```
.ycm_extra_conf.py | 132 ++++++++++
Testing.txt        |  73 ++++++
c/create.c         | 201 +++++++++------
c/ctsw.c           | 174 ++++++-------
c/di_calls.c       | 109 ++++++++
c/disp.c           | 736 +++++++++++++++++++++++++++++++++++------------------
c/init.c           | 179 +++++++++----
c/kbd.c            | 330 ++++++++++++++++++++++++
c/mem.c            | 365 ++++++++++++++++++++++----
c/msg.c            | 121 ++++++++-
c/signal.c         |  39 ++-
c/sleep.c          | 209 ++++++---------
c/syscall.c        | 223 +++++++++++++---
c/user.c           | 485 +++++++++++++++++------------------------
compile/Makefile   |   4 +
h/i386.h           |   2 +
h/kbd.h            |  75 ++++++
h/xeroskernel.h    | 309 +++++++++++++---------
18 files changed, 2692 insertions(+), 1074 deletions(-)
```

```
import os
import ycm_core


# These are the compilation flags that will be used in case there's no
# compilation database set (by default, one is not set).
# CHANGE THIS LIST OF FLAGS. YES, THIS IS THE DROID YOU HAVE BEEN LOOKING FOR.
flags = [
'-Wall',
'-Wstrict-prototypes',
'-I',
'./h',
'./compile'
'./lib'
'./boot'
# You 100% do NOT need -DUSE_CLANG_COMPLETER in your flags; only the YCM
# source code needs it.
'-DUSE_CLANG_COMPLETER',
# THIS IS IMPORTANT! Without a "-std=<something>" flag, clang won't know which
# language to use when compiling headers. So it will guess. Badly. So C++
# headers will be compiled as C headers. You don't want that so ALWAYS specify
# a "-std=<something>".
# For a C project, you would set this to something like 'c99' instead of
# 'c++11'.
'-std=gnu99',
# ...and the same thing goes for the magic -x option which specifies the
# language that the files to be compiled are written in. This is mostly
# relevant for c++ headers.
# For a C project, you would set this to 'c' instead of 'c++'.
'-x',
'c',
]


# Set this to the absolute path to the folder (NOT the file!) containing the
# compile_commands.json file to use that instead of 'flags'. See here for
# more details: http://clang.llvm.org/docs/JSONCompilationDatabase.html
#
# You can get CMake to generate this file for you by adding:
#   set( CMAKE_EXPORT_COMPILE_COMMANDS 1 )
# to your CMakeLists.txt file.
#
# Most projects will NOT need to set this to anything; you can just change the
# 'flags' list of compilation flags. Notice that YCM itself uses that approach.
compilation_database_folder = ''

if os.path.exists( compilation_database_folder ):
  database = ycm_core.CompilationDatabase( compilation_database_folder )
else:
  database = None

SOURCE_EXTENSIONS = [ '.cpp', '.cxx', '.cc', '.c', '.m', '.mm' ]

def DirectoryOfThisScript():
  return os.path.dirname( os.path.abspath( __file__ ) )


def MakeRelativePathsInFlagsAbsolute( flags, working_directory ):
  if not working_directory:
    return list( flags )
  new_flags = []
  make_next_absolute = False
  path_flags = [ '-isystem', '-I', '-iquote', '--sysroot=' ]
```

```
64    for flag in flags:
65      new_flag = flag
66
67      if make_next_absolute:
68        make_next_absolute = False
69        if not flag.startswith( '/' ):
70          new_flag = os.path.join( working_directory, flag )
71
72      for path_flag in path_flags:
73        if flag == path_flag:
74          make_next_absolute = True
75          break
76
77        if flag.startswith( path_flag ):
78          path = flag[ len( path_flag ): ]
79          new_flag = path_flag + os.path.join( working_directory, path )
80          break
81
82      if new_flag:
83        new_flags.append( new_flag )
84    return new_flags
85
86
87  def IsHeaderFile( filename ):
88    extension = os.path.splitext( filename )[ 1 ]
89    return extension in [ '.h', '.hxx', '.hpp', '.hh' ]
90
91
92  def GetCompilationInfoForFile( filename ):
93    # The compilation_commands.json file generated by CMake does not have entries
94    # for header files. So we do our best by asking the db for flags for a
95    # corresponding source file, if any. If one exists, the flags for that file
96    # should be good enough.
97    if IsHeaderFile( filename ):
98      basename = os.path.splitext( filename )[ 0 ]
99      for extension in SOURCE_EXTENSIONS:
100       replacement_file = basename + extension
101       if os.path.exists( replacement_file ):
102         compilation_info = database.GetCompilationInfoForFile(
103           replacement_file )
104         if compilation_info.compiler_flags_:
105           return compilation_info
106     return None
107   return database.GetCompilationInfoForFile( filename )
108
109
110 def FlagsForFile( filename, **kwargs ):
111   if database:
112     # Bear in mind that compilation_info.compiler_flags_ does NOT return a
113     # python list, but a "list-like" StringVec object
114     compilation_info = GetCompilationInfoForFile( filename )
115     if not compilation_info:
116       return None
117
118     final_flags = MakeRelativePathsInFlagsAbsolute(
119       compilation_info.compiler_flags_,
120       compilation_info.compiler_working_dir_ )
121
122     # NOTE: This is just for YouCompleteMe; it's highly likely that your project
123     # does NOT need to remove the stdlib flag. DO NOT USE THIS IN YOUR
124     # ycm_extra_conf IF YOU'RE NOT 100% SURE YOU NEED IT.
125     try:
126       final_flags.remove( '-stdlib=libc++' )
```

```
127       except ValueError:
128         pass
129     else:
130       relative_to = DirectoryOfThisScript()
131       final_flags = MakeRelativePathsInFlagsAbsolute( flags, relative_to )
132     return { 'flags': final_flags }
```

```
1
2   This file is to include your testing documentation. The file is to be
3   in plain text ASCII and properly spaced and edited so that when it is
4   viewed on a Linux machine it is readable. Not all ASCII files created
5   on a Windows machine display as expected due to the differences with
6   respect to how new lines and carriage returns are treated. Line widths
7   are to be limited to 80 characters.
8
9
10  Took code from scan codes to ASCII.
11
12   Testing
13   1. Showing prioritization of signals
14   2. syssighandler
15   3. syskill
16      create a root process (process 1).
17      Set handler that prints "signal 2 is running" at signal 28 for process 1.
18      Set handler that prints "signal 1 is running" at signal 31 for process 1.
19      create two more process (process 2 and 3)
20      have process 2
21          do syskill signal 31 to process 1.
22          do syskill signal 28 to process 1.
23      sysyield
24          handler run with trampoline code and signal 31 should run first then sin
    gnal 28
25          test passes:
26              sample output is: signal 1 running
27                                signal 2 running
28      set handler that prints "signal 2 is running" at signal 31 for process 1
29      set handler that prints "signal 1 is running" at signal 28 using oldhandler
    from 31 for process 1.
30
31      created two more process (process 4 and 5)
32      have process 4:
33          do syskill signal 31 to process 1
34          do syskill signal 28 to process 1
35              test passes:
36                  sample output is: signal 2 running
37                                    signal 1 running
38
39      this test prioritazation: 31 always prints before 28.
40      test syskill: we send singal from process to process and set up handler usin
    g the trampoline.
41      test sighandler we use it to set specific signal that print, and use the old
    hadler to set a new handler.
42
43   Testing:
44   5) sysopen invalid arguments.
45      create process, have prcess open device with a major number that is not part
     of divice table.
46      test passes, -1 is returned
47          sample out put: error opening device.
48
49   Testing:
50   6). syswrite with invalid fd
51      created frocess, opend device with major number 0 save the fd returned.
52      attempt a syswrtie with fd + 1.
53      test passed
54          sample output is: invalid fd, -1 returned.
55
56   Testing:
57   7). sysioctl test for invalid arguments
58      created process, open device with major number 0,
```

```
    make call to sysioctl with command 50(invalid command)
    test passed
        sample output is: ivalid command, not in range for this device, -1 returned.

 Testing:
 8). sysread when more character buffered in kernal than read request
    created proces, open device with major number 1.
    sysread specifying int bufflen of 1
    have shell as for input, type in four charcers.
    passes test
        only one charcater printed and checked to be in buffered passed by sysread.

Testing other:
    running the shell, we tested sysread by taking input from the keyboard user
    both echo on and echo off.. buffer from sysread compared agaist buffer made
    with the user input and buffers matched exacly.
    passed.

    ioctl tested by running shell, using it to switch from echo to non echo. Tested
    by viewing the output.. no echo no output shown. echo output shown.. passed.
    also ran shell used ctrl-d.. buffer kept characted prior to ctrl-d and then stopped reading
    disabaling interrupts. they used sysioctl to change the ctrl-d to x. then did same test
    as with ctrl-d typing x and inputs read up to x being typed at which point interrpts disabled.
    passed.
```

```c
/* create.c : create a process
*/

#include <xeroskernel.h>
#include <i386.h>
#include <limits.h>

/* Your code goes here. */

// Needed to prevent overflow of the PIDs
#define MAX_REUSE_COUNT ((INT_MAX - 32) / 32)
#define ENABLE_INTERRUPTS 0x3200

int create(void (*func)(void), int stackSize);
int nextPid(int reused_count, int index);
int createIdle(void (*func)(void), int stackSize);
struct pcb * setup_pcb(void (*func)(void), int stackSize);
void initFDT( struct FD *fd);

/*
 * ===  FUNCTION  ======================================================================
 *         Name:  create
 *  Description:  wrapper that creates regular process
 *  return     :  PID of process if successful or 0 if not
 * ======================================================================================
 */
int create(void (*func)(void), int stackSize) {
    struct pcb * process = setup_pcb(func, stackSize);
    if (!process) {
        return 0;
    }
    process->pid = nextPid(process->reuseCount, process->index);
    ready(process, &readyQueueHead, &readyQueueTail, STATE_READY);
    return process->pid;
}
//TODO
//TODO: remove function I don't think we need it anymore we can call idle first and use index starting from zero this way
//idle will always be assigned a zero.
int createIdle(void (*func)(void), int stackSize) {
    /*
     * ===  FUNCTION  ======================================================================
     *         Name:  createIdle
     *  Description:  wrapper that creates idle process
     *  Return     :  PID of idle = 0 if successful, -1 if not sucessful
     * ======================================================================================
     */
    struct pcb * process = setup_pcb(func, stackSize);
    if (!process) {
        return -1;
    }
    process->pid = 0;
    ready(process, &idleProcessHead, &idleProcessTail, STATE_READY);
    return 0;
}

/*
 * ===  FUNCTION  ======================================================================
```

```
58   *         Name:  setup_pcb
59   *  Description:  allocates an unused process control block and returns a pointe
     r to the
60   *                process control block
61   *       return:  a pointer to a process control block if successful or NULL if
     unsuccessful.
62   * ================================================================================
     ========
63   */
64  struct pcb * setup_pcb(void (*func)(void), int stackSize) {
65      struct pcb* newPcb = next(&stopQueueHead, &stopQueueTail);
66      if (!newPcb) {
67          kprintf("stopQueue has no available pcb function: create, file: create.c ");
68          return NULL;
69      }
70      // Top of the stack (lowest address);
71      unsigned long * stack = kmalloc(stackSize);
72      if (!stack) {
73          kprintf("kmalloc returned a null pointer function: create, file: create.c ");
74          return NULL;
75      }
76
77      unsigned long topStack = (unsigned long) stack;
78
79      unsigned long sizeStackAligned = (stackSize & 0xfffffff0); // don't add 16 s
     ince that is the header
80      // kprintf("\n\n Top of stack: %d", topStack);
81      // Bottom of the stack (highest address)
82      unsigned long bottomStack = (unsigned long) (topStack + sizeStackAligned);
83      //kprintf("\n\n Bottom of stack: %d", bottomStack);
84      unsigned long safetyMargin = (unsigned long) sizeof(unsigned long);
85      // Allocate space for the PCB
86      unsigned long cpuStatePointer = (unsigned long) (bottomStack - sizeof(struct
      CPU) - safetyMargin);
87      unsigned long *return_address = (unsigned long*) (bottomStack - safetyMargin
     );
88      // Setup default signal to be ignore for each function
89      int i;
90      for (i = 0; i < SIGNALMAX; i++) {
91              newPcb->sigFunctions[i] = NULL;
92      }
93      initFDT(newPcb->FDT);
94      *return_address = (unsigned long) &sysstop;
95      newPcb->memoryStart = (unsigned long*) topStack;
96      newPcb->cpuState = (struct CPU*) cpuStatePointer;
97      newPcb->sp = (unsigned long) cpuStatePointer;
98      newPcb->cpuState->edi = 0;
99      newPcb->cpuState->esi = 0;
100     newPcb->cpuState->ebp = 0;
101     newPcb->cpuState->esp = 0;
102     newPcb->cpuState->ebx = 0;
103     newPcb->cpuState->edx = 0;
104     newPcb->cpuState->ecx = 0;
105     newPcb->cpuState->eax = 0;
106     newPcb->cpuState->iret_eip = (unsigned long) (func);
107     newPcb->cpuState->iret_cs = getCS();
108     newPcb->cpuState->eflags = ENABLE_INTERRUPTS;
109     newPcb->reuseCount += 1;
110     newPcb->signalBitMask = 0;
111     if (newPcb->reuseCount > MAX_REUSE_COUNT) {
112         newPcb->reuseCount = 1;
113     }
114     return newPcb;
```

```
115 }
116
117
118
119 /*
120  * ===  FUNCTION  ======================================================================
    ========
121  *         Name:  nextPid
122  *  Description:  obtains a process id for the created process
123  *       Return:  the next available process id.
124  * ======================================================================================
    ========
125  */
126 int nextPid(int reuseCount, int index){
127     return (PCBTABLESIZE * reuseCount) + index;
128 }
129 void initFDT( struct FD *fd){
130     int i;
131     for(i = 0; i < FDTSIZE; i++){
132         fd[i].index = i;
133         fd[i].status = 0;
134         fd[i].majorNum = -1;
135     }
136 }
```

```
1  /* ctsw.c : context switcher
2  */
3
4  #include <xeroskernel.h>
5
6  /* Your code goes here - You will need to write some assembly code. You must
7     use the gnu conventions for specifying the instructions. (i.e this is the
8     format used in class and on the slides.) You are not allowed to change the
9     compiler/assembler options or issue directives to permit usage of Intel's
10    assembly language conventions.
