Code Modification

```
## Makefile

(-)CS333_PROJECT ?= 1
(+)CS333_PROJECT ?= 2

## uproc.h

(+)#include "types.h"

## defs.h

(+)#include "uproc.h"

// proc.c
int             cpuid(void);
void            exit(void);
int             fork(void);
int             growproc(int);
int             kill(int);
struct cpu*     mycpu(void);
struct proc*    myproc();
void            pinit(void);
void            procdump(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            setproc(struct proc*);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(void);
void            wakeup(void*);
void            yield(void);
(+)#ifdef CS333_P2
(+)int              getprocs(uint max, struct uproc* upTable);
(+)#endif
#ifdef CS333_P3
void            printFreeList(void);
void            printList(int);
void            printListStats(void);
#endif // CS333_P3

## proc.c

(+)#ifdef CS333_P2
(+)  #include "uproc.h"
(+)#endif

static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);
  int found = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED) {
      found = 1;
      break;
    }
  if (!found) {
    release(&ptable.lock);
```

```
    return 0;
  }
  p->state = EMBRYO;
  p->pid = nextpid++;
  release(&ptable.lock);

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;

  #ifdef CS333_P1
    p->start_ticks = ticks;
  #endif // CS333_P1

  (+)#ifdef CS333_P2
  (+)  p->cpu_ticks_total = 0;
  (+)  p->cpu_ticks_in = 0;
  (+)#endif // CS333_P2

  return p;
}

void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();

  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

  (+)#ifdef CS333_P2
  (+)  p->uid = DEFAULT_UID;
  (+) p->gid = DEFAULT_GID;
```

```
  (+)#endif

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

  // this assignment to p->state lets other cores
  // run this process. the acquire forces the above
  // writes to be visible, and the lock is also needed
  // because the assignment might not be atomic.
  acquire(&ptable.lock);
  p->state = RUNNABLE;
  release(&ptable.lock);
}

int
fork(void)
{
  int i;
  uint pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  (+)#ifdef CS333_P2
  (+)  np->uid = curproc->uid;
  (+)  np->gid = curproc->gid;
  (+)#endif

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);
  np->state = RUNNABLE;
  release(&ptable.lock);

  return pid;
}

void
scheduler(void)
```

```
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  #ifdef PDX_XV6
    int idle;  // for checking if processor is idle
  #endif // PDX_XV6

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

    #ifdef PDX_XV6
      idle = 1;  // assume idle unless we schedule a process
    #endif // PDX_XV6
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      #ifdef PDX_XV6
        idle = 0;  // not idle this timeslice
      #endif // PDX_XV6
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      (+)#ifdef CS333_P2
      (+)  p->cpu_ticks_in = ticks;
      (+)#endif // CS333_P2
      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);

    #ifdef PDX_XV6
      // if idle, wait for next interrupt
      if (idle) {
        sti();
        hlt();
      }
    #endif // PDX_XV6
  }
}

(+)#ifdef CS333_P2
(+)int
(+)getprocs(uint max, struct uproc* upTable){
(+)  struct proc* p;
(+)  int procsNumber = 0;
(+)  acquire(&ptable.lock);
(+)  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
(+)    if (procsNumber < max) {
(+)      if(p->state != UNUSED && p->state != EMBRYO) {
(+)        if(p->state >= 0 && p->state < NELEM(states) && states[p->state]){
(+)          safestrcpy(upTable[procsNumber].state, states[p->state],STRMAX);
(+)        } else {
```

```
(+)         safestrcpy(upTable[procsNumber].state,"???",STRMAX);
(+)       }
(+)       upTable[procsNumber].pid = p->pid;
(+)       upTable[procsNumber].uid = p->uid;
(+)       upTable[procsNumber].gid = p->gid;
(+)       upTable[procsNumber].ppid = p->parent ? p->parent->pid : p->pid;
(+)       upTable[procsNumber].elapsed_ticks = ticks - p->start_ticks;
(+)       upTable[procsNumber].CPU_total_ticks = p->cpu_ticks_total;
(+)       upTable[procsNumber].size = p->sz;
(+)       safestrcpy(upTable[procsNumber].name, p->name, STRMAX);
(+)       procsNumber++;
(+)     }
(+)   } else {
(+)     break;
(+)   }
(+) }
(+) release(&ptable.lock);
(+) return procsNumber;
(+)}
(+)#endif // CS333_P2
```

proc.h

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  uint pid;                    // Process ID
  struct proc *parent;         // Parent process. NULL indicates no parent
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  uint start_ticks;
  (+)uint uid;
  (+)uint gid;
  (+)uint cpu_ticks_total;
  (+)uint cpu_ticks_in;
};
```

ps.c

