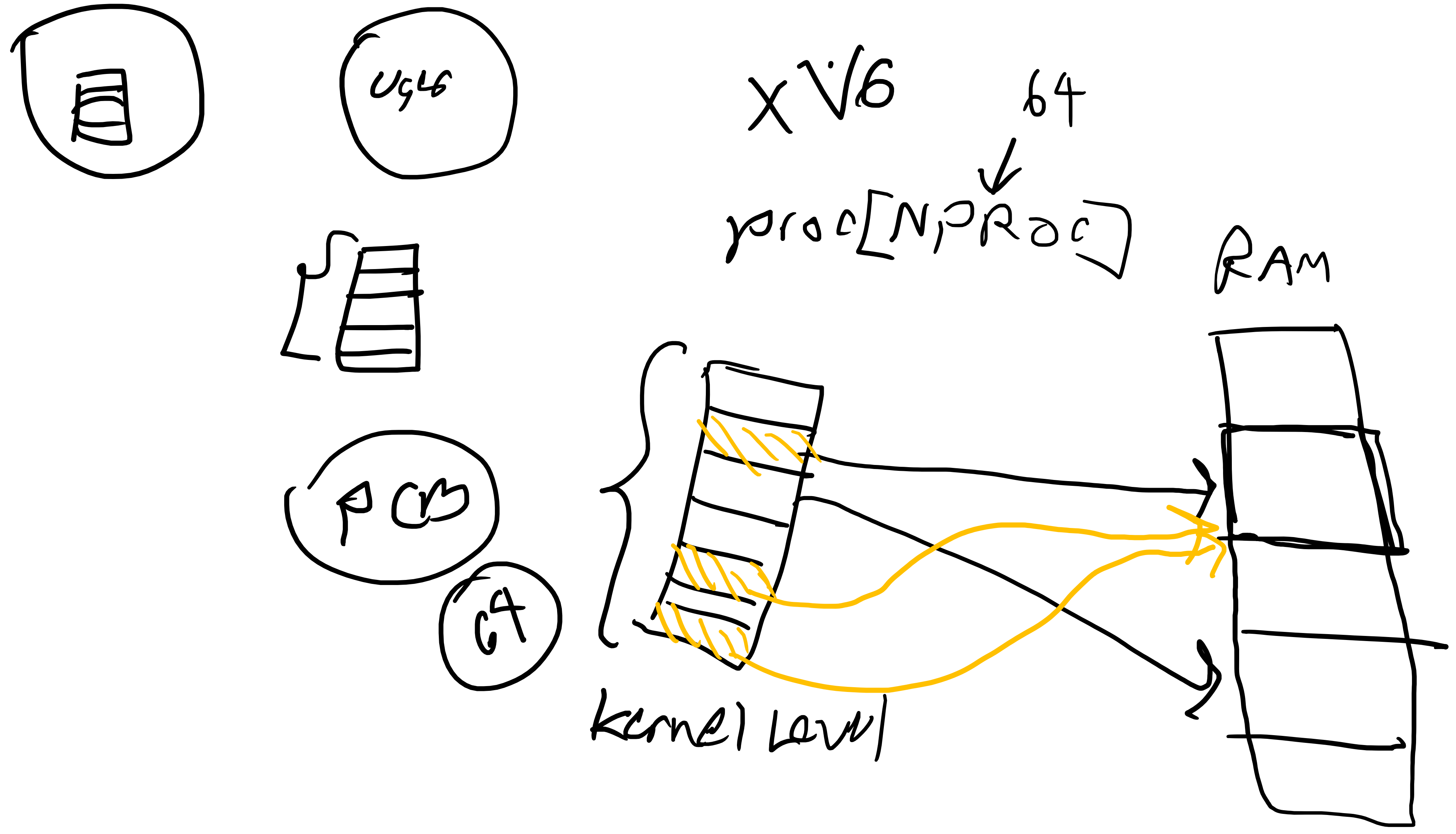
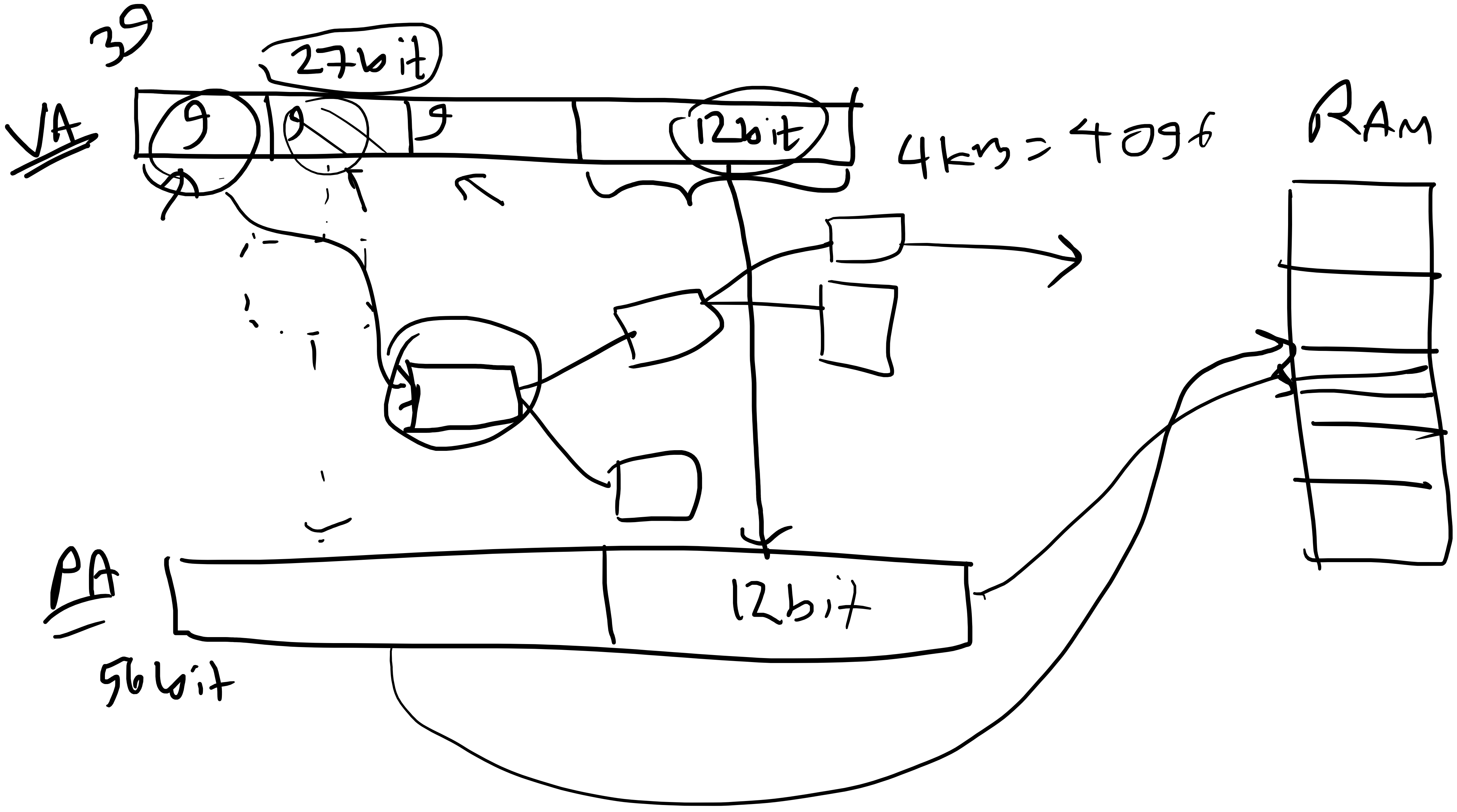
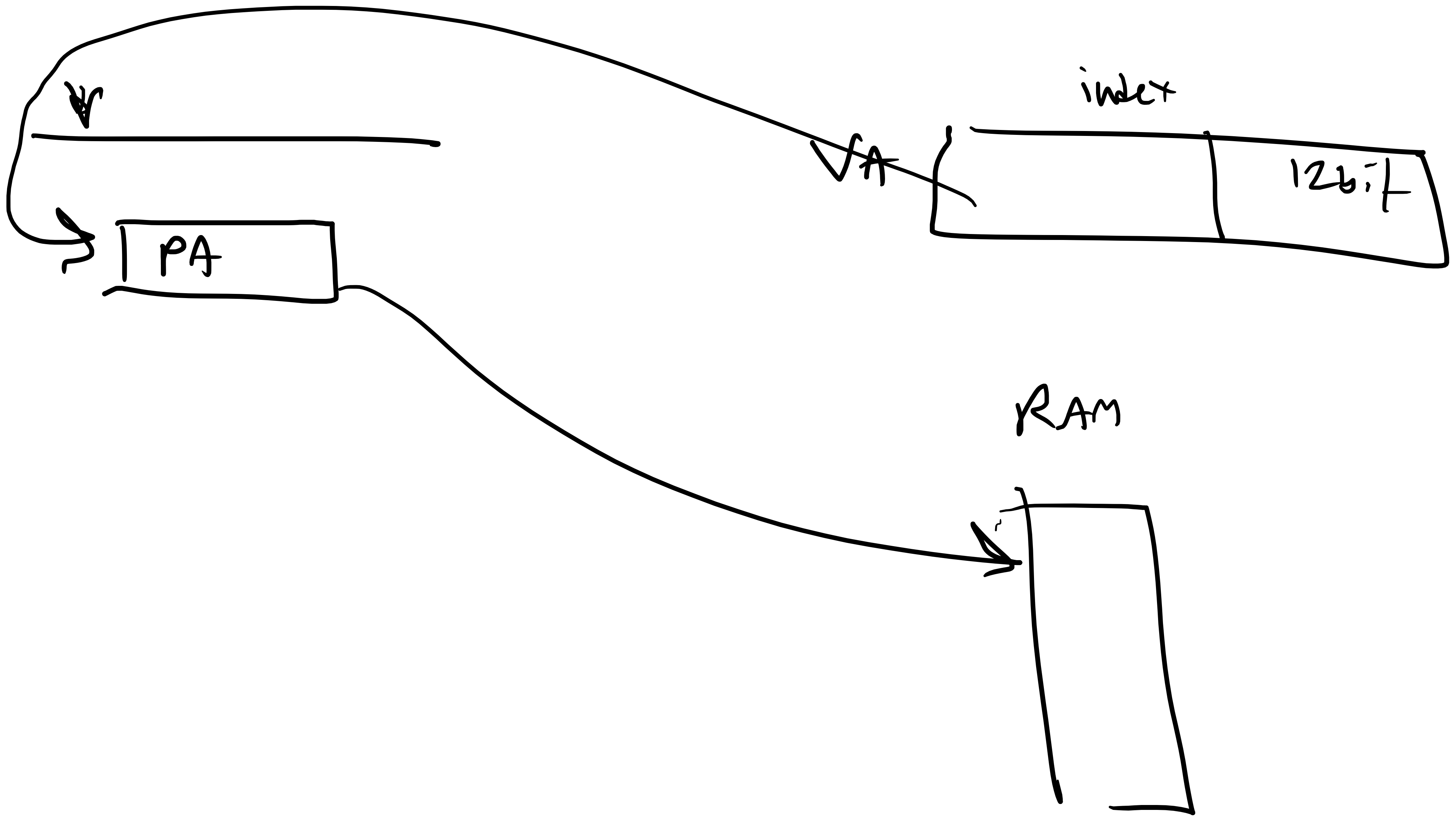
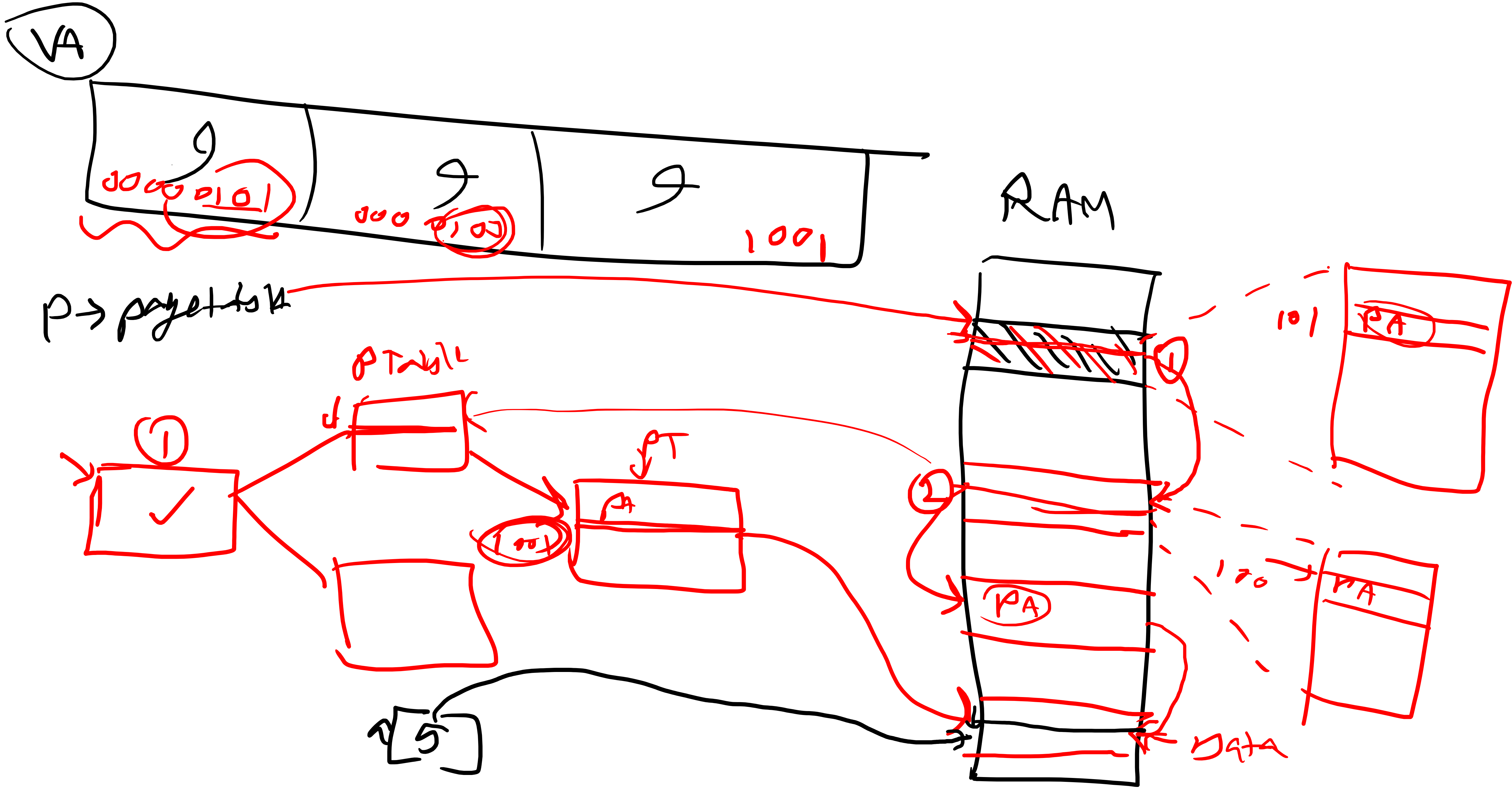

