CS 333: Operating Systems Lab
Autumn 2022
Lab8: the *clone* primitive
Lab8: Kernel Threads in xv6 (clone, join, and target functions)

This lab explores the introduction of multi-threading and locking in xv6. In this lab, we will modify the xv6 code to create the `clone` and `join` system calls for multi-threading and add user target functions to wrap them.

1. clone

In this subsection, you have to implement a `clone()` system call. This system call creates a new kernel thread for a process. Recall that while kernel threads are independently schedulable entities like processes, all the threads of a process share the same virtual address space and file descriptors.

In order to allow for multi-threading in xv6, we need to come up with a design for Thread Control Blocks (TCBs). For this lab, use **`struct proc`** itself as the PCB-cum-TCB for all the threads. You will need to add some thread-relevant fields to your PCB, such as:

- A thread-ID like the process-ID. For this lab, use the **`pid`** field of the **`struct proc`** as the thread-ID for different threads.
- A thread-group-ID, which can be the main thread's process-ID. This will help us tie together threads that belong to a process.
- A count of the number of threads of a process
- A pointer to the user stack of the thread (each thread has its own user and kernel stacks)

The signature of the clone system call is as follows:

```
int clone(void(*fn)(int*), int *arg, void *stack)
```

This system call will effectively create a kernel thread and the thread will start execution at the function **fn** with **arg** as the function's argument using the given stack argument as its stack. Note that we are designing the system call to work only with functions that take one argument.

For the arguments of clone system call:
- **`fn`** -> indicates the start point of execution of the thread
- **`arg`** -> pointer to the argument of function fn
- **`stack`** -> base address of the stack allocated for thread

On success, the clone call returns the thread-id of the new thread and if unsuccessful, it returns -1.

You must create a wrapper function for calling in user space on top of the system call defined above.

`int create_thread(void(*fn)(int*), int *arg)`

In this wrapper function, `malloc` **one page** and assign the malloced page as the stack argument, and THEN call the `clone` system call to start the thread. On success, this call returns the thread-id of the new thread and if unsuccessful, it returns -1.

Notes:

1. You may encounter `undefined reference to malloc` error while implementing the above function and calling make. This can be prevented by adding ***umalloc.o*** to "_forktest" in MakeFile in the following manner (see the 3rd line, umalloc.o after usys.o).

```
_forktest: forktest.o $(ULIB)
    # forktest has less library code linked in - needs to be small
    # in order to be able to max out the proc table.
    $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o _forktest forktest.o ulib.o usys.o umalloc.o
    $(OBJDUMP) -S _forktest > forktest.asm
```

2. The stack pointer is passed from the thread wrapper function (which allocates memory on the virtual space of the calling process). This stack pointer (virtual address of main process, pointing to a single page) will be used as the user stack of the cloned process. Implement the wrapper function **create_thread** in the file `ulib.c`.

   Why/How will this work?

3. Following are the expectations around the system call—
   - All the threads of a process must have an identical virtual address space and share the same physical pages (and hence share a single page table).
   - All the threads of a process must share the same set of file descriptors. For this lab, you can implement it like the way `fork` does it (by copying over the FDs to the new proc table entry) as opposed to sharing it. This would mean that a fopen or fclose call needs its effects to be replicated across threads but for this lab, you may assume that these calls will not happen.
   - Every thread has its own user stack for user functions.
   - To initialize a thread, the instruction pointer and the stack pointer of the new thread need to be initialized.
   - A new thread once created and initialized, when scheduled, should start executing at the user specified function.
   - Each thread function MUST end execution with an explicit call to **exit**().
     Why is this required? and if not present, how to handle correct termination of a thread?

   - Based on your logic, you might need to make a few changes to at least the `fork`, `wait`, `exit` and `kill` implementations to properly initialise and clean-up the TCB entries.
   - When the main process is about to **exit**, it should kill and reap all its threads before exiting.
   - For a process to be reaped, all of its threads must have been reaped before (refer

to the **join** system call described in the next subsection).