11    */
12 void _ISREntryPoint(void);
13 void _TimerEntryPoint(void);
14 void _KeyboardEntryPoint(void);
15 static void *k_stack;
16 static unsigned long ESP;
17 static int rc, interrupt;
18
19 /*
20  * ===  FUNCTION  ======================================================================
21  *          Name:  contextswitch
22  *  Description:  switches between kernal and process (and vice versa)
23  *        Return:  returns an int representing the type of system call, -1 if
24  *                 given process is null
25  * ======================================================================================
26  */
27 int contextswitch(struct pcb* process){
28     if(!process){
29         return -1;
30     }
31
32     ESP = process->sp;
33     unsigned long *eax_register = (unsigned long*) (ESP + 28);
34     *eax_register = process->rc;
35
36     __asm__ __volatile__("pushf\n\t"
37             "pusha\n\t"
38             "movl %%esp, k_stack\n\t"
39             "movl ESP, %%esp\n\t"
40             "popa\n\t"
41             "iret\n"
42             "_KeyboardEntryPoint:\n\t"
43             "cli\n\t"
44             "pusha\n\t"
45             "movl $2, %%ecx\n\t"
46             "jmp _CommonEntryPoint\n"
47             "_TimerEntryPoint:\n\t"
48             "cli\n\t"
49             "pusha\n\t"
50             "movl $1, %%ecx\n\t"
51             "jmp _CommonEntryPoint\n"
52             "_ISREntryPoint:\n\t"
53             "cli\n\t"
54             "pusha\n\t"
55             "movl $0, %%ecx\n"
56             "_CommonEntryPoint:\n\t"
57             "movl %%esp, ESP\n\t"
58             "movl k_stack, %%esp\n\t"
59             "movl %%eax, rc\n\t"
60             "movl %%ecx, interrupt\n\t"
61             "popa\n\t"
```

```
62                "popf"
63                :
64                :
65                :"%eax", "%ecx"
66                );
67
68      process->args = (unsigned long*) (ESP + 44);
69
70      if (interrupt == 1) {
71          unsigned long *eax_register = (unsigned long*) (ESP + 28);
72          process->rc = *eax_register;
73          rc = TIMER_INT;
74      } else if (interrupt == 2) {
75          unsigned long *eax_register = (unsigned long*) (ESP + 28);
76          process->rc = *eax_register;
77          rc = KEYBOARD;
78      } else {
79          rc = *(process->args);
80      }
81
82      process->sp = ESP;
83
84      return rc;
85  }
86
87  /*
88   * ===  FUNCTION  ======================================================
    ========
89   *         Name:  contextinit
90   *  Description:  sets entry points to inturrupt table
91   * ======================================================================
    ========
92   */
93  void contextinit(void){
94      set_evec(67, (unsigned long) _ISREntryPoint);
95      set_evec(32, (unsigned long) _TimerEntryPoint);
96      set_evec(33, (unsigned long) _KeyboardEntryPoint);
97  }
```

```
1
2   /* di_calls.c : device independent calls
3   */
4
5   #include <xeroskernel.h>
6
7
8   int di_open(struct pcb *process, int device_no);
9   int di_close(struct pcb *process, int fd);
10  int di_write(struct pcb *process, int fd, unsigned char *buff, int size);
11  int di_read(struct pcb *process, int fd, unsigned char *buff, int size);
12  int di_ioctl(struct pcb *process, int fd, unsigned long command, int val);
13  void initFDT( struct pcb *process );
14  int addToQueue(struct FD *fd, struct FD **head, struct FD **tail);
15  struct FD* nextFd(struct FD *head);
16  int validDescr(struct pcb *process, int fd);
17
18  int di_open(struct pcb *process, int device_no){
19      if(device_no >= DEVICETABLESIZE){
20          return -1;
21      }
22      struct FD *fdNew = nextFd(process->FDT);
23      struct devsw *devopenptr;
24      if(!fdNew){
25          return -1;
26      }
27      fdNew->majorNum = device_no;
28      fdNew->status = 1; // 1 marks fd entry as currently open by device
29      devopenptr = &deviceTable[device_no];
30      fdNew->dvBlock = devopenptr;
31      int result = (devopenptr->dvopen)(devopenptr, device_no);
32      if(result){
33          return result;
34      }
35      return fdNew->index;
36  }
37
38  int di_close(struct pcb *process, int fd){
39      struct devsw *devcloseptr;
40      int result = validDescr(process, fd);
41      if(!result){
42          return -1;
43      }
44      devcloseptr = process->FDT[fd].dvBlock;
45      process->FDT[fd].status = 0;
46      process->FDT[fd].majorNum = -1;
47      int res = (devcloseptr->dvclose)(devcloseptr);
48      process->FDT[fd].dvBlock = NULL;
49      return res;
50  }
51
52  int di_write(struct pcb *process, int fd, unsigned char *buff, int size){
53      struct devsw *devwriteptr;
54      int result = validDescr(process, fd);
55      if(!result || !buff || !size){
56          return -1;
57      }
58      devwriteptr = process->FDT[fd].dvBlock;
59      return (devwriteptr->dvwrite)(devwriteptr);
60  }
61
62  int di_read(struct pcb *process, int fd, unsigned char *buff, int size){
63      struct devsw *devreadptr;
```

```
64      int result = validDescr(process, fd);
65      if(!result || !buff || !size){
66          return -1;
67      }
68      devreadptr = process->FDT[fd].dvBlock;
69      return (devreadptr->dvread)(devreadptr, process, buff, size);
70  }
71  int di_ioctl(struct pcb *process, int fd, unsigned long command, int val){
72      struct devsw *devioctlptr;
73      int result = validDescr(process, fd);
74      if(!result){
75          return -1;
76      }
77      devioctlptr = process->FDT[fd].dvBlock;
78      return (devioctlptr->dvioctl)(devioctlptr, command, val);
79  }
80
81  /*
82   * ===  FUNCTION  ======================================================
    ========
83   *         Name:  nextFd
84   *  Description:  removes a FD struct from given list identified by the given he
    ad.
85   *       Return:  return FD struct, null if non availble in given list
86   * ======================================================================
    ========
87   */
88  struct FD* nextFd(struct FD *head){
89      if(!head){
90          return NULL;
91      }
92      int i;
93      for(i = 0; i < FDTSIZE; i++){
94          if(head[i].majorNum != -1){
95              return NULL;
96          }
97      }
98      return head;
99  }
100 int validDescr(struct pcb *process, int fd){
101     struct FD *fdEntry = &(process->FDT[fd]);
102     if(fd < 0 || fd > 3 || !fdEntry->status){
103         return 0; // invalid
104     }
105     return 1;
106 }
107
108
109
```

```
1  /* disp.c : dispatcher
2  */
3
4  #include <xeroskernel.h>
5  #include <i386.h>
6  #include <stdarg.h>
7
8  struct pcb *readyQueueHead;
9  struct pcb *readyQueueTail;
10 struct pcb *recvAnyQueueHead;
11 struct pcb *recvAnyQueueTail;
12 struct pcb *stopQueueHead;
13 struct pcb *stopQueueTail;
14 struct pcb *idleProcessHead;
15 struct pcb *idleProcessTail;
16
17
18 #define MAGIC_NUMBER 9999
19
20 extern long freemem; /* set in i386.c */
21 extern char * maxaddr;
22
23 void dispatch(void);
24 void cleanup(struct pcb *process);
25 struct pcb* next(struct pcb **head, struct pcb **tail);
26 int ready(struct pcb *process, struct pcb **head, struct pcb **tail, int state);
27 int killProcess(int pid, int currentPid);
28 void removeNthPCB(struct pcb *process);
29 void clearWaitingProcesses(struct pcb **head, struct pcb **tail, int retCode);
30 void testCleanup(void);
31 void setupSignal(struct pcb *process);
32 int registerHandler(int signal, void(*newHandler)(void *), void(**oldHandler)(vo
   id*), struct pcb *pcb);
33 void wait(int pid, struct pcb *p);
34 struct pcb* runIdleIfReadyEmpty(struct pcb **head);
35 int getCPUtimes(struct pcb *p, struct processStatuses *ps);
36
37 /*
38  * ===  FUNCTION  ======================================================
   ========
39  *         Name:  dispatch
40  *  Description:  process systems call and schedules the next process, if given
   non existent
41  *                system call prints error message and loops forever.
42  * ======================================================================
   ========
43  */
44 void dispatch(void) {
45     struct pcb *process = next(&readyQueueHead, &readyQueueTail);
46
47     while (1) {
48         setupSignal(process);
49         int request = contextswitch(process);
50
51         switch( request ) {
52             case(CREATE):
53                 {
54                     void (*func)(void) = (void (*)(void)) *(process->args + 1);
55                     int stack  = (int) *(process->args + 2);
56                     int res = create(func, stack);
57                     process->rc = res;
58                     break;
59                 }
```

```
60                  case(TIMER_INT):
61                      {
62                          tick();
63                          process->cpuTime++;
64                          ready(process, &readyQueueHead, &readyQueueTail, STATE_READY
    );
65                          process = next(&readyQueueHead, &readyQueueTail);
66                          end_of_intr();
67                          break;
68                      }
69                  case(YIELD):
70                      {
71                          ready(process, &readyQueueHead, &readyQueueTail, STATE_READY
    );
72                          process = next(&readyQueueHead, &readyQueueTail);
73                          break;
74                      }
75                  case(STOP):
76                      {
77                          cleanup(process);
78                          process = next(&readyQueueHead, &readyQueueTail);
79                          break;
80                      }
81                  case(GETPID):
82                      {
83                          process->rc = process->pid;
84                          break;
85                      }
86                  case(PUTS):
87                      {
88                          char * str = (char *) *(process->args + 1);
89                          kprintf(str);
90                          break;
91                      }
92                  case(KILL):
93                      {
94                          int pid = (int) *(process->args + 1);
95                          int sig_no = (int) *(process->args + 2);
96                          // Old kill code, need to change after A3 is completed
97                          //process->rc = killProcess(pid, process->pid);
98                          process->rc = signal(pid, sig_no);
99                          ready(process, &readyQueueHead, &readyQueueTail, STATE_READY
    );
100                         process = next(&readyQueueHead, &readyQueueTail);
101                         break;
102                     }
103                 case(SEND):
104                     {
105                         //call the send in msg.c
106                         int pid = (int) *(process->args + 1);
107                         unsigned long num = (unsigned long) *(process->args + 2);
108                         process->rc = send(pid, num, process);
109                         process = next(&readyQueueHead, &readyQueueTail);
110                         break;
111
112                     }
113                 case(RECEIVE):
114                     {
115                         unsigned int * from_pid = (unsigned int *) *(process->args +
     1);
116                         unsigned long * num = (unsigned long *) *(process->args + 2)
    ;
117                         process->rc = recv(from_pid, num, process);
```

```
118                         process = next(&readyQueueHead, &readyQueueTail);
119                         break;
120                     }
121                 case(SLEEP):
122                     {
123                         unsigned int ms = (unsigned int) *(process->args + 1);
124                         sleep(ms, process);
125                         process = next(&readyQueueHead, &readyQueueTail);
126                         break;
127                     }
128                 case(SIG_HANDLER):
129                     {
130                         int sig_no = (int) *(process->args + 1);
131                         void (*handler)(void*) = (void (*)(void*)) *(process->args +
 2);
132                         void (**oldHandler)(void*) = (void (**)(void*)) *(process->a
rgs + 3);
133                         process->rc = registerHandler(sig_no, handler, oldHandler, p
rocess);
134                         ready(process, &readyQueueHead, &readyQueueTail, STATE_READY
);
135                         process = next(&readyQueueHead, &readyQueueTail);
136                         break;
137                     }
138                 case(SIG_RETURN):
139                     {
140                         unsigned long *oldSP = (unsigned long *) *(process->args + 1
);
141                         int retCode = (int) *(oldSP - 1);
142                         process->rc = retCode;
143                         process->sp = (unsigned long) oldSP;
144                         break;
145                     }
146                 case(WAIT):
147                     {
148                         int pid = (int) *(process->args + 1);
149                         wait(pid, process);
150                         process = next(&readyQueueHead, &readyQueueTail);
151                         break;
152                     }
153                 case(CPU_TIMES):
154                     {
155                         struct processStatuses *ps = (struct processStatuses*) *(pro
cess->args + 1);
156                         process->rc = getCPUtimes(process, ps);
157                         break;
158                     }
159                 case(OPEN):
160                     {
161                         int device_no = (int) *(process->args + 1);
162                         process->rc = di_open(process, device_no);
163                         break;
164                     }
165                 case(CLOSE):
166                     {
167                         int fd = (int) *(process->args + 1);
168                         process->rc = di_close(process, fd);
169                         break;
170                     }
171                 case(WRITE):
172                     {
173                         int fd = (int) *(process->args + 1);
174                         void *buff = (void*) *(process->args + 2);
```

```
175                     int bufflen = (int) *(process->args + 3);
176                     process->rc = di_write(process, fd, buff, bufflen);
177                     break;
178                 }
179             case(READ):
180                 {
181                     int fd = (int) *(process->args + 1);
182                     void *buff = (void*) *(process->args + 2);
183                     int bufflen = (int) *(process->args + 3);
184                     di_read(process, fd, buff, bufflen);
185                     process = next(&readyQueueHead, &readyQueueTail);
186                     break;
187                 }
188             case(IOCTL):
189                 {
190                     int fd = (int) *(process->args + 1);
191                     unsigned long command = (unsigned long) *(process->args + 2)
    ;
192                     int val = (int) *(process->args + 3);
193                     process->rc = di_ioctl(process, fd, command, val);
194                     break;
195                 }
196             case (KEYBOARD):
197                 {
198                     kbd_read_in();
199                     end_of_intr();
200                     break;
201                 }
202             default:
203                 {
204                     kprintf("ERROR, request is: %d function: dispatch, file: disp.c", request);
205                     for(;;);
206                 }
207         }
208     }
209 }
210 /*
211  * ===  FUNCTION  ======================================================
    ========
212  *          Name:  cleanup
213  *  Description:  frees the stack for process and places the pcb on the stopped
    queue
214  * ======================================================================
    ========
215  */
216 void cleanup(struct pcb *process) {
217     //testCleanup();
218     //kprintf("PID: %d, RETURNING MEMORY: %d\n", process->pid, process->memorySt
    art);
219     process->pid = -1;
220     int i;
221     for(i = 0; i < FDTSIZE; i++){
222         int mN = process->FDT[i].majorNum;
223         if (mN > -1) {
224             di_close(process, i);
225         }
226     }
227     kfree(process->memoryStart);
228     clearWaitingProcesses(&(process->sendQHead), &(process->sendQTail), -1);
229     clearWaitingProcesses(&(process->recvQHead), &(process->recvQTail), -1);
230     clearWaitingProcesses(&(process->waitQHead), &(process->waitQTail), 0);
231     ready(process, &stopQueueHead, &stopQueueTail, STATE_STOPPED);
232     //testCleanup();
```

```
233 }
234
235 /*
236  * ===  FUNCTION  ======================================================================
    ========
237  *         Name:  clearWaitingProcesses
238  *  Description:  removes process from the given head and tail and puts it on th
    e ready queue with a return code that was passed in as retCode
239  * ======================================================================================
    ========
240  */
241 void clearWaitingProcesses(struct pcb **head, struct pcb **tail, int retCode) {
242     while (*head && *tail) {
243         struct pcb *process = next(head, tail);
244         process->rc = retCode;
245         ready(process, &readyQueueHead, &readyQueueTail, STATE_READY);
246     }
247     *head = NULL;
248     *tail = NULL;
249 }
250
251 /*
252  * ===  FUNCTION  ======================================================================
    ========
253  *         Name:  killProcess
254  *  Description:  kills process of given pid returns 0, else returns -2 if pid =
    = itself
255  *                  else -1 if index or pid is invalid
256  * ======================================================================================
    ========
257  */
258 int killProcess(int pid, int currentPid){
259     int index = (pid % PCBTABLESIZE);
260     struct pcb* process = pcbTable + index;
261     if (index < 0 || process->pid != pid) {
262         return -1;
263     }
264     if(pid == currentPid){
265         return -2;
266     }
267     removeNthPCB(process);
268     cleanup(process);
269     return 0;
270 }
271
272
273 void wait(int pid, struct pcb *p) {
274     if (pid < 0) {
275         p->rc = -1;
276         ready(p, &readyQueueHead, &readyQueueTail, STATE_READY);
277     }
278     int index = (pid % PCBTABLESIZE);
279     struct pcb* process = pcbTable + index;
280     if (index < 0 || process->pid != pid) {
281         p->rc = -1;
282         ready(p, &readyQueueHead, &readyQueueTail, STATE_READY);
283     }
284     p->state = STATE_WAITING;
285     ready(p, &(process->waitQHead), &(process->waitQTail), STATE_WAITING);
286
287
288 }
289
```

```
290 // This function is the system side of the sysgetcputimes call.
