WEENSY OS

```
#include "mpos-kern.h"
#include "x86.h"
#include "lib.h"

/*****************************************************************************
* mpos-kern
*
*   This is the miniprocos's kernel.
*   It sets up a process descriptor for the initial application, then runs
*   that application, and responds to its system calls.
*
*****************************************************************************/

// The kernel is loaded starting at 0x100000.
// The miniprocos applications are also available in RAM in packed form.
// The kernel loads one of those applications into memory starting at 0x200000.
// It also allocates 1/4 MB for each possible miniprocess's stack, starting at
// 0x280000.
// Each process's stack grows down from the top of its stack space.

#define PROC1_STACK_ADDR    0x280000
#define PROC_STACK_SIZE     0x040000

// MINIPROCOS MEMORY MAP
//
// +--------------------------+--------------+----------------------------+-/
// |   Base Memory (640K)    | I/O Memory  | Kernel            Kernel |
// |        (unused)          |             | Code + Data        Stack |
// +--------------------------+--------------+----------------------------+-/
// 0                   0xA0000      0x100000                 0x200000
//
//        /-+----------------+------------+------------+------------+---/
//          |  Application  | Miniproc 1 | Miniproc 2 | Miniproc 3 |
//          | Code + Globals |     Stack |     Stack |     Stack |
//        /-+----------------+------------+------------+------------+---/
//      0x200000        0x280000    0x2C0000    0x300000    0x340000
//                          |          |          |
//                 PROC1_STACK_ADDR    |    PROC1_STACK_ADDR
//                                     |    + 2*PROC_STACK_SIZE
//                                     |
//                            PROC1_STACK_ADDR
//                           + PROC_STACK_SIZE
//
// There is also a shared 'cursorpos' variable, located at 0x60000 in the
// kernel's data area.  (This is used by 'app_printf' in mpos-app.h.)


// A process descriptor for each possible miniprocess.
// Note that proc_array[0] is never used.
// The main application process descriptor is proc_array[1].
static process_t proc_array[NPROCS];

// A pointer to the currently running process.
// This is kept up to date by the run() function, in mpos-x86.c.
process_t *current;



/*****************************************************************************
* start
*
*   Initialize the hardware and process descriptors to empty, then load
*   and run the first process.
*
*****************************************************************************/

void
start(void)
{
const char *s;
int whichprocess;
pid_t i;

// Initialize process descriptors as empty
memset(proc_array, 0, sizeof(proc_array));
for (i = 0; i < NPROCS; i++) {
proc_array[i].p_pid = i;
proc_array[i].p_state = P_EMPTY;
}

// The first process has process ID 1.
current = &proc_array[1];

// Set up x86 hardware, and initialize the first process's
// special registers.  This only needs to be done once, at boot time.
// All other processes' special registers can be copied from the
// first process.
segments_init();
special_registers_init(current);

// Erase the console, and initialize the cursor-position shared
// variable to point to its upper left.
console_clear();

// Figure out which program to run.
cursorpos = console_printf(cursorpos, 0x0700, "Type '1' to run mpos-app, or '2' to run mpos-app2.");
do {
whichprocess = console_read_digit();
} while (whichprocess != 1 && whichprocess != 2);
console_clear();

// Load the process application code and data into memory.
// Store its entry point into the first process's EIP
// (instruction pointer).
program_loader(whichprocess - 1, &current->p_registers.reg_eip);

// Set the main process's stack pointer, ESP.
current->p_registers.reg_esp = PROC1_STACK_ADDR + PROC_STACK_SIZE;

// Mark the process as runnable!
current->p_state = P_RUNNABLE;

// Switch to the main process using run().
run(current);
}



/*****************************************************************************
* interrupt
*
*   This is the weensy interrupt and system call handler.
*   New system calls are implemented by code in this function.
*
*****************************************************************************/

static pid_t do_fork(process_t *parent);

void
interrupt(registers_t *reg)
{
// The processor responds to a system call interrupt by saving some of
// the application's state on the kernel's stack, then jumping to
// kernel assembly code (in mpos-int.S, for your information).
// That code saves more registers on the kernel's stack, then calls
// interrupt().  The first thing we must do, then, is copy the saved
```

```
// registers into the 'current' process descriptor.
current->p_registers = *reg;

switch (reg->reg_intno) {

case INT_SYS_GETPID:
// The 'sys_getpid' system call returns the current
// process's process ID.  System calls return results to user
// code by putting those results in a register.  Like Linux,
// we use %eax for system call return values.  The code is
// surprisingly simple:
current->p_registers.reg_eax = current->p_pid;
run(current);

case INT_SYS_FORK:
// The 'sys_fork' system call should create a new process.
