**XV6 Kernel thread support // adding clone and join system calls in xv6**

**#clone()**

Located in proc.c, clone() will direct the thread to point to the user stack, receive the fcn

to the appropriate register and pass the arguments of that fcn as well. The PC is also given a

fake address of 0xFFFFFFFF. It finally sets the state of that thread's PID to RUNNABLE while

using the acquire() and release() lock functions.

**#join()**

Located in proc.c, join() mimics wait(), while making sure that the thread will release itself

and kill the process.

**#Thread Library**

The threadLibrary was built in ulib.c. thread\_create(), thread\_join() make use of the system calls clone() and join(). thread\_create creates a new user stack, and supplies it to the clone() function. There is a struct of threadTable array that stores all the threads we create. As the join() begins, the threads are removed from the array.

**#Test assets**

The MAKEFILE was modified to add two test assets: testUserCalls.c and testThreadLib.c.

testUserCalls will directly test clone() and join() system calls by simply keeping a volatile global

counter that should be incremented by '1' by our fcn addToCounter(). It will create 10 threads, and then join them.

testThreadLib will make use of the threadLibrary built in ulib.c and add the value of our counter to itself 10 times through a for loop.

#Qemu: Compiling yields no errors or warnings. The following XV6 files were modified:

**1. MAKEFILE**

UPROGS=\

\_cat\

\_echo\

\_forktest\

\_grep\

\_init\

\_kill\

\_ln\

\_ls\

\_mkdir\

\_rm\

\_sh\

\_stressfs\

\_usertests\

\_wc\

\_zombie\

\_testUserCalls\

\_testThreadLib\

EXTRA=\

mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\

ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\

printf.c umalloc.c\

README dot-bochsrc \*.pl toc.\* runoff runoff1 runoff.list\

.gdbinit.tmpl gdbutil\

**2. defs.h**

// proc.c

**int clone(void(\*fnc)(void\*), void\*, void\*);**

**int join(void \*\*);**

**3. user.h**

**int clone(void(\*fnc)(void\*), void \* arg, void \* stack);**

**int join(void \*\* stack);**

**// ulib.c**

int stat(char\*, struct stat\*);

char\* strcpy(char\*, char\*);

void \*memmove(void\*, void\*, int);

char\* strchr(const char\*, char c);

int strcmp(const char\*, const char\*);

void printf(int, char\*, ...);

char\* gets(char\*, int max);

uint strlen(char\*);

void\* memset(void\*, int, uint);

void\* malloc(uint);

void free(void\*);

int atoi(const char\*);

//Lets add the thread library here so the user an access them

**int thread\_create(void(\*fcn)(void\*),void \*arg);**

**int thread\_join(void);**

//Now the locks

**void lock\_acquire(struct \_\_lock\_t \*);**

**void lock\_release(struct \_\_lock\_t \*);**

**void lock\_init(struct \_\_lock\_t \*);**

**4. ulib.c**

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* NEW THREAD LIBRARY \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

struct user\_thread threads[64];

//We are going initialize the table, and create 64 of them.

int thread\_table\_init = 0;

//Function call that creates the thread

int thread\_create(void(\*fcn)(void\*), void \*arg){

//There might be an instance where the initial values of the threads

//have not been initialized, so lets make sure our table is initialized

if(thread\_table\_init != 0){

int x;

for(x=0; x < NPROC; x++){

threads[x].pid = 0;

threads[x].ustack = 0;

threads[x].used = 0;

}

thread\_table\_init = 1;

}

//init the pid, and allocate space

int pid;

void \* newStack = malloc(KSTACKSIZE);

//Call system call clone, and add the threads to the list, return the pid

pid = clone((void \*) fcn, (void \*)arg, (void\*) newStack);

//Lets add it to our table;

int x;

for(x=0; x < NPROC; x++){ //For the number of processes we have

if(threads[x].used == 0){

//If this thread is not being used, then get the address from the pid

//and point to the user stack, and mark it as used.

