

Thread Library Implementation in XV6

- Define System Call Clone. It calls the implementation of clone function defined in proc.c

```
93  int sys_clone(void) {
94      void *func;
95      void *stack;
96      int size;
97      void *arg;
98
99      if (argptr(0, (char**)&func, sizeof(void*)) < 0) return -1;
100     if (argptr(1, (char**)&stack, sizeof(void*)) < 0) return -1;
101     if (argint(2, &size) < 0) return -1;
102     if (argptr(3, (char**)&arg, sizeof(void*)) < 0) return -1;
103
104     return clone((void*)(func), (void*)stack, size, (void*)arg);
105 }
```

- A new data type to store the lock in types.h

C types.h > ...

```
1  #ifndef __TYPE_H__
2  #define __TYPE_H__
3
4  typedef unsigned int    uint;
5  typedef unsigned short ushort;
6  typedef unsigned char   uchar;
7  typedef uint    pde_t;
8
9  typedef struct {
10     |   uint locked;
11 }lock_t;
12
13 #endif
```

- A thread library is created in ulib.c. Two pages for stack are allocated at first. If the virtual address of the start of the stack is not page-aligned, it is rounded up. Then the clone() system call is made in which, only one argument of type "unsigned int" is passed to the function, so the size is assigned to be eight. The new thread starts executing at the address specified by start_routine().

Spinlock initialization, acquire and release are defined.

```

100 void*
101 memmove(void *vdst, const void *vsrc, int n)
102 {
103     char *dst;
104     const char *src;
105
106     dst = vdst;
107     src = vsrc;
108     while(n-- > 0)
109         *dst++ = *src++;
110     return vdst;
111 }
112
113 // thread library
114 void lock_init(lock_t *lock) {
115     lock->locked = 0;
116 }
117
118 void lock_acquire(lock_t *lock) {
119     while(xchg(&(lock->locked), 1) != 0);
120 }
121
122 void lock_release(lock_t *lock) {
123     xchg(&(lock->locked), 0);
124 }
125
126 void *thread_create(void*(start_routine)(void*), void *arg) {
127     void *stack = malloc(2 * PGSIZE);
128
129     if((uint)stack % PGSIZE)
130         stack = stack + (PGSIZE - (uint)stack % PGSIZE);
131
132     int size = 8;
133     int tid = clone(start_routine, stack, size, arg);
134
135     if (tid < 0) {
136         printf(1, "# Clone failed\n");
137         return 0;
138     }
139
140     return 0;
141 }

```

- Implementation of Clone.
 - Unlike fork(), it uses parent's address space.
 - Thread should have the same file descriptor as parent.
 - A user stack has been created to store the passed arguments and a kernel function copyout() to copy ustack to the virtual address in page table pgdir. Register \$esp stores the address of the top of the stack. After deciding the address of the top of the stack, register \$eip is set to point to the next instruction the system is about to execute, and the base pointer register \$ebp is set to be the same as \$esp.

```

C proc.c > clone(void *f(void *), void *, int, void *)
538 int clone(void *f(void *), void *stack, int size, void *arg) {
539     int i, pid;
540     struct proc *np;
541     struct proc *currproc = myproc();
542
543     // Allocate process.
544     if((np = allocproc()) == 0){
545         return -1;
546     }
547
548     if ((uint)stack % PGSIZE != 0 || stack == 0)
549         return -1;
550
551     // step 1: share the same address space with parent
552     np->state = UNUSED;
553     np->sz = currproc->sz;
554     np->parent = currproc;
555     *np->tf = *currproc->tf;
556     np->pgdir = currproc->pgdir;
557
558     np->tf->eip = (uint)f; // set instruction pointer
559     np->tf->eax = 0;        // clear %eax so that fork returns 0 in the child.
560
561     // step 2: use the same file descriptor
562     for (i = 0; i < NOFILE; i++)
563         if (currproc->ofile[i])
564             np->ofile[i] = filedup(currproc->ofile[i]);
565     np->cwd = idup(currproc->cwd);
566
567     safestrcpy(np->name, currproc->name, sizeof(currproc->name));
568
569     acquire(&ptable.lock);
570     uint ustack[2];
571     ustack[0] = 0xffffffff; // fake return PC
572     ustack[1] = (uint)arg;
573
574     // test
575     np->tf->esp = (uint)(stack+PGSIZE - 4); //put esp to right spot on stack
576     *((uint*)(np->tf->esp)) = (uint)arg; //arg to function
577     *((uint*)(np->tf->esp) - 4) = 0xFFFFFFFF; //return to nowhere
578     np->tf->esp = (np->tf->esp) - 4;
579     if (copyout(np->pgdir, np->tf->esp, ustack, size) < 0) {
580         cprintf("Stack copy failed.\n");
581         return -1;
582     }
583
584     np->state = RUNNABLE;
585
586     np->tf->ebp = currproc->tf->esp; // set base pointer
587
588     pid = np->pid;
589     release(&ptable.lock);
590
591     return pid;
592 }