// You will have to complete the do_fork() function!
current->p_registers.reg_eax = do_fork(current);
run(current);

case INT_SYS_YIELD:
// The 'sys_yield' system call asks the kernel to schedule a
// different process.  (MiniprocOS is cooperatively
// scheduled, so we need a special system call to do this.)
// The schedule() function picks another process and runs it.
schedule();

case INT_SYS_EXIT:
// 'sys_exit' exits the current process, which is marked as
// non-runnable.
// The process stored its exit status in the %eax register
// before calling the system call.  The %eax REGISTER has
// changed by now, but we can read the APPLICATION's setting
// for this register out of 'current->p_registers'.

current->p_state = P_EMPTY;
current->p_exit_status = current->p_registers.reg_eax;

if(current->p_wait_pid)
{
proc_array[current->p_wait_pid].p_registers.reg_eax = current->p_exit_status;
proc_array[current->p_wait_pid].p_state = P_RUNNABLE;
current->p_wait_pid = 0;
}
schedule();

case INT_SYS_WAIT: {
// 'sys_wait' is called to retrieve a process's exit status.
// It's an error to call sys_wait for:
// * A process ID that's out of range (<= 0 or >= NPROCS).
// * The current process.
// * A process that doesn't exist (p_state == P_EMPTY).
// (In the Unix operating system, only process P's parent
// can call sys_wait(P).  In MiniprocOS, we allow ANY
// process to call sys_wait(P).)

pid_t p = current->p_registers.reg_eax;
if (p <= 0 || p >= NPROCS || p == current->p_pid
|| proc_array[p].p_state == P_EMPTY)
current->p_registers.reg_eax = -1;
else if (proc_array[p].p_state == P_ZOMBIE)
current->p_registers.reg_eax = proc_array[p].p_exit_status;
else{
current->p_state = P_BLOCKED;
proc_array[p].p_wait_pid = current->p_pid;
}

schedule();
}

default:
while (1)
/* do nothing */;

}
}



/*****************************************************************************
* do_fork
*
*   This function actually creates a new process by copying the current
*   process's state.  In MiniprocOS, a process's state consists of
*   (1) its registers, and (2) its stack -- that's it.  All processes share
*   THE SAME code and global variables.  (So really we should call them
*   "miniprocesses" or something.)
*   The parent process is passed in as an argument.  The function should
*   return the process ID of the child process, or -1 if the function can't
*   create a child process.
*   Your job is to fill it in!
*
*****************************************************************************/

static void copy_stack(process_t *dest, process_t *src);

static pid_t
do_fork(process_t *parent)
{
// YOUR CODE HERE!
// First, find an empty process descriptor.  If there is no empty
//   process descriptor, return -1.  Remember not to use proc_array[0].
// Then, initialize that process descriptor as a running process
//   by copying the parent process's registers and stack into the
//   child.  Copying the registers is simple: they are stored in the
//   process descriptor in the 'p_registers' field.  Copying the stack
//   is a little more involved; see the copy_stack function, below.
//   The child process's registers will be equal to the parent's, with
//   two differences:
//   * reg_esp    The child process's stack pointer will point into
//               its stack, rather than the parent's.  copy_stack
//               should arrange this.
//   * ???????    There is one other difference.  What is it?  (Hint:
//               What should sys_fork() return to the child process?)
// You need to set one other process descriptor field as well.
// Finally, return the child's process ID to the parent.

///// USER DEFINED VALUES
process_t *child = NULL;
int foundEmptyProcessDescriptor=0; // 0 = NO, 1 = YES
int i=1; // do not start at proc_array[0]

///// FIND EMPTY PROCESS DESCRIPTOR
for(i; i< NPROCS; i++) // loop through all process descriptors
{
if (proc_array[i].p_state == P_EMPTY) // if a process descriptor is empty
{
child = &proc_array[i]; // child points to that empty process descriptor
foundEmptyProcessDescriptor=1; // set flag
break;
}
}

///// FORK CHILD
if (foundEmptyProcessDescriptor)
{
```

```
child->p_registers = parent->p_registers; // copy registers
copy_stack(child,parent); // copy stack
child->p_state = P_RUNNABLE; // set child as runnable
child->p_registers.reg_eax = 0; // return 0 to child
return child->p_pid; // parent receives child pid
}
else
{
return -1;
}
}

static void
copy_stack(process_t *dest, process_t *src)
{
uint32_t src_stack_bottom, src_stack_top;
uint32_t dest_stack_bottom, dest_stack_top;

// YOUR CODE HERE!
