CS 380 Finalized thread test WSL QEMU assignment

**user.h**

#ifndef \_USER\_USER\_H\_

#define \_USER\_USER\_H\_

#include "types.h"

#include "user.h"

// Ticket lock type

typedef struct {

unsigned next;

unsigned now\_serving;

} lock\_t;

// clone/join syscalls

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

int join(void \*\*);

// Thread library

int thread\_create(void (\*start\_routine)(void\*, void\*), void \*arg1, void \*arg2);

int thread\_join(void);

// Lock interface

void lock\_init(lock\_t \*lk);

void lock\_acquire(lock\_t \*lk);

void lock\_release(lock\_t \*lk);

#endif // \_USER\_USER\_H\_

**syscall.h**

// add under existing SYS\_ definitions:

#define SYS\_clone 22

#define SYS\_join 23

**proc.h**

struct proc {

...

// new fields for threads:

int is\_thread; // true if this proc shares pgdir

int ref\_count; // number of sharers of pgdir

...

};

// function prototypes

int clone(void (\*fcn)(void\*,void\*), void \*arg1, void \*arg2, void \*stack);

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

**sysproc.c**

//make sure these are at the top:

#include "types.h"

#include "x86.h"

#include "defs.h"

#include "param.h"

#include "memlayout.h"

#include "mmu.h"

#include "proc.h"

int

sys\_clone(void)

{

void (\*fcn)(void\*, void\*);

void \*arg1, \*arg2, \*stack;

if(argptr(0, (void\*)&fcn, sizeof(fcn)) < 0 ||

argptr(1, (void\*)&arg1, sizeof(arg1)) < 0 ||

argptr(2, (void\*)&arg2, sizeof(arg2)) < 0 ||

argptr(3, (void\*)&stack, sizeof(stack)) < 0)

return -1;

return clone(fcn, arg1, arg2, stack);

}

int

sys\_join(void)

{

void \*\*stack;

if(argptr(0, (void\*)&stack, sizeof(stack)) < 0)

return -1;

return join(stack);

}

**Proc.c**

// clone(): create a new thread sharing the caller’s address space

int

clone(void (\*fcn)(void\*, void\*), void \*arg1, void \*arg2, void \*stack)

{

int i;

struct proc \*curproc = myproc();

struct proc \*np;

// Allocate process slot

if((np = allocproc()) == 0)

return -1;

// Share address space; bump reference count

np->pgdir = curproc->pgdir;

acquire(&ptable.lock);

curproc->ref\_count++;

np->is\_thread = 1;

release(&ptable.lock);

// Inherit size, parent pointer, trapframe

np->sz = curproc->sz;

np->parent = curproc;

\*np->tf = \*curproc->tf;

// Build the new thread’s user‐stack so that when it starts:

// esp points to fake return PC, arg2, arg1

{

uint sp = (uint)stack + PGSIZE;

// push arg1

sp -= sizeof(void\*);

\*(uint\*)sp = (uint)arg1;

// push arg2

sp -= sizeof(void\*);

\*(uint\*)sp = (uint)arg2;

// push fake return PC

sp -= sizeof(void\*);

\*(uint\*)sp = 0xFFFFFFFF;

np->tf->esp = sp;

np->tf->ebp = sp;

np->tf->eip = (uint)fcn; // start execution at fcn(arg1,arg2)

}

// Duplicate file descriptors & cwd

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

if(curproc->ofile[i])

np->ofile[i] = filedup(curproc->ofile[i]);

np->cwd = idup(curproc->cwd);

safestrcpy(np->name, curproc->name, sizeof(curproc->name));

np->state = RUNNABLE;

return np->pid;

}

// join(): wait for a thread in this address space to exit

int

join(void \*\*stack\_store)

{

struct proc \*curproc = myproc();

struct proc \*p;

int have\_threads, pid;

acquire(&ptable.lock);

for(;;){

have\_threads = 0;

// Scan for any child thread

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

if(p->parent != curproc || !p->is\_thread)

continue;

have\_threads = 1;

if(p->state == ZOMBIE){

// found one to join

pid = p->pid;

\*stack\_store = (void\*)p->tf->esp; // return user‐stack pointer

kfree((char\*)p->kstack);

p->kstack = 0;

p->state = UNUSED;

curproc->ref\_count--;

release(&ptable.lock);

return pid;

}

}

// no threads or we've been killed: fail

if(!have\_threads || curproc->killed){

release(&ptable.lock);

return -1;

}

sleep(curproc, &ptable.lock);

}

}

**ulib.c**

#include "types.h"

#include "stat.h"

#include "user.h"

#include "x86.h"

// thread\_create: allocate stack and clone

int

thread\_create(void (\*start\_routine)(void\*,void\*), void \*arg1, void \*arg2)

{

void \*stack = malloc(PGSIZE);

if(!stack) return -1;

int pid = clone(start\_routine, arg1, arg2, stack);

if(pid < 0){ free(stack); return -1; }

// In child: clone returns 0

if(pid == 0) return 0;

return pid;

}

// thread\_join: join and free stack

int

thread\_join(void)

{

void \*stack;

int pid = join(&stack);

if(pid > 0) free(stack);

return pid;

}

// Ticket lock routines

void

lock\_init(lock\_t \*lk)

{

lk->next = 0;

lk->now\_serving = 0;

}

void

lock\_acquire(lock\_t \*lk)

{

unsigned ticket = xchg(&lk->next, lk->next + 1);

while(xchg(&lk->now\_serving, lk->now\_serving) != ticket)

; // spin

}

void

lock\_release(lock\_t \*lk)

{

lk->now\_serving++;

}

**Threads\_test.c**

#include "types.h"

#include "stat.h"

#include "user.h"

#define NTHREADS 5

volatile int counter = 0;

lock\_t lk;

void

task(void \*a1, void \*a2)

{

int id = (int)a1;

lock\_acquire(&lk);

counter++;

printf(1, "thread %d incremented counter to %d\n", id, counter);

lock\_release(&lk);

exit();

}

int

main(void)

{

int i;

int tid;

lock\_init(&lk);

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

tid = thread\_create(task, (void\*)i, 0);

if(tid < 0)

printf(2, "thread\_create failed\n");

}

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

thread\_join();

}

printf(1, "expected threads = %d, actual counter = %d\n", NTHREADS, counter);

exit();

}

**XV6 changes**

// this might not be necessary, the chat GPT recommended this as a possible fix

//but I haven’t tested if it works without it.

Forktest becomes:

\_forktest: forktest.o ulib.o usys.o umalloc.o + $(LD) -m elf\_i386 -N -e main -Ttext 0 \ + -o $@ forktest.o ulib.o usys.o umalloc.o