthread->cwd);
580
581   safestrcpy(np->name, mthread->name, sizeof(mthread->name));
582
583   np->tid = nexttid++;    // set tid
584   *thread = np->tid;      // send tid to arg thread
585   np->mthread = mthread;     // set main thread
586   np->pid = mthread->pid;
```

```
74 static struct proc*
75 allocproc(void)
76 {
77   struct proc *p;
78   char *sp;
79
80   acquire(&ptable.lock);
81
82   for(p = ptable.proc; p < &ptabl
83     if(p->state == UNUSED)
84       goto found;
85
86   release(&ptable.lock);
87   return 0;
88
89 found:
90   p->state = EMBRYO;
91   p->pid = nextpid++;
92   p->tid = 0;
93   p->mthread = 0;
94   p->retval = 0;
95   release(&ptable.lock);
```

```
35 enum procstate { UNUSED, EMBRYO
36
37 // Per-process state
38 struct proc {
39   uint sz;
40   pde_t* pgdir;
41   char *kstack;
42   enum procstate state;
43   int pid;
44   struct proc *parent;
45   struct trapframe *tf;
46   struct context *context;
47   void *chan;
48   int killed;
49   struct file *ofile[NOFILE];
50   struct inode *cwd;
51   char name[16];
52   int tid;
53   struct proc *mthread;
54   void *retval;
55 };
```

우선 thread 에 필요한 변수들을 proc 구조체에 선언해주었고 처음 프로세스 할당 할 때 모두 0 으로 초기화 하였습니다.

thread_create 함수부터 살펴보겠습니다. 위의 스크린샷은 fork() 함수를 참고하여 작성한 초반부입니다. Thread의 경우 allocproc() 에 의한 pid 증가가 필요 없기에 증가한 pid를 감소시켰고 main thread를 지정하여 create된(호출되어 생성된) thread가 어떤 thread로 부터 생성되었는 지 명시하였습니다. Thread가 생성된 직후는 main thread와 상태가 똑같해야 하므로 모든 상태를 복사하여 할당하였습니다. 부모 프로세스는 main thread의 부모 프로세스와 같아야 하므로 동일한 값을 할당했으며 생성된 thread는 main thread와 페이지 영역을 공유하므로 main thread의 페이지 영역을 수정하여 할당하기 위해 pgdir 변수에 할당해 두었습니다. Thread 에 Tid를 할당한 후 매개변수에 해당 tid를 전달하였고 파일 영역과 이름을 복사하여 thread에 할당하고 main thread와 상태가 동일하게 설정하였습니다.

```
587
588   // modify exec
589   sz = mthread->sz;
590
591   // one for user stack, one for guard page
592   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
593     goto bad;
594   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
595   sp = sz;
596
597   ustack[0] = 0xFFFFFFFF;
598   ustack[1] = (uint)arg;  // store arg reiceived
599
600   sp -= 8;
601   if (copyout(pgdir, sp, ustack, 8) < 0)
602   {
603     deallocuvm(pgdir,sz,sz + 2*PGSIZE);
604     goto bad;
605   }
606
607   np->sz = sz;       // store sz to new thread
608   mthread->sz = sz;    // store modified sz to curr process(main thread)
609   np->pgdir = pgdir;        // share pgdir
610   np->tf->eip = (uint)start_routine; // store start_routine to instruction potine
611   np->tf->esp = sp;                        // store sp to stack pointer
612
613   acquire(&ptable.lock);
614
615   np->state = RUNNABLE;
616
617   struct proc* p;
618   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
619     if (np->pid == p->pid) { // same pid = same main thread
620       p->sz = sz;             // update sz
621     }
622   }
623
624   release(&ptable.lock);
625
626   return 0;
```

그 다음 exec() 함수에서 필요한 부분들을 가져왔습니다. 스택사이즈를 main thread에서 가져온 후 생성된 Thread 의 stack영역을 한 페이지 사이즈 만큼 추가하기 위해 스택사이즈(sz)를 페이지 사이즈 2만큼 추가 할당하였고 한 페이지 사이즈는 clearpteu 함수로 가드용 페이지로 사용하였습니다. 이후 스택 포인터를 확장하여 fake return pc와 받아온 인자를 페이지 영역에 넣어주었습니다. 생성된 thread에 변경된 스택사이즈 페이지 영역을 할당하였고 trap frame의 eip, esp 에 해당 쓰레드가 작업을 시작할 위치와 스택 포인터를 할당하여 잘 작동하도록 하였습니다. 마지막으로 변경된 스택 사이즈를 동일한 프로세스를 실행중인 thread 모두 update 하여 create 작업을 마무리하였습니다.