Ug6
XV6
64
proc[NPROC]
RAM
PCB
64
kernel level

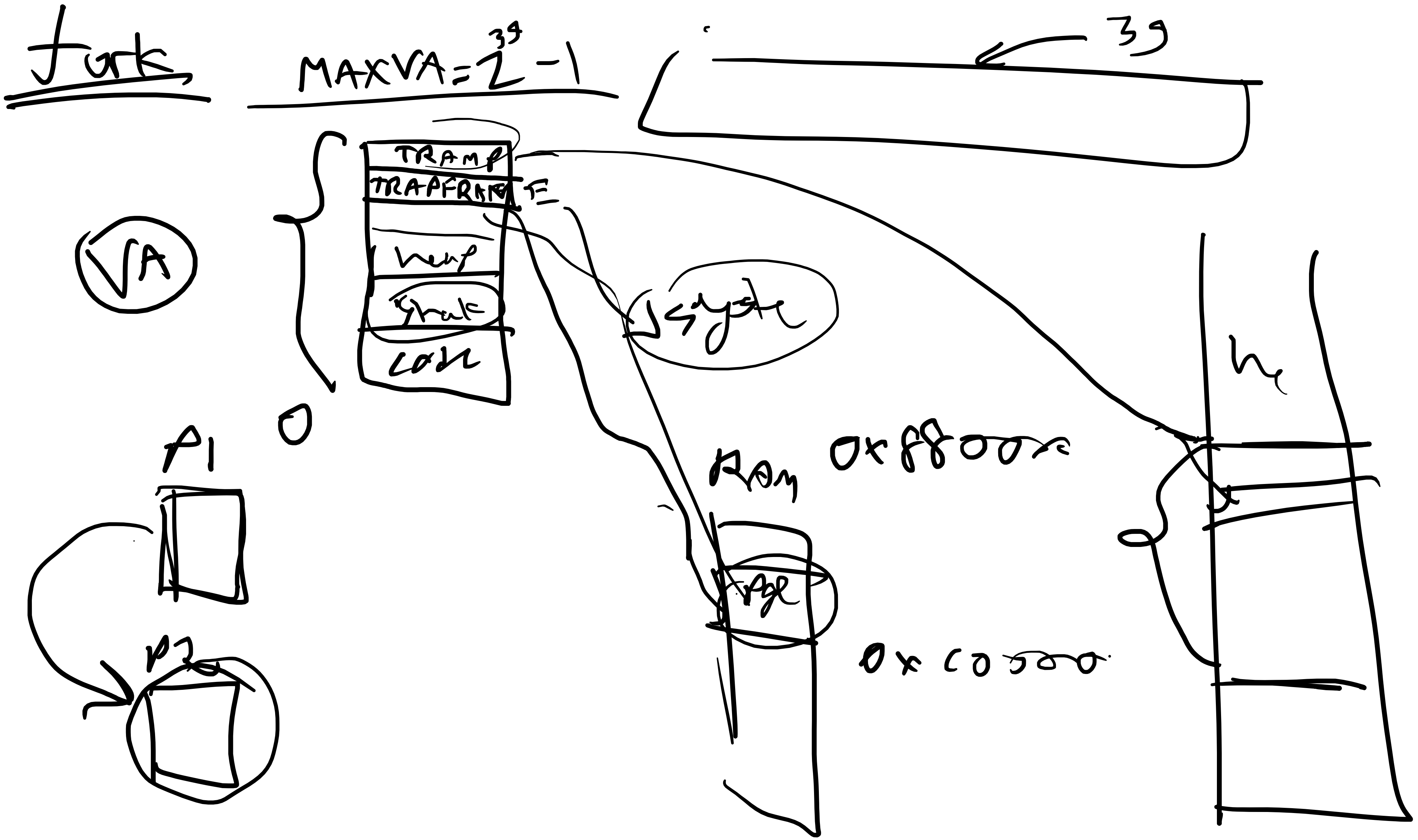

39
27bit
VA
9
9
9
12bit
4KB = 4096
RAM
PA
12bit
56bit

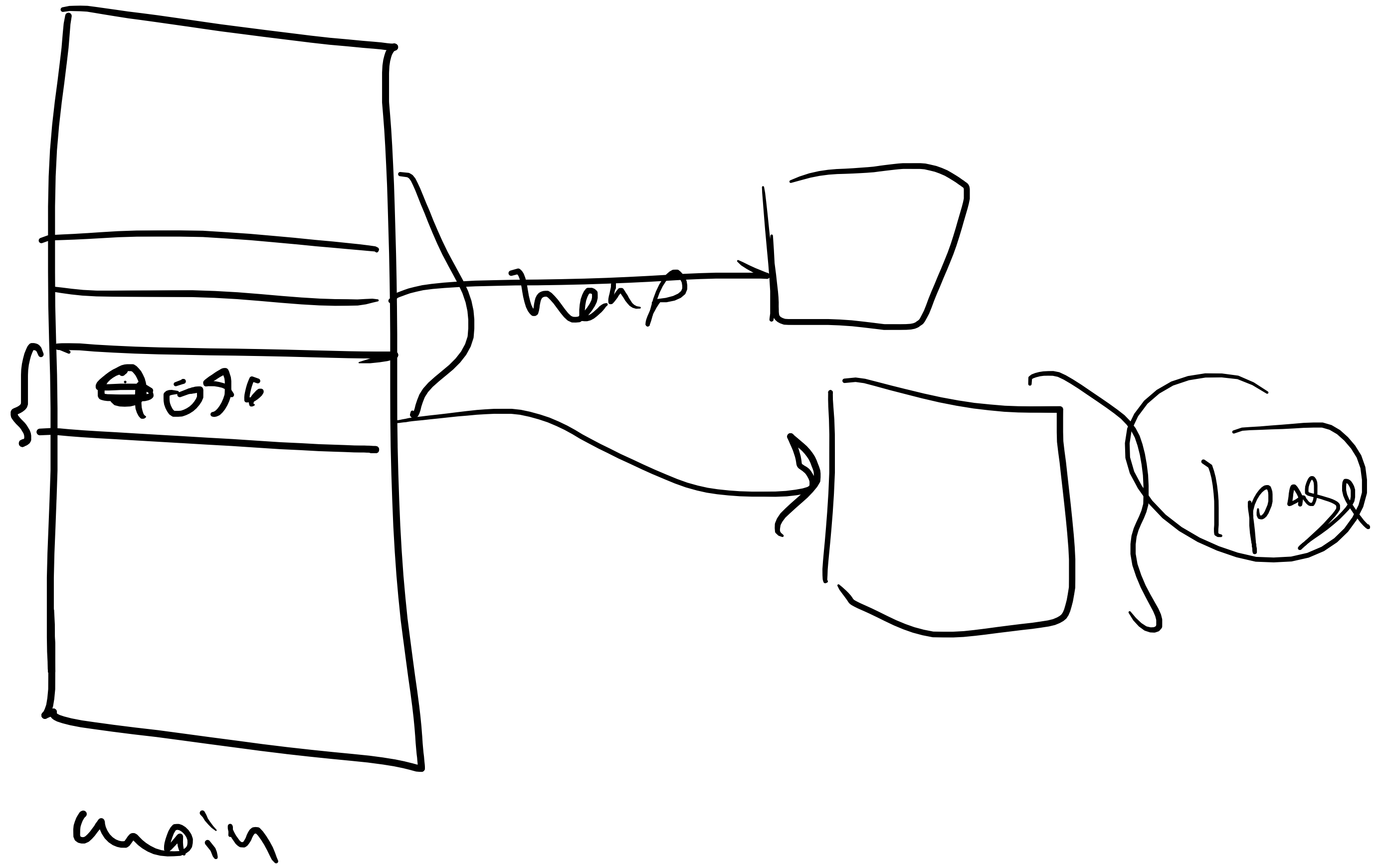


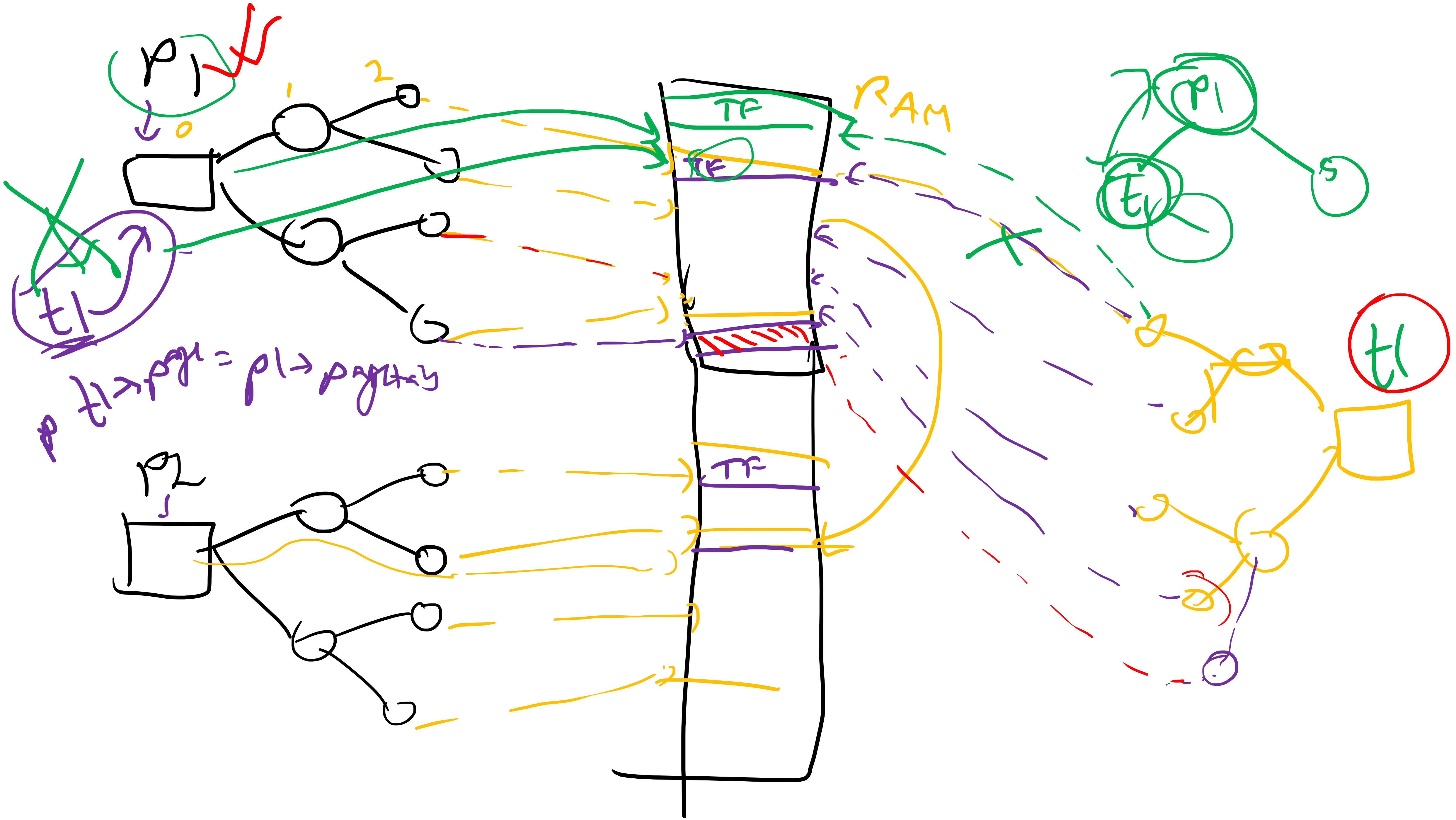
vr
PA
index
VA
12bit
RAM

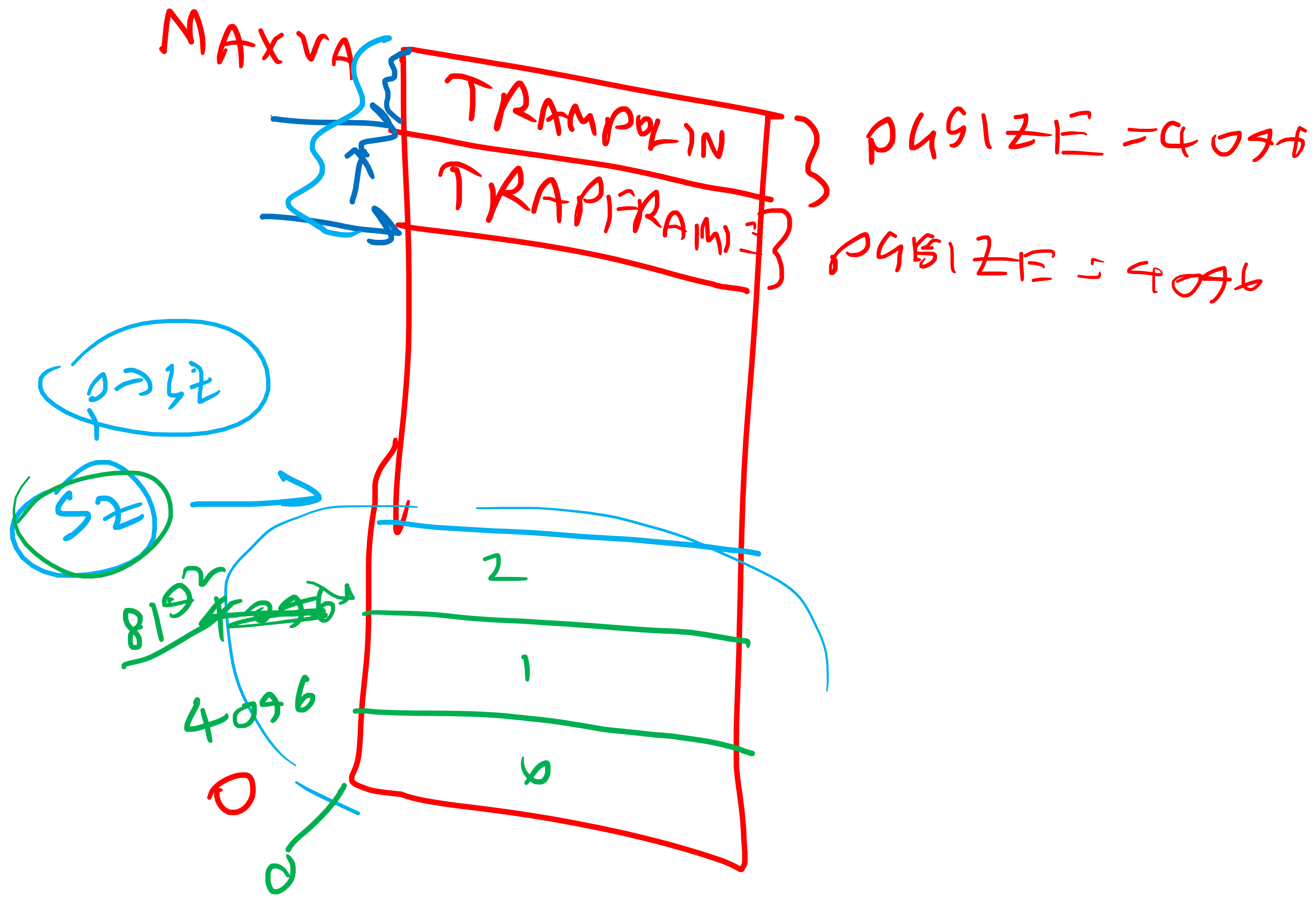

VA
9
9
9
000000101
000 0101
1001
p->pagetable
RAM
PTable
PT
PA
1001
101
PA
PA
100
PA
5
Data
1
2
1

