/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* EXCEPTIONS.C \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Exceptions are synchronous events generated when processor detects any predefined condition while executing instructions.

Exception.c handles different exceptions that can occur while exceuting processes, such as syscall, tlbTrap and progTrap.

The kaya exceptions.c also deals with 8 different syscalls starting from the creation of a process upto suspending when needed.

In addition, this module will help identify whether a process should be waiting or signaling. On top of dealing with these

situations, exceptions also handles clock exceptions - "your time is over or wait for I/O device," as these features are very

important for the control flow of process handling.

==== Written By Solomon G and Edwin Cervantes ==== \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* #INCLUDE MODULES \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* .h files to include \*/

#include "../h/const.h"

#include "../h/types.h"

/\* .e files to include \*/

#include "../e/initial.e"

#include "../e/scheduler.e"

#include "../e/pcb.e"

#include "../e/asl.e"

/\* include the µmps2 library \*/

#include "/usr/local/include/umps2/umps/libumps.e"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* GLOBAL VARIABLES \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

cpu\_t TODStart;

cpu\_t currentTOD;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* LOCAL FUNCTIONS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

HIDDEN void sysCall1(state\_PTR state);

HIDDEN void sysCall2();

HIDDEN void sysCall3(state\_PTR state);

HIDDEN void sysCall4(state\_PTR state);

HIDDEN void syscall5(state\_PTR state);

HIDDEN void sysCall6(state\_PTR state);

HIDDEN void sysCall7(state\_PTR state);

HIDDEN void sysCall8(state\_PTR state);

HIDDEN void passUpOrDie(state\_PTR state, int trap);

HIDDEN void copyState(state\_PTR src, state\_PTR dest);

HIDDEN void sysCall2Helper(pcb\_PTR p);

HIDDEN void getSyscall(int sysReq, state\_PTR called);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Parameters: state\_PTR \*/

/\* Returns: N/A \*/

/\* BAM conduct a context switch. AKA wake up somewhere else \*/

void contextSwitch(state\_PTR state){

LDST(state);

}

/\* Parameters: N/A\*/

/\* Returns: N/A\*/

/\* PROGTRAP defined as 1\*/

void progTrapHandler(){

state\_PTR oldS = (state\_PTR) PRGMTRAPOLDAREA; /\*Find the area\*/

passUpOrDie(oldS, PROGTRAP); /\*Pass up the process to its appropriate handler or terminate it \*/

}

/\* Parameters: N/A \*/

/\* Returns: N/A \*/

/\* TLBTRAP defined as 0 \*/

void tlbHandler(){

state\_PTR oldS = (state\_PTR) TBLMGMTOLDAREA; /\*Find the area \*/

passUpOrDie(oldS, TLBTRAP); /\*Pass up the process to its appropriate handler or terminate it \*/;

}

/\* Parameters: N/A \*/

/\* Returns: N/A \*/

void syscallHandler(){

unsigned int tempCause, calledStatus;

int sysReq;

state\_PTR called, prog;

called = (state\_PTR) SYSCALLOLDAREA; /\* Getting from memory, look at const.h \*/

/\* Pulling what syscall number was called, stored in a0 of old. Made it unsigned because our numbers can't be negative \*/

sysReq = called -> s\_a0;

calledStatus = called -> s\_status; /\* Just storing our status \*/

/\* Program Trap check \*/

/\* Steps:

-Copy the state from oldSys over to oldProg

- Set the cause register in oldProg to reflect a Privileged Instruction exception

-ProgTrap() \*/

if( (sysReq>0) && (sysReq<9) && ((calledStatus & USERMODEON) != ALLOFF)){

prog = (state\_PTR) PRGMTRAPOLDAREA;

copyState(called, prog);

tempCause = (prog -> s\_cause) & ~(PRIVATEINSTUC);

(prog -> s\_cause) = (tempCause | (10 << 2));

progTrapHandler();