```
646 void
647 thread_exit(void *retval)
648 {
649   struct proc* curr = myproc();
650   int fd;
651
652   if (curr == initproc)       // if curr process is initproc
653     panic("init exiting");
654
655   curr->retval = retval;      //  store retval
656
657   // close all open files
658   for (fd = 0; fd < NOFILE; fd++) {
659     if (curr->ofile[fd]) {
660       fileclose(curr->ofile[fd]);
661       curr->ofile[fd] = 0;
662     }
663   }
664
665   begin_op();
666   iput(curr->cwd);
667   end_op();
668   curr->cwd = 0;
669
670   acquire(&ptable.lock);
671
672   // main thread be sleeping in wait()
673   wakeup1(curr->mthread);
674
675   // jump into the scheduler, never to return.
676   curr->state = ZOMBIE;
677   sched();
678   panic("zombie exit");
679 }
```

다음은 thread_exit() 함수의 구현입니다. 원래 xv6의 exit() 함수를 참고하여 구현하였으며 return value를 할당하는 작업을 추가적으로 수행합니다. Thread의 열려 있는 파일들을 모두 닫은 이후 해당 thread가 종료되었으므로 대기중인 main thread를 깨워준 후 zombie 상태로 진입한 후 작업을 마칩니다. Exit() 함수와 다른 점은 thread는 자식 thread가 존재하지 않기 때문에 자식 프로세스 관리하는 부분이 없는 게 다른 점 입니다.

```
694 int
695 thread_join(thread_t thread, void **retval)
696 {
697   struct proc *p;
698   struct proc *curr = myproc();
699   int haveTh;
700
701   acquire(&ptable.lock);
702   for(;;) {
703     // Scan through table looking for exiting thread.
704     haveTh = 0;
705     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
706       if (p->tid != thread || p->mthread != curr)
707         continue;
708       haveTh = 1;
709       if (p->state == ZOMBIE) {
710         // Found Thread tid is arg Thread
711         *retval = p->retval;
712         kfree(p->kstack);
713         p->kstack = 0;
714         p->pid = 0;
715         p->tid = 0;
716         p->parent = 0;
717         p->mthread = 0;
718         p->name[0] = 0;
719         p->killed = 0;
720         p->state = UNUSED;
721         p->retval = 0;
722         release(&ptable.lock);
723         return 0;
724       }
725     }
726     if (!haveTh || curr->killed){
727       release(&ptable.lock);
728       return -1;
729     }
730     sleep(curr, &ptable.lock);
731   }
732 }
```

다음은 thread_join() 함수입니다. Xv6의 wait함수를 참고하여 작성하였습니다. 차이점은 wait 함수의 경우 현재 기다리는 자식 프로세스를 종료시키는 작업을 수행하는데 페이지 영역을 초기화합니다. 하지만 thread_join의 경우 페이지 영역을 해당 프로세스 작업을 수행하는 모든 쓰레드들이 공유하기에 초기화하면 안됩니다. 그렇기에 freevm(p->pgdir)을 삭제하여 작성했습니다 추가적으로 thread 에 추가한 변수들을 초기화하고 매개변수로 받아온 tid에 해당하는 thread의 상태를 종료합니다.

