MakeFile

```
CS333_PROJECT ?= 2
PRINT_SYSCALLS ?= 0
CS333_CFLAGS ?= -DPDX_XV6
ifeq ($(CS333_CFLAGS), -DPDX_XV6)
CS333_UPROGS += _halt _uptime
endif
```

```
ifeq ($(CS333_PROJECT), 2)
CS333_CFLAGS += -DCS333_P1 -DUSE_BUILTINS -DCS333_P2
CS333_UPROGS += _date _time _ps
CS333_TPROGS += _testsetuid  _testuidgid _p2-test
endif
```

defs.h

```
#ifdef CS333_P2
#include "uproc.h"
#endif
struct buf;
struct context;
struct file;
struct inode;
struct pipe;
struct proc;
```

```
// 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
```

proc.c

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "x86.h"
#include "proc.h"
#include "spinlock.h"

#ifdef CS333_P2
#include "uproc.h"
#endif
```

```
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;

  p->start_ticks = ticks;

  #ifdef CS333_p2
    p->cpu_ticks_total = 0;
    p->cpu_ticks-in = 0;
  #endif
  return p;
}
```

```
  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
```

```
// 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
```

```
// 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();
```

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

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
#ifdef CS333_P2
     p->cpu_ticks_total += (ticks - p->cpu_ticks_in);
#endif // CS333_P2
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

```
#if defined(CS333_P2)
void
procdumpP2P3P4(struct proc *p, char *state_string)
{
 cprintf("TODO for Project 2, delete this line and implement procdumpP2P3P4() in
proc.c to print a row\n");
 return;
 uint elapsed = ticks-p->start_ticks;
  uint elapsedLeft = (elapsed) / 1000;
  uint elapsedRight = elapsed % 1000;
  char *zeros = "";
  char *cpuZeros = "";
  uint cpuTicksTotal = p->cpu_ticks_total;
  uint cpuSecond = cpuTicksTotal / 1000;
  uint cpuMs = cpuTicksTotal % 1000;
  uint ppid = p->parent ? p->parent->pid : p->pid;

  if (elapsedRight < 10) {
   zeros = "00";
  } else if (elapsedRight < 100) {
    zeros = "0";
  }
```

```
  if (cpuMs < 10) {
    cpuZeros = "00";
  } else if (cpuMs < 100) {
    cpuZeros = "0";
  }

  cprintf(
    "\n%d\t%s\t%d\t%d\t%d\t%d.%s%d\t%d.%s%d\t%s\t%d\t",
    p->pid,
    p->name,
    p->gid,
    ppid,
    elapsedLeft,
    zeros,
    elapsedRight,
    cpuSecond,
    cpuZeros,
   cpuMs,
    state_string,
    p->sz
    );
}
```

```
#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
```

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)
```

user.h

```
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
```

uproc.h

```
#ifdef CS333_P2
#include "types.h"

struct uproc {
  uint pid;
  uint uid;
  uint gid;
  uint ppid;
#ifdef CS333_P4
  uint priority;
#endif // CS333_P4
  uint elapsed_ticks;
  uint CPU_total_ticks;
  char state[STRMAX];
  uint size;
  char name[STRMAX];
};
#endif
#endif
```

time.c

```c
#ifdef CS333_P2
#include "types.h"
#include "user.h"

int main(int argc, char *argv[]) {
  if(argc == 1) {
    printf(1, "(null) ran in 0.00\n");
  } else {
    int start = uptime();
    int pid = fork();

    if (pid > 0) {
      pid = wait();
    } else if (pid == 0) {
      exec(argv[1], argv+1);
      printf(1, "ERROR: Unknown Command\n");
      kill(getppid());
      exit();
    } else {
      printf(1, "ERROR: Fork error return -1\n");
    }

   int end = uptime();
    int timelapse = end - start;
    int seconds = timelapse/1000;
    int ms = timelapse%1000;
   char *msZeros = "";

    if (ms < 10) {
      msZeros = "00";
    } else if (ms < 100) {
      msZeros = "0";
    }

    printf(
      1,
      "%s ran in %d.%s%d\n",
      argv[1],
      seconds,
      msZeros,
      ms
    );
  }
```

```
  exit();
}
#endif
```

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
```

sysproc.c

```
#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
```

```
sys_date ( void )
{
  struct rtcdate *d ;
  if (argptr ( 0 ,( void*)&d , sizeof ( struct rtcdate)) < 0)
    return -1;
```

```
  cmostime(d);
  return 0;
}

#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
```

proc.h

```
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)

  #ifdef CS333_P2
    uint uid;
    uint gid;
    uint cpu_ticks_total;
    uint cpu_ticks_in;
  #endif
};
```

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

```
[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
```

```
};
```