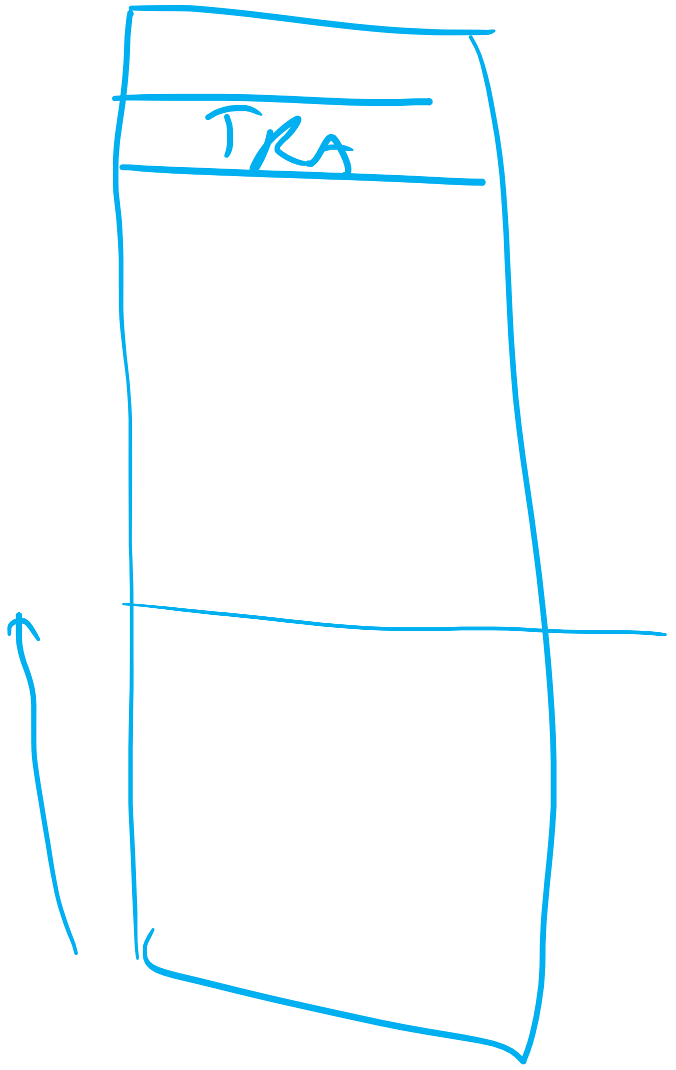
fork

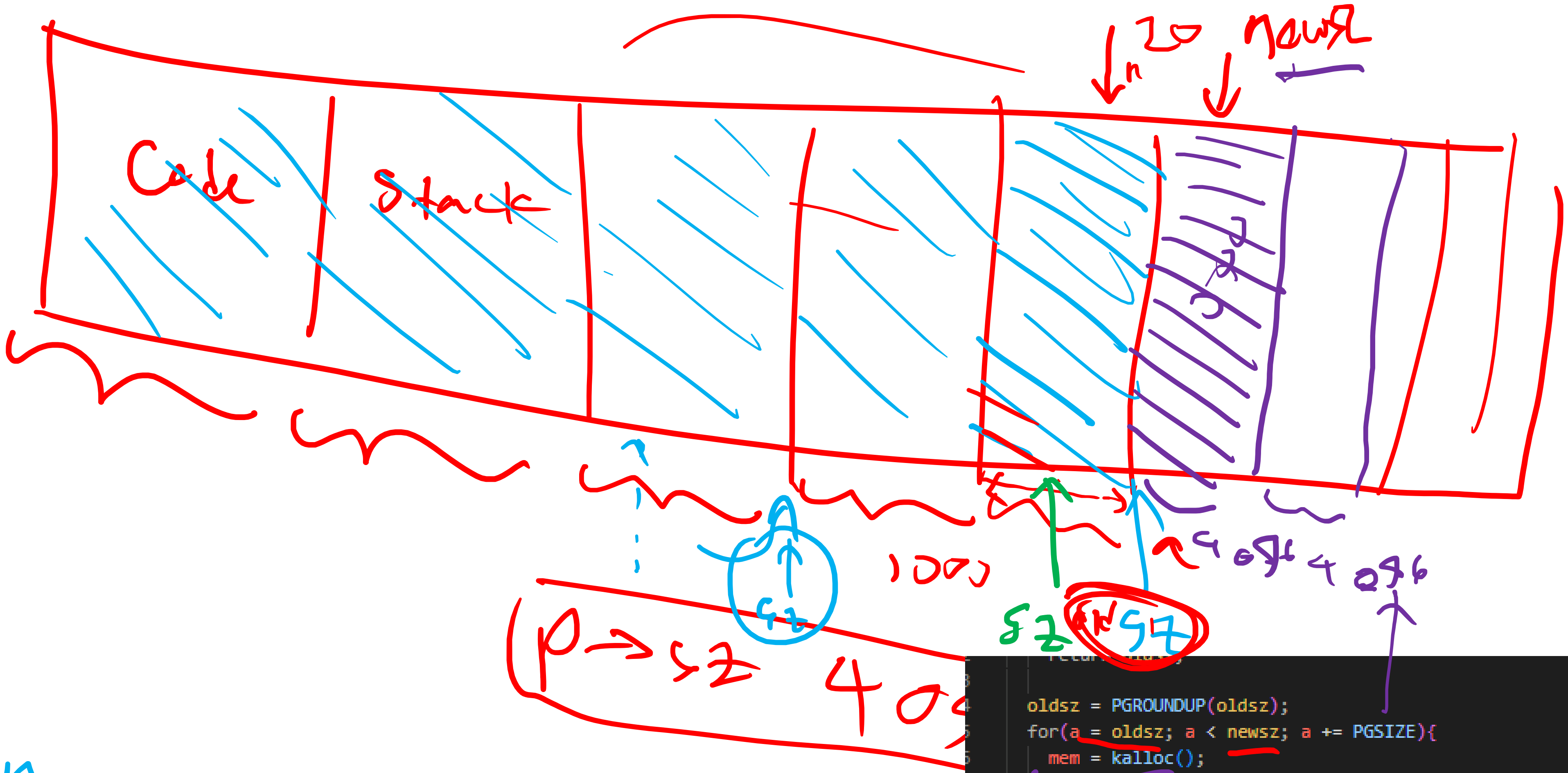

fork
MAXVA = $2^{39}-1$
39
VA
TRAMP
TRAPFRAME
heap
stack
code
0
P1
P2
usystem
RAM
0x8800
page
0x C0000
ke

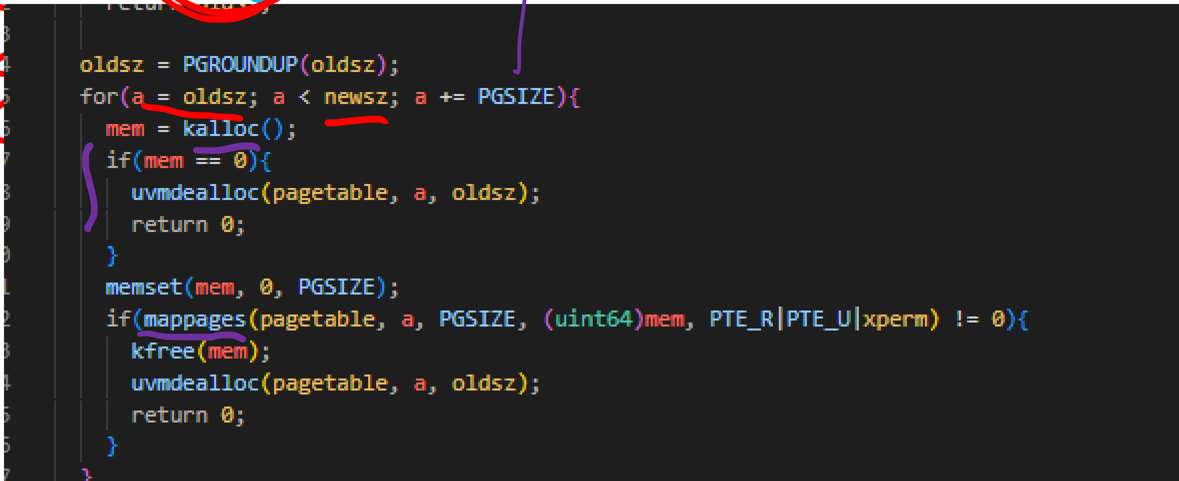

4096
heap
1 page
main


P1
0
1
2
t1
p t1->pgd = p1->pgdtable
P2
TF
TF
TF
RAM
P1
t1
t1

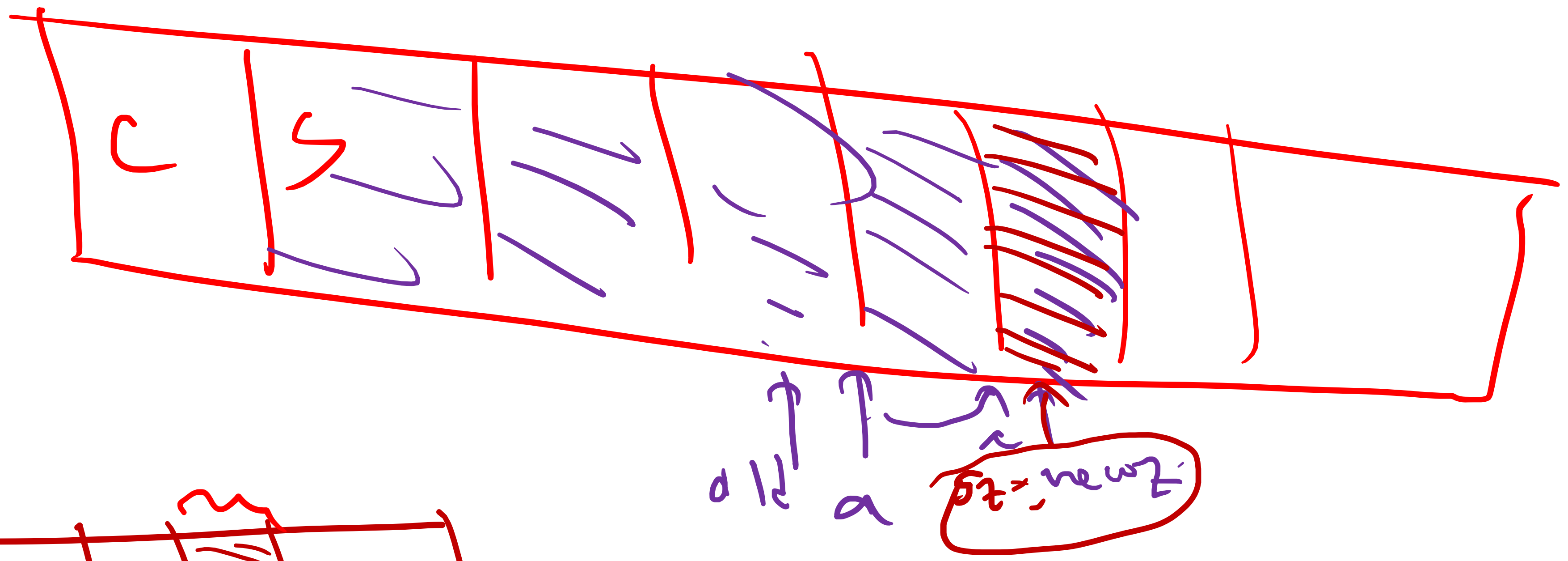

MAXVA
TRAMPOLIN
TRAPFRAME
PGSIZE = 4096
PGSIZE = 4096
0 → SZ
SZ
2
1
0
8192
~~4096~~
4096
0
0

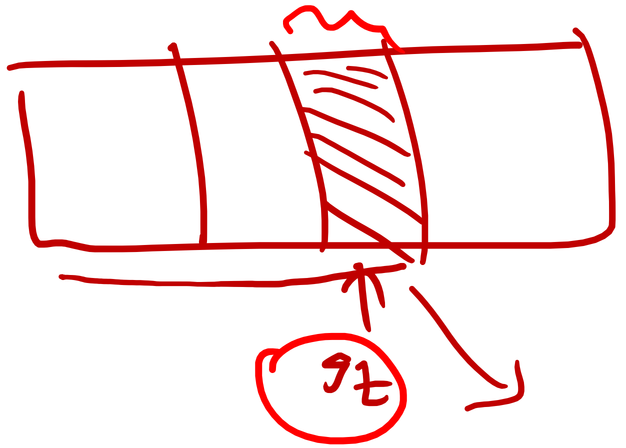

TRA

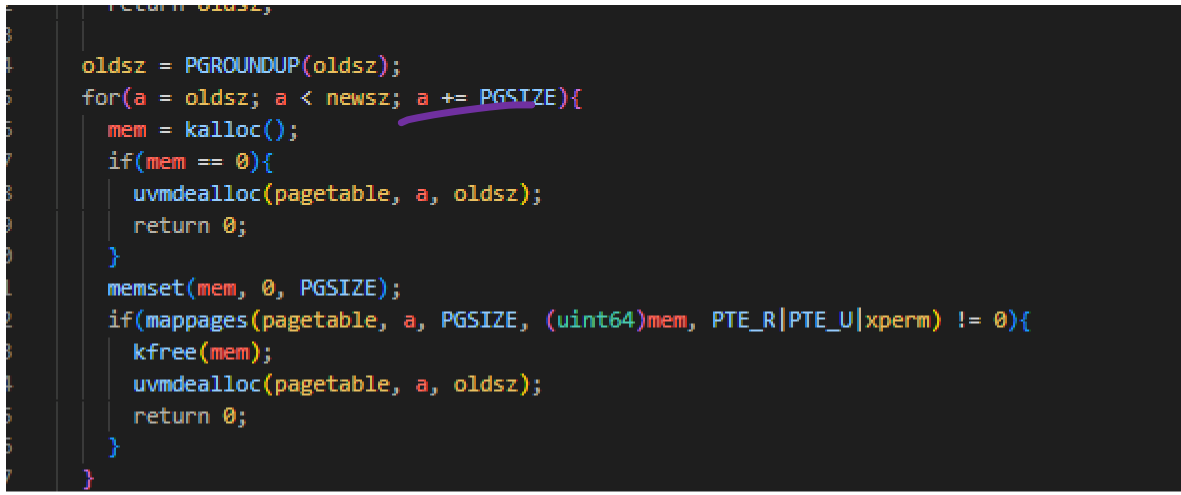



```
    oldsz = PGROUNDUP(oldsz);
    for(a = oldsz; a < newsz; a += PGSIZE){
      mem = kalloc();
      if(mem == 0){
        uvmdealloc(pagetable, a, oldsz);
        return 0;
      }
      memset(mem, 0, PGSIZE);
      if(mappages(pagetable, a, PGSIZE, (uint64)mem, PTE_R|PTE_U|xperm) != 0){
        kfree(mem);
        uvmdealloc(pagetable, a, oldsz);
        return 0;
      }
```