```
746 void
747 clearThreads(int pid, int tid)
748 {
749   struct proc *p;
750   int fd;
751
752   acquire(&ptable.lock);          // exit other thread (same pid d
753
754   for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
755     if (pid != p->pid || tid == p->tid)
756       continue;
757     release(&ptable.lock);
758
759     for (fd = 0; fd < NOFILE; fd++) {
760       if (p->ofile[fd]) {
761         fileclose(p->ofile[fd]);
762         p->ofile[fd] = 0;
763       }
764     }
765     begin_op();
766     iput(p->cwd);
767     end_op();
768     p->cwd = 0;
769
770     acquire(&ptable.lock);
771
772     kfree(p->kstack);
773     p->kstack = 0;
774     p->pid = 0;
775     p->parent = 0;
776     p->name[0] = 0;
777     p->killed = 0;
778     p->state = UNUSED;
779     p->tid = 0;
780     p->mthread = 0;
781   }
782   release(&ptable.lock);
783 }
```

다음은 clearThreads 함수인데 이 함수는 한 thread가 종료되거나 kill 될 시 해당 thread가 작업중인 프로세스를 작업중인 다른 thread 들을 모두 종료시키기 위함입니다. 해당 함수는 wait() 과 exit() 두 함수를 참고하여 작성하였으며 같은 pid를 가지는 모든 thread 상태를 초기화합니다.

이 함수는 원래 xv6의 system call 인 exec() 함수, exit() 함수에서 사용합니다. 그 이유는 system call exit() 호출 시 원래 작업이 잘 수행되도록 하기 위함 입니다. 위와 동일하게 페이지 영역은 초기화하지 않았습니다.

```
240 void
241 exit(void)
242 {
243   struct proc *curproc = myproc();
244   struct proc *p;
245   int fd;
246
247   if(curproc == initproc)
248     panic("init exiting");
249
250   // Close all open files.
251   for(fd = 0; fd < NOFILE; fd++){
252     if(curproc->ofile[fd]){
253       fileclose(curproc->ofile[fd]);
254       curproc->ofile[fd] = 0;
255     }
256   }
257
258   begin_op();
259   iput(curproc->cwd);
260   end_op();
261   curproc->cwd = 0;
262
263   clearThreads(curproc->pid, curproc->tid);
264
265   acquire(&ptable.lock);
266
```

위는 exit() 함수입니다. 중간에 clearThreads 함수를 호출하여 한 thread가 종료될 시 프로세스를 공유하는 thread들의 자원을 회수하는 작업을 수행합니다.

비슷하게 exec() 호출 시 프로세스를 종료하고 다른 프로세스를 실행해야 하기 때문에 모든 thread들을 종료하고 그 thread들 중 하나를 선택하여 새로운 프로세스를 실행합니다. 아래가 exec() 속 clearThreads() 함수호출 입니다. 추가적으로 exec에서 thread에 필요한 변수들을 초기화하는 작업도 수행합니다. 아래는 exec 함수 내부의 clearThreads 함수 호출입니다.

```
 94   safestrcpy(curproc->name, last, sizeof(curproc->name));
 95
 96   // Commit to the user image.
 97   clearThreads(curproc->pid, curproc->tid);
 98   oldpgdir = curproc->pgdir;
 99   curproc->tid = 0;           // add additon variable
100   curproc->mthread = 0;
101   curproc->retval = 0;
102   curproc->pgdir = pgdir;
103   curproc->sz = sz;
104   curproc->tf->eip = elf.entry;  // main
105   curproc->tf->esp = sp;
106   switchuvm(curproc);
107   freevm(oldpgdir);
108   return 0;
109
```

```
168   acquire(&ptable.lock);
169
170   sz = curproc->sz;
171   if(n > 0){
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
179   for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
180     if (p->pid != curproc->pid)   // update stacksize threads have same pid
181       continue;
182     p->sz = sz;
183   }
184
185   release(&ptable.lock);
186   switchuvm(curproc);
187   return 0;
```

추가적으로 sbrk입니다. 쓰레드들은 모두 스택사이즈를 공유하기에 lock을 통해 동시에 접근하여 스택사이즈를 수정하는 것을 막아주었고 이후 모든 proc를 돌면서 같은 pid를 가진 thread들의 스택사이즈 크기를 update 하여 동일하게 합니다.

