COMP-4736 Intro to Operating Systems

Lab 6

Jonghoon Jang(A01240621)

Jacob Seol(A01002532)

Clone

proc.h
To keep track of the stack allocated for a thread created using the ‘clone()’ system call. So, a new stack is allocated in the process's address space, and the address of this stack is stored in threadstack in the newly created process's struct proc

Add `void *threadstack;`

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  void *threadstack;           // Address of thread stack to be freed
  enum procstate state;        // Process state
  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)
};
```

proc.c
Allocate a new process using allocproc, and then copy the relevant data from the current process to the new process using pointers and struct assignments. Also set up the new thread's stack by pushing the function arguments and a fake return address onto it, and then modifying the process's trapframe to point to the new stack and the function to be executed. Lastly, set the process state to RUNNABLE and release the lock on the process table.

```
int
clone(void(*fcn)(void*,void*), void *arg1, void *arg2, void* stack)
```

```
{
  // Get the current process
  struct proc *p = myproc();

  // Allocate a new process structure for the new thread
  struct proc *np;

  // Allocate process.
  if((np = allocproc()) == 0)
    return -1;

  // Copy process data to the new thread
  np->pgdir = p->pgdir;  // Use the same page directory as the parent process
  np->sz = p->sz;  // Use the same memory size as the parent process
  np->parent = p;  // Set the parent process to the current process
  *np->tf = *p->tf;  // Copy the trap frame of the parent process to the new
thread's trap frame

  void * stackArg1, *stackArg2, *stackReturnAddress;

  // Push fake return address to the stack of thread
  stackReturnAddress = stack + PGSIZE - 3 * sizeof(void *);
  *(uint*)stackReturnAddress = 0xFFFFFFF; // fake return PC

  // Push first argument to the stack of thread
  stackArg1 = stack + PGSIZE - 2 * sizeof(void *);
  *(uint*)stackArg1 = (uint)arg1;

  // Push second argument to the stack of thread
  stackArg2 = stack + PGSIZE - 1 * sizeof(void *);
  *(uint*)stackArg2 = (uint)arg2;

  // Put address of new stack in the stack pointer (ESP)
  np->tf->esp = (uint) stack;

  // Save address of stack
  np->threadstack = stack;

  // Initialize stack pointer to appropriate address
  np->tf->esp += PGSIZE - 3 * sizeof(void*);
  np->tf->ebp = np->tf->esp;

  // Set instruction pointer to given function
  np->tf->eip = (uint) fcn;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  // Duplicate file descriptors
```

```
  int i;
  for(i = 0; i < NOFILE; i++) // Iterate through all file descriptors
    if(p->ofile[i]) // If the file descriptor is not null
      np->ofile[i] = filedup(p->ofile[i]);  // Duplicate file descriptors
  np->cwd = idup(p->cwd);  // Duplicate current working directory

  // Copy the name of the parent process to the new thread's name
  safestrcpy(np->name, p->name, sizeof(p->name));

  // Acquire the process table lock to modify the process table
  acquire(&ptable.lock);

  // Set the state of the new thread to runnable
  np->state = RUNNABLE;

  // Release the process table lock
  release(&ptable.lock);

  // Return the process ID of the new thread
  return np->pid;
}
```

sysproc.c

Add sys_clone which is the system call handler for clone system call. It extracts the four arguments passed by the user to the clone system call using the argint function and then calls the clone function with the extracted arguments.

```
sys_clone(void)
{
  int fcn, arg1, arg2, stack;
  if(argint(0, &fcn)<0 || argint(1, &arg1)<0 || argint(2, &arg2)<0 ||
argint(3, &stack)<0)
    return -1;
  return clone((void *)fcn, (void *)arg1, (void *)arg2, (void *)stack);
}
```

defs.h

Add the function prototypes to the defs.h to make new clone system call visible.

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

syscall.c

Defining the system call numbers and the corresponding functions, so the kernel can handles and executes the system calls when invoked by a user-space program

```
extern int sys_uptime(void);
extern int sys_clone(void);
```

```
[SYS_close]   sys_close,
[SYS_clone]   sys_clone,
```

syscall.h
Assigning a unique number to the SYS_clone marco, to be identified in the kernel

```
#define SYS_clone  22
```

user.h
Adding the function head, it allows user-space code to use the function

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

usys.S
It defines the implementation of the system call interface that is used to make requests to the operating system from user space.

