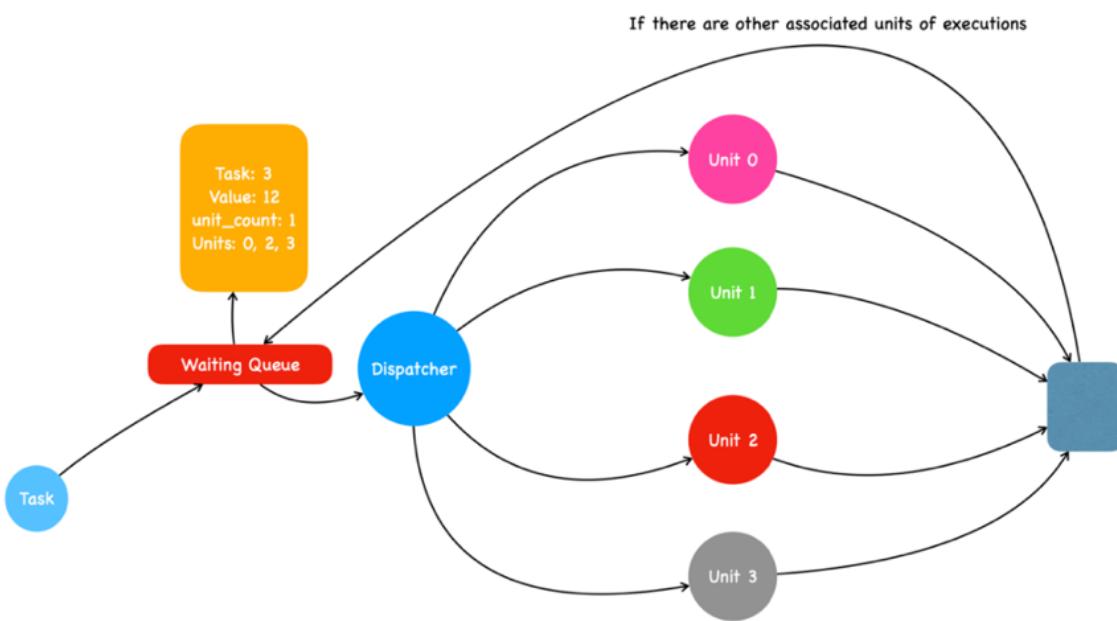


## ”به نام یزدان پاک“

### توصیف پروژه

یک سیستم پیچیده مانند یک سازمان بزرگ را درنظر بگیرید که واحدهای پردازشی جدایی برای هر کار دارد. در این سازمان برای آنکه هر کدام از کارها بطور کامل انجام شوند نیازمند مجموعه‌ای از پردازش‌ها در هر کدام از واحدهای پردازشی هستند. تمامی کارها در صف انتظار منتظر می‌مانند تا توزیع کننده، کار مورد نظر را انتخاب کند و به واحد پردازشی مربوطه تحويل دهد.

شکل زیر ساختار سیستم موردنظر را نشان می‌دهد:



شما باید بتوانید چنین مکانیزمی را در پروژه خود طراحی کنید.

برای پیاده سازی فاز دوم پروژه در ابتدا کلاسی تحت عنوان `thread_create` برای ساخت `thread` ها میسازیم.

در این کلاس با استفاده از تابع `clone()` که در ادامه در فایل `proc.c` پیاده سازی میکنیم و `#define` کردن `PAGESIZE` به عنوان `constant` میتوان به مقدار مورد نیاز `thread` با `id` های متفاوت تولید کرد.

توجه کنید که مقدار فضا سازی برای پوینتر `fptr` باید ۲ برابر `PAGESIZE` باشد تا در مشخص کردن `stack` به مشکل نخوریم.

```
C thread_create.c
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4
5 #define PAGESIZE 4096
6
7 int
8 thread_create(void (*fn) (void *), void *arg)
9 {
10 void *fptr = malloc(2 * (PAGESIZE));
11 void *stack;
12
13 if(fptr == 0)
14 return -1;
15
16 int mod = (uint)fptr % PAGESIZE;
17
18 if(mod == 0)
19 | stack = fptr;
20 else
21 | stack = fptr + (PAGESIZE - mod);
22
23 int thread_id = clone((void*)stack);
24
25 if(thread_id < 0)
26 | printf(1, "clone failed\n");
27 else if(thread_id == 0){
28 | | (fn)(arg);
29 | | free(stack);
30 | | exit();
31 }
32 |
33 | return thread_id;
34 }
```