```
734 int
735 sys_thread_join(void)
736 {
737   int thread, retval;
738
739   if (argint(0, &thread) < 0)
740     return -1;
741   if (argint(1, &retval) < 0)
742     return -1;
743   return thread_join(thread, (void**)retval);
744 }
```

```
681 int
682 sys_thread_exit(void)
683 {
684   int retval;
685
686   if (argint(0, &retval) < 0)
687     return -1;
688
689   thread_exit((void*) retval);
690
691   return 0;
692 }
```

```
632 int
633 sys_thread_create(void)
634 {
635   int thread, start_routine, arg;
636
637   if (argint(0, &thread) < 0)
638     return -1;
639   if (argint(1, &start_routine) < 0)
640     return -1;
641   if (argint(2, &arg) < 0)
642     return -1;
643   return thread_create((thread_t *)thread, (void*)start_routine, (void*)arg);
644 }
```

다음은 각 thread_ 함수들의 system call 구현입니다.

2. Lock UnLock 구현

```
1 #include <stdio.h>
2 #include <pthread.h>
3
4 #define n 5
5
6 int shared_resource = 0;
7 int number[n];
8 int flag[n];
9
10 #define NUM_ITERS 20
11 #define NUM_THREADS 5
```

우선 멀티 스레드 환경에서 스레드들의 우선순위를 표현하는 number 배열을 만들고, 우선순위 할당 여부를 표현하기 위한 배열 flag를 만듭니다.

```
16 void lock(int tid)
17 {
18     flag[tid] = 1;
19     int max_number = 0;
20     for (int i = 0; i < n; i++) {
21         if (number[i] > max_number) {
22             max_number = number[i];
23         }
24     }
25     number[tid] = max_number + 1;
26     flag[tid] = 0;
27
28     for (int i = 0; i < n; i++) {
29         while (flag[i]) { /* busy-wait */ }
30         while (number[i] != 0 &&
31             (number[i] < number[tid] ||
32             (number[i] == number[tid] && i < tid))) { /* busy-wait */ }
33     }
34 }
35
36 void unlock(int tid)
37 {
38     number[tid] = 0;
39 }
```

다음은 락 함수 구현입니다. Flag 배열의 현재 스레드의 번호 인덱스를 1로 설정합니다. 이는 아직 우선순위가 부여되지 않음을 나타냅니다. 이후 모든 스레드의 우선순위를 돌면서 확인하고 현재 스레드를 가장 낮은 우선순위로 설정하고 우선순위가 할당되었으므로 이를 표시하기 위해 flag 인덱스를 0으로 표현합니다.

그 후 모든 스레드를 확인하며 while(flag[i])를 통해 어떤 스레드가 우선순위가 할당중인 과정에 있다면 대기하고 우선순위 부여 후 우선순위를 비교합니다.

우선순위 비교는 다른 스레드의 number 배열 속 요소가 0이라면 아직 우선순위가 부여되지 않았거나 이미 critical section을 진행이 다 되었음을 나타내므로 현재 쓰레드가 실행되어야 합니다. 그렇기에 number[i] == 0 이라면 while문을 탈출하고 critical sectiond에 진입하도록 number[i] != 0 조건을 사용하고, 다른 스레드가 가진 우선순위가 존재하면 값을 비교하여 낮은 숫자의 스레드가 우선순위가 높기에 대기하도록 number[i] < number[tid] 조건문을 사용하였습니다. 만약 동일한 우선순위를 가졌다면 (멀티 스레드의 우선순위 부여 과정 중 발생할 수 있음) 더 낮은 스레드 번호를 가진 스레드가 먼저 실행되도록 (number[i] < number[tid] || (number[i] == number[tid] && i < tid)) 조건문을 사용하였습니다. (현재 우선순위가 낮거나 우선순위가 동일한데 tid가 더 크다 라면 참이므로 while문 루프)

이를 통해 쓰레드들을 critical section에 진입 의지를 가진 것 들만 비교할 수 있고 진입 의지를 가진 것들이 순서대로 우선순위가 부여되며 critical section 에 진입할 수 있게 되고 우선순위가 부여되지 않은 스레드(진입 의지 x)는 고려되지 않고, 동일한 우선순위를 가졌다면 tid를 통해 실행 순서를 부여하여 deadlock이 발생하지 않게 됩니다.