// This function copies the 'src' process's stack into the 'dest'
// process's stack region.  Then it sets 'dest's stack pointer to
// correspond to 'src's stack pointer.

// For example, assume that 'src->p_pid == 1' and 'dest->p_pid == 2'.
// Then this code should change this memory setup:

//        Miniproc 1 Stack    Miniproc 2 Stack
//   /--+-------------------+-------------------+--/
//      |        ABXLQPAOSRJ|                   |
//   /--+-------------------+-------------------+--/
//  0x280000     ^      0x2C0000            0x300000
//               |
//    src->p_registers.reg_esp
//         == 0x29A4CC

// into this:

//        Miniproc 1 Stack    Miniproc 2 Stack
//   /--+-------------------+-------------------+--/
//      |        ABXLQPAOSRJ|        ABXLQPAOSRJ|
//   /--+-------------------+-------------------+--/
//  0x280000     ^      0x2C0000     ^      0x300000
//               |                   |
//    src->p_registers.reg_esp   dest->p_registers.reg_esp
//         == 0x29A4CC           == 0x2DA4CC
//                           == 0x300000 + (0x29A4CC - 0x2C0000)

// You may implement this however you like, but we found it easiest
// to express with variables that locate the bottom and top of each
// stack.  In our examples, the variables would equal:

//   /--+-------------------+-------------------+--/
//      |        ABXLQPAOSRJ|        ABXLQPAOSRJ|
//   /--+-------------------+-------------------+--/
//  0x280000     ^      0x2C0000     ^      0x300000
//               |         ^         |         ^
//        src_stack_bottom  | dest_stack_bottom |
//          == 0x29A4CC     |  == 0x2DA4CC      |
//                   src_stack_top      dest_stack_top
//                    == 0x2C0000        == 0x300000

// Your job is to figure out how to calculate these variables,
// and then how to actually copy the stack.  (Hint: use memcpy.)
// We have done one for you.

///// CALCULATE STACK VARIABLES
src_stack_top = PROC1_STACK_ADDR + (PROC_STACK_SIZE) * (src->p_pid);
src_stack_bottom = src->p_registers.reg_esp;
dest_stack_top = PROC1_STACK_ADDR + (dest->p_pid)*PROC_STACK_SIZE;
dest_stack_bottom = dest_stack_top + (src_stack_bottom - src_stack_top);

///// COPY STACK
memcpy((void *)dest_stack_bottom,(void *)src_stack_bottom, (src_stack_top-src_stack_bottom));
///// SET DEST ESP
dest->p_registers.reg_esp = dest_stack_bottom;
}



/*****************************************************************************
* schedule
*
*   This is the weensy process scheduler.
*   It picks a runnable process, then context-switches to that process.
*   If there are no runnable processes, it spins forever.
*
*****************************************************************************/

void
schedule(void)
{
pid_t pid = current->p_pid;
while (1) {
pid = (pid + 1) % NPROCS;
if (proc_array[pid].p_state == P_RUNNABLE)
run(&proc_array[pid]);
}
}
```

WEENSY OS

```
#ifndef WEENSYOS_MPOS_KERN_H
#define WEENSYOS_MPOS_KERN_H
#include "mpos.h"
#include "x86.h"

// Process state type
typedef enum procstate {
P_EMPTY = 0,                    // The process table entry is empty
// (i.e. this is not a process)
P_RUNNABLE,                     // This process is runnable
P_BLOCKED,                      // This process is blocked
P_ZOMBIE                // This process has exited, but no one
// has called sys_wait() yet
} procstate_t;

// Process descriptor type
typedef struct process {
pid_t p_pid;                    // Process ID

registers_t p_registers;        // Current process state: registers,
// stack location, EIP, etc.
// 'registers_t' defined in x86.h
procstate_t p_state;            // Process state; see above
int p_exit_status;              // Process's exit status (if it has
// exited and p_state == P_ZOMBIE)

pid_t p_wait_pid; // pid that we wait on
} process_t;


// Top of the kernel stack
#define KERNEL_STACK_TOP        0x80000

// Functions defined in mpos-kern.c
void interrupt(registers_t *reg);
void schedule(void);

// Functions defined in mpos-x86.c
void segments_init();
void special_registers_init(process_t *proc);
void console_clear(void);
int console_read_digit(void);
// Function defined in mpos-loader.c
void program_loader(int programnumber, uint32_t *entry_point);

extern process_t *current;
void run(process_t *proc) __attribute__((noreturn));

#endif
```