```

- Test program frisbee

C frisbee.c > ...

```
6  #include "fcntl.h"
7  #include "syscall.h"
8  #include "traps.h"
9  #include "memlayout.h"
10 #include "semaphore.h"
11
12 #define NUMTHREADS 20
13
14 lock_t lock;
15 int thrower;
16
17 void player(void *arg_ptr);
18
19 int main(int argc, char *argv[]) {
20     int i;
21
22     lock_init(&lock);
23
24     for (i = 0; i < NUMTHREADS; i++) {
25         thread_create((void*)player, (void*)&i);
26         sleep(10);
27     }
28     while(wait() >= 0) ;
29     exit();
30 }
31
32 void player(void *arg_ptr) {
33     int i, self;
34     int *num = (int*) arg_ptr;
35     self = *num;
36
37     for (i = 0; i < 10; i++) {
38         if (thrower != self) {
39             lock_acquire(&lock);
40             printf(1, "%d caught frisbee from %d\n", self, thrower);
41             thrower = self;
42             printf(1, "  throwing frisbee\n", self);
43             sleep(20);
44             lock_release(&lock);
45         }
46         sleep(20);
47     }
48     exit();
49 }
```


- Output
 - With command frisbee 20 20

```
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ frisbee 20 20
]# Pass number no: 1 | Thread 0 is passing the token to thread 1
# Pass number no: 2 | Thread 1 is passing the token to thread 2
# Pass number no: 3 | Thread 2 is passing the token to thread 3
# Pass number no: 4 | Thread 3 is passing the token to thread 4
# Pass number no: 5 | Thread 4 is passing the token to thread 5
# Pass number no: 6 | Thread 5 is passing the token to thread 6
# Pass number no: 7 | Thread 6 is passing the token to thread 7
# Pass number no: 8 | Thread 7 is passing the token to thread 8
# Pass number no: 9 | Thread 8 is passing the token to thread 9
# Pass number no: 10 | Thread 9 is passing the token to thread 10
# Pass number no: 11 | Thread 10 is passing the token to thread 11
# Pass number no: 12 | Thread 11 is passing the token to thread 12
# Pass number no: 13 | Thread 12 is passing the token to thread 13
# Pass number no: 14 | Thread 13 is passing the token to thread 14
# Pass number no: 15 | Thread 14 is passing the token to thread 15
# Pass number no: 16 | Thread 15 is passing the token to thread 16
# Pass number no: 17 | Thread 16 is passing the token to thread 17
# Pass number no: 18 | Thread 17 is passing the token to thread 18
# Pass number no: 19 | Thread 18 is passing the token to thread 19
# Pass number no: 20 | Thread 19 is passing the token to thread 20
# Simulation of Frisbee game has finished, 20 rounds were played in total!
```

- Default (with command frisbee)

```
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ frisbee
# Default thread number: 4, pass round: 6
# Pass number no: 1 | Thread 0 is passing the token to thread 1
# Pass number no: 2 | Thread 1 is passing the token to thread 2
# Pass number no: 3 | Thread 2 is passing the token to thread 3
# Pass number no: 4 | Thread 3 is passing the token to thread 4
# Pass number no: 5 | Thread 4 is passing the token to thread 1
# Pass number no: 6 | Thread 1 is passing the token to thread 2
# Simulation of Frisbee game has finished, 6 rounds were played in total!
```