هنگام اجرا شدن سیستم عامل تابع main اجرا می‌شود و در طی اجرای یک سری تابع بعضی از بخش‌های اولیه سیستم عامل initialize می‌باشد. با اجرا شدن تابع userinit اولین user process سیستم عامل را ایجاد می‌کند.

```c
// start() jumps here in supervisor mode on all CPUs.
void
main()
{
  if(cpuid() == 0){
    consoleinit();
    printfinit();
    printf("\n");
    printf("xv6 kernel is booting\n");
    printf("\n");
    kinit();         // physical page allocator
    kvminit();       // create kernel page table
    kvminithart();   // turn on paging
    procinit();      // process table
    trapinit();      // trap vectors
    trapinithart();  // install kernel trap vector
    plicinit();      // set up interrupt controller
    plicinithart();  // ask PLIC for device interrupts
    binit();         // buffer cache
    iinit();         // inode table
    fileinit();      // file table
    virtio_disk_init(); // emulated hard disk
    userinit();      // first user process
    __sync_synchronize();
    started = 1;
```

```c
// Set up first user process.
void
userinit(void)
{
  struct proc *p;

  p = allocproc();
  initproc = p;

  // allocate one user page and copy initcode's instructions
  // and data into it.
  uvmfirst(p->pagetable, initcode, sizeof(initcode));
  p->sz = PGSIZE;

  // prepare for the very first "return" from kernel to user.
  p->trapframe->epc = 0;      // user program counter
  p->trapframe->sp = PGSIZE;  // user stack pointer

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  p->state = RUNNABLE;

  release(&p->lock);
}
```

همانطور که در کد userinit دیده می‌شود یک struct از نوع proc وجود دارد که اطلاعات مربوط به یک process در آن ذخیره می‌شود. ابتدا تابع allocproc در جدول process ها به دنبال یک process ه unused یا استفاده نشده می‌گردد در صورت پیدا کردن یک سری عملیات‌های اولیه مثل تعیین کردن pid یا تغییر state به حالت USED و غیره را انجام داده و اشاره گر به این process را برمی‌گرداند. اگر این عمل ناموفق باشد و هیچ process با state ه UNUSED پیدا نکند تابع 0 برمی‌گرداند.

```c
  struct proc *p;

  p = allocproc();
  initproc = p;

```

```c
// Look in the process table for an UNUSED proc.
// If found, initialize state required to run in the kernel,
// and return with p->lock held.
// If there are no free procs, or a memory allocation fails, return 0.
You, last week | 1 author (You)
static struct proc*
allocproc(void)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == UNUSED) {
      goto found;
    } else {
      release(&p->lock);
    }
  }
  return 0;
```

در مرحله بعد تابع uvmfirst اجرا می‌شود در این مرحله ابتدا یک page table ایجاد شده و کدهای اولیه یا همان initcode که در واقع instruction های اولیه هستند در آن ریخته می‌شود.

```c
// allocate one user page and copy initcode's instructions
// and data into it.
uvmfirst(p->pagetable, initcode, sizeof(initcode));
p->sz = PGSIZE;
```

```c
// Load the user initcode into address 0 of pagetable,
// for the very first process.
// sz must be less than a page.
void
uvmfirst(pagetable_t pagetable, uchar *src, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("uvmfirst: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pagetable, 0, PGSIZE, (uint64)mem, PTE_W|PTE_R|PTE_X|PTE_U);
  memmove(mem, src, sz);
}
```

در مراحل انتهایی یک سری عملیات‌ها مثل مقداردهی به program counter و stack pointer تعیین نام و working directory پردازه و همچنین تغییر وضعیت پردازه به RUNNABLE اتفاق می افتد در انتها هم قفل پردازه که در تابع allocproc انجام شده بود باز می‌شود.

```c
  // prepare for the very first "return" from kernel to user.
  p->trapframe->epc = 0;      // user program counter
  p->trapframe->sp = PGSIZE;  // user stack pointer

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  p->state = RUNNABLE;

  release(&p->lock);
```

بخش دوم:

چون تابع kfreemem را در ابتدای بوت شدن سیستم عامل صدا می‌زنیم مقدار فضای آزاد بسیار نزدیک به میزان کل فضای مموری است.

```
xv6 kernel is booting

OUTLINE
xv6 free memory: 134078464
Headings you add to the document will
hart 2 starting
hart 1 starting
init: starting sh
$ kfreemem
Free memory: 133382144
Total memory: 134217728
$
```

هنگام کامپایل شدن سیستم عامل اسکریپت perl ه usys.pl اجرا می‌شود و فایل usys.S یک فایل اسمبلی است که شامل فراخوانی سیستمی مورد نظر ما است.

```
.global kfreemem
kfreemem:
 li a7, SYS_kfreemem
 ecall
 ret
```

هنگام اجرای تابع systemcall مقدار a7 فراخوانی شده و عدد مربوط به system call مورد نظر به دست می‌آید و تابع مربوط به آن فراخوانی می‌شود.

```
void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // Use num to lookup the system call function for num, call it,
    // and store its return value in p->trapframe->a0
    p->trapframe->a0 = syscalls[num]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```