```
58 int main() {
59     pthread_t threads[n];
60     int tids[n];
61 
62     // Initialize flag and number arrays
63     for (int i = 0; i < n; i++) {
64         flag[i] = 0;
65         number[i] = 0;
66     }
67 
68     for (int i = 0; i < n; i++) {
69         tids[i] = i;
70         pthread_create(&threads[i], NULL, thread_func, &tids[i]);
71     }
72 
73     for (int i = 0; i < n; i++) {
74         pthread_join(threads[i], NULL);
75     }
76 
77     printf("shared: %d\n", shared_resource);
```

처음 number와 flag는 모두 0으로 초기화(critical section 진입 의지 x, 우선순위 부여 중 상태 x) 하였습니다.

```
35 
36 void unlock(int tid)
37 {
38     number[tid] = 0;
39 }
```

UnLock 과정은 Number 배열의 쓰레드 번호의 인덱스 요소가 0이라 설정하여 구현하였습니다. 인덱스 요소가 0 이라는 것은 critical section 에 진입할 의지가 없는 것이므로 number를 해제하여 다른 쓰레드들이 critical section에 진입할 수 있도록 해주었습니다.

Result

Thread_test.c

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ 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 staThread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Child of thread Child of thread 3 rt
1 start
Child of thread 2 startstart
Child of thread 4 starChild of thread 0 sta
t
rt
Child of thread 1 eChild of thread 3 end
nd
ThreadChild of thread 2 endChild of thread 4 end
Child of thread Thread 1 end
 3 end

0 end
ThreThread 4 end
ad 2 end
Thread 0 end
Test 2 passed

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

All tests passed!
$ thread_exit
```

Thread_test 입니다. 모두 정상적으로 작업하는 것을 확인했습니다.

Thread_exit.c

```
t
Thread 4 start
Executing...
Hello, thread!
$ thread_exit
Thread exit test start
Thread 0 staThread 1 start
Thread 2 start
rt
Thread Thread 4 start
3 start
Exiting...
$
```

Thread_Exit 입니다. Exiting 이후 바로 shell프로그램이 실행됨을 확인했습니다.

Thread_exec.c

```
Test 3 passed

All tests passed!
$ thread_exec
Thread exec test start
Thread 0 starThread 1 start
Thread 2 start
Thread 3 start
t
Thread 4 start
Executing...
Hello, thread!
$
```

Thread_exec 입니다. 정상적으로 hello_thread가 실행되는 것을 확인했습니다.

Thread_kill.c

```
Thread Thread 4 start
3 start
Exiting...
$ thread_kill
Thread kill test start
Killing process This code should be exeThis code should be executhis code shouldThis.
ted 5 times.
 be executed 5 times.
12
cuted 5 times.
This code should be executed 5 times.
Kill test finished
$
```

Process가 Kill 되었을 때 모든 thread가 정상적으로 종료되는 것을 확인했습니다.

Lock UnLock 적용 전 (race condition 발생)

```
➜  os ./pthread_lock_linux
tid : 2 , shared_resource : 95
tid : 1 , shared_resource : 96
tid : 0 , shared_resource : 97
tid : 3 , shared_resource : 98
tid : 4 , shared_resource : 99
shared: 99
➜  os
```

```
root@259998a39222:/OS/xv6-public# ./pthread_lock_linux
tid : 3 , shared_resource : 98
tid : 0 , shared_resource : 98
tid : 1 , shared_resource : 98
tid : 2 , shared_resource : 98
tid : 4 , shared_resource : 98
shared: 98
root@259998a39222:/OS/xv6-public# ./pthread_lock_linux
tid : 0 , shared_resource : 99
tid : 2 , shared_resource : 99
tid : 3 , shared_resource : 98
tid : 1 , shared_resource : 99
tid : 4 , shared_resource : 99
shared: 99
root@259998a39222:/OS/xv6-public# ./pthread_lock_linux
tid : 4 , shared_resource : 99
tid : 1 , shared_resource : 99
tid : 0 , shared_resource : 99
tid : 3 , shared_resource : 99
tid : 2 , shared_resource : 99
shared: 99
```