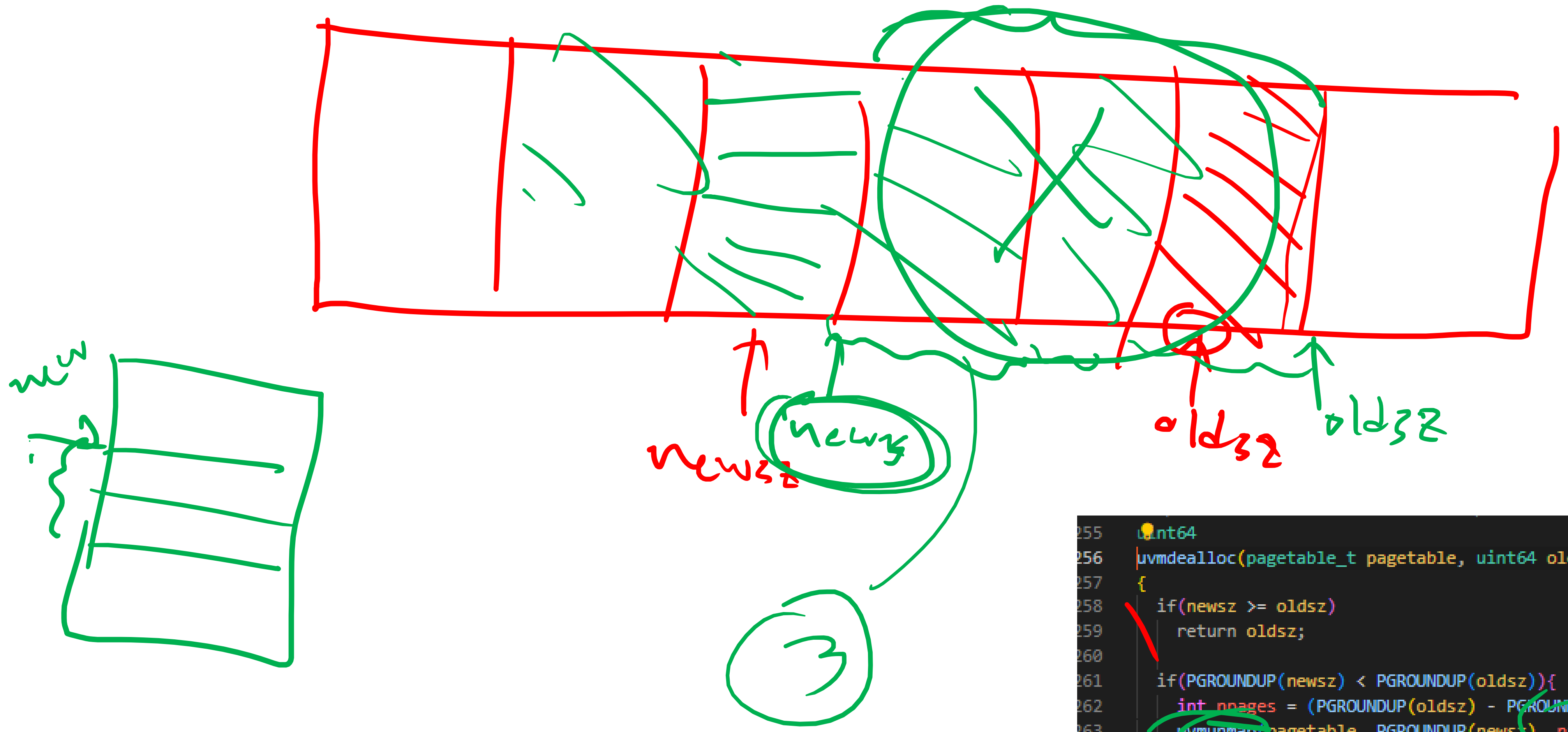
sz = PGROUNDUP(sz) =

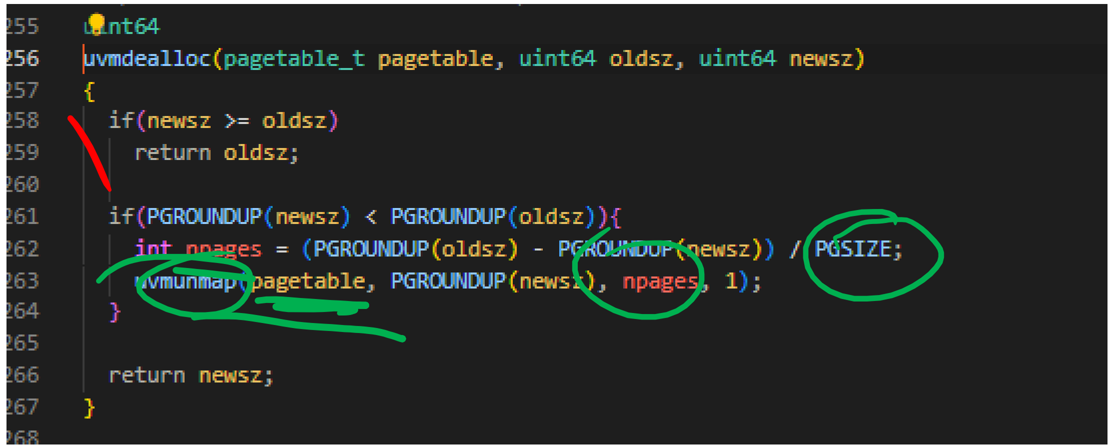




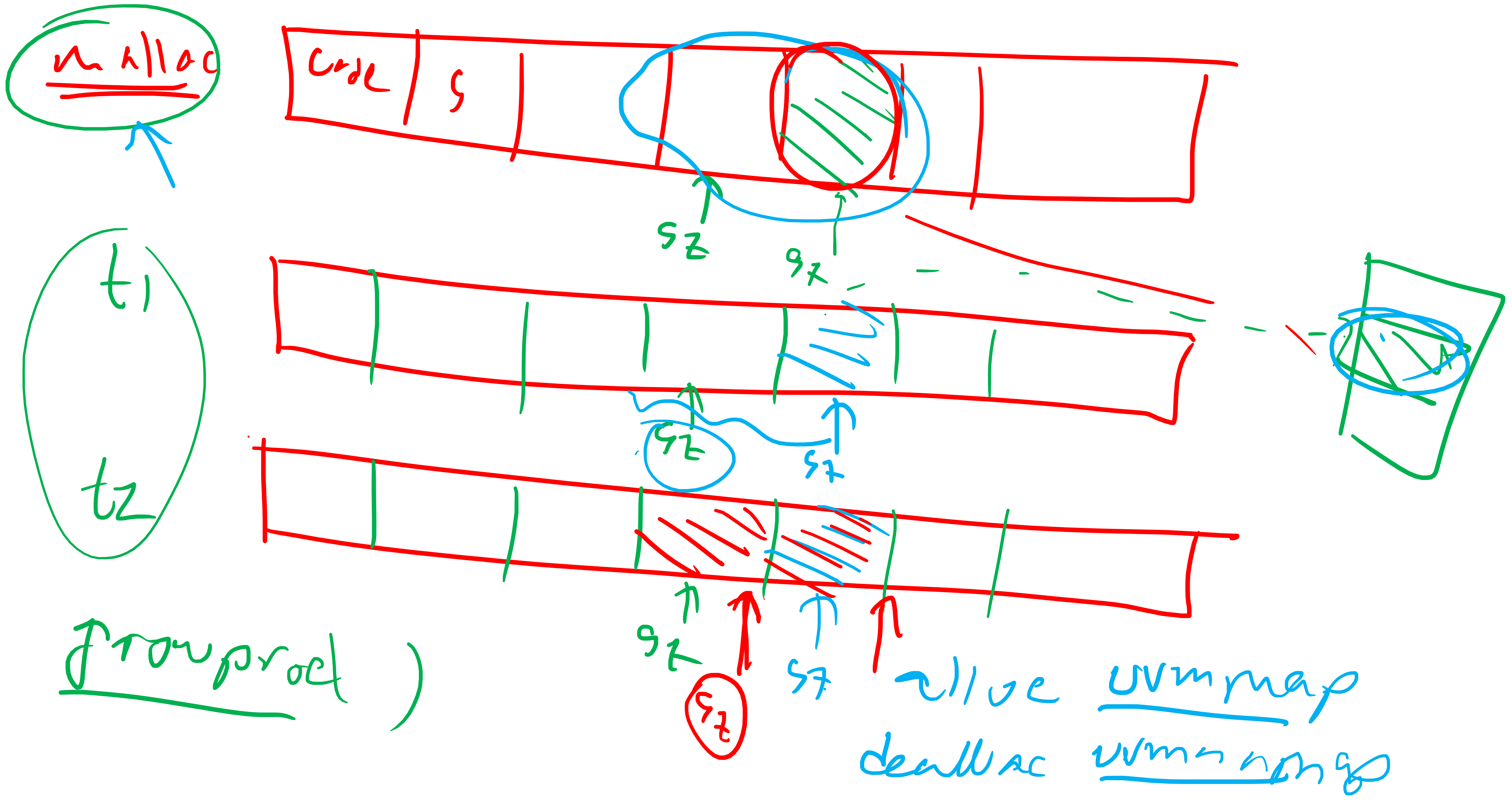


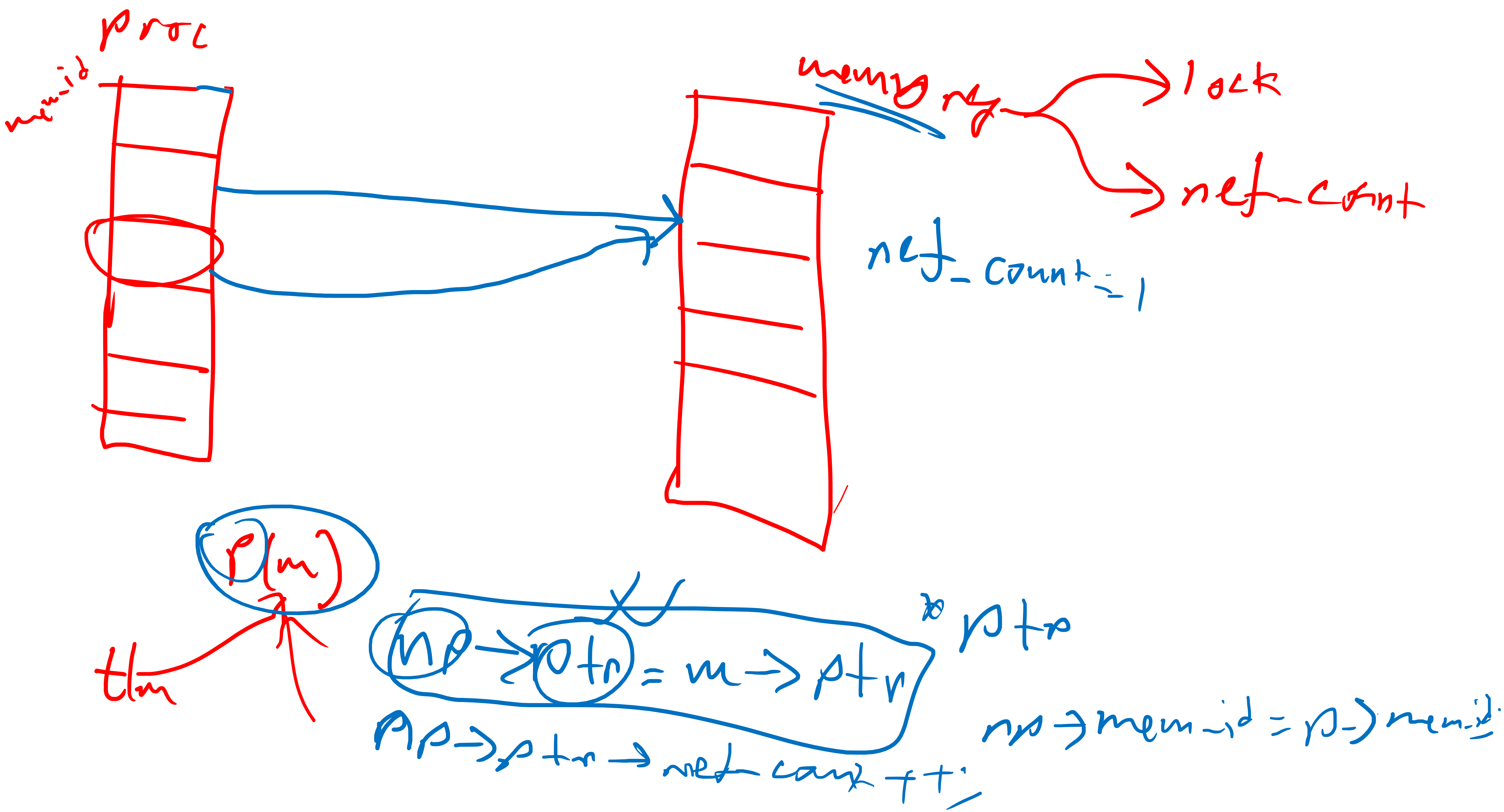
```
    return oldsz;

  oldsz = PGROUNDUP(oldsz);
  for(a = oldsz; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      uvmdealloc(pagetable, a, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pagetable, a, PGSIZE, (uint64)mem, PTE_R|PTE_U|xperm) != 0){
      kfree(mem);
      uvmdealloc(pagetable, a, oldsz);
      return 0;
    }
  }
```



