Project 2: Add threading support to XV6

Designer
Aryan Abdollahi Sabet Nejad
aryan_abdollahi@comp.iust.ac.ir
aryansabetnejad@gmail.com

1 Overview

In this project, we make some major changes to xv6 to add support for kernel threads to the OS. A single process may execute multiple kernel threads, which may operate concurrently on a multi-core machine. To add threading support for xv6, we must add some system calls:

1. `int clone(void *stack , ...)`
2. `int join(int)`

You'll also add some user-space functions that wrap clone() for convenience:

1. `int thread_create(void (*worker)(void *, void *), void *arg1, void *arg2)`
2. `int lock_init(lock*)`
3. `void lock_acquire(lock*)`
4. `void lock_release(lock*)`

2 Details and Challenges

2.1 System Calls

- `int clone(void *stack , ...)`
  The `clone()` system call creates new kernel threads and returns the thread ID of the newly created thread or -1 on failure. Unlike `fork`, `clone` shares the address space, requiring careful initialization of the new thread. The challenge lies in allocating a separate stack for each thread while sharing the address space.
- `int join()`
  The `join()` system call allows a thread to wait for a specific thread to finish execution. It returns 0 on success and -1 on failure. This system call is similar to the existing `wait()` call but is designed for thread synchronization.

2.2 User-Space Functions

- `int thread_create(void (*worker)(void *, void *), void *arg1, void *arg2)`
  The `thread_create()` function serves as a wrapper around `clone()`. It returns the thread ID or -1 on failure. The challenge involves allocating page-aligned space for the new stack,

calling `clone`, and managing the execution of the provided function pointer in the child thread. Proper stack deallocation and exit calls must be implemented.

- `int lock_init(lock*)`

  The `lock_init()` function initializes a lock variable and returns 0 on success or -1 on failure. This function is crucial for ensuring thread safety.
- `void lock_acquire(lock*)`

  The `lock_acquire()` function acquires a lock, and it is implemented using the provided lock implementation. This function ensures proper synchronization among threads.
- `void lock_release(lock*)`

  The `lock_release()` function releases a lock, allowing other threads to acquire it. Challenges involve ensuring the proper release of the lock and preventing race conditions.

2.3 Implementation Notes

- **Memory Management:** Careful allocation and deallocation of thread stacks are crucial. Challenges include managing page-aligned space and avoiding memory leaks.
- **Synchronization:** Implementing synchronization using locks requires addressing potential race conditions.
- **Error Handling:** Implement robust error handling in system calls and user-space functions to gracefully handle exceptional scenarios.

2.4 Challenges Ahead

- **Shared Memory Management:** Managing shared memory between threads requires careful initialization to prevent conflicts and ensure data consistency.
- **Concurrency Control:** Preventing race conditions and ensuring proper synchronization among threads pose significant challenges.
- **Stack Allocation:** Allocating and deallocating stacks dynamically while maintaining thread safety is a complex task.
- **Lock Implementation:** Ongoing challenges may arise in refining the lock implementation to address potential issues during thread synchronization.

> **Choosing Thread Model & Scheduling:** Given the adjustment to the `struct proc`, the Round-Robin (RR) scheduler may become less meaningful in this context. Consider making tweaks to achieve fairness between processes and threads respectively. Tailor the scheduling mechanism to account for the unique characteristics of threads within a process, ensuring equitable resource allocation.

3 Hints

3.1 `clone()` System Call

Start Simple: Begin by copying the syntax and basic behavior of the `fork()` system call for your `clone()` implementation. This example will provide a foundation that you can build upon:

```
int stack[4096] __attribute__ ((aligned (4096)));
int main(int argc, char *argv[]) {
  printf(1, "Stack␣", stack);
  int tid = clone(stack);
  if (tid < 0) {
    printf(2, "Error!\n");
  } else if (tid == 0) {
    // Child thread logic here
  } else {
    // Parent thread logic here
```

```
        }
        exit();
    }
```

3.2 `lock` Implementation Options:

- **Read Kernel's Spinlock Code:** Begin by implementing simple spinlocks for the `lock` functions. This approach involves busy-waiting, which may be suitable for initial testing and understanding.
- **Kernel-Level Locks:** Consider employing kernel-level locks for a more sophisticated and efficient locking mechanism. You can provide the kernel's spinlock for user-space usage by implementing it through a syscall.
- **Assembly Instructions:** Explore assembly instructions such as compare-and-swap (CAS) to implement atomic operations. This can be particularly useful for avoiding race conditions in critical sections.

Remember to incrementally test and debug your implementation, and feel free to experiment with different approaches as you progress.

User-space functions are located at *ulib.c* and *user.h* . The syntax of these functions is quite similar to `pthread` library of Linux.

4 Reminder on Installation of XV6

The following procedure has been tested on both Ubuntu VM & WSL2 (Windows 11):

```
user@user:~$ sudo apt-get update && sudo apt-get install
    --yes build-essential git qemu-system-x86
user@user:~$ git clone https://github.com/mit-pdos/
    xv6-public
user@user:~$ make qemu-nox
```

XV6 is also runnable on Mac (both Apple Silicon & Intel chips) by changing the `TOOLPREFIX` in *Makefile* & removing `-m32` from compiler & linker flags (only for Apple Silocon chips).

```
# Cross-compiling (Mac OS X) on Apple Silicon
TOOLPREFIX = i686-elf-gcc
# Cross-compiling (Mac OS X) on Intel x86
TOOLPREFIX = i386-elf-gcc
```

Then:

```
user@user:~$ brew install i686-elf-gcc
user@user:~$ make
user@user:~$ make qemu-nox
```

5 Submission

To submit your work, simply commit all your changes (adding new files as needed) and push your work to your GitHub repository. also, upload a zipped file of the Persian documentation & repository on Quera. Make sure you sign the filename with your student IDs. If you don't do this we won't be able to associate your submission with you!

- **Incremental Progress:** Focus on gradual development to maintain compilability. Debug each added functionality individually.

- **Active Participation:** All team members must actively contribute to the project. To present the collaboration graph on the project presentation day, install Gitgraph or GitLens extensions for VSCode. Inactive team members may receive a lower score.
- **Make a private repository:** Keep your repository private till presentation day. Ensure that your first commit is the latest commit from the xv6-public repository. This enables us to use the `Diff` command on Linux to track changes.
- **Testing Approach:** Test each system call individually or integrate them into different combination scenarios. Untested functionalities will not be scored.
- **Documentation of Challenges:** Write comprehensive documentation detailing the challenges faced during the project, along with your problem-solving approaches. Include parts of the project that were not strictly outlined and describe how you addressed them. We are eager to understand your thought process and the solutions you implemented.
- **Submission Readiness:** Perform a thorough compilation check before submission. Uncompilable code will not be scored.