Lock UnLock 적용 후

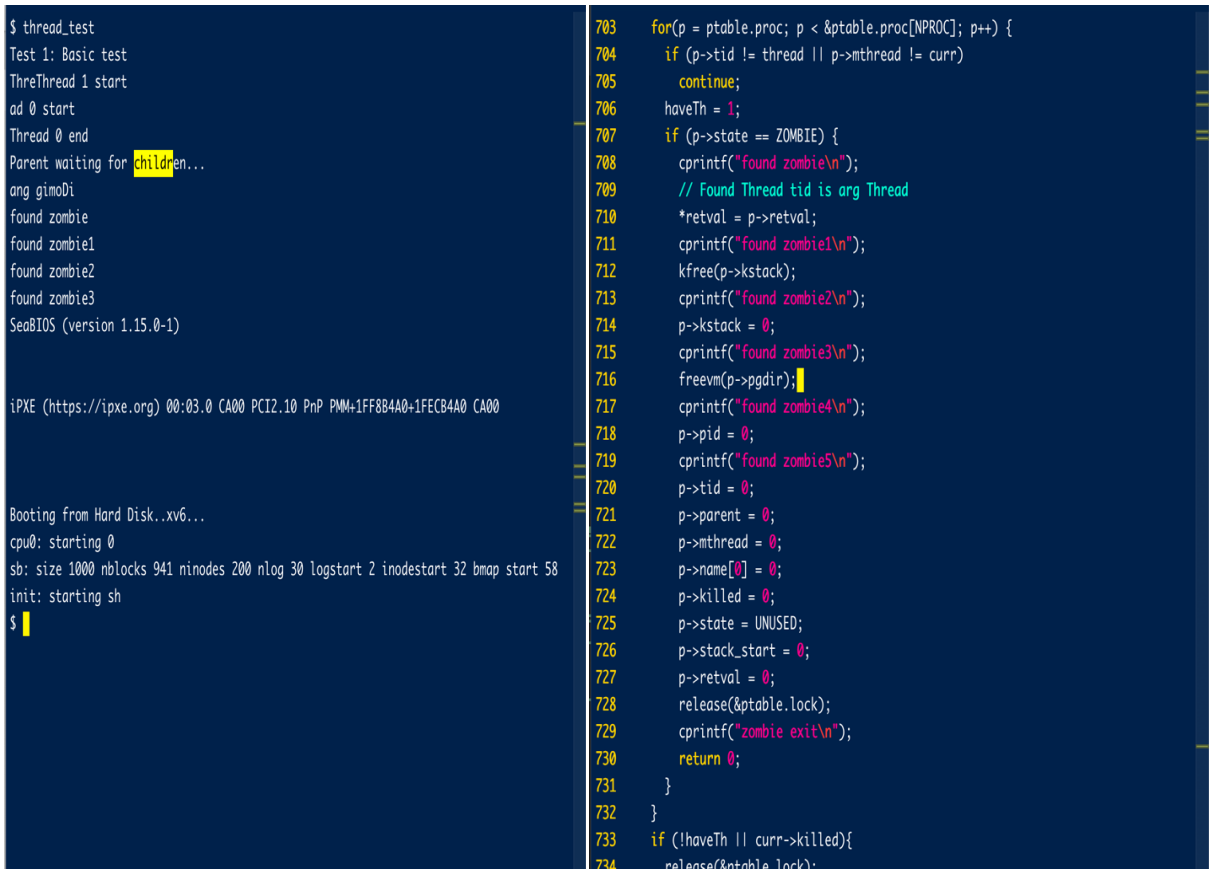
```
root@259998a39222:/OS/xv6-public# ./pthrec
tid : 0 , shared_resource : 20
tid : 1 , shared_resource : 40
tid : 2 , shared_resource : 60
tid : 3 , shared_resource : 80
tid : 4 , shared_resource : 100
shared: 100
```

```
➜  os ./pthread_lock_linux
tid : 0 , shared_resource : 20
tid : 1 , shared_resource : 40
tid : 2 , shared_resource : 60
tid : 3 , shared_resource : 80
tid : 4 , shared_resource : 100
shared: 100
```