حال باید کلاس threadsTest را پیاده سازی کرد این کلاس برای استفاده از thread\_create است.

```
C threadsTest.c
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  void childPrint(void* args){
6      printf(1, "hi, childs function executed properly with argument : %d\n", *(int*) args);
7  }
8
9  int main(void){
10     int argument = 0x0F01; // 3841 in decimal
11     int thread_id = thread_create(&childPrint, (void*)&argument);
12     if(thread_id < 0)
13         printf(1, "thread_create failed\n");
14     join();
15     printf(1, "thread_id is : %d\n", thread_id);
16     exit();
17 }
```

در ادامه کلاس threads.c را پیاده سازی میکنیم که برای پیاده سازی thread ها یا process هاست.

```
C threads.c
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  int stack[4096] __attribute__ ((aligned (4096)));
6  int x = 0;
7
8  int main(int argc, char *argv[]){
9      // int tid = fork();
10     int tid = clone(stack);
11     if (tid < 0)
12         printf(2, "error!\n");
13     else if (tid == 0){
14         for(;){
15             x++;
16             sleep(100);
17         }
18     }
19     else{
20         for(;){
21             printf(1, "x = %d\n", x);
22             sleep(100);
23         }
24     }
25     exit();
26 }
```

اگر بخواهیم به صورت process های مختلف به حل مسیله بپردازیم از (fork() با id های متفاوت استفاده میکنیم که در این صورت خروجی به شکل زیر میباشد:

(خط ۱۰ باید کامنت شود و به جای آن به خط ۹ بپردازیم)

اجرای این برنامه نسبت به threads و پیاده سازی با clone() کند تر است.

همچنین از آنجایی که process های مختلف shared memory ندارند پس زمانی که یکی از آنها  $x++$  میکند و  $x$  را از ۰ به ۱ تغییر میدهد دیگر process ها از این تغییر مطلع نمیشوند و همچنان

```
Machine View
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ threads
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
-
```

$x=0$  در نظر میگیرند.

اگر بخواهیم به صورت استفاده از چندین thread به حل مسیله بپردازیم از (clone) استفاده میکنیم که در این صورت خروجی به شکل زیر میباشد:

اجرای برنامه با threads به مراتب سریع تر است.

و از آنجایی که در thread ها shared memory داریم زمانی که یکی از آنها  $x++$  میکند دیگر

ها از آن باخبر اند و برنامه خود را با  $x$  آپدیت شده اجرا میکنند.

```
Machine View
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ threads
x = 0
x = 1
x = 2
x = 3
x = 4
x = 5
x = 6
x = 7
x = 8
x = 9
x = 10
x = 11
x = 12
x = 13
x = 14
```

حال باید تابع های () و clone() را در proc.c پیاده سازی کرد.

تابع clone() برای thread ها طوری پیاده سازی میشود که همان بدنه fork() است اما طوری که ماهیت thread پیدا کند اینگونه که stackTop را مشخص میکنیم تا به سمت پایین رشد کند.(متغیر stackTop را proc.h تعریف میکنیم) و در کل این تابع برای کپی کردن اطلاعات پدر برای فرزند است.

```
639 int
640 clone(void *stack)
641 {
642     int pid;
643     struct proc *curproc = myproc();
644     struct proc *np;
645     if((np = allocproc()) == 0)
646         return -1;
647
648     curproc->threads++;
649     np->stackTop = (int)((char*)stack + PGSIZE);
650     acquire(&phtable.lock);
651     np->pgdir = curproc->pgdir;
652     np->sz = curproc->sz;
653     release(&phtable.lock);
654
655     int bytesOnStack = curproc->stackTop - curproc->tf->esp;
656     np->tf->esp = np->stackTop - bytesOnStack;
657     memmove((void*)np->tf->esp, (void*)curproc->tf->esp, bytesOnStack);
658
659     np->parent = curproc;
660     *np->tf = *curproc->tf;
661     np->tf->eax = 0;
662     np->tf->esp = np->stackTop - bytesOnStack;
663     np->tf->ebp = np->stackTop - (curproc->stackTop - curproc->tf->ebp);
664
665     int i;
666     for(i = 0; i < NOFILE; i++)
667         if(curproc->ofile[i])
668             np->ofile[i] = filedup(curproc->ofile[i]);
669     np->cwd = idup(curproc->cwd);
670     safestrncpy(np->name, curproc->name, sizeof(curproc->name));
671     pid = np->pid;
672     acquire(&phtable.lock);
673     np->state = RUNNABLE;
674     release(&phtable.lock);
675     return pid;
676 }
```