```
(+)#ifdef CS333_P2
(+)#include "types.h"
(+)#include "user.h"
(+)#include "uproc.h"

(+)#define MAX 16

(+)int
(+)main(void)
(+){
(+)  struct uproc *proc = malloc(sizeof(struct uproc)*MAX);
(+) int procsNumber = getprocs(MAX, proc);
(+)  printf(1,"PID\tName\t\tUID\tGID\tPPID\tElapsed\tCPU\tState\tSize\n");

(+)  int i;
(+)  for(i = 0; i<procsNumber; i++){
(+)    struct uproc currentProc = proc[i];
(+)    uint elapsedTicks = currentProc.elapsed_ticks;
```

```
(+)    uint elapsedTicksSecond = elapsedTicks/1000;
(+)    uint elapsedTicksMs = elapsedTicks%1000;
(+)    char* zeros = "";
(+)    uint cpuTotalTicks = currentProc.CPU_total_ticks;
(+)    uint cpuTotalTicksSecond = cpuTotalTicks/1000;
(+)   uint cpuTotalTicksMs = cpuTotalTicks % 1000;
(+)    char* cpuZeros = "";

(+)    if (elapsedTicksMs < 10) {
(+)      zeros = "00";
(+)    } else if (elapsedTicksMs < 100) {
(+)      zeros = "0";
(+)    }

(+)    if(cpuTotalTicksMs < 10){
(+)      cpuZeros = "00";
(+)    } else if (cpuTotalTicksMs < 100) {
(+)      cpuZeros = "0";
(+)    }

(+)    printf(
(+)      1,
(+)      "%d\t%s\t\t%d\t%d\t%d\t%d.%s%d\t%d.%s%d\t%s\t%d\n",
(+)      currentProc.pid,
(+)      currentProc.name,
(+)      currentProc.uid,
(+)      currentProc.gid,
(+)      currentProc.ppid,
(+)      elapsedTicksSecond,
(+)      zeros,
(+)      elapsedTicksMs,
(+)      cpuTotalTicksSecond,
(+)      cpuZeros,
(+)      cpuTotalTicksMs,
(+)      currentProc.state,
(+)      currentProc.size
(+)    );
(+)  }

(+)  free(proc);
(+)  exit();
(+)}
(+)#endif
```

syscall.c

```
extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
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);
#ifdef PDX_XV6
extern int sys_halt(void);
#endif // PDX_XV6
#ifdef CS333_P1
extern int sys_date(void);
#endif //CS333_P1
(+)#ifdef CS333_P2
(+)extern int sys_getuid(void);
(+)extern int sys_getgid(void);
(+)extern int sys_getppid(void);
(+)extern int sys_setuid(void);
(+)extern int sys_setgid(void);
(+)extern int sys_getprocs(void);
(+)#endif //CS333_P2

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
#ifdef PDX_XV6
[SYS_halt]    sys_halt,
#endif // PDX_XV6
#ifdef CS333_P1
[SYS_date]    sys_date,
#endif // CS333_P1
(+)#ifdef CS333_P2
(+)[SYS_getuid]    sys_getuid,
(+)[SYS_getgid]    sys_getgid,
(+)[SYS_getppid]   sys_getppid,
(+)[SYS_setuid]    sys_setuid,
(+)[SYS_setgid]    sys_setgid,
(+)[SYS_getprocs]  sys_getprocs,
(+)#endif // CS333_P2
};

## syscall.h

#define SYS_fork        1
#define SYS_exit        SYS_fork+1
#define SYS_wait        SYS_exit+1
#define SYS_pipe        SYS_wait+1
#define SYS_read        SYS_pipe+1
#define SYS_kill        SYS_read+1
```