291 // It places into a the structure being pointed to information about
292 // each currently active process.
293 //  p - a pointer into the pcbtab of the currently active process
294 //  ps  - a pointer to a processStatuses structure that is
295 //        filled with information about all the processes currently in the syste
    m
296 //
297
298
299 int getCPUtimes(struct pcb *p, struct processStatuses *ps) {
300
301   int i, currentSlot;
302   currentSlot = -1;
303
304   // Check if address is in the hole
305   if (((unsigned long) ps) >= HOLESTART && ((unsigned long) ps <= HOLEEND)) {
306     return -1;
307   }
308
309   //Check if address of the data structure is beyone the end of main memory
310   if ((((char * ) ps) + sizeof(struct processStatuses)) > maxaddr)  {
311     return -2;
312   }
313
314   // There are probably other address checks that can be done, but this is OK fo
    r now
315
316
317   for (i=0; i < PCBTABLESIZE; i++) {
318     struct pcb *currentProcess = &pcbTable[i];
319     if (currentProcess->state != STATE_STOPPED) {
320       // fill in the table entry
321       currentSlot++;
322       ps->pid[currentSlot] = currentProcess->pid;
323       ps->status[currentSlot] = p == currentProcess ? STATE_RUNNING : currentPro
    cess->state;
324       ps->cpuTime[currentSlot] = currentProcess->cpuTime * TICKLENGTH;
325     }
326   }
327
328   return currentSlot;
329 }
330
331
332 int registerHandler(int signal, void(*newHandler)(void *), void(**oldHandler)(vo
    id*), struct pcb *p) {
333     if (signal < 0 || signal > SIGNALMAX) {
334         return -1;
335     }
336     if (((char *) newHandler) > maxaddr) {
337         return -2;
338     }
339     if (((unsigned long) newHandler) > HOLESTART && ((unsigned long) newHandler)
     < HOLEEND) {
340         return -2;
341     }
342
343     if (oldHandler) {
344         *oldHandler = p->sigFunctions[signal];
345     }
346     p->sigFunctions[signal] = newHandler;
347
```

```
348     return 0;
349 }
350
351 void setupSignal(struct pcb* process) {
352     if (!process->signalBitMask) {
353             return;
354     }
355     unsigned long sigBM = process->signalBitMask;
356     int signalNo = 0;
357     // Determine largest signal number to process
358     while (sigBM >>= 1) {
359         signalNo++;
360     }
361
362     void (*handler)(void*) = process->sigFunctions[signalNo];
363     // Handler is null, so we ignore signal
364     if (!handler) {
365         // Set the bit in the signal to be zero
366         unsigned long one = 1;
367         unsigned long newSignalBitMask = process->signalBitMask;
368         process->signalBitMask = newSignalBitMask & ~(one << signalNo);
369         return;
370     }
371
372     // PREPARE ARGUMENTS FOR SIGTRAMP
373     unsigned long * sp = (unsigned long *) process->sp;
374     sp--;
375     *sp = process->rc;
376     sp--;
377     *sp = process->sp; // old context
378     sp--;
379     *sp = (unsigned long) handler; //handler function
380     sp--;
381     // Instead of return address (for testing)
382     *sp = MAGIC_NUMBER;
383
384
385     // SETUP NEW CONTEXT
386     struct CPU* context = (struct CPU*) sp;
387
388     // Move pointer down so that we can fit the CPU State
389     context--;
390
391     context->edi = 0;
392     context->esi = 0;
393     context->ebp = 0;
394     context->esp = 0;
395     context->ebx = 0;
396     context->edx = 0;
397     context->ecx = 0;
398     context->eax = 0;
399     context->iret_eip = (unsigned long) (&sigtramp);
400     context->iret_cs = getCS();
401     context->eflags = 0x3200;
402
403     // Set up process stack pointer to look like it begins where the new context
    is;
404     process->sp = (unsigned long) context;
405
406     // Set the bit in the signal as delivered
407     unsigned long one = 1;
408     unsigned long newSignalBitMask = process->signalBitMask;
409     process->signalBitMask = newSignalBitMask & ~(one << signalNo);
```

```
410
411     return;
412 }
413
414 /*
415  * ===  FUNCTION  ======================================================================
416  *         Name:  removeNthPCB
417  *  Description:  removes the given process from what ever queue it is currently in.
418  * ======================================================================================
419  */
420 void removeNthPCB(struct pcb *process){
421     //process is head and tail
422     if(!(process->prev) && !(process->next)){
423         *(process->head) = NULL;
424         *(process->tail) = NULL;
425     }
426     //process is the head
427     else if(!(process->prev)){
428         struct pcb *temp = process->next;
429         *(process->head) = temp;
430         temp->prev = NULL;
431         process->next = NULL;
432     }
433     //process is the tail
434     else if(!(process->next)){
435         struct pcb *temp = process->prev;
436         *(process->tail) = temp;
437         temp->next = NULL;
438         process->prev = NULL;
439     }
440     //process not head or tail, in middle of queue
441     else{
442         struct pcb *tempPrev = process->prev;
443         struct pcb *tempNext = process->next;
444         tempPrev->next = tempNext;
445         tempNext->prev = tempPrev;
446         process->next = NULL;
447         process->prev = NULL;
448     }
449     process->head = NULL;
450     process->tail = NULL;
451 }
452
453
454 /*
455  * ===  FUNCTION  ======================================================================
456  *         Name:  ready
457  *  Description:  adds a pcb process to given list identified by head and tail.
458  *       Return:  1 if sucessful 0 if unsucessful
459  * ======================================================================================
460  */
461 int ready(struct pcb *process, struct pcb **head, struct pcb **tail, int state){
462     if(!process){
463         return -1;
464     }
465     if(process == idleProcessHead){
466         return -2;
467     }
```

```
468     if(!*head && !*tail){
469         *head = process;
470         *tail = process;
471         process->prev = NULL;
472         process->next = NULL;
473         process->head = head;
474         process->tail = tail;
475         process->state = state;
476         return 1;
477     }
478     if(*head && *tail){
479         (*tail)->next = process;
480         process->prev = *tail;
481         *tail = process;
482         process->head = head;
483         process->tail = tail;
484         (*tail)->next = NULL;
485         process->state = state;
486         return 1;
487     }
488     kprintf("\n\n one of QueueHead or QueueTail is NULL\n file: disp.c\n function: ready");
489     return 0;
490 }
491
492 /*
493  * ===  FUNCTION  ======================================================================
494  *         Name:  next
495  *  Description:  removes a pcb struct from given list identified by the given head.
496  *       Return:  return pcb struct, null if non availble in given list
497  * ======================================================================================
498  */
499 struct pcb* next(struct pcb **head, struct pcb **tail){
500     if(!*head){
501         // gets idle process only if dealing with ready queue
502         struct pcb* idleProcess = runIdleIfReadyEmpty(head);
503         return idleProcess;
504     }
505     //if they are the same only one thing on list
506     if (*head == *tail) {
507         *tail = NULL;
508     }
509     struct pcb *nextProcess = *head;
510     *head = (*head)->next;
511     nextProcess->next = NULL;
512     nextProcess->prev = NULL;
513     (*head)->prev = NULL;
514     nextProcess->head = NULL;
515     nextProcess->tail = NULL;
516     return nextProcess;
517 }
518 /*
519  * ===  FUNCTION  ======================================================================
520  *         Name:  runIdleIfReadyEmpty
521  *  Description:  if readyQueue is empty runs the idle prcess
522  *      Return :  pcb  * to the idleProcessHead if readyQueue Empty, else NULL
523  * ======================================================================================
524  */
525 struct pcb* runIdleIfReadyEmpty(struct pcb **head){
```

```
526     if(head == &readyQueueHead){
527         return idleProcessHead;
528     }
529     return NULL;
530 }
531
532 //Test function for cleanup
533 void testCleanup(void){
534
535     struct pcb *sqht = stopQueueHead;
536     int i = 1;
537     while(sqht){
538         //kprintf("(sqht: %d)-->", sqht->state);
539         sqht = sqht->next;
540         i++;
541     }
542     kprintf("\nThere are %d, processes in the stopped queue", i);
543 }
```

```
/* initialize.c - initproc */

#include <i386.h>
#include <xeroskernel.h>
#include <xeroslib.h>

extern  int     entry( void );  /* start of kernel image, use &start    */
extern  int     end( void );    /* end of kernel image, use &end        */
extern  long    freemem;        /* start of free memory (set in i386.c) */
extern char     *maxaddr;       /* max memory address (set in i386.c)   */

struct pcb *pcbTable;
struct devsw *deviceTable;

void initProcessTable( void );
static void idleproc( void );
void initDeviceTable( void );

/************************************************************************/
/***                            NOTE:                                 ***/
/***                                                                  ***/
/***   This is where the system begins after the C environment has    ***/
/***   been established.  Interrupts are initially DISABLED.  The     ***/
/***   interrupt table has been initialized with a default handler    ***/
/***                                                                  ***/
/***                                                                  ***/
/************************************************************************/

/*------------------------------------------------------------------------
 *  The init process, this is where it all begins...
 *------------------------------------------------------------------------
 */
void initproc( void )                           /* The beginning */
{

    char str[1024];
    int a = sizeof(str);
    int b = -69;
    int i;

    kprintf( "\n\nCPSC 415, 2016W1 \n32 Bit Xeros 0.01 \nLocated at: %x to %x\n",
            &entry, &end);

    kprintf("Some sample output to illustrate different types of printing\n\n");

    /* A busy wait to pause things on the screen, Change the value used
       in the termination condition to control the pause
       */

    for (i = 0; i < 3000000; i++);

    /* Build a string to print) */
    sprintf(str,
            "This is the number -69 when printed signed %d unsigned %u hex %x and a string %s.\n     Sample printing of 1024 in signed %d, unsigned %u and hex %x.",
            b, b, b, "Hello", a, a, a);

    /* Print the string */

    kprintf("\n\nThe %dstring is: \"%s\"\n\nThe formula is %d + %d = %d.\n\n\n",
            a, str, a, b, a + b);

    for (i = 0; i < 4000000; i++);
```

```
63      /* or just on its own */
64      kprintf(str);
65
66      /* Add your code below this line and before next comment */
67      kprintf("\n");
68      kmeminit();
69      contextinit();
70      initPIT(TIMESLICE);
71      pcbTable = (struct pcb*) kmalloc(PCBTABLESIZE*sizeof(struct pcb));
72      if (!pcbTable) {
73          kprintf("\n\nCould not allocate memory for pcbtable. File: init.c. Function: initproc()");
74          for(;;);
75      }
76      initProcessTable();
77
78      deviceTable = (struct devsw*) kmalloc(DEVICETABLESIZE*sizeof(struct devsw));
79      if(!deviceTable) {
80          kprintf("\n\nCould not allocate memory for deviceTable. File: init.c. Function initproc()");
81      }
82      initDeviceTable();
83
84
85      create(&idleproc, 8000);
86      create(&root, 8000);
87      //testing function for stopping process.