```
SYSCALL(clone)
```

testcase.c

```
#include "types.h"
#include "stat.h"
#include "user.h"

void thread(void *arg1, void *arg2) {
  printf(1, "Hello from thread! arg1: %d arg2: %d\n", *(int*)arg1,
*(int*)arg2);
  exit();
}

int main(int argc, char *argv[]) {
  printf(1, "Hello from main!\n");
  int arg1 = 10;
  int arg2 = 20;
  void *stack = malloc(4096);
  int pid = clone(thread, &arg1, &arg2, stack);
  if(pid < 0) {
    printf(2, "Error: clone failed\n");
    exit();
  }
  printf(1, "clone created a new thread with pid %d\n", pid);
  wait();
  exit();
}
```

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ cat output.txt
Hello from main!
clone created a new thread with pid 5

Hello from thread! arg1: 10 arg2: 20
$
```

As we can see from the output, the child thread is successfully created by the parent process.

JOIN

Loop through the process table looking for zombie child threads with the same parent process and page directory as the current process. When findind a zombie thread, free its kernel stack, reset its process table entry, and return the thread's PID and thread stack pointer to the calling function. If there are no children or the current process has been killed, it returns -1. If there are still live children, call sleep to wait for one of them to exit.

proc.c

```c
// Wait for a child thread to exit and return its pid.
int join(void** stack)
{
  struct proc *p;
  int foundChildThreads, pid;
  // Get the current process
  struct proc *cp = myproc();

  // Acquire the lock for the process table
  acquire(&ptable.lock);

  for (;;) { // Loop forever
    // Set a flag to indicate whether there are any children
    foundChildThreads = 0;

    // Scan through the process table looking for child threads of the
calling process
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {

      // Check if this is a child thread (parent or shared address space)
      if (p->parent != cp || p->pgdir != p->parent->pgdir)
        continue;

      // If this is a child thread, set the flag to indicate that we have
children
      foundChildThreads = 1;
```

```
      // Check if the child thread is in a zombie state
      if (p->state == ZOMBIE) {
        // If we found a zombie child thread, get its process ID
        pid = p->pid;

        // Free the kernel stack that was allocated for the child thread
        kfree(p->kstack);
        p->kstack = 0;

        // Reset the process table entry for the child thread
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        stack = p->threadstack;
        p->threadstack = 0;

        // Release the process table lock and return the process ID of the
zombie child thread
        release(&ptable.lock);
        return pid;
      }
    }

    // If there are no children or the calling process has been killed,
release the process table lock and return an error code
    if (!foundChildThreads || cp->killed) {
      release(&ptable.lock);
      return -1;
    }

    // If there are still child threads running, go to sleep until one of
them exits
    sleep(cp, &ptable.lock);
  }
}
```

sysproc.c

```
int
sys_join(void) {
    void **stack;

    if(argptr(0, (char**)&stack, sizeof(stack)) < 0)
        return -1;
```

```
    return join(stack);
}
```

defs.h

```
int             join(void **);
```

syscall.c

```
extern int sys_clone(void);
extern int sys_join(void);
```

```
[SYS_clone]   sys_clone,
[SYS_join]    sys_join,
```

syscall.h

```
#define SYS_clone  22
#define SYS_join   23
```

user.h

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

usys.S

```
SYSCALL(clone)
SYSCALL(join)
```

Testcase for clone

```
#include "types.h"
#include "user.h"
#include "fcntl.h"

#define PGSIZE 4096

void thread_function(void *arg1, void *arg2) {
  printf(1, "Thread: arg1 = %d, arg2 = %d\n", *(int*)arg1, *(int*)arg2);
  exit();
}

int main(int argc, char *argv[]) {
  void *stack;
  int arg1 = 10;
  int arg2 = 20;

  stack = malloc(PGSIZE);
```

```
  if (stack == NULL) {
    printf(1, "Error: could not allocate stack.\n");
    exit();
  }

  printf(1, "Parent: calling clone...\n");
  int pid = clone(thread_function, &arg1, &arg2, stack);
  if (pid < 0) {
    printf(1, "Error: clone failed.\n");
    exit();
  }

  printf(1, "Parent: waiting for child to finish...\n");
  void *join_stack;
  int join_pid = join(&join_stack);
  if (join_pid < 0) {
    printf(1, "Error: join failed.\n");
    exit();
  }

  printf(1, "Parent: child finished.\n");

  exit();
}
```

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ testcasejoin > outputjoin.txt
$ cat outputjoin.txt
Parent: calling clone...
Parent: waiting for child to finish...
Thread: arg1 = 10, arg2 = 20
Parent: child finished.
```

As we can see the output, the parent process successfully created a child thread using **clone()** and the parent process waits until the child thread finishes its execution using **join()**

thread_create(), thread_join(), lock_acquire() and lock_release() functions

ulib.c
As ulib.c file provides support for user-level threads. The thread libraries added to the ulib.c

Add `#include "mmu.h"`