}

called -> s\_pc = called -> s\_pc + 4; /\* We must increment to the next instruction. We know to add 4 from Machine Org \*/

getSyscall(sysReq, called); /\* Lets go find out how to handle this Syscall \*/

}

/\* Parameters: int, state\_PTR \*/

/\* Returns: N/A \*/

HIDDEN void getSyscall(int sysReq, state\_PTR called){

switch(sysReq){

            case CREATEPROCESS:

                sysCall1(called);

            break;

            case TERMINATEPROCESS:

                sysCall2();

            break;

            case VERHOGEN:

                sysCall3(called);

            break;

            case PASSEREN:

                sysCall4(called);

            break;

            case SPECTRAPVEC:

                syscall5(called);

            break;

            case GETCPUTIME:

                sysCall6(called);

            break;

            case WAITCLOCK:

                sysCall7(called);

            break;

            case WAITIO:

                sysCall8(called);

            break;

            default:

                passUpOrDie(called, SYSTRAP); /\* SYSTRAP defined as 2 \*/

            break;

    }

}

/\* Syscall 1 - Create Process \*/

/\* Parameters: state\_PTR \*/

/\* Returns: N/A \*/

/\* Function: creates a new process by allocating a new pcb. \*/

HIDDEN void sysCall1 (state\_PTR state){

state\_PTR temp;

pcb\_PTR p;

p = allocPcb();/\* Get a new process \*/

if (p != NULL){

temp = (state\_PTR) state->s\_a1; /\* copy the content from the state's $a1 register to the new pcb\_t's state \*/

copyState(temp, &(p->p\_state));

insertChild(currentProcess, p);/\* if the process has a parent, it is inserted and then placed in the ready queue \*/

insertProcQ(&(readyQueue), p);

processCount++; /\* there is now n+1 running processes \*/

currentProcess->p\_state.s\_v0 = SUCCEEDED; /\* The success of the new process \*/

contextSwitch(state);

} else {

currentProcess->p\_state.s\_v0 = FAILED; /\* If there are no free processes, failure of a new allocated pcb\_t \*/

contextSwitch(state);

}

contextSwitch(state);

}

/\* Syscall 2 - Terminate Process \*/

/\* Parameters: N/A \*/

/\* Returns: N/A \*/

/\* Function: terminates an executing process. This function will also allow the process's children to be terminated recursively. \*/

HIDDEN void sysCall2() {

/\* Check if there are no children. If so, process is decremented. Remove current process, and free up a pcb\_t \*/

if(emptyChild(currentProcess)){

outChild(currentProcess);

freePcb(currentProcess);

processCount--;

} else {

sysCall2Helper(currentProcess);

}

currentProcess = NULL;

/\* call the scheduler to reschedule a new process \*/

scheduler();

}

/\* Syscall 3 - Verhogen \*/

/\* Parameters: state\_PTR \*/

/\* Returns: N/A \*/

/\* Function: performs a V operation on a specified synchronization semaphores. \*/

HIDDEN void sysCall3(state\_PTR state) {

pcb\_PTR p = NULL;

int\* semaphore = (int\*) state->s\_a1; /\* The semaphore is placed in the a1 register of the passed in state\_t \*/

(\*semaphore)++; /\* Increment the semaphore \*/

if((\*semaphore) <= 0){ /\*If semaphore is less than or equal to 0, unblock\*/

p = removeBlocked(semaphore); /\* Unblock the next process \*/

if(p != NULL){ /\* current process is then placed in the ready queue \*/

insertProcQ(&readyQueue, p); /\* place it in the ready queue \*/

p->p\_semAdd = NULL;

softBlockCount--; /\* Decrement softBlockCount \*/

}

}

/\* load the new state of the requested process - BAM \*/

contextSwitch(state);

}

/\* Syscall 4 \*/

/\* Parameters: state\_PTR\*/

/\* Returns: N/A\*/

/\* Function: performs a P operation on a specified synchronization semaphores. \*/

HIDDEN void sysCall4(state\_PTR state) {

cpu\_t time1;

int\* semaphore = (int\*) state->s\_a1; /\* get the semaphore in the s\_a1 \*/

(\*semaphore)--; /\* decrement the semaphore \*/

if ((\*semaphore) <= -1) {

int eTime = time1 - TODStart; /\* Elapsed time \*/

STCK(time1);

currentProcess->p\_time = currentProcess->p\_time + eTime; /\* New process time \*/

copyState(state, &(currentProcess->p\_state)); /\* copy from the old syscall area to the new process's state \*/

insertBlocked(semaphore, currentProcess); /\* Block it \*/

softBlockCount++;

currentProcess = NULL;

scheduler();

