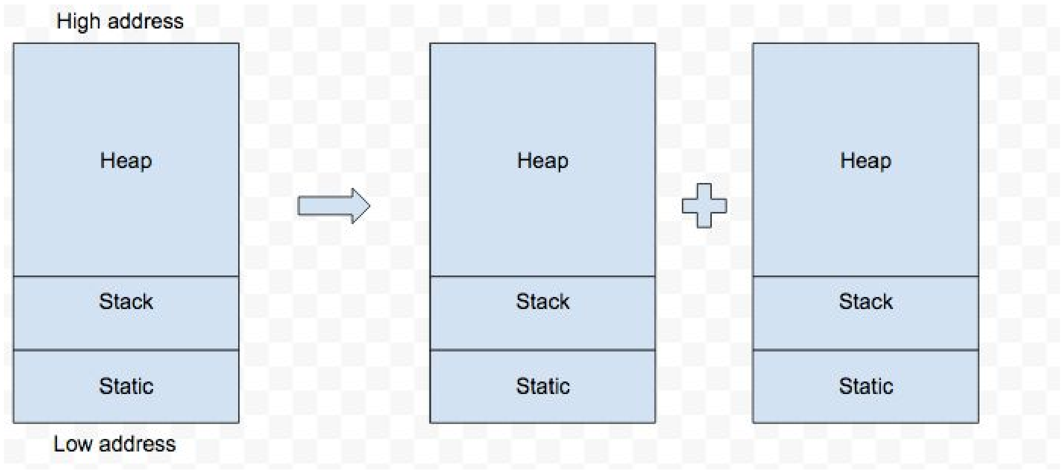
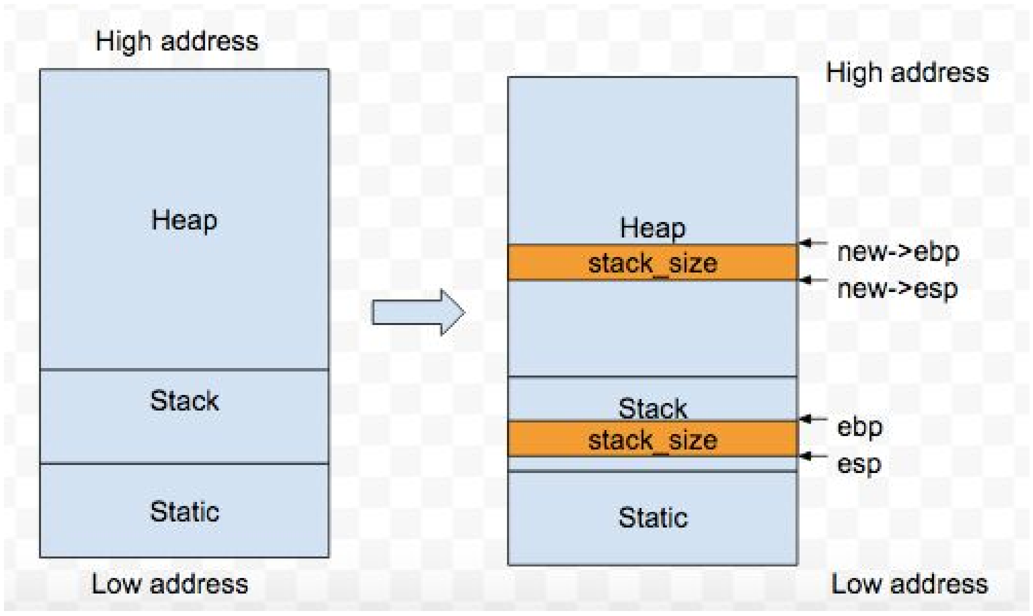
Name: Chenyang Yu **SUID:** 862052273
Source: https://github.com/mit-pdos/xv6-public.git
Environment: Ubuntu 16.04.3 LTS

