دانشکده مهندسی کامپیوتر

تیر ۱۴۰۱

پروژه پایانی سیستم های عامل

نام استاد : دکتر انتظاری

اعضای گروه:

فاطمه زهرا بخشنده ۹۸۵۲۲۱۵۷

امین علیاری ۹۸۵۲۱۳۵۱

به نام خدا

قسمت اول پروژه:

برای گسترش proc structure کنونی ۴ فیلد stime , etime , iotime , rtime را به فایل proc.h اضافه می‌کنیم.

```
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
  enum procstate state;
  int pid;
  struct proc *parent;
  struct trapframe *tf;
  struct context *context;
  void *chan;
  int killed;
  struct file *ofile[NOFILE];
  struct inode *cwd;
  char name[16];   // Process
  int stime;
  int etime;
  int rtime;
  int iotime;
};
```

زمان ساخت یک پردازه جدید در تابع allocproc از فایل proc.c مقادیر اولیه این ۴ متغیر را تعیین می‌کنیم.

برای تعیین زمان شروع از ticks استفاده می‌کنیم و مقدار ۰ را به سه متغیر دیگر اختصاص می‌دهیم.

۴ خط دستور زیر به انتهای تابع اضافه می‌شوند.

```
p->stime = ticks;
p->etime = 0;
p->iotime = 0;
p->rtime = 0;
return p;
```

به ازای هر کلاک تابع trap فراخوانی می‌شود پس برای بروزرسانی مقادیر iotime و rtime کد زیر را به تابع trap در فایل trap.c اضافه می‌کنیم.

```
if(myproc()) {
  if(myproc()->state == SLEEPING) {
    myproc()->iotime++;
  }
  else if(myproc()->state == RUNNING) {
    myproc()->rtime++;
  }
}
```

زمانی که پردازه به پایان می‌رسد تابع exit در فایل proc.c فراخوانی می‌شوند پس برای بروزرسانی مقدار etime کد زیر را به انتهای این تابع اضافه می‌کنیم.

```
curproc->etime = ticks;
```

برای اضافه کردن سیستم کال waitx آن را به فایل syscall.h که لیست سیستم کال ها و شماره مرتبط با آن ها ذخیره شده است، اضافه می‌کنیم.

```
#define SYS_waitx  22
```

در فایل syscall.c یک پوینتر به سیستم کال تعریف شده متصل می‌کنیم.

```
[SYS_waitx]   sys_waitx
```

در صورتی که سیستم کال با شماره ۲۲ صدا زده شود، تابعی که متغیر sys_waitx به آن اشاره می‌کند، اجرا می‌شود. نمونه اولیه این تابع را به فایل syscall.c اضافه می‌کنیم.

```
extern int sys_waitx(void);
```

در فایل usys.s برای برقراری ارتباط با سیستم کال دستور زیر را اضافه می‌کنیم.

```
SYSCALL(waitx)
```

کد زیر را به فایل defs.h اضافه می‌کنیم.

```
int             waitx(int *, int *);
```

تابعی که توسط user program فراخوانی می‌شود را به user.h اضافه می‌کنیم.

```
int waitx(int *wtime, int *rtime);
```

در فایل sysproc.c تابع سیستم کال sys_waitx(void) را پیاده سازی می‌کنیم. این تابع آرگومان های سیستم کال سطح پایین تر که waitx است را چک می‌کند و اگر تعداد آرگومان ها با تعداد خواسته شده برابر نبود 1- بر می‌گرداند.

```c
int sys_waitx(void)
{
  int *rtime;
  int *wtime;
  if(argptr(0, (char**)&wtime, sizeof(int)) < 0)
    return -1;

  if(argptr(1, (char**)&rtime, sizeof(int)) < 0)
    return -1;

  return waitx(wtime, rtime);
}
```

در فایل proc.c تابع سطح پایین waitx را پیاده سازی می‌کنیم. این تابع تقریبا مشابه تابع wait پردازه های زامبی را پیدا کرده و منابعشان را آزاد می‌کند است با این تفاوت که زمان انتظار و زمان اجرای پردازه را محاسبه و بروزرسانی می‌کند.

```c
if(p->state == ZOMBIE) {
  *wtime = p->etime - p->stime - p->rtime - p->iotime;
  *rtime = p->rtime;
```

پیاده سازی سیستم کال در این مرحله به پایان می‌رسد.