}

/\* if semaphore is not less than 0, load the new state - BAM \*/

contextSwitch(state);

}

/\* Syscall 5 - Specify Exceptions State Vector\*/

/\* Parameters: state\_PTR \*/

/\* Returns: N/A \*/

/\* Function: this sets up the new TLB, PROG and SYS TRAP for handling exceptions. If they have already been there, sys2 is called. \*/

HIDDEN void syscall5(state\_PTR state) {

int a1 = (int) state->s\_a1; /\* get the exception from the a1 register \*/

switch(a1) {

case TLBTRAP:/\* 0 \*/

/\* incase the new TLB has been established. We kill \*/

if(currentProcess->newTLB != NULL) {

sysCall2();

scheduler();

}

/\* store the syscall area state in the new tlb \*/

currentProcess->newTLB = (state\_PTR) state->s\_a3;

/\* store the syscall area state in the old tlb \*/

currentProcess->oldTLB = (state\_PTR) state->s\_a2;

break;

case PROGTRAP:/\* 1 \*/

/\* incase the new prog has been established, we kill \*/

if(currentProcess->newPgm != NULL) {

sysCall2();

scheduler();

}

/\* store the pgm area state in the new pgm \*/

currentProcess->newPgm = (state\_PTR) state->s\_a3;

currentProcess->oldPgm = (state\_PTR) state->s\_a2;

break;

case SYSTRAP:/\* 2 \*/

/\* incase the new systrap has been established, we kill\*/

if(currentProcess->newSys != NULL) {

sysCall2();

scheduler();

}

/\* store the syscall area state in the new pgm \*/

currentProcess->newSys = (state\_PTR) state->s\_a3;

/\* store the syscall area state in the old pgm \*/

currentProcess->oldSys = (state\_PTR) state->s\_a2;

break;

}

contextSwitch(state);

}

/\* Syscall 6 - calculate CPU time \*/

/\* Parameters: state\_PTR \*/

/\* Returns: N/A \*/

/\* Function: when syscall6 is called, processor's time used will be calculated and be returned in v0 \*/

HIDDEN void sysCall6(state\_PTR state) {

cpu\_t t;

int eTime;

STCK(t);

eTime = (t - TODStart); /\* Calculate elapsed time \*/

(currentProcess->p\_time) = (currentProcess->p\_time) + (eTime); /\* Do math \*/

(state->s\_v0) = currentProcess->p\_time; /\* Store the state in the pcb\_t's v0 register \*/

STCK(TODStart);

contextSwitch(state);

}

/\* Syscall 7 - wait for clock \*/

/\* Parameters: state\_PTR \*/

/\* Returns: N/A \*/

/\* Function: performs a P operation on the pseudo-clock timer semaphore.

The semaphore is also V'ed every 100 milliseconds by the nucleus. Thus,

Afterwards, we call the scheduler to schedule a new job \*/

HIDDEN void sysCall7(state\_PTR called) {

int \*semaphore = (int\*) &(semdTable[PSEUDOCLOCK]); /\* sempahore index of the clock \*/

(\*semaphore) = (\*semaphore)-1; /\* Decrement semaphore \*/

if ((\*semaphore) < 0) {

cpu\_t time1;

int eTime;

STCK(time1);

eTime = time1 - TODStart;

currentProcess->p\_time = currentProcess->p\_time + eTime;

copyState(called, &(currentProcess->p\_state)); /\* Copy from the old syscall area into the new pcb\_state \*/

insertBlocked(semaphore, currentProcess); /\* block the process \*/

softBlockCount++; /\* Increment the number of waiting processes \*/

currentProcess = NULL;

scheduler();

}

/\* if there's not P operation, BAM - context switch \*/

contextSwitch(called);

}

/\* Syscall 8 \*/

/\* Parameters: state\_PTR \*/

/\* Returns: N/A \*/

/\* Function: performs a P operation on the semaphore requested by a device. \*/

HIDDEN void sysCall8(state\_PTR state){

int lineNum, deviceNum, terminal, index;

int \*semaphore;

lineNum = state->s\_a1; /\* line number in the a1 register \*/

deviceNum = state->s\_a2; /\* device number in the a2 register \*/

terminal = state->s\_a3; /\* terminal read/write in the a1 register \*/

/\*These devices can't do IO request \*/

if (lineNum < DISKNUM || lineNum > TERMINT){

sysCall2(); /\* terminate \*/

}

/\* calculate the index of the device sema4 that did IO request and if it's a terminal read \*/

if (lineNum == TERMINT && terminal == TRUE){

index = PERDEV \* (lineNum - DEVNOSEM + terminal) + deviceNum;