88      //int restest = create(&testStop, 7989);
89      //kprintf("Root PID is %d", res);
90      dispatch();
91
92
93      for (i = 0; i < 2000000; i++);
94      /* Add all of your code before this comment and after the previous comment *
    /
95      /* This code should never be reached after you are done */
96      kprintf("\n\nWhen the kernel is working properly ");
97      kprintf("this line should never be printed!\n");
98      for(;;) ; /* loop forever */
99  }
100
101 /*
102  * ===  FUNCTION  ======================================================
    ========
103  *         Name:  initProcessTable
104  *  Description:  initiate the process table putting it on the stoppedQueue
105  * ======================================================================
    ========
106  */
107 void initProcessTable( void ){
108     // Allocate a process table with 32 pcbs and place them all in the stopped q
    ueue
109     struct pcb *pcbTableHead = pcbTable;
110     int i;
111     for (i = 0; i < PCBTABLESIZE; i++) {
112         pcbTableHead[i].pid = -1;
113         pcbTableHead[i].index = i;
114         pcbTableHead[i].reuseCount = -1;
115         pcbTableHead[i].head = &stopQueueHead;
116         pcbTableHead[i].tail = &stopQueueTail;
117         ready(pcbTableHead+i, &stopQueueHead, &stopQueueTail, STATE_STOPPED);
118     }
119     //struct pcb *pcbTableHeadTest = pcbTable;
120     /* test to make sure that stopped queue is set up with 32 pcbs
121        i = 0;
```

```
122         while(i < PCBTABLESIZE){
123         kprintf("file: init.c, functin: initproc state: %d\n",  (pcbTableHeadTest
    +i)->state);
124         i++;
125         }
126         */
127
128 }
129 /*
130  * ===  FUNCTION  ======================================================================
    ========
131  *         Name:  initDeviceTable
132  *  Description:  initiate the device table
133  * ======================================================================================
    ========
134  */
135 void initDeviceTable( void ){
136     struct devsw *deviceTableHead = deviceTable;
137     int i;
138     for(i = 0; i < DEVICETABLESIZE; i++){
139         deviceTableHead[i].dvnum = i;
140         deviceTableHead[i].dvopen = &kb_open;
141         deviceTableHead[i].dvclose = &kb_close;
142         deviceTableHead[i].dvread =  &kb_read;
143         deviceTableHead[i].dvwrite = &kb_write;
144         deviceTableHead[i].dvioctl = &kb_ioctl;
145     }
146     deviceTableHead[0].dvname = "ECHOING_KEYBOARD";
147     deviceTableHead[1].dvname = "NON_ECHOING_KEYBOARD";
148 }
149
150 static void idleproc( void ){
151     for(;;){
152         sysyield();
153     }
154 }
```

```
#include <xeroskernel.h>
#include <i386.h>
#include <kbd.h>

static char kbuf[MAX_KBUF_SIZE];
static int kBytesRead = 0;

static char TOGGLE_ECHO = 1;
static char KB_IN_USE = 0;
static int EOFINDICATOR = 0x4;
static int EOFFLAG = 0;

// Stores exactly one request for the keyboard
static struct dataRequest kbDataRequest;

static int state; /* the modifer key state of the keyboard */

unsigned int kbtoa( unsigned char code );
int done(void);
int kb_open(const struct devsw* const dvBlock, int majorNum);
int kb_close(const struct devsw* const dvBlock);
int kb_ioctl(const struct devsw* const dvBlock, unsigned long command, int val);
int kb_write(const struct devsw * const dvBlock);
int kb_read(const struct devsw * const dvBlock, struct pcb * const process, void
 *buff, int size);
int copyCharactersToBuffer(char *outBuf, int outBufSize, int outBufBytesRead, ch
ar * inBuf, int inBufSize, int *inBufBytesRead);
static int extchar(unsigned char code);

int kb_open(const struct devsw* const dvBlock, int majorNum) {
    if (KB_IN_USE) {
        //return failure
        return -1;
    }
    if (!majorNum) {
        // Make sure echo is turned off
        TOGGLE_ECHO = 0;
    }
    if (majorNum) {
        // Make sure echo is turned off
        TOGGLE_ECHO = 1;
    }
    kBytesRead = 0;
    KB_IN_USE = 1;
    EOFINDICATOR = 0x4;//standard EOF
    EOFFLAG = 0;
    enable_irq(1,0);
    return 0;
}

int kb_write(const struct devsw * const dvBlock) {
    //we can just always return -1 here since we don't write
    return -1;
}

int kb_close(const struct devsw* const dvBlock) {
    if (!KB_IN_USE) {
        //return failure
        return -1;
    }
    if (!dvBlock->dvnum) {
        // closing steps for device one
    }
```

```
62      if (dvBlock->dvnum == 1) {
63          //clsing steps for device two
64      }
65      kBytesRead = 0;
66      KB_IN_USE = 0;
67      return 0;
68  }
69
70  int kb_ioctl(const struct devsw* const dvBlock, unsigned long command, int val)
    {
71      if(command == 53){
72          EOFINDICATOR = val;
73          return 0;
74      }
75      else if(command == 56){
76          TOGGLE_ECHO = 1;
77          return 0;
78      }
79      else if(command == 55){
80          TOGGLE_ECHO = 0;
81          return 0;
82      }
83      else{
84          return -1; //error didn't get correct command
85      }
86  }
87
88  int kb_read(const struct devsw * const dvBlock, struct pcb * p, void *buff, int
    size) {
89      if (!p) {
90          return -1;
91      }
92      if (EOFFLAG) {
93          return 0;
94      }
95      int bytesRead = 0;
96      if (kBytesRead > 0) {
97          bytesRead = copyCharactersToBuffer(buff, size, 0, &kbuf[0], MAX_KBUF_SIZ
    E, &kBytesRead);
98      }
99      kbDataRequest.status = 1;
100     kbDataRequest.buff = buff;
101     kbDataRequest.size = size;
102     kbDataRequest.bytesRead = bytesRead;
103     kbDataRequest.done = &done;
104     if (bytesRead == size) {
105         // we're done with this sysread call;
106         done();
107         return 0;
108     }
109     kbDataRequest.blockedProc = p;
110     p->state = STATE_DEV_WAITING;
111     return 0;
112 }
113
114 int done(void) {
115     struct pcb * p =  kbDataRequest.blockedProc;
116     p->rc = kbDataRequest.bytesRead;
117     ready(p, &readyQueueHead, &readyQueueTail, STATE_READY);
118     kbDataRequest.status = 0;
119     kbDataRequest.blockedProc = NULL;
120     kbDataRequest.size = 0;
121     kbDataRequest.bytesRead = 0;
```

```
122     return 0;
123 }
124
125
126 //==================================
127 //          LOWER HALF
128 //==================================
129
130 int kbd_read_in() {
131     unsigned char ctrlByte = inb(CTRL_PORT);
132     unsigned char scanCode = inb(READ_PORT);
133     unsigned int character = kbtoa(scanCode);
134     if (!(ctrlByte & 1)) {
135         // nothing to read, spurious interrupt
136         return -2;
137     }
138     if (kBytesRead == MAX_KBUF_SIZE) {
139         // discard characters because buffer is full
140         kprintf("KEYBOARD BUFFER FULL\n");
141         return -3;
142     }
143
144     if (character == NOCHAR) {
145         // discard uneeded scan codes
146         return -4;
147     }
148     if (TOGGLE_ECHO) {
149         kprintf("%c", character);
150     }
151     // Put it in the buffer
152     if (kBytesRead < MAX_KBUF_SIZE) {
153         kbuf[kBytesRead] = (unsigned char) character;
154         kBytesRead++;
155     }
156     if (kbDataRequest.status) {
157         char * buff = kbDataRequest.buff;
158         int bytesRead = kbDataRequest.bytesRead;
159         int size = kbDataRequest.size;
160         // Copy as much from the buffer into the dataRequest
161         bytesRead = copyCharactersToBuffer(buff, size, bytesRead, &kbuf[0], MAX_
    KBUF_SIZE, &kBytesRead);
162         kbDataRequest.bytesRead = bytesRead;
163         if (bytesRead == size || character == '\n'|| character == EOFINDICATOR)
    {
164             kbDataRequest.done();
165         }
166     }
167     return 0;
168 }
169
170 int copyCharactersToBuffer(char *outBuf, int outBufSize, int outBufBytesRead, ch
    ar * inBuf, int inBufSize, int *inBufBytesRead) {
171     int bytesCopied = 0;
172     while (outBufBytesRead < outBufSize && bytesCopied < *inBufBytesRead) {
173         if(inBuf[bytesCopied] == EOFINDICATOR){
174             enable_irq(1,1);
175             EOFFLAG = 1;
176             break;
177         }
178         outBuf[outBufBytesRead] = inBuf[bytesCopied];
179         outBufBytesRead++;
180         bytesCopied++;
181     }
```

```
182     *inBufBytesRead -= bytesCopied;
183     return outBufBytesRead;
184 }
185 
186 
187 static int extchar(unsigned char code) {
188     return state &= ~EXTENDED;
189 }
190 
191 
192 unsigned int kbtoa( unsigned char code )
193 {
194     unsigned int ch;
195 
196     if (state & EXTENDED)
197         return extchar(code);
198 
199     if (code & KEY_UP) {
200         switch (code & 0x7f) {
201             case LSHIFT:
202             case RSHIFT:
203                 state &= ~INSHIFT;
204                 break;
205             case CAPSL:
206                 state &= ~CAPSLOCK;
207                 break;
208             case LCTL:
209                 state &= ~INCTL;
210                 break;
211             case LMETA:
212                 state &= ~INMETA;
213                 break;
214         }
215 
216         return NOCHAR;
217     }
218 
219 
220     /* check for special keys */
221     switch (code) {
222         case LSHIFT:
223         case RSHIFT:
224             state |= INSHIFT;
225             //kprintf("shift detected!\n");
226             return NOCHAR;
227         case CAPSL:
228             state |= CAPSLOCK;
229             return NOCHAR;
230         case LCTL:
231             state |= INCTL;
232             return NOCHAR;
233         case LMETA:
234             state |= INMETA;
235             return NOCHAR;
236         case EXTESC:
237             state |= EXTENDED;
238             return NOCHAR;
239     }
240 
241     ch = NOCHAR;
242 
243     if (code < sizeof(kbcode)){
244         if ( state & CAPSLOCK )
```

```
245             ch = kbshift[code];
246         else
247             ch = kbcode[code];
248     }
249     if (state & INSHIFT) {
250         if (code >= sizeof(kbshift))
251             return NOCHAR;
252         if ( state & CAPSLOCK )
253             ch = kbcode[code];
254         else
255             ch = kbshift[code];
256     }
257     if (state & INCTL) {
258         if (code >= sizeof(kbctl))
259             return NOCHAR;
260         ch = kbctl[code];
261     }
262     if (state & INMETA)
263         ch += 0x80;
264     return ch;
265 }
266
267
268
269 //================================================
270 //   HELPER FUNCTIONS (NOT USING CURRENTLY)
271 //================================================
272
273 /*
274  * ===  FUNCTION  ======================================================================
    ========
275  *         Name:  ready
276  *  Description:  adds a pcb process to given list identified by head and tail.
277  *       Return:  1 if sucessful 0 if unsucessful
278  * ======================================================================================
    ========
279  */
280 int queueRequest(struct dataRequest *dr, struct dataRequest **head, struct dataR
    equest **tail, int status){
281     if(!dr){
282         return -1;
283     }
284     if(!*head && !*tail){
285         *head = dr;
286         *tail = dr;
287         dr->prev = NULL;
288         dr->next = NULL;
289         dr->status = status;
290         return 1;
291     }
292     if(*head && *tail){
293         (*tail)->next = dr;
294         dr->prev = *tail;
295         *tail = dr;
296         (*tail)->next = NULL;
297         dr->status = status;
298         return 1;
299     }
300     kprintf("\n\n one of QueueHead or QueueTail is NULL\n file: disp.c\n function: ready");
301     return 0;
302 }
303
304 /*
```

```
305  * ===  FUNCTION  ======================================================
     ========
306  *         Name:  next
307  *  Description:  removes a pcb struct from given list identified by the given h
     ead.
308  *       Return:  return pcb struct, null if non availble in given list
309  * ======================================================================
     ========
310  */
311 struct dataRequest* nextRequest(struct dataRequest **head, struct dataRequest **
     tail){
312     if(!*head){
313         // gets idle process only if dealing with ready queue
314         return NULL;
315     }
316     //if they are the same only one thing on list
317     if (*head == *tail) {
318         *tail = NULL;
319     }
320     struct dataRequest *nextDr = *head;
321     *head = (*head)->next;
322     nextDr->next = NULL;
323     nextDr->prev = NULL;
324     (*head)->prev = NULL;
325     return nextDr;
326 }
327
328
329
330
```

```
1  /* mem.c : memory manager
2  */
3
4  #include <xeroskernel.h>
5  #include <xeroslib.h>
6  #include <i386.h>
7
8  #define SANITYCHECK 314159265 // contant used for sanity check
9  extern long freemem; /* set in i386.c */
10 extern  int     entry( void );  /* start of kernel image, use &start    */
11 extern  int     end( void );    /* end of kernel image, use &end        */
12
13 //structure used to store information about memory chunks
14 struct memHeader {
15     unsigned long size;// size of the memory space allocated
16     struct memHeader *prev; //previous memHeader on the freelist
17     struct memHeader *next; //next memHeader on the freelist
18     char *sanityCheck; // veryfy nothing is wrong with the memHeader returned by
    user
19     unsigned char dataStart[0]; // start address of allocated memory
20 };
21
22
23 struct memHeader *freeListHead;
24 struct memHeader *freeListTail;
25
26
27
28
29 struct memHeader *removeFreeListMemberWithSize(int size);
30 int insert(struct memHeader *memSlot);
31 struct memHeader *removeFromList(struct memHeader *prevHead, struct memHeader *h
   ead);
32 struct memHeader *findNextMemHeaderWithAddress(long addr);
33 struct memHeader *findPreviousMemHeaderWithAddress(long addr);
34 void startTestMem(void);
35 void printCurrentList(char *descriptor);
36
37 /*
38  * ===  FUNCTION  ======================================================
   ========
39  *         Name:  kmeminit
40  *  Description:  initilizes the free memomry linked list
41  * ======================================================================
   ========
42  */
43 void kmeminit(void) {
44     struct memHeader *freeSpace1 = (struct memHeader*) freemem;
45     freeSpace1->size = HOLESTART - freemem + sizeof(struct memHeader);
46     freeSpace1->next = (struct memHeader*) HOLEEND;
47     freeSpace1->sanityCheck = (char*) SANITYCHECK;
48
49     struct memHeader *freeSpace2 = (struct memHeader*) HOLEEND;
50     freeSpace2->size = 0x400000 - HOLEEND + sizeof(struct memHeader);
51     freeSpace2->prev = (struct memHeader*) freemem;
52     freeSpace2->sanityCheck = (char*) SANITYCHECK;
53
54     int result = insert(freeSpace1);
55     if (!result) {
56         kprintf("\n\ninsert freespace1 failed\n file: mem.c\n function: kmeminit");
57     }
58     result = insert(freeSpace2);
59     if (!result) {
```

```
60          kprintf("\n\ninsert freespace2 failed\n file: mem.c\n function: kmeminit");
61      }
62      /*
63          long addr1 = (long) (&(freeListHead->size));
64          long addr2 = (long) (&(freeListTail->size));
65          kprintf("\n\nfreespace1 allocated at %x\n, with size %u\n", addr1 , freeL
    istHead->size);
66          kprintf("\n\nfreespace2 allocated at %x\n, with size %u\n", addr2, freeLi
    stTail->size);
67          */
68
69      //startTestMem();
70  }
71
72
73  /*
74   * ===  FUNCTION  ======================================================================
    ========
75   *          Name:  kmalloc
76   *  Description:  allocates memorys
77   *        Return:  return pointer to allocated chunk if abs(size) is available.