threads[x].pid = (int) &pid;

threads[x].ustack = newStack;

threads[x].used = 1;

}

}

return pid;

}

//Function call that will JOIN the threads

int thread\_join(void){

int x,p;

for(x=0; x < NPROC; x++){

if(threads[x].used == 1){

p = join(&threads[x].ustack);

//If the ustack from joining is greater than 0, then remove this thread.

if(p > 0){

int y;

void \* ustack;

for(y=0; y < NPROC; y++){

if(threads[y].used && threads[y].pid == (int)&p){

//If this thread is being used, and it matches its pid, then remove it.

ustack = threads[y].ustack;

free(ustack); //Free allocation

threads[y].pid = 0;

threads[y].ustack = 0;

threads[y].used = 0;

}

}

break;

}

}

}

return p;

}

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* LOCKS ARE HERE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

void lock\_acquire(struct \_\_lock\_t \* lock){

volatile uint x = (uint) 1;

//Let's use the atomic function xchg to see if our lock flag is ready

//to be acquired, wait until its acquired.

while(xchg(&lock->flag, x) == x)

;

}

void lock\_release(struct \_\_lock\_t \* lock){

lock->flag = 0;

}

void lock\_init(struct \_\_lock\_t \* lock){

//init state of the lock is: 0 (not being used)

lock->flag = 0;

}

**5. usys.S**

**SYSCALL(clone)**

**SYSCALL(join)**

**6. syscall.h**

**// System call numbers**

**#define SYS\_clone 22**

**#define SYS\_join 23**

**7. syscall.c**

**extern int sys\_clone(void);**

**extern int sys\_join(void);**

**[SYS\_clone] sys\_clone,**

**[SYS\_join] sys\_join**

**8. sysproc.c**

int

sys\_clone(void){

//Init the variables;

void \* fcn = 0;

void \* arg = 0;

void \* stack = 0;

//Let's pass the user arguments to XV6 using argint()

if(argint(0, (int \*) &fcn) < 0){

return -1;

}

if(argint(1, (int \*) &arg) < 0){

return -1;

}

if(argint(2, (int \*) &stack) < 0){

return -1;

}

//Now let's return the function clone();

return clone(fcn, arg, stack);

}

int

sys\_join(void){

//Init the stack;

void \*\* stack = 0;

//Let's pass the user arguments to XV6 using argint()

if(argint(0, (int \*) &stack) < 0){

return -1;

}

return join(stack);

}

**9. proc.c**

int

clone(void(\*fcn)(void\*), void \*arg, void\*stack){

//Initialize our variables

int pid;

struct proc \*newProcess;

int \*ustack = stack + PGSIZE - 4;

//Make room for the process.

if((newProcess = allocproc()) == 0){

return -1;

}

if (proc->isThread == 0) {

newProcess->parent = proc;

}

else {

newProcess->parent = proc->parent;

}

//Set the size of it

newProcess->sz = proc->sz;

\*newProcess->tf = \*proc->tf;

newProcess->isThread = 1;

//Lets copy the process' stuff

// newProcess->parent = 0;

newProcess->pthread = proc;

newProcess->ustack = stack;

newProcess->pgdir = proc->pgdir;

//get the registers, and give it the function to run.

newProcess->tf->eax = 0;

newProcess->tf->ebp = (int) ustack - 4;

newProcess->tf->esp = (int) ustack - 4;

newProcess->ustack = stack;

//We are going to move the newProcess to the top of the stack,

//And push the arguments in while giving it a phony return address.

newProcess->tf->esp = (int)stack + 4092;

\*((int \*)(newProcess->tf->esp)) = (int)arg;

\*((int \*)(newProcess->tf->esp - 4)) = 0xFFFFFFFF;

newProcess->tf->esp -= 4;

//Assign the function to the eip register.

