Student Name: Chen-yang Yu (862052273)Po-Chang Kuo (862029279)

Part1:

In order to modify the existed system call or add a new one, we have to modify the following file:

In defs.h:

Change the declaration

```c
//PAGEBREAK: 16
// proc.c
int             cpuid(void);
//void          exit(void); OS153_lab1
void            exit(int status);
int             fork(void);
int             growproc(int);
```

...

```c
void            userinit(void);
//int           wait(void); OS153_lab1
int             wait(int *status);
void            wakeup(void*);
void            yield(void);
int             waitpid(int pid, int *status, int options);//OS153_lab1
void            SetPriority(int priority);//OS153_lab1
```

In proc.h

The declaration of process state

Add the status to save the exit status when process exit

Add waitpid_pid[ ] and waitpid_count to save if there is any process waiting for me

Add priority for scheduling

```c
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
```

...

```c
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int status;                  // exit status OS153_lab1
  int waitpid_pid[64];         // pid of process who is waiting for you
  int waitpid_count;           // how many process waiting for you
  int priority;                // process priority OS153_lab1
};
```

In sysproc.c

The real implement of system call method (use argxxx() to pass argument)

```c
int sys_SetPriority()
{
  //OS153_lab1
  int priority;
  if (argint(0,&priority) < 0)
    return -1;
  else
    SetPriority(priority);
  return 0;  // not reached
}

int sys_waitpid()
{
  int pid;
  int * status;
  int options;
  if( argint(0,&pid) <  0)
        return -1;
  else if( argptr(1,(char **) &status, sizeof(int*)) < 0)
        return -1;
  else if( argint(2,&options) <0 )
        return -1;
  else
        return waitpid(pid, status, options);

}
int
```

```c
int
sys_exit()
{
//exit(); OS153_lab1
  int status;
  if (argint(0,&status) < 0)
    return -1;
  else
    exit(status);
  return 0;  // not reached
}

int
sys_wait(void)
{
  int* status;
  if( argptr(0,(char **) &status, sizeof(int*)) < 0)
      return -1;
  else
      return wait(status);
}
```

In syscall.c

Extern define the function that connect the shell and the kernel

(the position is defined in the syscall.h)

Use the position 22 to add the function call in the system call vector

```
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_waitpid(void);
extern int sys_SetPriority(void);
//OS153_lab1
```

```
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_waitpid] sys_waitpid,
[SYS_SetPriority] sys_SetPriority,
};
//OS153_lab1
```

In syscall.h

define the position of system call vector that connect your implement

Define the system call vector (#define SYS_waitpid 22) #define SYS_SetPriority 23

```
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
//OS153_lab1
#define SYS_waitpid 22
#define SYS_SetPriority 23
```

In user.h

define the function that can be called through the shell

```
// system calls
int fork(void);
//int exit(void) __attribute__((noreturn));
int exit(int status);
//int wait(void); OS153_lab1
int wait(int *status);
int pipe(int*);
```

```
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
//OS153_lab1
int waitpid(int pid, int *status, int options);
void SetPriority(int priority);
```

In usys.S

use the macro to define the connect the call of user to the system call function

```
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(waitpid)
SYSCALL(SetPriority)
```

a) Change the exit system call signature to void exit(int status).

In proc.c

Change the implement of exit() here:

Add curproc->status = status to save the status when a process exit

```
// Exit the current process.  Does not return.
// An exited process remains in the zombie state
// until its parent calls wait() to find out it exited.
//void exit(void); OS153_lab1
void
exit(int status)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;
  curproc->status=status; //OS153_lab1
  if(curproc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }
```

Other user space changing:

In echo.c/ cat.c/ forktest.c/ grep.c/ init.c/ kill.c/ ln.c/ rm.c/ sh.c/ trap.c/ usertest.c/ wc.c

If the exit ends abnormally(not enough argument, naming error...), we set it exit(1). Exit(0) if exit normally.

b) Update the wait system call signature to int wait(int *status).