```
// Creates a new thread by cloning the current process.
int
thread_create(void (*start_routine)(void *, void *), void* arg1, void* arg2)
{
  void* stack;

  // allocate a stack for the new thread
  stack = malloc(PGSIZE);

  // returns the pid of the new thread
  return clone(start_routine, arg1, arg2, stack);
}

// waits for a child thread to finish and returns the child's PID.
int
thread_join()
{
  void * stackPtr;
  // join returns the pid of the child thread that finished
  int x = join(&stackPtr);
  return x;
}

//  initializes a lock by setting the flag field of the lock structure to 0
int
lock_init(lock_t *lk)
{
  lk->flag = 0;
  return 0;
}

// acquires the lock by setting the flag field of the lock structure to 1
void
lock_acquire(lock_t *lk){
  while(xchg(&lk->flag, 1) != 0);
}

// releases the lock by setting the flag field of the lock structure to 0
void
lock_release(lock_t *lk){
  xchg(&lk->flag, 0);
}
```

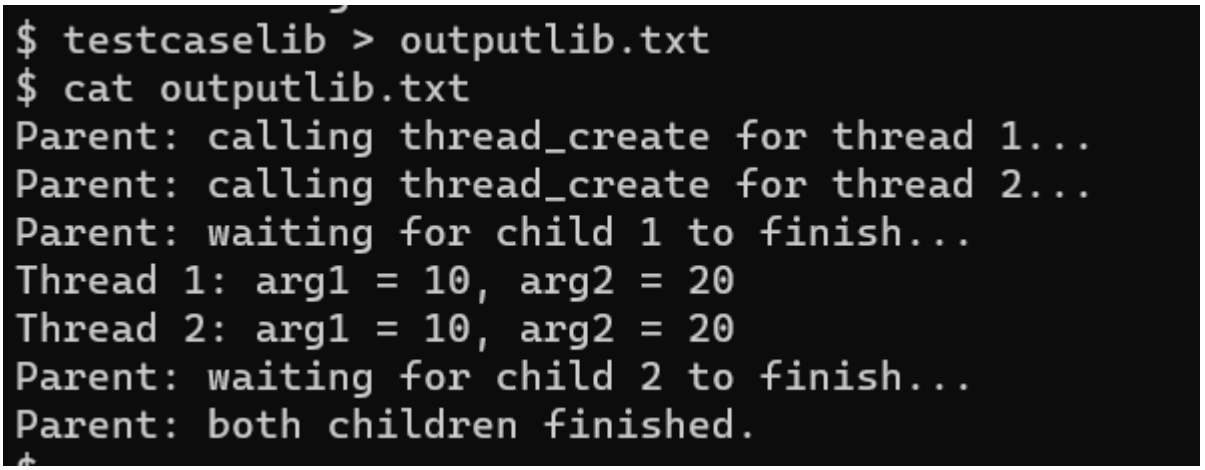
user.h

```
typedef struct __lock_t{
  uint flag;
}lock_t;
```

```
int thread_create(void (*start_routine)(void *,void*), void * arg1, void *
arg2);
int thread_join();
int lock_init(lock_t *lk);
void lock_acquire(lock_t *lk);
void lock_release(lock_t *lk);
```

testcase for thread_create and thread_join

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"
#define PGSIZE 4096

lock_t lock;

void thread_function(void *arg1, void *arg2) {
  lock_acquire(&lock);
  printf(1, "Thread: arg1 = %d, arg2 = %d\n", *(int*)arg1, *(int*)arg2);
  lock_release(&lock);
  exit();
}

int main(int argc, char *argv[]) {
  int arg1 = 10;
  int arg2 = 20;

  lock_init(&lock);

  printf(1, "Parent: calling thread_create...\n");
  int pid = thread_create(thread_function, &arg1, &arg2);
  if (pid < 0) {
    printf(1, "Error: thread_create failed.\n");
    exit();
  }

  printf(1, "Parent: waiting for child to finish...\n");
  int join_pid = thread_join();
  if (join_pid < 0) {
    printf(1, "Error: thread_join failed.\n");
    exit();
  }

  printf(1, "Parent: child finished.\n");

  exit();
}
```



The output shows that the parent thread created two new threads using **thread_create()** and it waits until its child ocesses to be finished using **thread_join()**. in the thread_functions, each child acquired lock using **lock_acquire()** and released lock using **lock_release()** after executing printing to console