Department of Computer Science, Hanyang University

Light-weight Process

Operating Systems: Three Easy Pieces

Programming Project with xv6

Final due date: May 30, 2017 (HARD DEADLINE)

1 Overview

1. Light-weight process

   A process is a computer program running on a computer. It is an independent execution object that is a target of a scheduler. Processes have independent resources for execution, such as address spaces and file descriptors. A light-weight process(shortly LWP) is a little different. A LWP is a process that shares resources such as address space with other LWPs, and allows multitasking to be done at the user level. The purpose of this project is to improve the multitasking capabilities of xv6 by implementing an abstraction of LWP.

2. POSIX thread

   POSIX threads (abbreviated as Pthread) are light-weight processes initially implemented in UNIX and the management of pthreads is done through standard pthread APIs provided for writing software allowing parallel execution. Pthread is a commonly used library on all UNIX-like operating systems. When we write code for a parallel application on Linux, we generally use pthread. This pthread is an implementation of LWP mentioned above. The behavior of pthread will be a good reference for implementing LWP in xv6. In addition, the tests for evaluation follow the behavior of these pthreads unless stated otherwise.

2 Implementation Specifications

1. Simplified Pthread

   The goal of this project is to create LWP similar to pthread for xv6. Same as pthread, but with a simplified version, each LWP shares only their address space but other resources. Each LWP shares its address space, but **it must have a separate stack for independent execution**. Also, in the case of pthread, it is possible to set the stack size, detach status, scheduling policy, etc. through attributes. However, in this project, it is just aimed to create a joinable LWP with the same properties as the normal process.

2. **The First Milestone : Basic LWP Operations** - Create, Join and Exit

   -Due Date: May 16

   The operations below are provided to users through system calls.

   a) **Create** : You must provide a method to create threads within the process. From that point on, the execution routine assigned to each thread starts.

   ```
   int thread_create(thread_t * thread, void * (*start_routine)(vo
   id *), void *arg);
   ```

   - thread - returns the thread id.
   - start_routine - the pointer to the function to be threaded. The function has a single argument: pointer to void.
   - arg : the pointer to an argument for the function to be threaded. To pass multiple arguments, send a pointer to a structure.
   - ret_val : On success, thread_create return 0. On error, it returns a non-zero value.

   b) **Exit** : You must provide a method to terminate the thread in it. As the `main` function do, you call the `thread_exit` function at the last of a thread routine. Through this function, you must able to return a result of a thread.

   ```
   void thread_exit(void *retval);
   ```

   - retval : Return value of the thread.

   c) **Join** : You must provide a method to wait for the thread specified by the argument to terminate. If that thread has already terminated, then this returns immediately. In the join function, you have to clean up the resources allocated to the thread such as a page table, allocated memories and stacks. You can get the return value of thread through this function.

   ```
   int thread_join(thread_t thread, void **retval);
   ```

   - thread : the thread id allocated on thread_create.
   - retval : the pointer for return value.
   - ret_val : On success, this function returns 0. On error, it returns a non-zero value.

3. **The Second Milestone : Interaction with other services in xv6**

   -Due Date: May 30

   a) Interaction with system calls in xv6

      A newly created LWP must interact properly with the services already provided by the operating system. Therefore, it is necessary to consider how to operate when calling other system calls in the multithreaded environment by LWP that you make. In particular, we will mainly evaluate operations related to process and it follows operations of pthread unless otherwise noted. The system calls to consider include exit, fork, exec, sbrk, kill, pipe, sleep, etc. The specific behavior is described in the test case section below.

   b) (Advanced) Interaction with the schedulers that you implemented in Project 2: MLFQ and Stride.

      A newly created LWP should be able to interact properly with the schedulers you made in Project 2. There can be a lot of ways to interact with two schedulers. We propose one variation among them and you have to follow this specification. The details are described in the test case section below.

3 Evaluation

1. Document

   You must write detailed document for your work on the gitlab wiki. It is evaluated based on that document. If you miss a description for something, it may be not evaluated. Please do careful work that describes your work. The document may include design, implementation, solved problem(evaluating list below) and considerations for evaluation.

2. How to evaluate and self-test

   The evaluation of this project is based on the requirements of the implementation specification above and can be verified through the test-case provided. The points are as follows.

Evaluation items	**Points**
Documents	20
Basic LWP operations	55
Interaction with system calls	35
(advanced) Interaction with the schedulers that you made in Project 2	10
Other corner cases	10
Total Points	130

3. Test Cases

   The test program will be updated in the next specification. Below is the detail descriptions how to operate for each test case. These descriptions are based on pthread and if there is no description for a specific situation, follow behaviors of pthread.

   a) **Basic Operations** : The behaviors of the LWPs spawned by the system calls should share their address space, have their own context and stack, and be able to generate a race condition if they share a resource. You also need to be able to manage LWPs through the create, join, and exit system calls presented in this project.

   b) **Exit** : When a LWP calls the exit system call, all LWPs are terminated and all resources used for each LWP must be cleaned up and the kernel can reuse it at a later time. Also, no LWP should survive for a long time after the exit system call is executed.

   c) **Fork** : Even if multiple LWPs call the fork system call at the same time, a new process must be created according to the fork's behavior and the address space of the LWP must be copied normally. You should also be able to wait for the child process normally by the wait system call. Note the parent-child relationship between processes after the fork.

   d) **Exec** : If you call the exec system call, the resources of all LWPs are cleaned up so that the image of another program can be loaded and executed normally in one LWP. At this time, the process executed by the exec must be guaranteed to be executed as a general process thereafter.

   e) **Sbrk** : When multiple LWPs simultaneously call the sbrk system call to extend the memory area, memory areas must not be allocated overlapping with each other, nor should they be allocated a space of a different size from the requested size. The area expended by it must be shared among LWPs.

   f) **Kill** : If more than one LWP is killed, all LWPs must be terminated and the resources for each LWPs in that process must be cleaned up. After the kill to a LWP is called, no LWP should survive for a long time.

   g) **Pipe** : All LWPs must share a pipe and when reading or writing and data should be synchronized and not be duplicated.

   h) **Sleep** : When a specific LWP executes a sleep system call, only the requested LWP should be in the sleeping state for the requested time. If a LWP is terminated, sleeping LWP also has to be terminated.

   i) **Other corner cases** : We will also conduct tests for other corner cases as well as the tests mentioned above. As there are many corner cases like creating a thread in a thread, please mimic the behavior of pthread well. The more defensive your code, the higher your score.

   j) **Interaction with the schedulers that you made in Project 2** : LWPs are scheduled by MLFQ scheduler like the normal process by default. If a LWP calls `set_cpu_` `share` function, all threads belonging the same process are scheduled by Stride scheduler and share the time slice for that process. When a process is already being scheduled by Stride scheduler, subsequent threads spawned in the process will share the time slice for the process.

Noted items :

1. A LWP is a process just sharing same address space. We recommend using `proc` structure to implement LWP.
2. Sharing address space means sharing same page table among the LWPs spawned by a process.
3. You can get many hints from fork, exec, exit and wait functions in xv6.
4. Pay attention to using the lock for ptable.
5. Recommend making various test cases using Pthread on a linux machine and get an intuition for implementation.