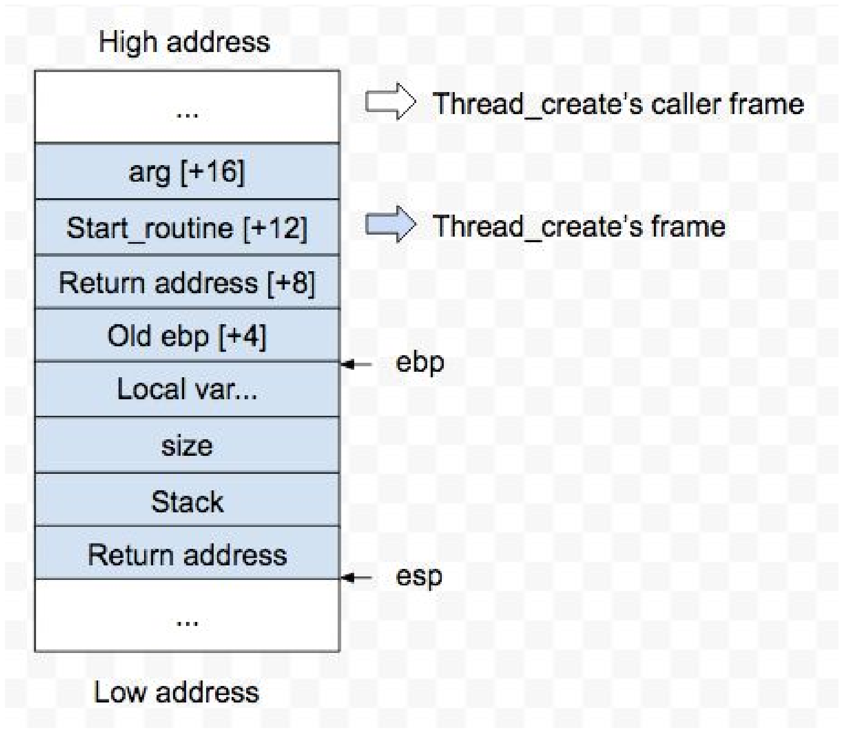
Part 1. Clone()
In this part, we implement a kernel thread in xv6. Compare with fork(), which creates new copy of process and owning memory space, clone() create a copy of process which share the static data. As the following Figure:
fork():



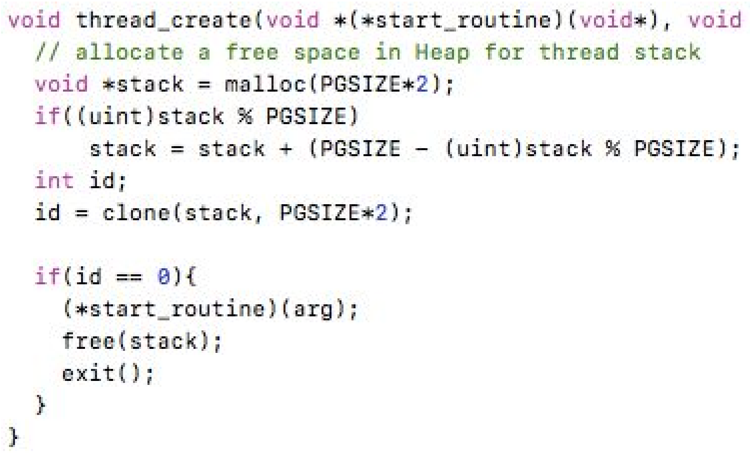
clone():



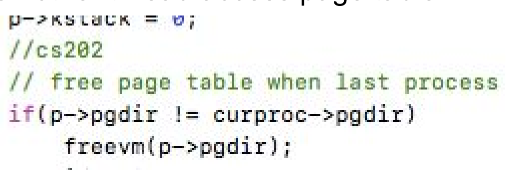
Since the stack in xv6 in grow from high address to low address (as we can see above, %ebp is in high address and %esp is in low address), when we cope the stack for each thread, besides the call stack (between %ebp and %esp), we have to also copy the previous %ebp and the return. These data locates at the bottom of stack as following figure(again, since the stack grows from high address to low address, these data are stored at higher address). Without these data, the thread will cause segment fault when finish function and return to the caller.