برای تست کردن سیستم کال باید یک user program را مطابق مراحل زیر اضافه کنیم.

فایل test.c را ایجاد کرده و آن را در پوشه xv6-public ذخیره می‌کنیم.

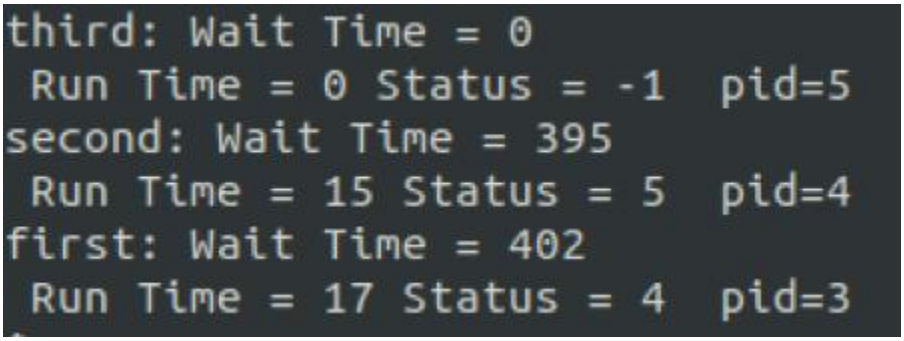
کد _test\ را به بخش UPROGS و \test.c را به بخش EXTRA فایل Makefile اضافه می‌کنیم. و با اجرای دو دستور make clean و make qemu سیستم عامل را باز می‌کنیم.

```
EXTRA=\
    mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
    test.c\
```

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _test\
```

```c
int main (int argc,char *argv[])
{
    int Status, a, b;
    int pid1=fork();
    if (pid1==0){
        int pid2=fork();
        sleep(300);
        if (pid2>0){
        Status=waitx(&a , &b);
        printf(1, "second: Wait Time = %d\n Run Time = %d Status = %d  pid=%d\n", a, b, Status,getpid());
        exit();
        }
        else{
            sleep(100);
            Status=waitx(&a,&b);
            printf(1, "third: Wait Time = %d\n Run Time = %d Status = %d  pid=%d\n", a, b, Status,getpid());
            exit();
        }
    }
    if (pid1>0){
        Status=waitx(&a , &b);
        printf(1, "first: Wait Time = %d\n Run Time = %d Status = %d  pid=%d\n", a, b, Status,getpid());
    }
  exit();
}
```

در این برنامه سه پردازه ایجاد می‌شوند که سومی فرزند دومی و دومی فرزند اولی است و اطلاعات مربوط به هرکدام توسط سیستم کال waitx استخراج می‌شود. و خروجی برنامه به شکل زیر است.



متغیر status نشان دهنده مقداری است که waitx برمی‌گرداند. از آنجا که پردازه سوم فرزندی ندارد waitx به آن 1- برمی‌گرداند و به پردازه دوم pid فرزند زامبی شده آن یعنی 5 و به پردازه اول pid فرزند زامبی شده آن یعنی 4 را برمی‌گرداند.

قسمت دوم پروژه:

قبل از ایجاد سیستم کال ها فیلد جدید priority را به struct proc در فایل proc.h اضافه می‌کنیم.

```
int stime;
int etime;
int rtime;
int iotime;
int priority;
```

در تابع allocproc در فایل proc.c مقدار پیش فرض priority را تعیین می‌کنیم.