78   * ======================================================================================
    ========
79   */
80  void* kmalloc(int size) {
81
82      if (size < 1) {
83          return NULL;
84      }
85
86      long finalSize = (size & 0xfffffff0) + 16;
87      struct memHeader *foundNode = removeFreeListMemberWithSize(finalSize);
88      //kprintf("final size: %u", finalSize);
89
90      if (!foundNode) {
91          kprintf("\n\ncould not find mem of: %d\n in function: kmalloc\n and file: mem.c", size);
92          return NULL;
93      } else {
94          //kprintf("foundNode %u", foundNode);
95          //kprintf("size memHeader %u", sizeof(struct memHeader));
96
97          struct memHeader *newFreeSpace = (struct memHeader*) (((long)foundNode)
    + sizeof(struct memHeader) + finalSize);
98          unsigned long newFreeSpaceLocation = (unsigned long) (newFreeSpace);
99          //kprintf("addr of nfs %u", (long) newFreeSpace);
100
101         // Don't insert a new free space header because we will be outside valid
     memory
102         foundNode->size = finalSize;
103         foundNode->sanityCheck = (char*) SANITYCHECK;
104         if ((newFreeSpaceLocation >= HOLESTART) || (newFreeSpaceLocation >= 0x40
    0000)) {
105             return (void*) &(foundNode->dataStart);
106         }
107
108         newFreeSpace->size = foundNode->size - finalSize - sizeof(struct memHead
    er);
109         newFreeSpace->sanityCheck = (char*) SANITYCHECK;
110         int result = insert(newFreeSpace);
111
112         if(!result){
113             kprintf("\n\ninsert newFreeSpace failed\n file: mem.c\n function: kmalloc");
114         }
```

```
115         if(!newFreeSpace->size){
116             kprintf("\n\ninserted memory with size 0\n file: mem.c\n function: kmalloc");
117         }
118 
119         return (void*) &(foundNode->dataStart);
120     }
121 }
122 
123 
124 /*
125  * ===  FUNCTION  ======================================================================
126  *         Name:  kfree
127  *  Description:  adds memHeader to free memory link list given a pointer to the returned address
128  *                if memHeader found to be invalid at given address goes into infinte loop.
129  * =====================================================================================
130  */
131 void kfree(void *ptr) {
132     struct memHeader *returnedMemHeader = (struct memHeader *) (((long) ptr) - sizeof(struct memHeader));
133     if(((long)returnedMemHeader->sanityCheck != SANITYCHECK)){
134         kprintf("\n\n not a match or memhead overwritten file: mem.c function: kfree");
135         for(;;);
136     }
137     //not sure if this is needed
138     returnedMemHeader->sanityCheck = NULL;
139     long returnedMemHeaderStartAddr = (long) returnedMemHeader;
140     long returnedMemHeaderEndAddr = returnedMemHeaderStartAddr + sizeof(struct memHeader) + returnedMemHeader->size;
141 
142     struct memHeader *previousMemHeader = findPreviousMemHeaderWithAddress(returnedMemHeaderStartAddr);
143     struct memHeader *nextMemHeader = findNextMemHeaderWithAddress(returnedMemHeaderEndAddr);
144     // found all three or just the previous
145     if (previousMemHeader) {
146         // we update previous memheader
147         previousMemHeader->size += (returnedMemHeader->size + sizeof(struct memHeader));
148         previousMemHeader->sanityCheck = NULL;
149         if (nextMemHeader) {
150             previousMemHeader->size += (nextMemHeader->size + sizeof(struct memHeader));
151             // insert into list
152         }
153         insert(previousMemHeader);
154         return;
155     }
156     // found the current and the next;
157     if (nextMemHeader) {
158         // update current
159         returnedMemHeader->size += (nextMemHeader->size + sizeof(struct memHeader));
160         // insert into list
161         insert(returnedMemHeader);
162         return;
163     }
164     // else just insert the returned memheader;
165     insert(returnedMemHeader);
166 }
```

```
167
168 /*
169  * ===  FUNCTION  ======================================================================
    ========
170  *         Name:  insert
171  *  Description:  performs actual insert of memHeader into freelist
172  * ======================================================================================
    ========
173  */
174 int insert(struct memHeader *memSlot) {
175     if(!freeListTail && !freeListHead){
176         freeListHead = memSlot;
177         freeListTail = memSlot;
178         return 1;
179     }
180     if(freeListHead && freeListTail){
181         struct memHeader *current = freeListHead;
182         struct memHeader *prevCurrent = NULL;
183         long addressMemSlot = (long) memSlot;
184         while(current){
185             if(addressMemSlot < (long) current){
186                 if(!prevCurrent){
187                     memSlot->next = current;
188                     current->prev = memSlot;
189                     freeListHead = memSlot;
190                     return 1;
191                 }
192                 else{
193                     prevCurrent->next = memSlot;
194                     current->prev = memSlot;
195                     memSlot->next = current;
196                     memSlot->prev = prevCurrent;
197                     return 1;
198                 }
199             }
200             prevCurrent = current;
201             current = current->next;
202         }
203         memSlot->prev = freeListTail;
204         freeListTail->next = memSlot;
205         freeListTail = freeListTail->next;
206         return 1;
207     }
208     kprintf("\n\n one of freeListHead or freeListTail is NULL\n file: mem.c\n function: insert");
209     return 0;
210 }
211
212 /*
213  * ===  FUNCTION  ======================================================================
    ========
214  *         Name:  removeFreeeListMemberWithSize
215  *  Description:  removes the first item in the freelist >= given size
216  *       Return:  returns the memHeader removed from freelist
217  * ======================================================================================
    ========
218  */
219 struct memHeader *removeFreeListMemberWithSize(int size) {
220     struct memHeader *head = freeListHead;
221     struct memHeader *prevHead = NULL;
222     while(head){
223         if(head->size >= size){
224             return removeFromList(prevHead, head);
225         }
```

```
226             prevHead = head;
227             head = head->next;
228         }
229         //kprintf("Error: Not enough free memory found.\n");
230         return NULL;
231     }
232 
233     struct memHeader *removeFromList(struct memHeader *prevHead, struct memHeader *h
    ead) {
234         //case: find first item on list;
235         if(!prevHead){
236             //case: head and tail are equal
237             if (freeListHead == freeListTail){
238                 freeListHead = NULL;
239                 freeListTail = NULL;
240             }
241             //case: head and tail are not equal
242             else{
243                 freeListHead = head->next;
244                 freeListHead->prev = NULL;
245                 head->next = NULL;
246             }
247         }
248         //case: find nth item in list
249         else{
250             if(head == freeListTail){
251                 freeListTail = freeListTail->prev;
252                 head->prev = NULL;
253             }
254             else{
255                 struct memHeader *oneAhead = head->next;
256                 if (oneAhead) {
257                     oneAhead->prev = prevHead;
258                 }
259                 prevHead->next = oneAhead;
260                 head->next = NULL;
261                 head->prev = NULL;
262             }
263         }
264         return head;
265 
266     }
267 
268     /*
269      * ===  FUNCTION  ======================================================
    ========
270      *         Name:  findNextMemHeaderWithAddress
271      *  Description:  finds a memHeader with given address that comes next
272      *       Return:  returns found memHeader if found else Null
273      * ======================================================================
    ========
274      */
275     struct memHeader *findNextMemHeaderWithAddress(long addr) {
276         struct memHeader *head = freeListHead;
277         struct memHeader *prevHead = NULL;
278         while(head){
279             long headAddress = (long) head;
280             if (headAddress == addr) {
281                 return removeFromList(prevHead, head);
282             }
283             prevHead = head;
284             head = head->next;
285         }
```

```
286     return NULL;
287 }
288 
289 /*
290  * ===  FUNCTION  ======================================================================
291  *         Name:  findPreviousMemHeaderWithAddress
292  *  Description:  finds a memHeader with given address that comes previous
293  *       Return:  returns found memHeader if found else Null
294  * ======================================================================================
295  */
296 struct memHeader *findPreviousMemHeaderWithAddress(long addr) {
297     struct memHeader *head = freeListHead;
298     struct memHeader *prevHead = NULL;
299     while(head){
300         long headAddress =  (((long)head) + sizeof(struct memHeader) + head->size);
301         if (headAddress == addr) {
302             return removeFromList(prevHead, head);
303         }
304         prevHead = head;
305         head = head->next;
306     }
307     return NULL;
308 }
309 
310 /*
311  * ===  FUNCTION  ======================================================================
312  *         Name:  startTestMem
313  *  Description:  starting method for testing memory allocation.
314  * ======================================================================================
315  */
316 void startTestMem(void){
317     printCurrentList("start");
318     //test a simple allocation
319     int* firstAllocation = (int*) kmalloc(1000);
320     //allocation should have moved over 1024 size 1008
321     printCurrentList("1000");
322     //allocate second item
323     int* secondAllocation = (int*) kmalloc(2000);
324     //allocation should have moved over 2032 size 2016
325     printCurrentList("2000");
326     int* thirdAllocation = (int*) kmalloc(5000);
327     //allocation should have moved over 5024 size 5008
328     printCurrentList("5000");
329     //kfree the second item
330     kfree(thirdAllocation);
331     printCurrentList("free3");
332     kfree(firstAllocation);
333     printCurrentList("free1");
334     kfree(secondAllocation);
335     printCurrentList("free2");
336 }
337 
338 void printCurrentList(char *descriptor){
339     int i = 0;
340     struct memHeader *current = freeListHead;
341     kprintf("list %s\n", descriptor);
342     while(current){
343         long address = (long) current;
```

```
344             kprintf("(mem: %u/size: %u)-->", address, current->size);
345             i = i + 1;
346             current = current->next;
347         }
348         kprintf("\n");
349     }
```

```
1  /* msg.c : messaging system
2     This file does not need to modified until assignment 2
3     */
4
5  #include <xeroskernel.h>
6  #include <stdarg.h>
7
8
9  int send(int dest_pid, unsigned long num, struct pcb * currentProcess);
10 int recv(unsigned int *from_pid, unsigned long *num, struct pcb * currentProcess
   );
11
12 /*
13  * ===  FUNCTION  ======================================================
   ========
14  *         Name:  send
15  *  Description:  sends num to dest_pid process returs 0 is successful, else ret
   urns -1 if invalid pid or index out of bounds
16  *                 -2 if sending to itself, and -3 if any other errors
17  * ======================================================================
   ========
18  */
19 int send(int dest_pid, unsigned long num, struct pcb * currentProcess) {
20     if (dest_pid < 1) {
21         ready(currentProcess, &readyQueueHead, &readyQueueTail, STATE_READY);
22         return -1;
23     }
24
25     int index = (dest_pid % PCBTABLESIZE);
26     struct pcb* foundProcess = pcbTable + index;
27
28     if (index < 0 || foundProcess->pid != dest_pid) {
29         ready(currentProcess, &readyQueueHead, &readyQueueTail, STATE_READY);
30         return -1;
31     }
32
33     if (dest_pid == currentProcess->pid) {
34         ready(currentProcess, &readyQueueHead, &readyQueueTail, STATE_READY);
35         return -2;
36     }
37
38     // check that the foundProcess is in the queue of receivers for the currentP
   rocess
39     // check that the foundProcess is in the queue of receivers who are receivin
   g any
40     if (foundProcess->head == &(currentProcess->recvQHead) || foundProcess->head
    == &recvAnyQueueHead) {
41         // put the value of num in the recv's address
42         unsigned int * from_pid = (unsigned int *) *(foundProcess->args + 1);
43         unsigned long * dest_num = (unsigned long *) *(foundProcess->args + 2);
44         if ((*from_pid != currentProcess->pid) && *from_pid) {
45             ready(currentProcess, &readyQueueHead, &readyQueueTail, STATE_READY)
   ;
46             return -3;
47         }
48
49         *dest_num = num;
50         *from_pid = currentProcess->pid;
51         // remove foundProcess from the queue it was in
52         removeNthPCB(foundProcess);
53         // put both processes on the ready queue
54         ready(currentProcess, &readyQueueHead, &readyQueueTail, STATE_READY);
55         ready(foundProcess, &readyQueueHead, &readyQueueTail, STATE_READY);
```

```
56
57      } else {
58          // put the currentprocess on the sender's queue of the process we found
59          ready(currentProcess, &(foundProcess->sendQHead), &(foundProcess->sendQT
    ail), STATE_SEND);
60      }
61      return 0;
62  }
63
64  /*
65   * ===  FUNCTION  ======================================================================
    ========
66   *         Name: recv
67   *  Description:  recv num from from_pid process returs 0 is successful, else re
    turns -1 if invalid pid or index out of bounds
68   *                 -2 if receiving from itself, and -3 if any other errors
69   * ======================================================================================
    ========
70   */
71  int recv(unsigned int *from_pid, unsigned long *num, struct pcb * currentProcess
    ) {
72      // Non-zero PID
73      if (*from_pid) {
74          int index = (*from_pid % PCBTABLESIZE);
75          struct pcb* foundProcess = pcbTable + index;
76          if (index < 0 || foundProcess->pid != *from_pid) {
77              ready(currentProcess, &readyQueueHead, &readyQueueTail, STATE_READY)
    ;
78              return -1;
79          }
80          if (*from_pid == currentProcess->pid) {
81              ready(currentProcess, &readyQueueHead, &readyQueueTail, STATE_READY)
    ;
82              return -2;
83          }
84
85          if ((unsigned long) (foundProcess->head) == (unsigned long) &(currentPro
    cess->sendQHead)) {
86
87              int dest_pid = (int) *(foundProcess->args + 1);
88              unsigned long from_num = (unsigned long) *(foundProcess->args + 2);
89              if (dest_pid != currentProcess->pid) {
90                  ready(currentProcess, &readyQueueHead, &readyQueueTail, STATE_RE
    ADY);
91                  return -3;
92              }
93
94              *num = from_num;
95              removeNthPCB(foundProcess);
96              ready(currentProcess, &readyQueueHead, &readyQueueTail, STATE_READY)
    ;
97              ready(foundProcess, &readyQueueHead, &readyQueueTail, STATE_READY);
98          } else {
99              ready(currentProcess, &(foundProcess->recvQHead), &(foundProcess->re
    cvQTail), STATE_RECV);
100         }
101     } else {
102         // from_pid is zero.
