#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]);

}

}

#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