Trouble shooting

1. thread_test, thread_exit 실행 시 xv6가 재부팅 되는 경우

   thread_test의 경우 thread_join 함수를 구현할 때 wait() 함수를 참고하여 작성하였기 때문에 wait() 함수의 자식 프로세스 자원 회수 중 freevm(p->pgdir) 함수를 가져다 사용하였습니다. 하지만 thread의 경우 페이지 영역을 공유하여 사용하기 때문에 다른 thread의 페이지를 회수하는 상황이 발생하여 재부팅 되는 것을 확인했습니다. Freevm() 함수를 삭제하여 실행하였더니 잘 실행되는 것을 확인했습니다. 아래는 thread_join 디버깅 스크린샷입니다.



```
$ thread_test
Test 1: Basic test
ThreThread 1 start
ad 0 start
Thread 0 end
Parent waiting for children...
ang gimoDi
found zombie
found zombie1
found zombie2
found zombie3
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00



Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
```

```
703   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
704     if (p->tid != thread || p->mthread != curr)
705       continue;
706     haveTh = 1;
707     if (p->state == ZOMBIE) {
708       cprintf("found zombie\n");
709       // Found Thread tid is arg Thread
710       *retval = p->retval;
711       cprintf("found zombie1\n");
712       kfree(p->kstack);
713       cprintf("found zombie2\n");
714       p->kstack = 0;
715       cprintf("found zombie3\n");
716       freevm(p->pgdir);
717       cprintf("found zombie4\n");
718       p->pid = 0;
719       cprintf("found zombie5\n");
720       p->tid = 0;
721       p->parent = 0;
722       p->mthread = 0;
723       p->name[0] = 0;
724       p->killed = 0;
725       p->state = UNUSED;
726       p->stack_start = 0;
727       p->retval = 0;
728       release(&ptable.lock);
729       cprintf("zombie exit\n");
730       return 0;
731     }
732   }
733   if (!haveTh || curr->killed){
```

   Thread_exit 의 경우는 exit() 함수 실행 시 clreadThreads 함수를 호출하였고 이 함수 역 wait() 를 참고하였기에 freevm(p->pgdir) 이 존재하였고 이부분을 제거하여 해결했습니다.

2. Exit() , clearThreads 함수에서 deallocuvm() 함수 실행 시 오류 발생

   본인이 생각 하기에 쓰레드를 종료 할 시 페이지 영역에 쓰레드를 위해 할당한 stacksize 2 만큼을 회수해야 한다고 생각하여 코드를 작성하였는데, 에러가 발생하였다. 어느 부분에서 에러가 발생한것인 지 모르겠어서 프린트로 디버깅 해보았는데 해당 부분이 에러를 발생시켰다. Dealloc 함수를 삭제하면서 이부분을 삭제하면 잘 돌아가지 않아야 하는거 아닌가? 라고 생각을 하였는데 실제로는 잘 돌아가는 것을 확인하였다.

3. Lock Unlock 구현 접근방식 잘못 & 아이디어 부재

처음에 xv6 환경에서 Lwp 구현상태에서 Lock Unlock을 구현하는 것으로 착각하여 한참 동안을 어떻게 구현해야 할까 고민했다. 하지만 컴파일 하는 과정에서 이 과제는 xv6상에서 구현하는 것이 아님을 깨달았고 헛수고 했음을 알았다. 물론 본인이 알아차리지 못한 것이 맞지만, 프로젝트 명세에 xv6상에서 구현하는 것이 아님을 강조해주었으면 좋았을 것 같다. 이후에 피터슨 알고리즘을 사용하여 lock unlock 을 구현하려 하였지만, 멀티 쓰레드 환경에서 힘들다는 것을 깨달았고, 어찌할 지 고민하던 중 멀티 쓰레드 환경에서 lock unlock을 구현하는 베이커리 알고리즘이 존재함을 알았고, 이를 통해 구현하였다.