103         // check the sender's queue of the current Process and take the first th
    ing from there
104         struct pcb *foundProcess = next(&(currentProcess->sendQHead), &(currentP
    rocess->sendQTail));
105         if (!foundProcess) {
```

```
106             ready(currentProcess, &recvAnyQueueHead, &recvAnyQueueTail, STATE_RE
    CV);
107             return 0;
108         }
109
110         int dest_pid = (int) *(foundProcess->args + 1);
111         unsigned long from_num = (unsigned long) *(foundProcess->args + 2);
112
113         if (dest_pid != currentProcess->pid) {
114             return -3;
115         }
116         *num = from_num;
117         *from_pid = foundProcess->pid;
118         ready(currentProcess, &readyQueueHead, &readyQueueTail, STATE_READY);
119         ready(foundProcess, &readyQueueHead, &readyQueueTail, STATE_READY);
120
121     }
122     return 0;
123
124 }
```

```
1  /* signal.c - support for signal handling
2     This file is not used until Assignment 3
3   */
4
5  #include <xeroskernel.h>
6  #include <xeroslib.h>
7
8
9  int signal(int pid, int sig_no);
10 void sigtramp(void (*handler)(void*), void *cntx);
11
12 int signal(int pid, int sig_no) {
13     int index = (pid % PCBTABLESIZE);
14     struct pcb* process = pcbTable + index;
15     if (index < 0 || process->pid != pid || pid < 0) {
16         return -1;
17     }
18     if (sig_no < 0 || sig_no > SIGNALMAX) {
19         return -2;
20     }
21
22     int state = process->state;
23     if (state == STATE_WAITING) {
24         removeNthPCB(process);
25         ready(process, &readyQueueHead, &readyQueueTail, STATE_READY);
26         process->rc = -2;
27     }
28     if (state == STATE_RECV || state == STATE_SEND || state == STATE_SLEEP)  {
29         removeNthPCB(process);
30         ready(process, &readyQueueHead, &readyQueueTail, STATE_READY);
31         process->rc  = -362;
32     }
33     if (state == STATE_DEV_WAITING) {
34         process->rc  = -362;
35     }
36     unsigned long originalMask = process->signalBitMask;
37     unsigned long one = 1;
38     process->signalBitMask = originalMask | (one << sig_no);
39     return 0;
40 }
41
42 void sigtramp(void (*handler)(void*), void *cntx) {
43     handler(cntx);
44     syssigreturn(cntx);
45 }
```

```
1  /* sleep.c : sleep device
2     This file does not need to modified until assignment 2
3     */
4
5  #include <xeroskernel.h>
6  #include <xeroslib.h>
7  struct pcb *sleepQueueHead; //points to sleep queue head
8  struct pcb *sleepQueueTail; //points to sleep queue tail
9
10 int insertIntoSleepQ(struct pcb * process, unsigned int tick);
11 unsigned int sleep(unsigned int ms, struct pcb * process);
12 void tick(void);
13
14 /*
15  * ===  FUNCTION  ======================================================
    ========
16  *         Name:  sleep
17  *  Description:  adds process to the the sleep queue if ms > 0
18  * ======================================================================
    ========
19  */
20 unsigned int sleep(unsigned int ms, struct pcb * process){
21     if(!ms){
22         process->rc = 0;
23         ready(process, &readyQueueHead, &readyQueueTail, STATE_READY);
24         return 0;
25     }
26     unsigned int tick = ms/TICKLENGTH + 1;
27     process->state = STATE_SLEEP;
28     insertIntoSleepQ(process, tick);
29     return 1;
30 }
31
32 /*
33  * ===  FUNCTION  ======================================================
    ========
34  *         Name:  tick
35  *  Description:  subtracts one tick off the head of the sleep queue, and adds p
    rocess that have tick <= 0 to ready Queue
36  * ======================================================================
    ========
37  */
38 void tick(void){
39     if (sleepQueueHead) {
40         //kprintf("SleepQueueHead->tick: %d\n", sleepQueueHead->tick);
41         sleepQueueHead->tick -= 1;
42     }
43     while (sleepQueueHead && sleepQueueHead->tick <= 0) {
44         struct pcb *process = next(&sleepQueueHead, &sleepQueueTail);
45         process->rc = 0;
46         ready(process, &readyQueueHead, &readyQueueTail, STATE_READY);
47     }
48 }
49
50 /*
51  * ===  FUNCTION  ======================================================
    ========
52  *         Name:  insertIntoSleepQ
53  *  Description:  inserts into sleep queue following delta list convention
54  * ======================================================================
    ========
55  */
56 int insertIntoSleepQ(struct pcb * process, unsigned int tick){
```

```
57      if(!sleepQueueTail && !sleepQueueHead){
58          sleepQueueHead = process;
59          sleepQueueTail = process;
60          process->tick = tick;
61          process->head = &sleepQueueHead;
62          process->tail = &sleepQueueTail;
63          return 1;
64      }
65
66      if(sleepQueueHead && sleepQueueTail){
67          struct pcb *current = sleepQueueHead;
68          struct pcb *prev = NULL;
69          while(current && tick >= current->tick){
70              tick -= current->tick;
71              prev = current;
72              current = current->next;
73          }
74          process->tick = tick;
75
76          if(!prev){
77              sleepQueueHead = process;
78          } else {
79              prev->next = process;
80          }
81          if (current) {
82              current->prev = process;
83              current->tick -= tick;
84          } else {
85              sleepQueueTail = process;
86          }
87
88          process->next = current;
89          process->prev = prev;
90          process->head = &sleepQueueHead;
91          process->tail = &sleepQueueTail;
92          return 1;
93      }
94      kprintf("\n\n one of sleepQueueHead or sleepQueueTail is NULL\n file: sleep.c\n function: insert");
95      return 0;
96  }
97
```

```
1  /* syscall.c : syscalls
2  */
3
4  #include <xeroskernel.h>
5  #include <stdarg.h>
6  #include <stdio.h>
7
8  /* Your code goes here */
9
10
11 int syscall(int call);
12 int syscall2(int call, ...);
13 int syscall3(int call, ... );
14 int syscall4(int call, ... );
15 void sysyield( void );
16 unsigned int syscreate( void (*func)(void), int stack);
17 void sysstop( void *cntx );
18 int sysgetpid( void );
19 void sysputs(char *str);
20 int syskill(int pid, int signalNumber);
21 int syssend(int dest_pid, unsigned long num);
22 int sysrecv(unsigned int *from_pid, unsigned long *num);
23 int syssleep( unsigned int milliseconds );
24 int syssighandler(int signal, void(*newHandler)(void*), void(**oldHandler)(void*
   ));
25 int syssigreturn(void *old_sp);
26
27 /*
28  * ===  FUNCTION  ======================================================
   ========
29  *         Name:  syscall
30  *  Description:  pushes one arguments on to stack
31  * ======================================================================
   ========
32  */
33 int syscall(int call){
34     int result = 0;
35     __asm__ __volatile__("pushl 8(%%ebp)\n\t"
36             "int $67\n\t"
37             "movl %%eax, %0\n\t"
38             "popl %%eax"
39             :"=r"(result)
40             :
41             :"%eax"
42             );
43     return result;
44 }
45
46 /*
47  * ===  FUNCTION  ======================================================
   ========
48  *         Name:  syscall2
49  *  Description:  pushes two arguments on to stack
50  * ======================================================================
   ========
51  */
52 int syscall2(int call, ... ){
53     int result = 0;
54     __asm__ __volatile__("pushl 12(%%ebp)\n\t"
55             "pushl 8(%%ebp)\n\t"
56             "int $67\n\t"
57             "movl %%eax, %0\n\t"
58             "popl %%eax\n\t"
```

```
59              "popl %%eax"
60              : "=r"(result)
61              :
62              :"%eax"
63              );
64      return result;
65  }
66
67  /*
68   * ===  FUNCTION  ======================================================
    ========
69   *         Name:  syscall3
70   *  Description:  pushes three arguments on to stack
71   * ======================================================================
    ========
72   */
73  int syscall3(int call, ... ){
74      int result = 0;
75      __asm__ __volatile__("pushl 16(%%ebp)\n\t"
76              "pushl 12(%%ebp)\n\t"
77              "pushl 8(%%ebp)\n\t"
78              "int $67\n\t"
79              "movl %%eax, %0\n\t"
80              "popl %%eax\n\t"
81              "popl %%eax\n\t"
82              "popl %%eax"
83              : "=r"(result)
84              :
85              :"%eax"
86              );
87      return result;
88  }
89
90  /*
91   * ===  FUNCTION  ======================================================
    ========
92   *         Name:  syscall4
93   *  Description:  pushes four arguments on to stack
94   * ======================================================================
    ========
95   */
96  int syscall4(int call, ... ){
97      int result = 0;
98      __asm__ __volatile__(
99              "pushl 20(%%ebp)\n\t"
100             "pushl 16(%%ebp)\n\t"
101             "pushl 12(%%ebp)\n\t"
102             "pushl 8(%%ebp)\n\t"
103             "int $67\n\t"
104             "movl %%eax, %0\n\t"
105             "popl %%eax\n\t"
106             "popl %%eax\n\t"
107             "popl %%eax"
108             : "=r"(result)
109             :
110             :"%eax"
111             );
112     return result;
113 }
114
115 unsigned int syscreate( void (*func)(void), int stack){
116     if (!func) {
117         return 0;
```

```
118     }
119     return syscall3(CREATE, func, stack);
120 }
121
122 void sysyield( void ){
123     syscall(YIELD);
124 }
125
126 void sysstop( void* cntx ){
127     syscall(STOP);
128 }
129
130 int sysgetpid( void ) {
131     return syscall(GETPID);
132 }
133
134 void sysputs(char *str) {
135     if (!str) {
136         return;
137     }
138     syscall2(PUTS, str);
139 }
140
141 int syskill(int pid, int signalNumber) {
142     int res = syscall3(KILL, pid, signalNumber);
143     if (res == -1) {
144         return -712;
145     } else if (res == -2) {
146         return -651;
147     } else {
148         return 0;
149     }
150 }
151
152 int syssend(int dest_pid, unsigned long num) {
153     if (dest_pid < 1) {
154         return -1;
155     }
156     return syscall3(SEND, dest_pid, num);
157 }
158
159 int sysrecv(unsigned int *from_pid, unsigned long *num) {
160     if (!from_pid || !num) {
161         return -3;
162     }
163     return syscall3(RECEIVE, from_pid, num);
164 }
165
166 int syssleep( unsigned int milliseconds ){
167     return syscall2(SLEEP, milliseconds);
168
169 }
170
171 int sysgetcputimes(struct processStatuses *ps) {
172     if (!ps) {
173         return -2;
174     }
175     return syscall2(CPU_TIMES, ps);
176 }
177
178 int syssighandler(int signal, void(*newHandler)(void *), void(**oldHandler)(void
    *)) {
179         return syscall4(SIG_HANDLER, signal, newHandler, oldHandler);
```

```
180
181 }
182
183 int syssigreturn(void *old_sp) {
184     return syscall2(SIG_RETURN,  old_sp);
185 }
186
187 int syswait(int pid) {
188     return syscall2(WAIT, pid);
189 }
190
191 int sysopen(int device_no){
192     return syscall2(OPEN, device_no);
193 }
194
195 int sysclose(int fd){
196     return syscall2(CLOSE, fd);
197 }
198
199 int syswrite(int fd, void *buff, int bufflen){
200     return syscall4(WRITE, fd, buff, bufflen);
201 }
202
203 int sysread(int fd, void *buff, int bufflen){
204     return syscall4(READ, fd, buff, bufflen);
205 }
206
207 int sysioctl(int fd, unsigned long command, ...){
208         int val = 0;
209         va_list list;
210         va_start(list, command);
211         val += va_arg(list, int);
212         va_end(list);
213         return syscall4(IOCTL, fd, command, val);
214 }
```

```
1  /* user.c : User processes
2  */
3
4  #include <xeroskernel.h>
5  #include <xeroslib.h>
6
7  #define BUF_MAX 100
8  char *username = "cs415\n";
9  char *password = "EveryoneGetsAnA\n";
10 char *pscom = "ps";
11 char *excom = "ex";
12 char *kcom = "k";
13 char *acom = "a";
14 char *tcom = "t";
15 char *psand = "ps&";
16 char *exand = "ex&";
17 char *kand = "k&";
18 char *aand = "a&";
19 char *tand = "t&";
20 int shellPid;
21 int alarmTicks;
22
23
24 void shell(void);
25 void  root(void);
26 void psf(void);
27 void exf(void);
28 void kf(int pid);
29 void alarmHandler(void *cntx);
30 void alarm(void);
31 void t(void);
32 int parseString(char *inBuf, int inBufSize, char *outBuf, int outBufSize);
33
34 void  root( void ) {
35     int error = 0;
36     char ubuf[BUF_MAX];
37     char pbuf[BUF_MAX];
38
39     while (1) {
40         // Banner
41         sysputs("\nWelcome to Xeros - an experimental OS\n");
42
43         // Open keyboard in non echo mode
44         int fd = sysopen(0);
45         if (fd == -1) {
46             kprintf("Error opening keyboard\n");
47             for(;;);
48         }
49
50         // Turn keyboard echoing on;
51         error = sysioctl(fd, 56);
52         if (error == -1) {
53             kprintf("Error turning keyboard echoing on\n");
54             for(;;);
55         }
56
57         sysputs("Username: ");
58         int bytes = sysread(fd, &ubuf[0], BUF_MAX - 1);
59         if (!bytes) {
60             kprintf("Sysread returned EOF\n");