/\* calculate index without the terminal read \*/

} else {

index = PERDEV \* (lineNum - DEVNOSEM) + deviceNum;

}

semaphore = &(semdTable[index]);

/\* p operation \*/

(\*semaphore)--;

if ((\*semaphore) < 0){

cpu\_t time2;

STCK(time2);

int eTime;

eTime = time2 - TODStart;

currentProcess->p\_time = currentProcess->p\_time + eTime;

/\* block the current process \*/

insertBlocked(semaphore, currentProcess);

copyState(state, &(currentProcess->p\_state));

softBlockCount = softBlockCount + 1;

currentProcess = NULL;

/\* call scheduler to get a new processor \*/

scheduler();

}

contextSwitch(state);

}

/\* Allows caller to store the address of 2 processor states

Only occurs for TLBtrap, or ProgTrap or SYScall >= 9 occurs.

Has a sys5 for that trap been called? - Check if sys/tlb/progNEW != NULL in the current process, meaning sys5 was executed.

No: Terminate process and all its offspring(sys2)

Yes: Copy the state that caused the exception from oldXXX -> location specified in their PCB

LDST(current->newSys) \*/

/\* Parameters: state\_PTR, int \*/

/\* Returns: N/A \*/

HIDDEN void passUpOrDie(state\_PTR oldS, int trap){

switch(trap){

case TLBTRAP: /\* 0 \*/

if(currentProcess -> newTLB != NULL){

copyState(oldS, currentProcess -> oldTLB);

contextSwitch(currentProcess -> newTLB);

}

break;

case PROGTRAP: /\* 1 \*/

if(currentProcess -> newPgm != NULL){

copyState(oldS, currentProcess -> oldPgm);

contextSwitch(currentProcess ->newPgm);

}

break;

case SYSTRAP: /\* 2 \*/

if(currentProcess -> newSys != NULL){

copyState(oldS, currentProcess -> oldSys);

contextSwitch(currentProcess -> newSys);

}

break;

}

/\* no cases match Therefore none of the handlers have been "set up". KILL EVERYONE \*/

sysCall2();

}

/\* sysCall2Helper - terminate children \*/

/\* Parameters: pcb\_PTR \*/

/\* Returns: N/A \*/

/\* Function: performs a tail recursively termination on current process, killing all of its children. \*/

HIDDEN void sysCall2Helper(pcb\_PTR first){

while(!emptyChild(first)){

sysCall2Helper(removeChild(first)); /\*Our favorite, recursion.\*/

}

if (first->p\_semAdd != NULL){ /\* check the pcb\_b has a semaphore address \*/

int\* sema4 = first->p\_semAdd; /\* get the sema4 \*/

outBlocked(first); /\* when sempahore address is found, call outBlocked \*/

/\* Handle semaphore count when unblocking

i.e. softBlockCount and/or (\*sem)--

\*/

if (sema4 >= &(semdTable[0]) && sema4 <= &(semdTable[PSEUDOCLOCK])){

softBlockCount =softBlockCount - 1;

} else {

(\*sema4)++;

}

}

if ((first) == currentProcess){

outChild(currentProcess);

currentProcess = NULL;

}

if ((first->p\_semAdd) == NULL){

outProcQ(&(readyQueue), first);

}

freePcb(first);/\* no more offspring, free the process \*/

processCount--;

}

/\* Function: copy state. Helper function to copy the contents (SHALLOW COPY) \*/

HIDDEN void copyState(state\_PTR src, state\_PTR dest) {

int i;

/\* id \*/

dest->s\_asid = src->s\_asid;

/\* register cause \*/

dest->s\_cause = src->s\_cause;

/\* pc \*/

dest->s\_pc = src->s\_pc;

/\* status register \*/

dest->s\_status = src->s\_status;

/\* each register \*/

for (i=0; i < STATEREGNUM; i++){

dest->s\_reg[i] = src->s\_reg[i];

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END OF INITIAL.C \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/