The reason why we allocate 2 page for each stack is for page alignment. In order to prevent the starting address locating at the middle of a page and causing fragment.

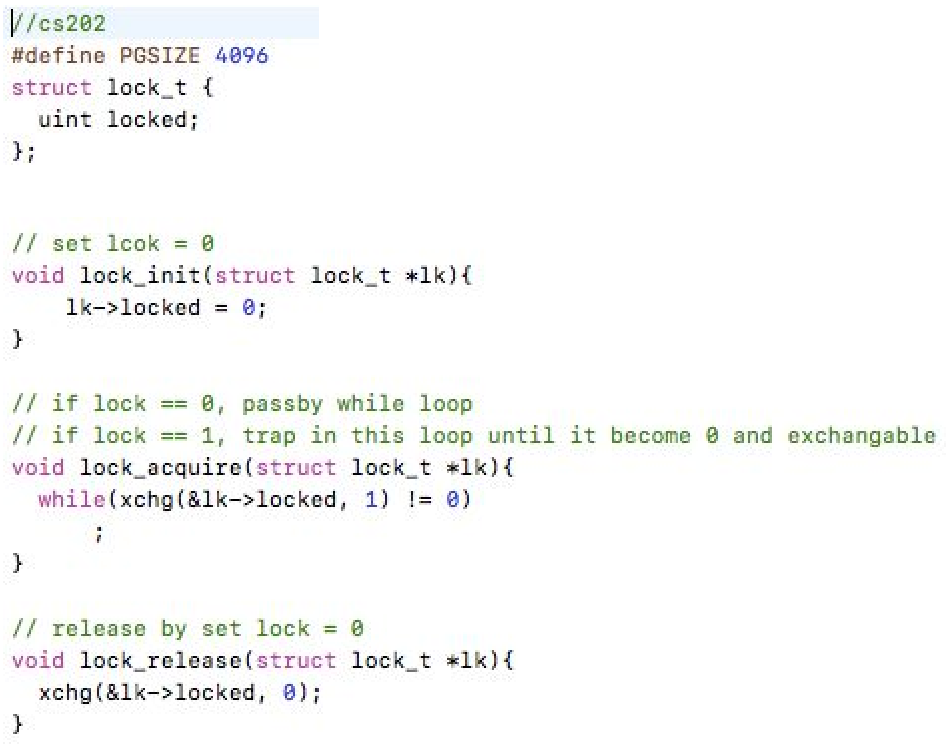

```
void thread_create(void *(*start_routine)(void*), void *arg){
  // allocate a free space in Heap for thread stack
  void *stack = malloc(PGSIZE*2);
  if((uint)stack % PGSIZE)
      stack = stack + (PGSIZE - (uint)stack % PGSIZE);
  int id;
  id = clone(stack, PGSIZE*2);

  if(id == 0){
    (*start_routine)(arg);
    free(stack);
    exit();
  }
}
```

We also modify the wait() to preventing some thread free the page table and cause segment fault when other thread access page table.


```
p->kstack = 0;
//cs202
// free page table when last process
if(p->pgdir != curproc->pgdir)
    freevm(p->pgdir);
```