61             for(;;);
62         }
63         if (bytes == -1) {
```

```
64              kprintf("Sysread returned an error\n");
65              for(;;);
66          }
67          ubuf[bytes] = NULLCH;
68          // Turn keyboard echoing off;
69          error = sysioctl(fd, 55);
70          if (error == -1) {
71              kprintf("Error turning keyboard echoing off\n");
72              for(;;);
73          }
74          sysputs("Password: ");
75          bytes = sysread(fd, &pbuf[0], BUF_MAX - 1);
76          if (!bytes) {
77              kprintf("Sysread returned EOF\n");
78              for(;;);
79          }
80          if (bytes == -1) {
81              kprintf("Sysread returned an error\n");
82              for(;;);
83          }
84          pbuf[bytes] = NULLCH;
85          error = sysclose(fd); // Just writing this in for testing even though we
     dont actually have to close the fd
86          if (error == -1) {
87              kprintf("Error turning closing device\n");
88              for(;;);
89          }
90          int usercheck = strcmp(&ubuf[0], username);
91          int passcheck = strcmp(&pbuf[0], password);
92          //kprintf("\n user check %d, pass check %d", usercheck, passcheck);
93          //kprintf("\n user in %s, pass in %s", ubuf, pbuf);
94          if (usercheck == 0 && passcheck == 0) {
95              break;
96          }
97      }
98      char buf[BUF_MAX];
99      sprintf(buf, "\n");
100     sysputs(buf);
101
102     shellPid = create(&shell, 8000);
103     int retCode = syswait(shellPid);
104     sprintf(buf, "Syswait retcode%d\n", retCode);
105     sysputs(buf);
106
107 }
108
109 void shell(void) {
110     char stdinput[BUF_MAX];
111     // Open keyboard in echo mode
112     int fd = sysopen(1);
113     if (fd == -1) {
114         kprintf("Error opening keyboard\n");
115         for(;;);
116     }
117
118     while (1) {
119         sysputs("> ");
120         int bytes = sysread(fd, &stdinput[0], BUF_MAX - 1);
121         if (!bytes) {
122             break;
123         }
124         if (bytes == -1) {
125             kprintf("Sysread returned an error\n");
```

```
126             for(;;);
127         }
128         stdinput[bytes++] = NULLCH;
129         char command[bytes];
130         int bytesParsed = parseString(stdinput, bytes, command, bytes);
131         if (bytesParsed == -2) {
132             // GO back to the the beginning of the loop
133             sysputs("Ignoring command");
134             continue;
135         }
136         command[bytesParsed++] = NULLCH;
137         if (!strcmp(command, pscom) || !strcmp(command, psand)) {
138                 psf();
139         } else if (!strcmp(command, excom) || !strcmp(command, exand)) {
140             break;
141         } else if (!strcmp(command, kcom) || !strcmp(command, kand)) {
142             if (bytesParsed < BUF_MAX) {
143                 char arg[BUF_MAX];
144                 bytesParsed += parseString(&stdinput[bytesParsed], BUF_MAX - byt
    esParsed, arg, BUF_MAX);
145                 arg[bytesParsed++] = NULLCH;
146                 int pid = atoi(arg);
147                 int res = syskill(pid, 9);
148                 if (res == -712) {
149                     sprintf(arg, "No such process\n");
150                     sysputs(arg);
151                 }
152             }
153         } else if (!strcmp(command, acom) || !strcmp(command, aand)) {
154             if (bytesParsed < BUF_MAX) {
155                 char arg[BUF_MAX];
156                 bytesParsed += parseString(&stdinput[bytesParsed], BUF_MAX - byt
    esParsed, arg, BUF_MAX);
157                 arg[bytesParsed++] = NULLCH;
158                 int ticks = atoi(arg);
159                 alarmTicks = ticks;
160                 syssighandler(15, &alarmHandler, NULL);
161                 int alarmPid = syscreate(&alarm, 8000);
162                 if (!strcmp(command, acom)) {
163                     syswait(alarmPid);
164                 }
165             }
166         } else if (!strcmp(command, tcom) || !strcmp(command, tand)) {
167             int tpid = syscreate(&t, 8000);
168             if (!strcmp(command, tcom)) {
169                 syswait(tpid);
170             }
171         } else {
172             sysputs("Command not found\n");
173         }
174     }
175     sysputs("Exiting shell...\n");
176 }
177
178 int parseString(char *inBuf, int inBufSize, char *outBuf, int outBufSize) {
179         int bytesRead = 0;
180         char * endInBuf = inBuf + inBufSize;
181         char * endOutBuf = outBuf + outBufSize;
182         while (inBuf < endInBuf && *inBuf == ' ') {
183             inBuf++;
184         }
185         while (inBuf < endInBuf && *inBuf != ' ' && *inBuf != '\n' && outBuf < en
    dOutBuf ) {
```

```
186             *outBuf = *inBuf;
187             outBuf++;
188             inBuf++;
189             bytesRead++;
190         }
191         return bytesRead;
192 }
193
194 void psf(void) {
195     struct processStatuses ps;
196     int procs =  sysgetcputimes(&ps);
197     char buf[100];
198     sprintf(buf, "%4s  %4s  %10s\n", "Pid", "Status", "CpuTime");
199     sysputs(buf);
200     for (int i = 0; i <= procs; i++) {
201         int status = ps.status[i];
202         switch(status) {
203         case STATE_STOPPED:
204             sprintf(buf, "%4d  %4s  %10d\n", ps.pid[i], "STOPPED", ps.cpuTime[i])
    ;
205             break;
206         case STATE_READY:
207             sprintf(buf, "%4d  %4s  %10d\n", ps.pid[i], "READY", ps.cpuTime[i]);
208             break;
209         case STATE_SLEEP:
210             sprintf(buf, "%4d  %4s  %10d\n", ps.pid[i], "SLEEP", ps.cpuTime[i]);
211             break;
212         case STATE_RUNNING:
213             sprintf(buf, "%4d  %4s  %10d\n", ps.pid[i], "RUNNING", ps.cpuTime[i]
    );
214             break;
215         case STATE_RECV:
216             sprintf(buf, "%4d  %4s  %10d\n", ps.pid[i], "RECV", ps.cpuTime[i]);
217             break;
218         case STATE_SEND:
219             sprintf(buf, "%4d  %4s  %10d\n", ps.pid[i], "SENDING", ps.cpuTime[i])
    ;
220             break;
221         case STATE_WAITING:
222             sprintf(buf, "%4d  %4s  %10d\n", ps.pid[i], "WAITING", ps.cpuTime[i])
    ;
223             break;
224         case STATE_DEV_WAITING:
225             sprintf(buf, "%4d  %4s  %10d\n", ps.pid[i], "DEV_WAITING", ps.cpuTim
    e[i]);
226             break;
227         }
228         sysputs(buf);
229     }
230 }
231 void kf(int pid) {}
232
233 void alarmHandler(void *cntx) {
234     char buf[100];
235     sprintf(buf, "ALARM ALARM ALARM\n");
236     sysputs(buf);
237     syssighandler(15, NULL, NULL);
238 }
239
240 void alarm(void) {
241     int sleepTime = alarmTicks * TICKLENGTH;
242     syssighandler(9, &sysstop, NULL);
243     syssleep(sleepTime);
```

```
244     syskill(shellPid, 15);
245 }
246
247 void t(void) {
248     char buf[5];
249     syssighandler(9, &sysstop, NULL);
250     sprintf(buf, "T\n");
251     for (;;) {
252         syssleep(10000);
253         sysputs(buf);
254     }
255 }
```

```
.SUFFIXES: .o .c
#
# Makefile for PC Xeros to compile on a PC running Linux.
#

CCPREFIX =


# Things that need not be changed, usually
OS       = LINUX
DEFS     = -DBSDURG  -DVERBOSE -DPRINTERR
INCLUDE  = -I../h
CFLAGS   = -Wall -Wstrict-prototypes -fno-builtin -c  ${DEFS} ${INCLUDE}
SDEFS    = -D${OS} -I../h -DLOCORE -DSTANDALONE -DAT386
LIB      = ../lib
AS       = $(CCPREFIX)as --32
AR       = $(CCPREFIX)ar
XEROS    = ./xeros
GCC      = $(CCPREFIX)gcc -m32 -march=i386 -std=gnu99
CC       = $(GCC) -D__KERNEL__ -D__ASSEMBLY__
LD       = $(CCPREFIX)ld -m elf_i386
CPP      = $(CC) -E
AWK      = awk


# Use the following line if you want to boot Xeros from floppy diskette
#
BRELOC  = 0x100000
TEXTSPOT= 0x000000
BOOTPLOC= 0x150000

# Ignore these
#BRELOC = 0x150000
#TEXTSPOT= 0x150000

# Linker line, do not modify this, please.
LDSTR    = -e start -Ttext ${TEXTSPOT}


#Do NOT modify these lines
SOBJ = startup.o intr.o
IOBJ = init.o i386.o evec.o kprintf.o
UOBJ = mem.o disp.o ctsw.o syscall.o create.o user.o msg.o sleep.o signal.o
UOBJ += kbd.o di_calls.o

#Add your sources here
MY_OBJ =


# Don't modiy any of this unless you are really sure
all: xeros

xeros: Makefile ${SOBJ} ${IOBJ} ${UOBJ} ${MY_OBJ} ${LIB}/libxc.a
        $(LD) ${LDSTR} ${SOBJ} ${IOBJ} ${UOBJ} ${MY_OBJ} ${LIB}/libxc.a -o ${XER
OS}

clean:
        rm -rf *.o *.bak *.a core errs ${XEROS} ${XEROS}.boot

cleanall:
        rm -rf *.o *.bak *.a core errs ${XEROS} ${XEROS}.boot
        (cd ${LIB}/libxc; make clean)

```

```
63 ${LIB}/libxc.a:
64         (cd ${LIB}/libxc; make install)
65
66 intr.o: ../c/intr.S ../c/xint.s
67         ${CPP} ${SDEFS} ../c/intr.S | ${AS} -o intr.o
68
69 startup.o: ../c/startup.S Makefile
70         ${CPP} ${SDEFS} -DBRELOC=${BRELOC} -DBOOTPLOC=${BOOTPLOC} -DLINUX_XINU .
   ./c/startup.S | ${AS} -o startup.o
71
72 ${IOBJ}:
73         ${CC} ${CFLAGS} ../c/`basename $@ .o`.[c]
74
75 ${UOBJ}:
76         ${CC} ${CFLAGS} ../c/`basename $@ .o`.[c]
77
78 init.o: ../c/init.c ../h/i386.h ../h/xeroskernel.h ../h/xeroslib.h
79 i386.o: ../c/i386.c ../h/i386.h ../h/icu.h ../h/xeroskernel.h ../h/xeroslib.h
80 evec.o: ../c/evec.c ../h/i386.h ../h/xeroskernel.h ../h/xeroslib.h
81 kprintf.o: ../c/kprintf.c ../h/i386.h ../h/xeroskernel.h ../h/xeroslib.h
82 mem.o: ../c/mem.c ../h/xeroskernel.h ../h/xeroslib.h
83 disp.o: ../c/disp.c ../h/xeroskernel.h ../h/xeroslib.h
84 ctsw.o: ../c/ctsw.c ../h/xeroskernel.h ../h/xeroslib.h
85 syscall.o: ../c/syscall.c ../h/xeroskernel.h ../h/xeroslib.h
86 create.o: ../c/create.c ../h/xeroskernel.h ../h/xeroslib.h
87 user.o: ../c/user.c ../h/xeroskernel.h ../h/xeroslib.h
88 msg.o: ../c/msg.c ../h/xeroskernel.h ../h/xeroslib.h
89 sleep.o: ../c/sleep.c ../h/xeroskernel.h ../h/xeroslib.h
90 signal.o: ../c/signal.c ../h/xeroskernel.h ../h/xeroslib.h
91 # at the bottom of the compile/Makefile, add:
92 kbd.o: ../c/kbd.c ../h/xeroskernel.h ../h/kbd.h ../h/xeroslib.h ../h/i386.h
93 di_calls.o: ../c/di_calls.c ../h/xeroskernel.h
```

```
1  /* i386.h - DELAY */
2
3  #define NBPG             4096
4  #define KERNEL_STACK     (4*4096)
5
6
7  #define NID              48
8  #define NGD               8
9
10 #define IRQBASE          32      /* base ivec for IRQ0                    */
11
12 struct idt {
13         unsigned short  igd_loffset;
14         unsigned short  igd_segsel;
15         unsigned int    igd_rsvd : 5;
16         unsigned int    igd_mbz : 3;
17         unsigned int    igd_type : 5;
18         unsigned int    igd_dpl : 2;
19         unsigned int    igd_present : 1;
20         unsigned short  igd_hoffset;
21 };
22
23 #define IGDT_TASK         5      /* task gate IDT descriptor              */
24 #define IGDT_INTR        14      /* interrupt gate IDT descriptor         */
25 #define IGDT_TRAPG       15      /* Trap Gate                             */
26
27
28 /* Segment Descriptor */
29
30 struct sd {
31         unsigned short  sd_lolimit;
32         unsigned short  sd_lobase;
33         unsigned char   sd_midbase;
34         unsigned int    sd_perm : 3;
35         unsigned int    sd_iscode : 1;
36         unsigned int    sd_isapp : 1;
37         unsigned int    sd_dpl : 2;
38         unsigned int    sd_present : 1;
39         unsigned int    sd_hilimit : 4;
40         unsigned int    sd_avl : 1;
41         unsigned int    sd_mbz : 1;             /* must be '0' */
42         unsigned int    sd_32b : 1;
43         unsigned int    sd_gran : 1;
44         unsigned char   sd_hibase;
45 };
46
47 #define sd_type          sd_perm
48
49 /* System Descriptor Types */
50
51 #define SDT_INTG         14      /* Interrupt Gate        */
52
53 /* Segment Table Register */
54 struct segtr {
55         unsigned int    len : 16;
56         unsigned int    addr : 32;
57 };
58
59 /*
60  * Delay units are in microseconds.
61  */
62 #define DELAY(n)                                            \
63 {                                                           \
```

```
64          extern int cpudelay;                                \
65          register int i;                                     \
66          register long N = (((n)<<4) >> cpudelay);           \
67                                                              \
68          for (i=0;i<=4;i++)                                  \
69             {                                                \
70             N = (((n) << 4) >> cpudelay);                    \
71             while (--N > 0) ;                                \
72             }                                                \
73  }
74
75  #define HOLESIZE        (600)
76  #define HOLESTART       (640 * 1024)
77  #define HOLEEND         ((1024 + HOLESIZE) * 1024)
78  /* Extra 600 for bootp loading, and monitor */
79
80  /* Code grokked from cs452 (waterloo) libs
81   */
82  #define TIMER_IRQ       0       /* IRQ of counter 0 on timer 1 */
83  #define TIMER_1_PORT    0x040   /* 8253 Timer #1 */
84  #define TIMER_2_PORT    0x048   /* 8253 Timer #2 (EISA only) */
85
86  #ifndef TIMER_FREQ
87  #define TIMER_FREQ      1193182
88  #endif
89  #define TIMER_DIV(x) ((TIMER_FREQ+(x)/2)/(x))
90
91  /*
92   * Macros for specifying values to be written into a mode register.