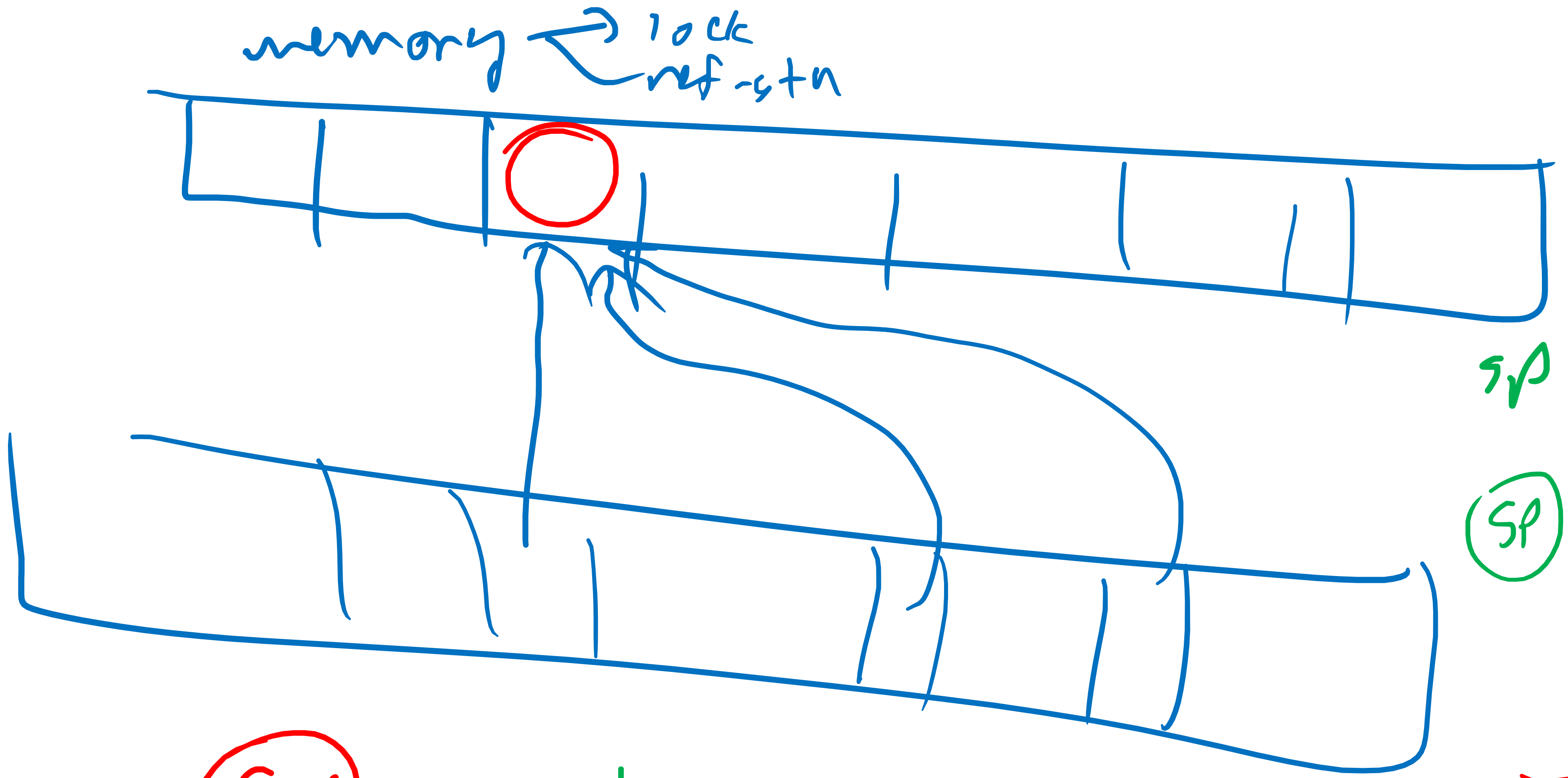


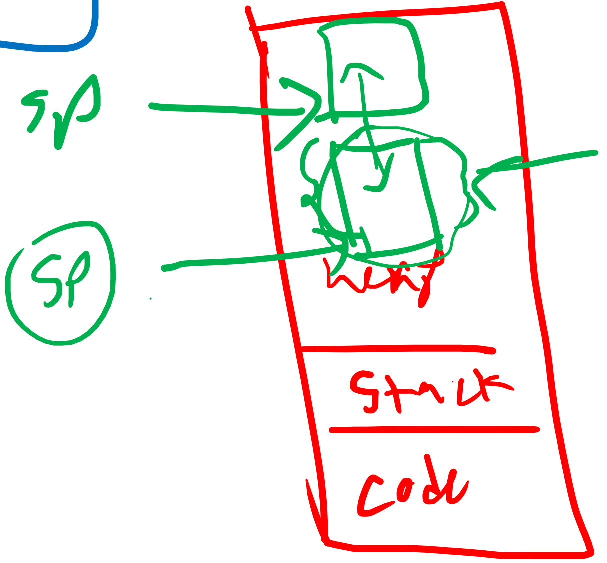
```
uint64
uvmdealloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz)
{
  if(newsz >= oldsz)
    return oldsz;

  if(PGROUNDUP(newsz) < PGROUNDUP(oldsz)){
    int npages = (PGROUNDUP(oldsz) - PGROUNDUP(newsz)) / PGSIZE;
    uvmunmap(pagetable, PGROUNDUP(newsz), npages, 1);
  }

  return newsz;
}
```


malloc
code
s
sz
sz
t1
t2
sz
sz
sz
sz
sz
sz
growproc)
alloc uvmmap
dealloc uvmunmap



```
np->ptr = m->ptr
np->ptr->ref_count++;
np->mem_id = p->mem_id
```