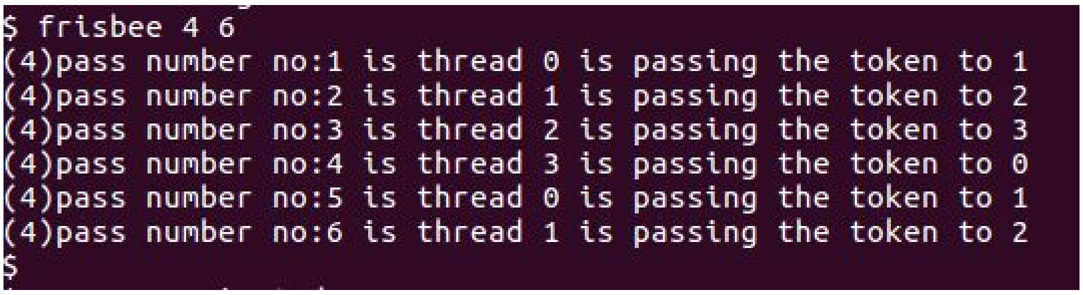
Part2. Spin lock

We use three function call to implement spinlock mechanism. The lock_init() set the flag into zero. The lock_acuire() check if the flag is zero and change the flag to one (this instruction is atomic). When flag is changed to one, the thread after ward will trap into the while loop and the critical section is protected. The lock_release() set the flag to zero and open the critical section for any thread who acquires the lock.

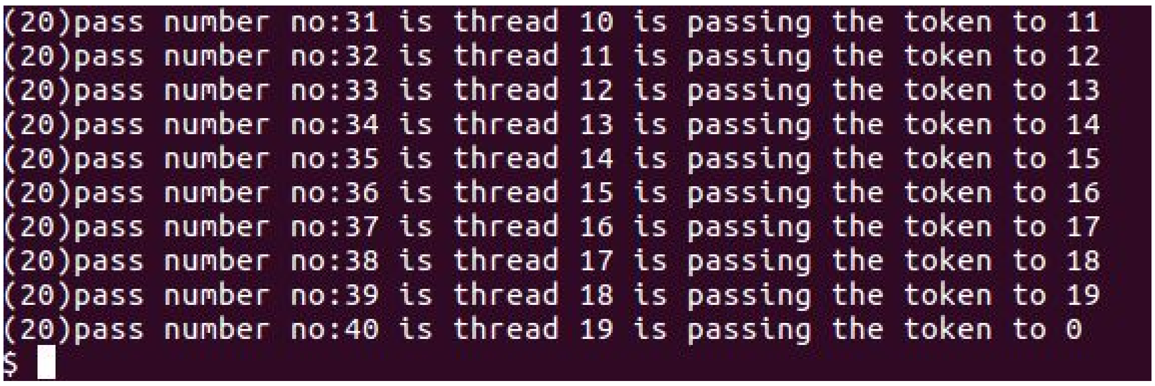


```
//cs202
#define PGSIZE 4096
struct lock_t {
  uint locked;
};


// set lcok = 0
void lock_init(struct lock_t *lk){
    lk->locked = 0;
}

// if lock == 0, passby while loop
// if lock == 1, trap in this loop until it become 0 and exchangable
void lock_acquire(struct lock_t *lk){
  while(xchg(&lk->locked, 1) != 0)
      ;
}

// release by set lock = 0
void lock_release(struct lock_t *lk){
  xchg(&lk->locked, 0);
}
```

Result:



```
$ frisbee 4 6
(4)pass number no:1 is thread 0 is passing the token to 1
(4)pass number no:2 is thread 1 is passing the token to 2
(4)pass number no:3 is thread 2 is passing the token to 3
(4)pass number no:4 is thread 3 is passing the token to 0
(4)pass number no:5 is thread 0 is passing the token to 1
(4)pass number no:6 is thread 1 is passing the token to 2
$
```



```
(20)pass number no:31 is thread 10 is passing the token to 11
(20)pass number no:32 is thread 11 is passing the token to 12
(20)pass number no:33 is thread 12 is passing the token to 13
(20)pass number no:34 is thread 13 is passing the token to 14
(20)pass number no:35 is thread 14 is passing the token to 15
(20)pass number no:36 is thread 15 is passing the token to 16
(20)pass number no:37 is thread 16 is passing the token to 17
(20)pass number no:38 is thread 17 is passing the token to 18
(20)pass number no:39 is thread 18 is passing the token to 19
(20)pass number no:40 is thread 19 is passing the token to 0
$
```

Each thread with different pid keep waiting until the turn becoming the same with itself (get the frisbee) and plus the turn. Making the next thread get frisbee until the pass reach the MAXpass we setup.