93   */
94  #define TIMER_CNTR0     (TIMER_1_PORT + 0) /* timer 0 counter port */
95  #define TIMER_CNTR1     (TIMER_1_PORT + 1) /* timer 1 counter port */
96  #define TIMER_CNTR2     (TIMER_1_PORT + 2) /* timer 2 counter port */
97  #define TIMER_MODE      (TIMER_1_PORT + 3) /* timer mode port */
98  #define         TIMER_SEL0      0x00    /* select counter 0 */
99  #define         TIMER_SEL1      0x40    /* select counter 1 */
100 #define         TIMER_SEL2      0x80    /* select counter 2 */
101 #define         TIMER_INTTC     0x00    /* mode 0, intr on terminal cnt */
102 #define         TIMER_ONESHOT   0x02    /* mode 1, one shot */
103 #define         TIMER_RATEGEN   0x04    /* mode 2, rate generator */
104 #define         TIMER_SQWAVE    0x06    /* mode 3, square wave */
105 #define         TIMER_SWSTROBE  0x08    /* mode 4, s/w triggered strobe */
106 #define         TIMER_HWSTROBE  0x0a    /* mode 5, h/w triggered strobe */
107 #define         TIMER_LATCH     0x00    /* latch counter for reading */
108 #define         TIMER_LSB       0x10    /* r/w counter LSB */
109 #define         TIMER_MSB       0x20    /* r/w counter MSB */
110 #define         TIMER_16BIT     0x30    /* r/w counter 16 bits, LSB first */
111 #define         TIMER_BCD       0x01    /* count in BCD */
112
113
114 /* Some helpful prototypes */
115 void initPIT( int divisor );
116 void end_of_intr( void );
117 void enable_irq( unsigned int,  int);
118
119
```

```
1  // PROVIDED CODE
2  #define KEY_UP   0x80            /* If this bit is on then it is a key   */
3                                   /* up event instead of a key down event */
4
5  /* Control code */
6  #define LSHIFT  0x2a
7  #define RSHIFT  0x36
8  #define LMETA   0x38
9
10 #define LCTL    0x1d
11 #define CAPSL   0x3a
12
13
14 /* scan state flags */
15 #define INCTL           0x01    /* control key is down          */
16 #define INSHIFT         0x02    /* shift key is down            */
17 #define CAPSLOCK        0x04    /* caps lock mode               */
18 #define INMETA          0x08    /* meta (alt) key is down       */
19 #define EXTENDED        0x10    /* in extended character mode   */
20
21 #define EXTESC          0xe0    /* extended character escape    */
22 #define NOCHAR  256
23
24
25
26 /*  Normal table to translate scan code  */
27 unsigned char   kbcode[] = { 0,
28           27,  '1',  '2',  '3',  '4',  '5',  '6',  '7',  '8',  '9',
29          '0',  '-',  '=', '\b', '\t',  'q',  'w',  'e',  'r',  't',
30          'y',  'u',  'i',  'o',  'p',  '[',  ']', '\n',    0,  'a',
31          's',  'd',  'f',  'g',  'h',  'j',  'k',  'l',  ';', '\'',
32          '`',    0, '\\',  'z',  'x',  'c',  'v',  'b',  'n',  'm',
33          ',',  '.',  '/',    0,    0,    0,  ' ' };
34
35 /* captialized ascii code table to tranlate scan code */
36 unsigned char   kbshift[] = { 0,
37           0,  '!',  '@',  '#',  '$',  '%',  '^',  '&',  '*',  '(',
38          ')',  '_',  '+', '\b', '\t',  'Q',  'W',  'E',  'R',  'T',
39          'Y',  'U',  'I',  'O',  'P',  '{',  '}', '\n',    0,  'A',
40          'S',  'D',  'F',  'G',  'H',  'J',  'K',  'L',  ':',  '"',
41          '~',    0,  '|',  'Z',  'X',  'C',  'V',  'B',  'N',  'M',
42          '<',  '>',  '?',    0,    0,    0,  ' ' };
43 /* extended ascii code table to translate scan code */
44 unsigned char   kbctl[] = { 0,
45           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
46           0,   31,    0, '\b', '\t',   17,   23,    5,   18,   20,
47          25,   21,    9,   15,   16,   27,   29, '\n',    0,    1,
48          19,    4,    6,    7,    8,   10,   11,   12,    0,    0,
49           0,    0,   28,   26,   24,    3,   22,    2,   14,   13 };
50
51 // END OF PROVIDED CODE
52
53
54
55
56
57 #define MAX_KBUF_SIZE 4
58 #define READ_PORT 0x60
59 #define CTRL_PORT 0x64
60 struct dataRequest {
61         int status;
62     char *buff;
63     int size;
```

```
64      int bytesRead;
65      struct dataRequest *next;
66      struct dataRequest *prev;
67      struct pcb * blockedProc;
68      int (*done)(void);
69  };
70
71  int kb_open(const struct devsw* const dvBlock, int majorNum);
72  int kb_close(const struct devsw* const dvBlock);
73  int kb_ioctl(const struct devsw* const dvBlock, unsigned long command, int val);
74  int kb_write(const struct devsw * const dvBlock);
75  int kb_read(const struct devsw * const dvBlock, struct pcb * const process, void
     *buff, int size);
```

```
1  /* xeroskernel.h - disable, enable, halt, restore, isodd, min, max */
2
3  #ifndef XEROSKERNEL_H
4  #define XEROSKERNEL_H
5
6  /* Symbolic constants used throughout Xinu */
7
8  typedef char    Bool;        /* Boolean type                  */
9  typedef unsigned int size_t; /* Something that can hold the value of
10                                * theoretical maximum number of bytes
11                                * addressable in this architecture.
12                                */
13 #define FALSE   0       /* Boolean constants             */
14 #define TRUE    1
15 #define EMPTY   (-1)    /* an illegal gpq                */
16 #define NULL    0       /* Null pointer for linked lists */
17 #define NULLCH '\0'     /* The null character            */
18
19
20 /* Universal return constants */
21
22 #define OK            1         /* system call ok               */
23 #define SYSERR       -1         /* system call failed           */
24 #define EOF          -2         /* End-of-file (usu. from read) */
25 #define TIMEOUT      -3         /* time out  (usu. recvtim)     */
26 #define INTRMSG      -4         /* keyboard "intr" key pressed  */
27                                 /*  (usu. defined as ^B)        */
28 #define BLOCKERR     -5         /* non-blocking op would block  */
29
30 /* Functions defined by startup code */
31
32
33 void           bzero(void *base, int cnt);
34 void           bcopy(const void *src, void *dest, unsigned int n);
35 void           disable(void);
36 unsigned short getCS(void);
37 unsigned char  inb(unsigned int);
38 void           init8259(void);
39 int            kprintf(char * fmt, ...);
40 void           lidt(void);
41 void           outb(unsigned int, unsigned char);
42 void           set_evec(unsigned int xnum, unsigned long handler);
43
44
45 // Global Constants
46 #define PCBTABLESIZE 32
47 #define SIGNALMAX 31
48 #define DEVICETABLESIZE 2
49 #define FDTSIZE 4
50
51 //constants to track state that a process is in
52 #define STATE_STOPPED 0
53 #define STATE_READY 1
54 #define STATE_SLEEP 22
55 #define STATE_RUNNING 23
56 #define STATE_RECV 29
57 #define STATE_SEND 34
58 #define STATE_WAITING 47
59 #define STATE_DEV_WAITING 92
60
61 //Time slice constant
62 #define TIMESLICE 100 // must change both these values. one is dependent on the
   other
```

```
63  #define TICKLENGTH 10 // assuming 1 tick is 10ms based on the TIMESLICE constant
     above
64
65  // init.c functions
66  extern struct pcb *pcbTable;
67  extern struct devsw *deviceTable;
68  // mem.c functions
69  extern void kmeminit(void);
70  extern void *kmalloc(int size);
71  extern void kfree(void *ptr);
72  // disp.c functions
73  extern void dispatch(void);
74  extern struct pcb *readyQueueHead;// Head of pcb ready queue
75  extern struct pcb *readyQueueTail;// Tail of pcb ready queue
76  extern struct pcb *recvAnyQueueHead;//head of recv any queue
77  extern struct pcb *recvAnyQueueTail;//tail of recv any queue
78  extern struct pcb *stopQueueHead;// Head of pcb queue stopped
79  extern struct pcb *stopQueueTail;// Tail of pcb queue stopped
80  extern struct pcb *idleProcessHead; //points to idle process
81  extern struct pcb *idleProcessTail; //points to idle process
82  extern struct pcb *sleepQueueHead; //points to idle process
83  extern struct pcb *sleepQueueTail; //points to idle process
84  extern struct pcb* next(struct pcb **head, struct pcb **tail);
85  extern int ready(struct pcb *process, struct pcb **head, struct pcb **tail, int
    state);
86  extern void removeNthPCB(struct pcb *process);
87
88  //stores cpu context
89  struct CPU {
90      unsigned long edi;
91      unsigned long esi;
92      unsigned long ebp;
93      unsigned long esp;
94      unsigned long ebx;
95      unsigned long edx;
96      unsigned long ecx;
97      unsigned long eax;
98      unsigned long iret_eip;
99      unsigned long iret_cs;
100     unsigned long eflags;
101 };
102 struct FD{
103     int index;
104     int majorNum;
105     struct devsw *dvBlock;
106     int status;
107     char *name;
108     struct FD *prev;
109     struct FD *next;
110 };
111 //stores process information
112 struct pcb {
113     int pid; // process ID
114     int index; // index of pcb inside process table
115     int reuseCount;
116     int state;
117     unsigned long *args; // pointer to system call type and arguments on process
     stack
118     unsigned long *memoryStart; //pointer to the allocated stack for the process
119     unsigned long sp;//most current stack pointer for the process
120     int rc;
121     unsigned int tick;
122     long cpuTime;
```

```
123     unsigned long signalBitMask;
124     void (*sigFunctions[SIGNALMAX+1])(void*); //array of function pointers for s
    ignal
125     struct CPU *cpuState;// pointer to the cpu struct
126     struct pcb *next;// pointer to next pcp in the queue
127     struct pcb *prev;
128     struct pcb **head; // pointer to the global variable that is a pointer to th
    e head of the queue this pcb block belongs to
129     struct pcb **tail; // pointer to the global variable that is a pointer to th
    e tail of the queue this pcb block belongs to
130     struct pcb *sendQHead;// pointer to the list of pcb's that want to send to t
    his pcb
131     struct pcb *sendQTail;
132     struct pcb *recvQHead;// pointer to the list of pcb's that want to recv from
     this pcb
133     struct pcb *recvQTail;
134     struct pcb *waitQHead;// pointer to the list of pcb's that are waiting for t
    his process to die
135     struct pcb *waitQTail;
136     struct FD FDT[FDTSIZE]; //pointer to the FDT for this process always size fo
    ur initiated with sysopen
137 };
138
139 // enum representing type of system calls available
140 enum SystemEvents {
141     CREATE,
142     YIELD,
143     STOP,
144     GETPID,
145     PUTS,
146     KILL,
147     SEND,
148     RECEIVE,
149     TIMER_INT,
150     SLEEP,
151     CPU_TIMES,
152     SIG_HANDLER,
153     SIG_RETURN,
154     WAIT,
155     OPEN,
156     CLOSE,
157     WRITE,
158     READ,
159     IOCTL,
160     KEYBOARD
161 };
162
163 struct processStatuses {
164   int  pid[PCBTABLESIZE];      // The process ID
165   int  status[PCBTABLESIZE];   // The process status
166   long  cpuTime[PCBTABLESIZE]; // CPU time used in milliseconds
167 };
168
169 struct devsw{
170     int dvnum;
171     char *dvname;
172     int (*dvopen)(const struct devsw* const, int);
173     int (*dvclose)(const struct devsw* const);
174     int (*dvread)(const struct devsw* const, struct pcb*, void*, int);
175     int (*dvwrite)(const struct devsw* const);
176     int (*dvioctl)(const struct devsw* const, unsigned long, int);
177     int *dvcsr;
178     int *dvivec;
```

```
179     int *dvovec;
180     int (*dviint)(void);
181     int (*dvoint)(void);
182 };
183 // di_calls.c functions
184 extern int di_open(struct pcb *process, int device_no);
185 extern int di_close(struct pcb *process, int fd);
186 extern int di_write(struct pcb *process, int fd, unsigned char *buff, int size);
187 extern int di_read(struct pcb *process, int fd, unsigned char *buff, int size);
188 extern int di_ioctl(struct pcb *process, int fd, unsigned long command, int val)
    ;
189
190 // ctsw.c functions
191 extern int contextswitch(struct pcb* process);
192 extern void contextinit(void);
193 // create.c
194 extern int create(void (*func)(void), int stackSize);
195 extern int createIdle(void (*func)(void), int stackSize);
196 // syscall.c
197 extern int syscall(int call);
198 extern int syscall2(int call, ...);
199 extern void sysyield(void);
200 extern void sysstop(void *cntx);
201 extern unsigned int syscreate(void (*func)(void), int stack);
202 extern int sysgetpid( void );
203 extern void sysputs(char *str);
204 extern int syskill(int pid, int signalNumber);
205 extern int syssend(int dest_pid, unsigned long num);
206 extern int sysrecv(unsigned int *from_pid, unsigned long *num);
207 extern int syssleep( unsigned int milliseconds );
208 extern int sysgetcputimes(struct processStatuses *ps);
209 extern int syssighandler(int signal, void(*newHandler)(void*), void(**oldHandler
    )(void*));
210 extern int syssigreturn(void *old_sp);
211 extern int syswait(int pid);
212 extern int sysopen(int device_no);
213 extern int sysclose(int fd);
214 extern int syswrite(int fd, void *buff, int bufflen);
215 extern int sysread(int fd, void *buff, int bufflen);
216 extern int sysioctl(int fd, unsigned long command, ...);
217 // user.c
218 extern void root(void);
219 // msg.c
220 extern int send(int dest_pid, unsigned long num, struct pcb * currentProcess);
221 extern int recv(unsigned int *from_pid, unsigned long *num, struct pcb * current
    Process);
222 // sleep.c
223 extern unsigned int sleep(unsigned int ms, struct pcb * process);
224 extern void tick(void);
225 // signal.c
226 extern int signal(int pid, int sig_no);
227 extern void sigtramp(void (*handler)(void*), void *cntx);
228 // kbd.c
229 int kbd_read_in(void);
230 int kb_open(const struct devsw* const dvBlock, int majorNum);
231 int kb_close(const struct devsw* const dvBlock);
232 int kb_ioctl(const struct devsw* const dvBlock, unsigned long command, int val);
233 int kb_read(const struct devsw * const dvBlock, struct pcb * const process, void
     *buff, int size);
234 int kb_write(const struct devsw * const dvBlock);
235
236
237
```

```
238
239
240
241 /* Anything you add must be between the #define and this comment */
242 #endif
```