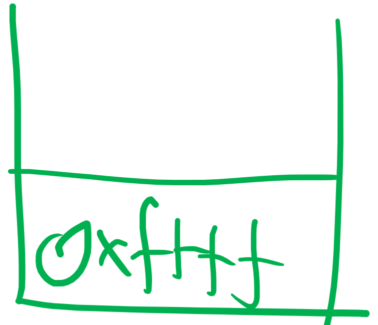




sp

$sp -= uint64$ 8

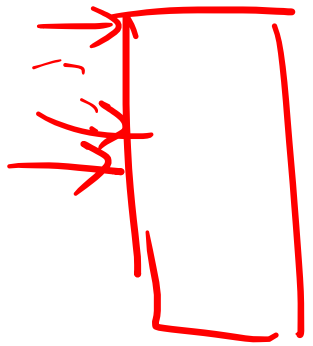
$\boxed{sp = sp \% 16}$

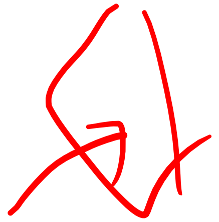


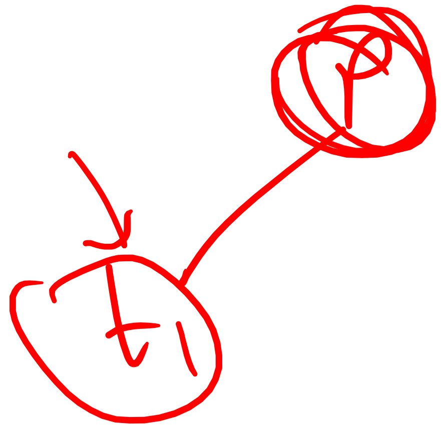
np →   a0 =

sp $-= sp \% 16$

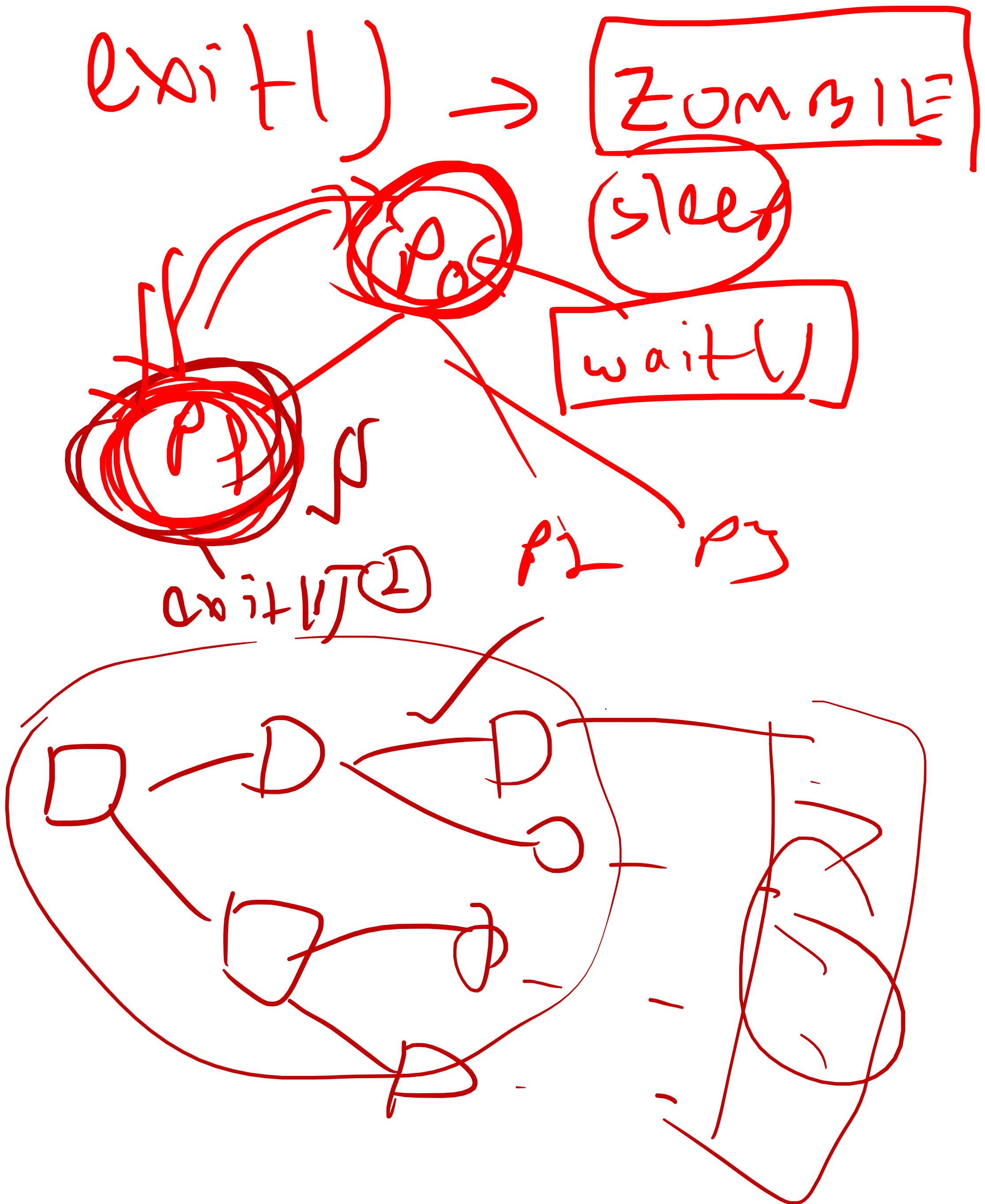
0000

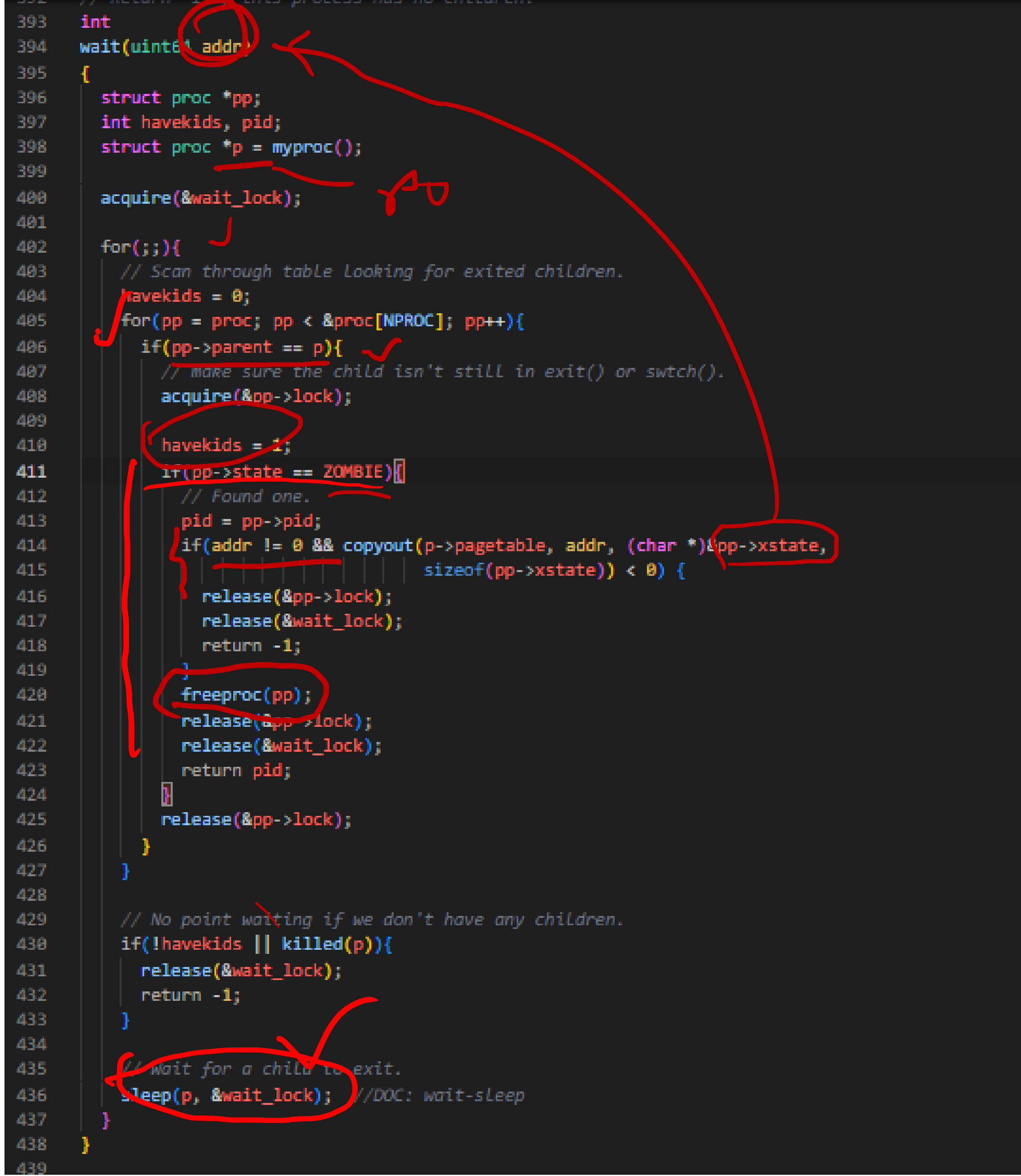




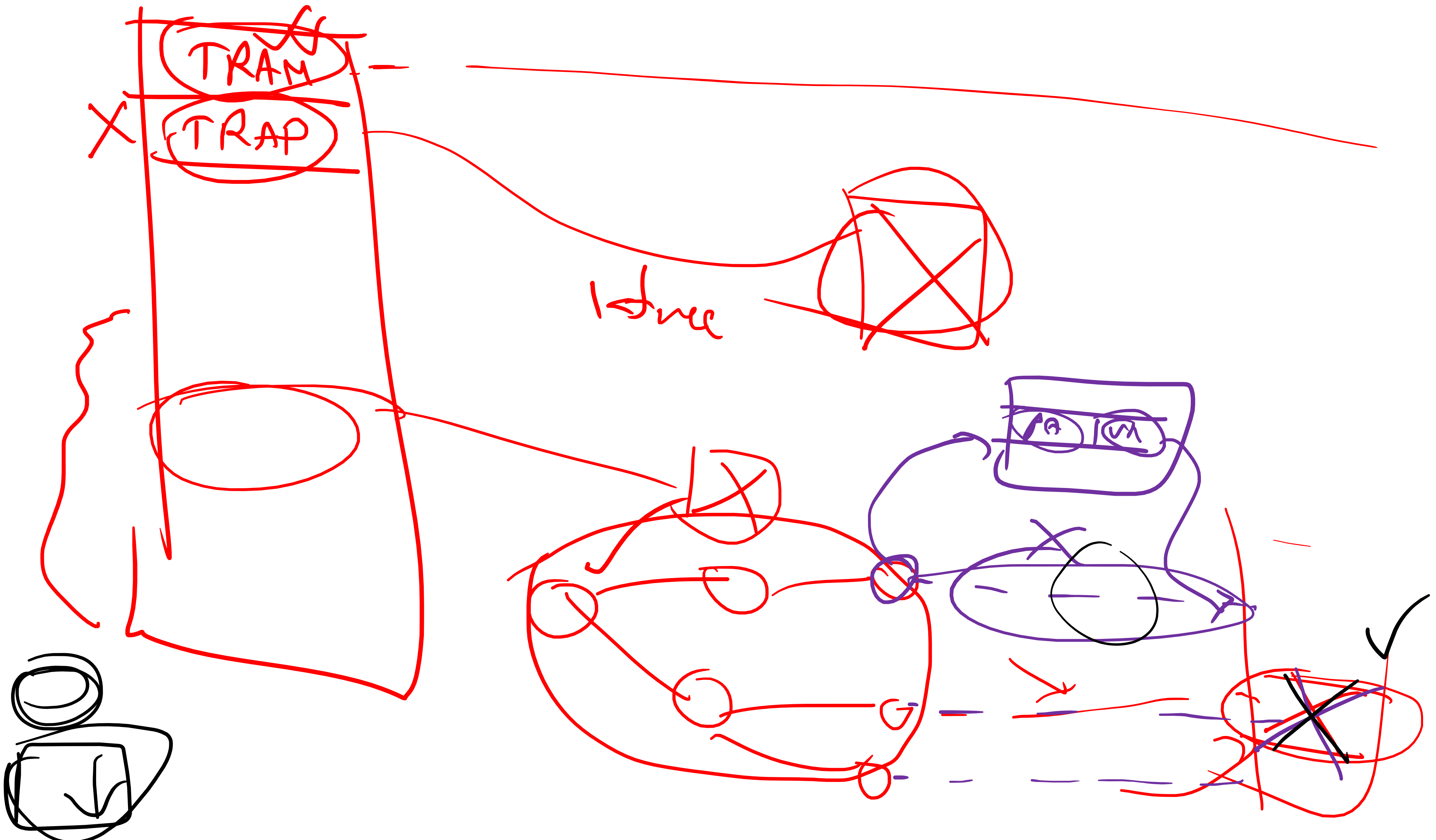


join()
exit()

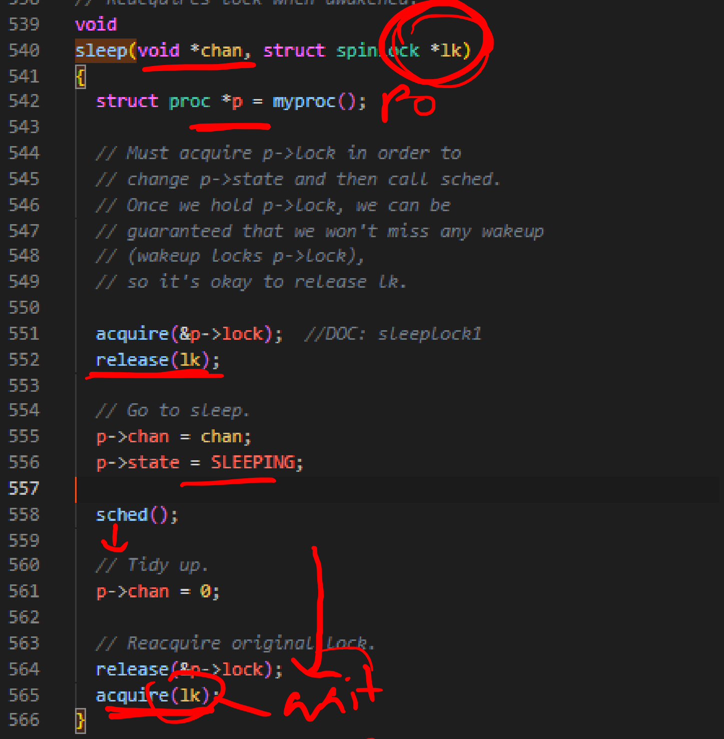




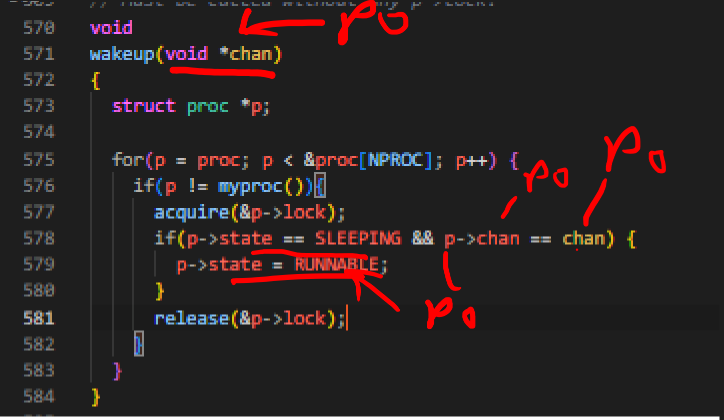
```
int
wait(uint64 addr)
{
  struct proc *pp;
  int havekids, pid;
  struct proc *p = myproc();

  acquire(&wait_lock);

  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(pp = proc; pp < &proc[NPROC]; pp++){
      if(pp->parent == p){
        // make sure the child isn't still in exit() or swtch().
        acquire(&pp->lock);

        havekids = 1;
        if(pp->state == ZOMBIE){
          // Found one.
          pid = pp->pid;
          if(addr != 0 && copyout(p->pagetable, addr, (char *)&pp->xstate,
                                  sizeof(pp->xstate)) < 0) {
            release(&pp->lock);
            release(&wait_lock);
            return -1;
          }
          freeproc(pp);
          release(&pp->lock);
          release(&wait_lock);
          return pid;
        }
        release(&pp->lock);
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || killed(p)){
      release(&wait_lock);
      return -1;
    }

    // Wait for a child to exit.
    sleep(p, &wait_lock);  //DOC: wait-sleep
  }
}
```