```
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->priority    = 60;
  release(&ptable.lock);
```

باید دو سیستم کال set_priority و proc_stats را به سیستم اضافه کنیم که اولی برای اختصاص دادن اولویت به پردازه و دومی برای نمایش آمار همه پردازه ها استفاده می‌شود.

مطابق قسمت قبل مراحل زیر را طی می‌کنیم.

به فایل syscall.h دو خط زیر را اضافه می‌کنیم.

```
#define SYS_set_priority  23
#define SYS_proc_stats 24
```

به فایل syscall.c چهار خط زیر را اضافه می‌کنیم.

```
[SYS_set_priority]    sys_set_priority,
[SYS_proc_stats]    sys_proc_stats
```

```
 extern int sys_set_priority(void);
 extern int sys_proc_stats(void);
```

به فایل usys.s دو خط زیر را اضافه می‌کنیم.

```
SYSCALL(set_priority)
SYSCALL(proc_stats)
```

دو تابع زیر را به فایل sysproc.c اضافه می‌کنیم.

```
int
sys_set_priority(void)
{
    int new_priority, pid;
    if(argint(0, &new_priority) < 0)
        return -1;
    if(argint(1, &pid) < 0)
        return -1;
    return set_priority(new_priority, pid);
}

int
sys_proc_stats(void)
{
  return proc_stats();
}
```

دو خط زیر را به فایل defs.h اضافه می‌کنیم.

```
  int               proc_stats(void);
  int               set_priority(int,int);
```

دو خط زیر را به فایل user.h اضافه می‌کنیم.

```
int proc_stats(void);
int set_priority(int new_priority, int pid);
```

حالا دو تابع سطح پایین proc_stats و set_priority را در فایل proc.c پیاده سازی می‌کنیم.

تابع set_priority مقدار جدید اولویت و pid پردازه را ورودی می‌گیرد و pid را با تمام پردازه ها تطابق می‌دهد تا پردازه مورد نظر را پیدا کند. سپس اولویت جدید را جایگزین اولویت قبلی می‌کند و برای انجام دوباره Rescheduling ، yield را فراخوانی می‌کند و طبق خواسته سوال مقدار قبلی اولویت را برمی‌گرداند.

```
int set_priority(int new_priority, int pid) {
    struct proc *p;
    int old_prio = 0;
    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if(p->pid != pid){
            continue;
        }
  if( p->priority==0){
    p->priority=60;
  }
        old_prio = p->priority;
        p-> priority = new_priority;

    }
    release(&ptable.lock);
        yield();
    return old_prio;
}
```

```
int
proc_stats()
{
  struct proc *p;
  sti();
  acquire(&ptable.lock);
  cprintf("Name Pid State   Priority\n");
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->pid!=0)
    {
      if(p->state == RUNNABLE)
        cprintf("%s %d  RUNNABLE  %d\n",p->name,p->pid,p->priority);
      else if(p->state == RUNNING)
        cprintf("%s %d  RUNNING   %d\n",p->name,p->pid,p->priority);
      else if(p->state == SLEEPING)
        cprintf("%s %d  SLEEPING  %d\n",p->name,p->pid,p->priority);
    }
  }
  release(&ptable.lock);
  return 0;
}
```

تابع proc_stats روی لیست تمام پردازه ها حلقه می‌زند و اطلاعات آن ها اعم از نام، pid، وضعیت را چاپ می‌کند.

پیاده سازی سیستم کال ها در این مرحله به پایان می‌رسد.

برای تغییر روش زمانبندی از Round Robin به Priority scheduling تابع scheduler موجود در فایل proc.c را به شکل زیر تغییر می‌دهیم.

این تابع روی همه پردازه های قابل اجرا حلقه می‌زند و پردازه ای را انتخاب می‌کند که قابل اجرا باشد و بالاترین اولویت(کوچکترین مقدار) را داشته باشد. سپس منابع حافظه و cpu را به این پردازه اختصاص می‌دهد و آن را وارد حالت Running می‌کند.