حال تابع `wait` را با نام `join()` پیاده سازی میکنیم در بدنه این تابع همان کد `wait()` است با این تفاوت که زمانی که برای `thread` هاست نباید برای `process` ها `wait` کند و وقتی برای `process` هاست نباید برای `thread` ها `wait()` کند (که این چک کردن درون `if` ها انجام میشود)

```
C proc.c
678
679     int
680     join(void)
681     {
682         struct proc *p;
683         int havekids, pid;
684         struct proc *curproc = myproc();
685
686         acquire(&ptable.lock);
687         for(;;){
688             havekids = 0;
689             for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
690                 if(p->parent != curproc)
691                     continue;
692                 if(p->threads != -1)
693                     continue;
694                 havekids = 1;
695                 if(p->state == ZOMBIE){
696                     pid = p->pid;
697                     kfree(p->kstack);
698                     p->kstack = 0;
699
700                     if(check_pgdir_share(p))
701                         freevm(p->pgdir);
702
703                     p->pid = 0;
704                     p->parent = 0;
705                     p->name[0] = 0;
706                     p->killed = 0;
707                     p->state = UNUSED;
708                     p->stackTop = 0;
709                     p->pgdir = 0;
710                     p->threads = -1;
711                     release(&ptable.lock);
712                     return pid;
713                 }
714             }
715             if(!havekids || curproc->killed){
716                 release(&ptable.lock);
717                 return -1;
718             }
719             sleep(curproc, &ptable.lock);
720         }
721     }
```

پس از ساخت کلاس ها و توابع `join()` و `clone()` باید تغییرات لازم را در کلاس ها و توابع مختلف که بیشتر آنها مانند فاز اول انجام داد.

برای مثال در کلاس `proc.c` تغییرات زیر باید اعمال شوند.

ساخت `:thread` برای `stack`

```
15 static struct proc *initproc;
16 struct spinlock thread;
```

تغییر تابع `print()` برای نشان دادن `:thread`:

```
24 void
25 pinit(void)
26 {
27     initlock(&ptable.lock, "ptable");
28     initlock(&thread, "thread");
29 }
```

در تابع `allocproc()` هم باید مقدار های `stackTop` و `threads` را برای `p` تعیین کرد:

```
--> 90     found:
91     p->state = EMBRYO;
92     p->pid = nextpid++;
93     p->stackTop = -1;
94     p->threads = -1;
```

در تابع `userinit()` هم باید مقدار `threads` را برای `p` تعیین کرد:

```
--> 137     p->threads = 1;
```

در ادامه باید تابع growproc() را تغییر داد:

```
185     curproc->sz = sz;
186     acquire(&ptable.lock);
187     struct proc *p;
188     int numberOfChildren;
189
190     if(curproc->threads == -1) {
191         curproc->parent->sz = curproc->sz;
192         numberOfChildren = curproc->parent->threads - 2;
193         if(numberOfChildren <= 0){
194             release(&ptable.lock);
195             release(&thread);
196             switchuvm(curproc);
197             return 0;
198         }
199
200     else
201         for(p = ptable.proc; p < &ptable.proc[NPROC];p++){
202             if(p!=curproc && p->parent == curproc->parent && p->threads == -1){
203                 p->sz = curproc->sz;
204                 numberOfChildren--;
205             }
206         }
207     }
208     else{
209         numberOfChildren = curproc->threads - 1;
210
211         if(numberOfChildren <= 0){
212             release(&ptable.lock);
213             release(&thread);
214             switchuvm(curproc);
215             return 0;
216         }
217         else
218             for(p = ptable.proc; p < &ptable.proc[NPROC];p++){
219                 if(p->parent == curproc && p->threads == -1){
220                     p->sz = curproc->sz;
221                     numberOfChildren--;
222                 }
223             }
224         }
225     }
```

در تابع () هم باید مقدار های stackTop و threads را برای np تعیین کرد:

```
258     np->stackTop = curproc->stackTop;
259     np->threads = 1;
```

همچنین در تابع () باید if برای thread های که موفق آمیز نبوده اند پیاده سازی کرد:

```
311     if(curproc->threads == -1){
312         curproc->parent->threads--;
313     }
314 }
```

تابع check\_pgdir\_share برای چک کردن این است که process پدری که مورد نیاز است را از

```
333 int
334 check_pgdir_share(struct proc *process)
335 {
336     struct proc *p;
337     for(p = ptable.proc; p < &ptable.proc[NPROC];p++){
338         if(p != process && p->pgdir == process->pgdir)
339             return 0;
340     }
341     return 1;
342 }
```

بین نبرد:

این قسمت در تابع () هم باید چک کند زمانی که process داریم thread ها را از بین نبرد و

```
383     if(p->threads == 1){
384         freevm(p->pgdir);
385         p->pid = 0;
386         p->parent = 0;
387         p->name[0] = 0;
388         p->killed = 0;
389         p->state = UNUSED;
390         p->stackTop = -1;
391         p->pgdir = 0;
392         p->threads = -1;
393     }
394     else{
395         p->pid = 0;
396         p->parent = 0;
397         p->name[0] = 0;
398         p->killed = 0;
399         p->state = UNUSED;
400         p->stackTop = -1;
401         p->pgdir = 0;
402         p->threads = -1;
403     }
```

بر عکس:

در کلاس defs.h باید clone() و join() را فراخوانی کرد:

```
125 int clone(void*);  
126 int join(void);
```

در کلاس proc.h باید threads و stackTop را فراخوانی کرد:

```
52 int stackTop;  
53 int threads;
```

در کلاس sysproc.c باید clone() و join() را تعریف کرد:

```
105  
106 int sys_clone(void)  
107 {  
108     int stackptr = 0;  
109     if(argint(0, &stackptr) < 0)  
110     | return -1;  
111     | return clone((void*) stackptr);  
112 }  
113  
114  
115 int sys_join(void)  
116 {  
117     | return join();  
118 }  
119 }
```

در کلاس user.h باید clone() و join() را فراخوانی کرد:

```
28 int clone(void*);  
29 int join(void);
```

در usys.S سیستم کال ها را مشخص میکنیم:

```
34 SYSCALL(clone)  
35 SYSCALL(join)
```

در syscall.c هم از سیستم کال ها استفاده میکنیم:

```
137 [SYS_clone]    sys_clone,  
138 [SYS_join]    sys_join,
```

در define را SYS\_join و SYS\_clone مه syscall.h میکنیم:

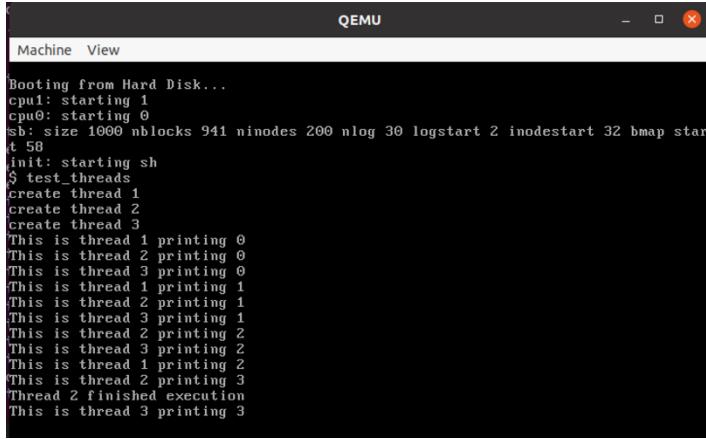
```
25 #define SYS_clone 24
26 #define SYS_join 25
```

در exec.c ب 4 threads و stackTop برای curproc مقدار میدهیم:

```
102 curproc->threads = 1;
103 curproc->stackTop = sp;
```

در ادامه فایل test\_threads را برای thread create کردن test کار میگیریم:

```
C test_threads.c
1
2
3
4
5     void
6 run(void* arg)
7 {
8     int id = *(int*) arg;
9     id++;
10    int i;
11    sleep(100);
12    for( i = 0 ; i < 4 ; i++){
13        printf(1, "This is thread %d printing %d\n", id, i);
14        if( id != 2 || i != 3)
15            sleep(100);
16        if (id == 1 && i == 1)
17            sleep(100);
18    }
19    printf(1, "Thread %d finished execution\n", id);
20    exit();
21 }
22
23 void
24 testThread(int n)
25 {
26     int tid, i;
27     for (i = 0; i < n; i++) {
28         tid = thread_create(run, &i);
29         sleep(10);
30         if (tid < 0) {
31             printf(1, "create thread failed\n");
32         } else {
33             printf(1, "create thread %d\n", i+1);
34             sleep(10);
35         }
36     }
37
38     for (i = 0; i < 5 ; i++) {
39         tid = join();
40     }
41     printf(1, "all threads joined\n");
42 }
43
44 int
45 main(int argc, char *argv[]){
46     testThread(3);
47     exit();
48 }
```