TRAM
TRAP
free

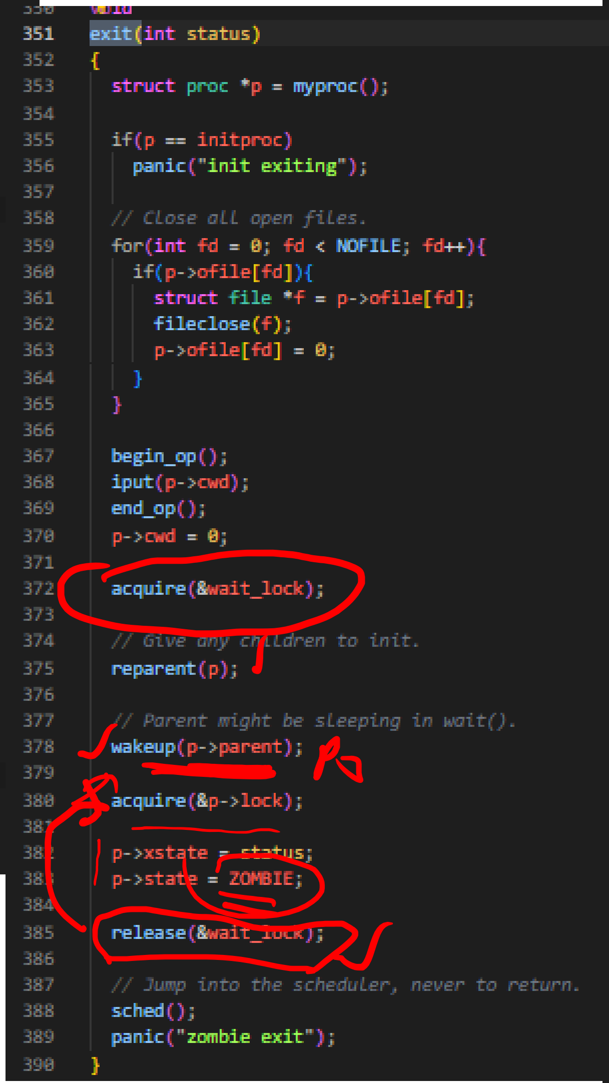


```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  // Must acquire p->lock in order to
  // change p->state and then call sched.
  // Once we hold p->lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup locks p->lock),
  // so it's okay to release lk.

  acquire(&p->lock);  //DOC: sleeplock1
  release(lk);

  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();

  // Tidy up.
  p->chan = 0;

  // Reacquire original lock.
  release(&p->lock);
  acquire(lk);
}
```

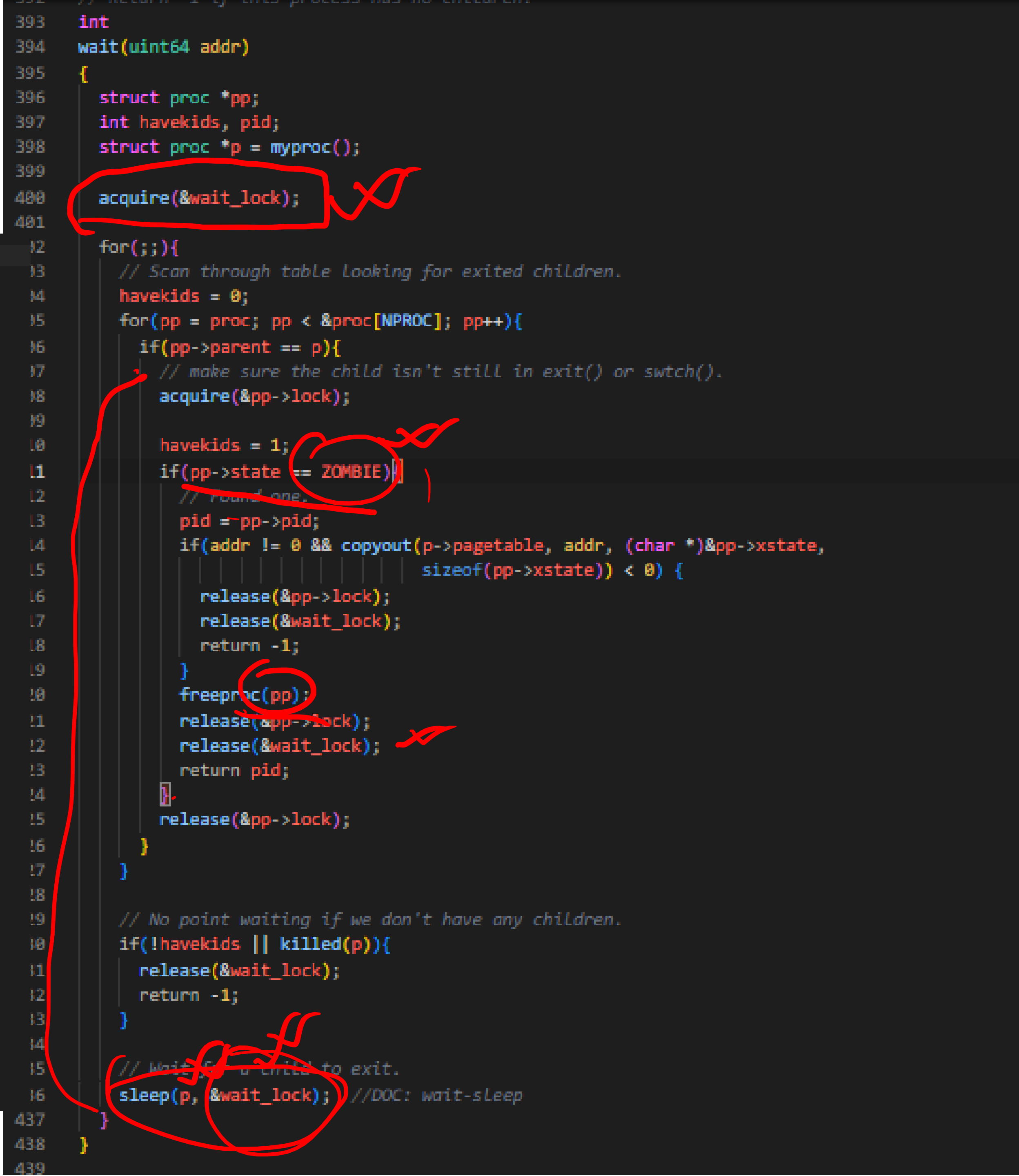


```
void
wakeup(void *chan)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    if(p != myproc()){
      acquire(&p->lock);
      if(p->state == SLEEPING && p->chan == chan) {
        p->state = RUNNABLE;
      }
      release(&p->lock);
    }
  }
}
```

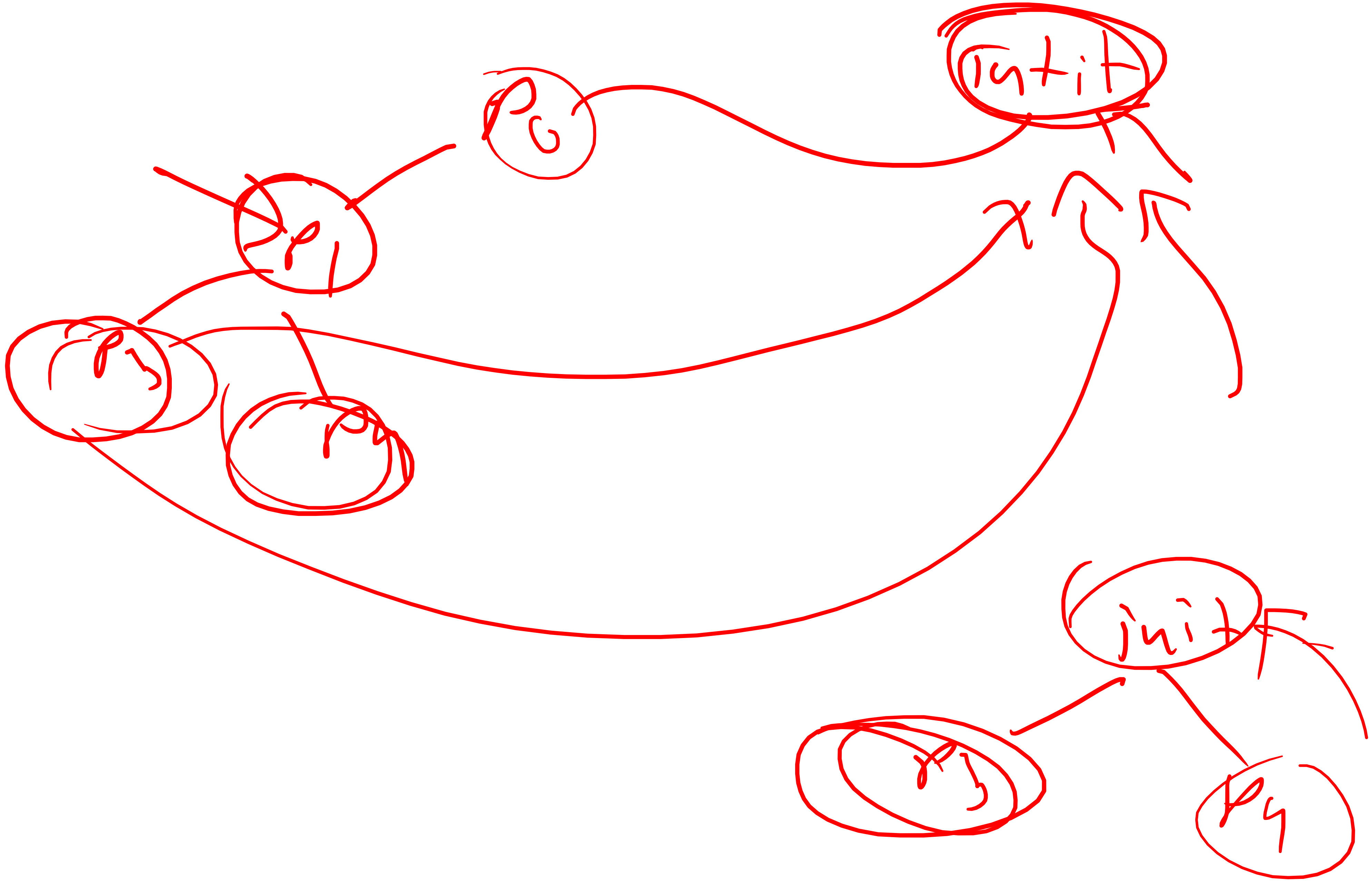
p0 ⇒ chan = p0



```
exit(int status)
{
  struct proc *p = myproc();

  if(p == initproc)
    panic("init exiting");

  // Close all open files.
  for(int fd = 0; fd < NOFILE; fd++){
    if(p->ofile[fd]){
      struct file *f = p->ofile[fd];
      fileclose(f);
      p->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(p->cwd);
  end_op();
  p->cwd = 0;

  acquire(&wait_lock);

  // Give any children to init.
  reparent(p);

  // Parent might be sleeping in wait().
  wakeup(p->parent);

  acquire(&p->lock);

  p->xstate = status;
  p->state = ZOMBIE;

  release(&wait_lock);

  // Jump into the scheduler, never to return.
  sched();
  panic("zombie exit");
}
```



```
int
wait(uint64 addr)
{
  struct proc *pp;
  int havekids, pid;
  struct proc *p = myproc();

  acquire(&wait_lock);

  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(pp = proc; pp < &proc[NPROC]; pp++){
      if(pp->parent == p){
        // make sure the child isn't still in exit() or swtch().
        acquire(&pp->lock);

        havekids = 1;
        if(pp->state == ZOMBIE){
          // Found one.
          pid = pp->pid;
          if(addr != 0 && copyout(p->pagetable, addr, (char *)&pp->xstate,
                                  sizeof(pp->xstate)) < 0) {
            release(&pp->lock);
            release(&wait_lock);
            return -1;
          }
          freeproc(pp);
          release(&pp->lock);
          release(&wait_lock);
          return pid;
        }
        release(&pp->lock);
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || killed(p)){
      release(&wait_lock);
      return -1;
    }

    // Wait for a child to exit.
    sleep(p, &wait_lock);  //DOC: wait-sleep
  }
}
```



init
p_0
p_1
p_3
p_4
init
p_3
p_4