```
struct proc *p, *p1;
struct cpu *c = mycpu();
c->proc = 0;
for(;;){
  sti();
  struct proc *highP;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
      continue;
    highP = p;
    for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
      if(p1->state != RUNNABLE)
        continue;
      if(highP->priority > p1->priority)
        highP = p1;
    }
    p = highP;
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    swtch(&(c->scheduler), p->context);
    switchkvm();
    c->proc = 0;
  }
  release(&ptable.lock);
}
```

برای تست کردن این قسمت دو user program به نام های stats و set_priority اضافه می‌کنیم که برای نمایش آمار همه پردازه ها و تغییر اولولیت پردازه ها استفاده می‌شوند.

```
$ stats
Name    Pid     State           Priority
init    1       SLEEPING        60
sh      2       SLEEPING        60
stats   5       RUNNING         60
$ set_priority 1 30
priority of pid 1 changed from 60 to 30
$ stats
Name    Pid     State           Priority
init    1       SLEEPING        30
sh      2       SLEEPING        60
stats   7       RUNNING         60
```

برای تست کردن scheduler جدید در فایل test.c تعدادی پردازه ایجاد می‌کنیم و مقادیر مختلفی برای اولویت آنها در نظر می‌گیریم.

```
$ test
Name    Pid     State           Priorit
init    1       SLEEPING        60
sh      2       SLEEPING        60
test    3       SLEEPING        100
test    4       RUNNABLE        60
test    5       RUNNING         40
test    6       RUNNABLE        90
test    7       RUNNABLE        80
test    9       RUNNABLE        60
test    10      RUNNING         50
test    11      RUNNABLE        60
process with pid 5 is done
Name    Pid     State           Priorit
init    1       SLEEPING        60
sh      2       SLEEPING        60
test    3       SLEEPING        100
test    4       RUNNING         50
test    6       RUNNABLE        90
test    7       RUNNABLE        80
test    8       RUNNABLE        60
test    9       RUNNABLE        60
test    10      RUNNING         50
test    11      RUNNABLE        60
process with pid 4 is done
```

قسمت سوم پروژه:

فیلد جدید queue_level را به struct proc در فایل proc.h اضافه می‌کنیم.

```
int stime;
int etime;
int rtime;
int iotime;
int priority;
int queue_level;
```

برای قرار دادن هر پردازه در صف از سیستم کال set_level استفاده می‌کنیم که عینا مشابه با set_priority پیاده سازی می‌شود و تنها تفاوت آن در تابع set_level در فایل proc.c است.

```
int set_level(int level, int pid) {
  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->pid != pid){
      continue;
    }
    p->queue_level=level;
  }
  release(&ptable.lock);
// yield();
  return level;
}
```

در فایل proc.c تابعی به نام queue تعریف می‌کنیم که خالی نبودن صف ها به ترتیب اولویتشان را بررسی می‌کند چرا که تنها اگر صف اول خالی باشد پردازه های

صف دوم می‌توانند اجرا شوند و تنها اگر صف دوم خالی باشد پردازه های صف سوم می‌توانند اجرا شوند.

```
int queue(void) {
  struct proc *p;
  for(p=ptable.proc; p<&ptable.proc[NPROC]; p++) {
    if(p->queue_level == 1 && p->state == RUNNABLE)
      return 1;
    if(p->queue_level == 2 && p->state == RUNNABLE)
      return 2;
    if(p->queue_level == 3 && p->state == RUNNABLE)
      return 3;
  }
  return 0;
}
```

در مرحله بعدی تابع scheduler در فایل proc.c را به گونه زیر بازنویسی می‌کنیم.

تابع تمامی پردازه ها را بررسی می‌کند اگر در صف اول پردازه قابل اجرایی وجود داشت پردازه ای را به روش guaranteed scheduling انتخاب کرده و اجرا می‌کند.

اگر صف اول خالی بود صف دوم را بررسی کرده و پردازه ای را به روش Round Robin انتخاب کرده و اجرا می‌کند و اگر صف دوم خالی بود با صف سوم هم مشابه صف دوم رفتار خواهد شد.