فاز اول پروژه سیستم عامل

سینا زمانی– آیسا میاهی نیا

استاد انتظاری

برای اضافه کردن سیستم کالی جدید به این سیستم عامل ابتدا باید یک سری تغییرات داخل فایل های زیر داد:

1. syscall.h
2. syscall.c
3. sysproc.c
4. usys.S
5. user.h

سپس یک instance از استراکت proc که در فایل proc.h تعریف شده است و همه ویژگی های تک تک process ها را دارا است، ساختیم. این instanceاز نوع پوینتر است و با استفاده از میتوان به process های دیگر هم دسترسی داشت. بنابراین یک حلقه روی آن زدیم و اگر state آن RUNABBLE یا RUNNING بود، pid و memsize(sz) آن process را در قالب شی از استراکت proc_info ذخیره کردیم و این ابجکت ها را در آرایه ای ریختیم.

```
int
proc_dump(void)
{
  struct proc *p;
  sti();
  acquire(&ptable.lock);
  int count  = 0;
  struct proc_info processes[NPROC];
  for(p = ptable.proc ; p < &ptable.proc[NPROC]; p++)
  {
    if(p->state == RUNNABLE || p->state == RUNNING )
    {
      struct proc_info pi = {p->pid , p->sz};
      processes[count] = pi;
      count++;
    }
  }
```

اکنون در آرایه مذکور pid و memsize های process های هدف سوال را در قالب ابجکت هایی از proc_info ذخیره کردیم.

هدف بعدی سوال سورت کردن این آرایه ابتدا براساس memsize و سپس براساس pid می باشد.

نهایتا اطلاعات آرایه را پرینت کردیم.

```
  for(int i = 0 ; i<count-1 ;i++)
  {
    for(int j = i +1 ; j<count;j++)
    {
      if(processes[i].memsize > processes[j].memsize)
      {
        struct proc_info tmp = processes[i];
        processes[i] = processes[j];
        processes[j] = tmp;
      }
      else if(processes[i].memsize == processes[j].memsize)
      {
        if(processes[i].pid > processes[j].pid)
        {
          struct proc_info tmp = processes[i];
          processes[i] = processes[j];
          processes[j] = tmp;
        }
      }
    }
  }
  cprintf("pid \t memsize \n");
  for(int i = 0 ; i<count ;i++)
  {
    cprintf("%d \t %d \n" , processes[i].pid ,processes[i].memsize );
  }
  release(&ptable.lock);
  return 0;
}
```

واضح است هدف سوال نمایش process های مختلف با تاکید بر اختلاف سایز ممووری مصرف شده آن هاست فلذا یک فایل جدید برای تست کردن این system call ایجاد نمودیم. برای اینکه کارایی system call را به بهترین شکل نشان دهیم نیاز است با استفاده از سیستم کال fork، پروسس های متعدد ایجاد کنیم و برای هرکدام مقدار مختلفی از حافظه را malloc میکنیم.

بدین منظور دو fork ایجاد می کنیم، منتها چالشی که در ادامه به آن بر میخوریم این است که هر چهار پروسس موجود سیستم کال proc_dump را اجرا می کند و چهار بار خروجی یکسان پرینت می شود که به این حالت zombie process می گویند. برای جلوگیری از این اتفاق بعد از ایجاد هر fork یک شرط میذاریم که باعث شود پراسس پرنت، مقداری حافظه malloc کند و به اجرای ادامه کد بپردازد اما پراسسس فرزند در شرط گیر کند و تا پایان برنامه متوقف شود با این کار ما بین پراسس ها تمایز ایجاد کردیم و نهایتا با کمک پراسس پرنت اصلی و سیستم کال Proc_dump این تفاوت را به نمایش گذاشتیم.

```c
struct employees{
    int emp_id;
    char emp_name;
};

int main()
{
    int* ptr;
    int pid = fork();
    if( pid > 0)
    {
        ptr = (int*)malloc(100000 * sizeof(int));
        *ptr = 1;
    }
    else
    {
        while (1)
        {
            continue;
        }
    }

    int pid1 = fork();
    if( pid1 > 0)
    {
        struct employees* employeesData = malloc(1000 * sizeof *employeesData);
    }
    else
    {
        while (1)
        {
            continue;
        }
    }
    proc_dump();
    exit();
}
```

برای مثال خروجی برنامه به ازای دو fork و mallocهای متفاوت به صورت زیر است:

```
Booting from Hard Disk...
xv6...
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
$ sampleTest
pid     memsize
4       12288
5       412296
3       445064
$
```