DEPARTMENT OF COMPUTER SCIENCE, HANYANG UNIVERSITY

Light-weight Process

Operating Systems: Three Easy Pieces

Programming Project with xv6

Final due date: May 29, 2020

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. MLFQ and Stride Scheduling

   Context switching between LWPs that are within the same group can be implemented in a more efficient way than a context switching between processes. You should think carefully about the difference between them. During the implementation, you have to reimplement the scheduler that you've made for project1 so that it works correctly with the LWP implementation.

   We define the LWP group that is a set of LWPs all executing inside the same process, and LWPs within the same LWP group should adopt the same scheduling policy. LWP group applies Round Robin scheduling internally, and context switching within the LWP group occurs every *1tick* in a given time quantum. The Procedure of context switching between LWPs within the same group should be implemented differently from that of a context switching between normal processes. When a context switching occurs between LWPs within the same group, **an LWP should switch its context with another LWP directly**; context switching between an LWP and the scheduler never occurs. The details are as follows.

   - MLFQ (Multi-level feedback queue) scheduling
     - All LWPs within the same LWP group should share *their time quantum and time allotment*. If an LWP is selected to run by the scheduler, It should use its time quantum within the LWP group.
     - Each level of queue adopts Round Robin policy with different time quantum.
       * The highest priority queue: 5 tick
       * Middle priority queue: 10 ticks
       * The lowest priority queue: 20 ticks
     - Each queue has different time allotment.
       * The highest priority queue: 20 ticks
       * Middle priority queue: 40 ticks
     - To prevent starvation, priority boosting needs to be performed periodically.
       * The priority boosting is the only way to move the process upward.

        * Frequency of the priority boosting: 200 ticks
    - Stride scheduling
        - The default time quantum is 5 ticks.
        - All LWPs within the same LWP group should share *their time quantum*. If an LWP is selected to run by the scheduler, It should use its time quantum within the LWP group.
        - When a LWP calls set_cpu_share(), all LWPs in this group have to be managed by stride scheduler.
        - The total sum of CPU share requested from processes in the stride queue can not exceed 80% of CPU time. Exception handling needs to be properly implemented to handle oversubscribed requests.

3. **The First Milestone : Design Basic LWP Operations**

   -Due Date: ASAP

   Carefully read the project2 specification. Review lectures related to your project and write down your understanding.

   a) **Process / Thread**
      - Write briefly about Process and Thread.
      - Write briefly about context switching in both sides.

   b) **POSIX thread** : Write briefly about pthread library.
      - `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *),void *arg);`
      - `int pthread_join(pthread_t thread, void **ret_val);`
      - `void pthread_exit(void *ret_val);`

   c) **Design Basic LWP Operations for xv6**
      - Write down your understanding of milestone 2.
      - Write a rough design for Milestone 2.

4. **The Second Milestone - First step : 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.

5. **The Second Milestone - Second step : Interaction with other services in xv6**

   -Due Date: May 29

   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) Interaction with the schedulers that you implemented in Project 1: MLFQ and Stride.

   A newly created LWP should be able to interact appropriately with the schedulers you made in Project 1. We proposed one way above, and you have to follow this specification.

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
Interaction with the schedulers that you made in Project 2	20
Other corner cases	10
Total Points	140

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** : All threads belonging to the same process are scheduled by the same scheduler. Threads in Stride and MLFQ scheduler share the time slice for their origin process.

Noted items :

1. An LWP is a process just sharing same address space. We recommend using `proc` structure to implement LWP.
2. Sharing address space means sharing the same page table among 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. We recommend you to make various test cases using Pthread on a linux machine and get an intuition for implementation.