```
#define SYS_exec        SYS_kill+1
#define SYS_fstat       SYS_exec+1
#define SYS_chdir       SYS_fstat+1
#define SYS_dup         SYS_chdir+1
#define SYS_getpid      SYS_dup+1
#define SYS_sbrk        SYS_getpid+1
#define SYS_sleep       SYS_sbrk+1
#define SYS_uptime      SYS_sleep+1
#define SYS_open        SYS_uptime+1
#define SYS_write       SYS_open+1
#define SYS_mknod       SYS_write+1
#define SYS_unlink      SYS_mknod+1
#define SYS_link        SYS_unlink+1
#define SYS_mkdir       SYS_link+1
#define SYS_close       SYS_mkdir+1
#define SYS_halt        SYS_close+1
#define SYS_date        SYS_halt+1
(+)#define SYS_getuid   SYS_date+1
(+)#define SYS_getgid   SYS_getuid+1
(+)#define SYS_getppid  SYS_getgid+1
(+)#define SYS_setuid      SYS_getppid+1
(+)#define SYS_setgid      SYS_setuid+1
(+)#define SYS_getprocs    SYS_setgid+1

## testsetuid.c

(+)#ifdef CS333_P2
(+)#include "types.h"
(+)#include "user.h"

(+)int
(+)main(int argc, char *argv[])
(+){
(+)  printf(1, "***** In %s: my uid is %d\n\n", argv[0], getuid());
(+)  exit();
(+)}
(+)#endif

## user.h

// ulib.c
int stat(char*, struct stat*);
char* strcpy(char*, char*);
void *memmove(void*, void*, int);
char* strchr(const char*, char c);
int strcmp(const char*, const char*);
void printf(int, char*, ...);
char* gets(char*, int max);
uint strlen(char*);
void* memset(void*, int, uint);
void* malloc(uint);
void free(void*);
int atoi(const char*);
#ifdef PDX_XV6
int atoo(const char*);
int strncmp(const char*, const char*, uint);
#endif // PDX_XV6

(+)#ifdef CS333_P2
(+)uint getuid(void);
(+)uint getgid(void);
(+)uint getppid(void);

(+)int setuid(uint);
```

```
(+)int setgid(uint);

(+)int getprocs(uint max, struct uproc* table);
(+)#endif

## usys.S

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(halt)
SYSCALL(date)
(+)SYSCALL(getuid)
(+)SYSCALL(getgid)
(+)SYSCALL(getppid)
(+)SYSCALL(setuid)
(+)SYSCALL(setgid)
(+)SYSCALL(getprocs)

## sysproc.c

#include "types.h"
#include "x86.h"
#include "defs.h"
#include "date.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#ifdef PDX_XV6
  #include "pdx-kernel.h"
#endif // PDX_XV6

(+)#ifdef CS333_P2
(+)  #include "uproc.h"
(+)#endif  // CS333_P2

(+)#ifdef CS333_P2
(+)int
(+)sys_getuid(void)
(+){
(+)  struct proc *curproc = myproc();
(+)  return curproc->uid;
(+)}

(+)int
```

```
(+)sys_getgid(void)
(+){
(+) struct proc *curproc = myproc();
(+) return curproc->gid;
(+)}

(+)int
(+)sys_getppid(void)
(+){
(+)  struct proc *curproc = myproc();
(+)  struct proc *parent = curproc->parent;
(+)  return parent != NULL ? parent->pid : 0;
(+)}

(+)int sys_setuid(void)
(+){
(+)  uint uid;
(+)  struct proc *curproc = myproc();

(+)  if(argint(0, (int*)&uid) >= 0) {
(+)    if(uid >= 0 && uid <= 32767) {
(+)      curproc->uid = uid;
(+)     return 0;
(+)   }
(+)  }

(+)  return -1;
(+)}

(+)int sys_setgid(void)
(+){
(+)  uint gid;
(+)  struct proc *curproc = myproc();

(+)  if(argint(0, (int*)&gid) >= 0) {
(+)    if(gid >= 0 && gid <= 32767) {
(+)      curproc->gid = gid;
(+)      return 0;
(+)    }
(+)  }

(+)  return -1;
(+)}

(+)int sys_getprocs(void)
(+){
(+)  uint max;
(+)  struct uproc* proc;

(+)  if (argint(0,(int*)&max) >= 0) {
(+)    if (max == 1 || max == 16 || max == 64 || max == 72) {
(+)      if (argptr(1, (void*)&proc, sizeof(struct uproc)) >= 0) {
(+)        return getprocs(max, proc);
(+)      }
(+)    }
(+)  }

(+)  return -1;
(+)}
(+)#endif //CS333_P2
```