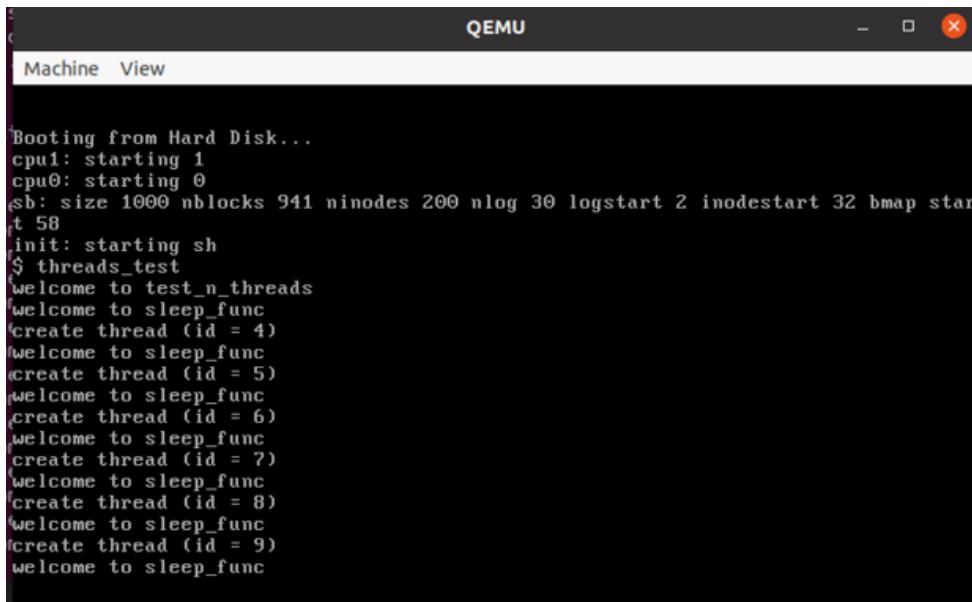
Boot from Hard Disk...  
cpu1: starting 1  
cpu0: starting 0  
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58  
init: starting sh  
\$ test\_threads  
create thread 1  
create thread 2  
create thread 3  
This is thread 1 printing 0  
This is thread 2 printing 0  
This is thread 3 printing 0  
This is thread 1 printing 1  
This is thread 2 printing 1  
This is thread 3 printing 1  
This is thread 2 printing 2  
This is thread 3 printing 2  
This is thread 1 printing 2  
This is thread 2 printing 3  
Thread 2 finished execution  
This is thread 3 printing 3

خروجی `test_threads` به شکل زیر است:

در آخر فایل `test_threads.c` را هم برای `thread create` کردن `test_threads` میگیریم:

```
5 void
6 sleep_func(void* arg)
7 {
8     printf(1, "welcome to sleep_func\n");
9     int time = *(int*)arg;
10    sleep(time);
11    exit();
12 }
13
14 void
15 create_n_threads(int n)
16 {
17     printf(1, "welcome to test_n_threads\n");
18     int tid, i, pid;
19     int threadCounter = 0;
20     int time = 100;
21     for (i = 0; i < n; i++) {
22         tid = thread_create(sleep_func, &time);
23         sleep(10);
24         if (tid < 0) {
25             printf(1, "create thread failed\n");
26         } else {
27             printf(1, "create thread (id = %d)\n", tid);
28             threadCounter++;
29         }
30     }
31
32     pid = fork();
33     if (pid < 0) {
34         printf(1, "create process failed\n");
35     } else if (pid == 0) {
36         printf(1, "fork: child\n");
37         exit();
38     } else {
39         printf(1, "create process (id = %d)\n", pid);
40         if (wait() != pid) {
41             printf(2, "create_n_threads: wait unexpected process (expected = %d)\n", pid);
42         }
43     }
44
45     for (i = 0; i < threadCounter; i++) {
46         tid = join();
47         printf(1, "tid = %d join \n", tid);
48     }
49 }
50
51 int
52 main(int argc, char *argv[])
53 {
54     create_n_threads(100);
55     exit();
56 }
```

خروجی threads\_test به شکل زیر است:



The screenshot shows a terminal window titled "QEMU" with the command "threads\_test" running. The output of the program is displayed, starting with boot messages and then the execution of the threads test. The terminal interface includes a menu bar with "Machine" and "View" options, and standard window controls (minimize, maximize, close) at the top.

```
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ threads_test
welcome to test_n_threads
welcome to sleep_func
create thread (id = 4)
welcome to sleep_func
create thread (id = 5)
welcome to sleep_func
create thread (id = 6)
welcome to sleep_func
create thread (id = 7)
welcome to sleep_func
create thread (id = 8)
welcome to sleep_func
create thread (id = 9)
welcome to sleep_func
```