newProcess->tf->eip = (int) fcn;

//Same as in fork() \*\*\*\*\*\*\*\*\*\*\*\* down here.

int x;

for(x = 0; x < NOFILE; x++){

if(proc->ofile[x]){

newProcess->ofile[x] = filedup(proc->ofile[x]);

}

}

newProcess->cwd = idup(proc->cwd);

safestrcpy(newProcess->name, proc->name, sizeof(proc->name));

pid = newProcess->pid;

//Let it run.

acquire(&ptable.lock);

newProcess->state = RUNNABLE;

release(&ptable.lock);

return pid;

}

//This is going to be like our wait() function.

int

join(void \*\* stack){

struct proc \*p;

int havekids;

int pid;

acquire(&ptable.lock);

for(;;){

havekids = 0;

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

if(p->parent != proc || p->isThread != 1){

continue;

}

havekids = 1;

if(p->state == ZOMBIE){

pid = p->pid;

kfree(p->kstack);

p->kstack = 0;

p->state = UNUSED;

p->pid = 0;

p->parent = 0;

p->pthread = 0;

\*stack = p->ustack;

p->ustack = 0;

p->name[0] = 0;

p->killed = 0;

release(&ptable.lock);

return pid;

}

}

if(!havekids || proc->killed){

release(&ptable.lock);

return -1;

}

sleep(proc, &ptable.lock);

}

return 0;

}

**10. proc.h**

struct proc {

uint sz; // Size of process memory (bytes)

pde\_t\* pgdir; // Page table

char \*kstack; // Bottom of kernel stack for this process

enum procstate state; // Process state

volatile int pid; // Process ID

struct proc \*parent; // Parent process

struct trapframe \*tf; // Trap frame for current syscall

struct context \*context; // swtch() here to run process

void \*chan; // If non-zero, sleeping on chan

int killed; // If non-zero, have been killed

struct file \*ofile[NOFILE]; // Open files

struct inode \*cwd; // Current directory

char name[16]; // Process name (debugging)

**struct proc \*pthread;**

**void \* ustack;**

**int isThread;**

};

**11. testUserCalls.c**

#include "types.h"

#include "stat.h"

#include "user.h"

#include "fs.h"

#include "fcntl.h"

#include "syscall.h"

#include "traps.h"

volatile int globalCounter = 0;

void addToCounter(){

globalCounter++;

printf(1, "counter is: %x\n", globalCounter);

exit();

}

int

main(int argc, char \*argv[])

{

int pid;

void \* stack[10];

printf(1, "Running testUserCalls:\n");

globalCounter++;

printf(1, "before cloning counter is: %x\n", globalCounter);

int x;

for(x=0; x<10; x++){

stack[x] = malloc(4096);

pid = clone((void \*) &addToCounter, (void \*) &globalCounter, (void \*) stack[x]);

printf(1, "user pid: %d\n", pid);

}

for(x=0; x<10; x++){

printf(1, "join pid %d\n", join((void \*\*) &stack[x]));

}

globalCounter++;

printf(1, "joined\n");

printf(1, "The globalCounter should be 12 (C): and it is = %x\n", globalCounter);

exit();

return 0;

}

**12. testThreadLib.c**

#include "types.h"

#include "user.h"

#include "fs.h"

#include "fcntl.h"

#include "syscall.h"

#include "traps.h"

//Our global counter

void printWord(void \* number){

\*(int\*) number = \*(int\*)number + (\*(int\*)number);

printf(1, "The Word is: %x\n", \*((int \*) number));

exit();

}

int

main(int argc, char \*argv[])

{

int counter = 1;

printf(1, "Testing Thread Library:\n");

int x;

for(x =0; x < 10; x++){

thread\_create((void \*)&printWord, &counter);

}

int y;

for(y =0; y < 10; x++){

thread\_join();

}

printf(1, "Finished Joined.\n");

exit();

return 0;

}