In proc.c

Add * status = p->status (or *status = -1, if we can't find a child)

```
  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        if(status)
            *status=p->status;
        release(&ptable.lock);
        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || curproc->killed){
      if(status)
        *status = -1;
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
```

Other user space changing:

forktest.c/ init.c/ sh.c/ stressfs.c/ usertest.c

(to wait(&status))

c) Add a waitpid system call:

In proc.c

The behavior is the same as wait(), except we wait for the input argument pid instead of the child process. We add the caller of waitpid() into a list of the input argument process (if this process exist), this process will wake up the caller (the process which is waiting).

```
//OS153_lab1
int waitpid(int pid, int *status, int options){

  struct proc *p;
  int Flag_found;
  struct proc *curproc = myproc();
  //cprintf("wait for pid: %d\n",pid);
  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited p = pid.
    Flag_found = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid != pid)
      continue;
    Flag_found = 1; // no continue
    if(p->state == ZOMBIE){
          // Found one.
          pid = p->pid;
          kfree(p->kstack);
          p->kstack = 0;
          freevm(p->pgdir);
          p->pid = 0;
          p->parent = 0;
          p->name[0] = 0;
          p->killed = 0;
          p->state = UNUSED;
      if(status){
            *status=p->status;
        cprintf("got a zombie pid: %d, status=%d\n",pid, p->status);
      }
          release(&ptable.lock);
      //cprintf("got a zombie pid: %d, status=%d , status=%d\n",pid,status, p->status);
          return pid;
        }else{ //found p=pid, but RUNNING
      cprintf("Found!! but running\n");
      p->waitpid_pid[p->waitpid_count]=curproc->pid;
      p->waitpid_count++;
      break;
    }// if p = pid is still alive, add current process as a fake parent
    // caller process might sleep
    // in this case, this p = pid has to konw who to wake up
    // add extra wakeup in exit()
    // No point waiting if we don't have any children.
    }
    if(!Flag_found || curproc->killed){
        if(status)
      *status = -1;
        release(&ptable.lock);
    cprintf("not found pid: %d, status=-1\n",pid);
        return -1;
    }
    cprintf("go to sleep\n");
    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }// for(;;)
}
```

We also have to modify the code in exit(). Since the caller of waitpid may sleep to wait the specific pid. When one process exit, it needs to check if there is any process waiting for itself (other than its parent). Wake it up if there is any.

```
  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);

  // OS153_lab1
  // below part is to check if any one call waitpid() and wait the caller
  // of exit() to termonate
  if(curproc->waitpid_count!=0){
      int i;
      for(i=0; i< curproc->waitpid_count;i++){
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
          if(p->pid==curproc->waitpid_pid[i])
            wakeup1(p);
        }
      }
  }

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }
```

In order to test what we implement, we use ./lab1 as following

The parent get the right exit status:

```
$ lab1 1

 This program tests the correctness of your lab#1

  Step 1: testing exit(int status) and wait(int* status):

This is child with PID# 44 and I will exit with status 0

 This is the parent: child with PID# 44 has exited with status 0

This is child with PID# 45 and I will exit with status -1

 This is the parent: child with PID# 45 has exited with status -1
$
```

The parent wait the child and get the exit status:

```
yang@yang-VirtualBox: /media/sf_VB_share/xv6
$
$ lab1 2

 This program tests the correctness of your lab#1

  Step 2: testing waitpid(int pid, int* status, int options):

 The is child with PID# 32 and I will exit with status 0

 The is child with PID# 33 and I will exit with status 0

 The is child with PID# 34

 The is child with PID# 36 and I will exit with status  and I will exit wit
h status 0
The is child 0
with PID# 35 and I will exit with status 0

 This is the parent: Now waiting for child with PID# 35
got a zombie pid: 35, status=0

 This is the partent: Child# 35 has exited with status 0

 This is the parent: Now waiting for child with PID# 33
got a zombie pid: 33, status=0

 This is the partent: Child# 33 has exited with status 0

 This is the parent: Now waiting for child with PID# 34
got a zombie pid: 34, status=0

 This is the partent: Child# 34 has exited with status 0

 This is the parent: Now waiting for child with PID# 32
got a zombie pid: 32, status=0

 This is the partent: Child# 32 has exited with status 0

 This is the parent: Now waiting for child with PID# 36
got a zombie pid: 36, status=0

 This is the partent: Child# 36 has exited with status 0
$
```

Part2:

Add a priority value to each process, and choose the highest priority in the ready queue. First we add a system call to assign the priority value:

In proc.c

```
//OS153_lab1
//go through the patble, find my_process and set the priority
void SetPriority(int priority)
{
  struct proc * p;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == curproc->pid){
#if debugFlag
      cprintf("found pid and set priority\n");
#endif
      if(priority<=63 && priority>=0){
        p->priority=priority;
        //p->state = RUNNABLE;
#if debugFlag
        cprintf("set %d\n",priority);
#endif
        break;
      }else{
        p->priority=63;
#if debugFlag
        cprintf("SET63\n");
#endif
        break;
      }
    }else{
      continue;
    }
  }//for
  release(&ptable.lock);
  //enable interrupts
  yield();
  //exit(0);

}
```

Modify the schedule method in scheduling()

(Using a compile flag to separate from original scheduling())

The new scheduling follows the original one: check the ptable and decide the next RUNNABLE process. We modify it as: find the highest priority RUNNABLE process (zero is the highest and 63 is the lowest), and check the ptable if there is any process with higher or equal priority.

We use hpriority() to go through the ptable and return the highest priority

```
#define MAX(x, y) (((x) > (y)) ? (x) : (y))
#define MIN(x, y) (((x) < (y)) ? (x) : (y))
#define swap(x,y) { x = x + y; y = x - y; x = x - y; }

//OS153_lab1
//go through the ptable and find the highest prioriry (lowest value)
int hpriority(void){
    int toppriority = 63;
    struct proc * p;
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if(p->state != RUNNABLE)
            continue;
        toppriority = MIN(toppriority, p->priority);
        //compare each and choose the minimal
    }
    release(&ptable.lock);

    return toppriority;
}
```

The schelder():

```c
for(;;){
    // Enable interrupts on this processor.
    sti();

    hpriorityValue = hpriority();
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        //procdump();
        if(p->state != RUNNABLE)
            continue;
        if(p->priority > hpriorityValue)
            continue;
        // Switch to chosen process.  It is the process's job
        // to release ptable.lock and then reacquire it
        // before jumping back to us.
        //cprintf("hp = %d\n",hpriorityValue);
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;

        //we need this to track the actual running time
        upTime = ticks;

        swtch(&(c->scheduler), p->context);
        switchkvm();

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
        //release(&ptable.lock);
        //break;

    }
    release(&ptable.lock);
}
```

In order to test what we implement, we use lab1.c as following, the highest priority process ends first as we expect

```
yang@yang-VirtualBox: /media/sf_VB_share/xv6
call:

 Step 2: Assuming that the priorities range between range between 0 to 63

 Step 2: 0 is the highest priority. All processes have a default priority of 20

 Step 2: The parent processes will switch to priority 0

Hello! this is child# 4 and I will change my priority to 60

Hello! this is child# 5 and I will change my priority to 40

Hello! this is child# 6 and I will change my priority to 20

child# 6 with priority 20 has finished!
[6]     Start time:      230
        End time:       395
        Turnaround time: 165
        Running time: 165
        Waiting time: 0

 This is the parent: child with PID# 6 has finished with status 0

 child# 5 with priority 40 has finished!
[5]     Start time:      220
        End time:       551
        Turnaround time: 331
        Running time: 163
        Waiting time: 168

 This is the parent: child with PID# 5 has finished with status 0

 child# 4 with priority 60 has finished!
[4]     Start time:      208
        End time:       726
        Turnaround time: 518
        Running time: 183
        Waiting time: 335

 This is the parent: child with PID# 4 has finished with status 0

 if processes with highest priority finished first then its correct
[3]     Start time:      200
        End time:       727
        Turnaround time: 527
        Running time: 9
        Waiting time: 518
$
```

Explain how you would implement priority donation/priority inheritance, or for 5% bonus implement it. (Bonus 1)

We need the priority donation/inheritance when priority inversion (the lower priority task blocks the higher priority task due to resource locked). By donate the priority to the resource holder process, we can keep the priority scheduling as much as we can.