2. join

1. In this subsection, we will implement a **join** system call. **join** can be called in the main process to reap a thread that has exited. This is a blocking system call which should work in a similar manner as the **wait** system call. The signature of the system call is as follows:

   ```
   int join()
   ```

   Like **wait**, **join** is a blocking system call that returns the thread-ID of a reaped thread, when called from the main thread. If none of the other threads have exited, then the thread calling join waits for one of the threads to exit. If there are no threads left to exit then join will return -1.

   Reaping a thread involves cleaning up its TCB(-cum-PCB), and updating the PCB of the process, which consists of steps like:

   - Freeing up the kernel-stack of the thread (we are not freeing the user-stack as explained later).
   - Setting the PCB and TCB entries of the kernel thread to their respective default values, like in the implementation of **wait** (for example, set the `state` field to `UNUSED`).
   - Decrementing the thread-count of the process.

   Based on our design, the join may lead to a page-sized memory leak due to the stack memory allocated earlier and not freed on thread termination. Our current design will keep things simple and let this be. As an extension, you can think of how to overcome these leaks cleanly.

   Note: The wrapper function should be available as part of every program that runs in xv6. Thus, you should add prototypes to **user/user.h** and the actual code to implement the library routines in **user/ulib.c**.

3. Testing

You can make sure that your cloned processes are properly reaped after the thread exits. This can be done by proper implementation of the join system call. Two test case files are added in the **testcases/** directory to test your implementation.

For a test case named **tc-<something>.c**, add **_tc-<something>** to UPROGS in the MakeFile similar to the previous labs and **tc-<something>.c** to EXTRA definition in MakeFile and then run make and make qemu-nox. On the command prompt of xv6, execute **tc-<something>** to run the test case.

1. Global Variable (tc-var.c)

   This test case contains a global variable **initialised** to **zero** and the main process creates N threads and each thread increments the global variable value by 1, and prints the value after incrementing. Finally, after joining all the threads, the main program prints the updated value of the variable, which could be N (or less than N, because of possible race conditions). For a given testcase, N is 5.

   Expected Output:

```
$ tc-var
Calling Process Print VAR value: 0
Thread Rank: 0, VAR: 1
Thread Rank: 1, VAR: 2
Thread Rank: 2, VAR: 3
Thread Rank: 3, VAR: 4
Thread Rank: 4, VAR: 5
All threads joined, VAR value: 5
```

2. Global Array Sums (tc-array.c)

   This test case contains two globally initialized arrays, and the **argument** for the function of the thread will be its "rank". Two threads are created with ranks 0 and 1. If cloned threads are working properly, the 0th ranked thread will calculate the sum of **all elements of the first half of the first array** and **all elements of the second half of the second array**. Similarly, rank 1 thread will calculate the sum of the remaining elements of both arrays, and both threads will print in the following manner. The main process will then print the cumulative sum of both arrays which should be equal to the sum of individual sums.

   Expected Output:

```
$ tc-array
Calling Process Print Check
Thread Rank: 0, Sum Value: 71
Thread Rank: 1, Sum Value: 53
All threads joined
Sum of thread calls is equal to that of both array sums, value: 124
```

Submission Instructions

- All submissions are to be done on moodle only.
- Name your submission as <rollnumber>_lab8.tar.gz (e.g 190050004_lab8.tar.gz)
- The tar should contain the following files in the following directory structure:
  <rollnumber>_lab8/

    |_ _< all modified files in xv6 such as

        syscall.c, syscall.h, sysproc.c, user.h, usys.S, proc.c, trap.c, defs.h, proc.h, . . . >

    |_ _Makefile

  Please adhere to it strictly.
- Your modified code/added code should be well commented on and readable.
- `tar -czvf <rollnumber>_lab8.tar.gz <rollnumber>_lab8`

Deadline: Monday, 14th October 2022, 11:59 PM via moodle.