Add a check field in acquire(): If any process attempts to access the critical section and finds out that the lock is hold by a lower priority process, donate the priority to the lock holder process

```
void acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");
  Flag = 0;
  while(xchg(&lk->locked, 1) != 0){
        if( lk->holder->priority < myproc()->priority ){
                Flag = 1; // donate
                donatePriority(lk->holder)
        }
}

  // Record info about lock acquisition for debugging.
  lk->cpu = cpu;
  lk->proc = proc;
  getcallerpcs(&lk, lk->pcs);
}
```

And give the priority back when release()

```
void release(struct spinlock *lk){
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;
  lk->proc = 0;
  if(Flag)
        returnPriority();
  xchg(&lk->locked, 0);

  popcli();
}
```

track the scheduling performance of each process. These values should allow you to compute the turnaround time and wait time for each process. Add a system call to extract these values or alternatively print them out when the process exits. 5\% bonus (Bonus 2)

Turnaround time: Simple use time tick at the beginning and the end of process

Waiting time: Turnaround time - run time

In proc.h

We add startTime and endTime to calculate the turnaround time. Add run_time to accumulate the running time of the process in order to calculate the wait time

```
  char name[16];               // Process name (debugging)
  int status;                   // exit status OS153_lab1
  int waitpid_pid[64];          // pid of process who is waiting for you
  int waitpid_count;            // how many process waiting for you
  int priority;                 // process priority OS153_lab1
  int startTime;                //
  int endTime;                  //
  int turnAroundTime;           //
  int runTime;                  // wait time = turnaroundtime-runtime
};

//OS153_lab1
//when sched:          p->runtime += ticks - upTime;
//when fork:           proc->runtime = 0 //init
//when exit:           proc->runtime += proc->endtime - upTime;
//when scheduler       upTime = ticks;

//sched: back to scheduler, means end of exec, calculate the time from
//last time we call scheduler

//exit: calculate the time from last time we call scheduler
//(scheduler -> exit)

//fork and put this process into ready (not exec yet)
int upTime;
```

In proc.c

In fork()

```
  acquire(&ptable.lock);

  np->state = RUNNABLE;
  np->startTime = ticks;

  np->runTime = 0;
#if debugFlag
  if(upTime>np->startTime)
        cprintf("\t>\n");
  else if(upTime<np->startTime)
        cprintf("\t<\n");
  else
        cprintf("\t=\n");
  //np->runTime += upTime - np->startTime;
  np->runTime += np->startTime - upTime;
  upTime = np->startTime;
#endif

  release(&ptable.lock);
```

In scheduler()

```
//OS153_lab1
//we need this to track the actual running time
upTime = ticks;
```

In sched()

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  //OS153_lab1
  p->runTime += ticks - upTime;
```

In exit()

```c
    //OS153_lab1: calculate the endtime and accumulate the last part of running time
    curproc->endTime = ticks;
    curproc->turnAroundTime = curproc->endTime - curproc->startTime;
    curproc->runTime += curproc->endTime - upTime;

    cprintf("[%d]\tStart time:      %d\n", curproc->pid, curproc->startTime);
    cprintf("\tEnd time:      %d\n", curproc->endTime);
    cprintf("\tTurnaround time: %d\n", curproc->turnAroundTime);
    cprintf("\tRunning time: %d\n", curproc->runTime);
    cprintf("\tWaiting time: %d\n", curproc->turnAroundTime - curproc->runTime);

    // Jump into the scheduler, never to return.
    curproc->state = ZOMBIE;
    sched();
    panic("zombie exit");
  }
```

Result:

Output turnaround time and waiting time when process exit();

```
 child# 6 with priority 20 has finished!
[6]     Start time:      230
        End time:      395
        Turnaround time: 165
        Running time: 165
        Waiting time: 0

 This is the parent: child with PID# 6 has finished with status 0

 child# 5 with priority 40 has finished!
[5]     Start time:      220
        End time:      551
        Turnaround time: 331
        Running time: 163
        Waiting time: 168

 This is the parent: child with PID# 5 has finished with status 0

 child# 4 with priority 60 has finished!
[4]     Start time:      208
        End time:      726
        Turnaround time: 518
        Running time: 183
        Waiting time: 335

 This is the parent: child with PID# 4 has finished with status 0

 if processes with highest priority finished first then its correct
[3]     Start time:      200
        End time:      727
        Turnaround time: 527
        Running time: 9
        Waiting time: 518
```