CS 333: Operating Systems Lab

Autumn 2022

Lab Quiz 3 (30 marks)

- **No internet. No phone. No friends** (strictly for the duration of the exam only!)
- This lab quiz has a total of 3 questions. You can attempt them in any order of your choice.
- Auxiliary files are in the `auxiliary_files/` folder.
- Test programs are in the `testing/` folder.

Q1. FCFS Process Scheduling (10 marks)

You have to implement a First Come First Serve (FCFS) process scheduling policy in xv6. The idea is simple: **the process which arrives at the scheduler first will be scheduled to run till completion**.

Modify the `scheduler` function to implement an FCFS scheduling system in place of the existing Round Robin scheduler. For this, you should need to maintain some scheduling related metadata for processes in their PCB, such as :

- The first time a process became `RUNNABLE` (the *arrival time* of a process)
- The first time a process is scheduled.
- The time at which a process exits.

Note that the second and third fields are not necessary for the scheduler to work properly, but we would like to have them for the sake of testing your implementation.

Using the above three fields, implement a system call `wait2`, which reaps a process **exactly like wait** does, and also gives out the arrival time, first scheduled time, and the exit time of the reaped process. The function signature of the system call is as follows:

```
int wait2(int *arrival_time, int *first_scheduled_time, int *exit_time)
```

Since it is a simple enough system call to implement, and is very critical to test your code, **you will receive 0 credit for this question if you don't implement `wait2` correctly**.

Now, coming back to our scheduler, here are some expectations and hints to help you:

- For this and only for this question, in your Makefile, replace the line CPUS := 2 with CPUS := 1 .

- A process is scheduled based on its **Arrival time**, and once scheduled, the process should run till completion, except for a scenario where it has to block/sleep. This also means that a process is not supposed to yield on a timer interrupt.

- Even if a process sleeps and wakes up, it will be scheduled based on its original Arrival time; you should not modify the Arrival time of a process if its state changes from `SLEEPING` to `RUNNABLE`. Although, in the test programs, we have made sure that the child processes initiated by the main process do not block, so that it is easier for you to

test your implementation.

- In both `wait` and `wait2`, while cleaning up a process' PCB, you must reset the fields related to the scheduling metadata of the process to a default value of 0.

Testing: You have been provided with a test program `q1tc.c`. In the test program, a parent forks `NCHILD` number of children, each of which performs some significant amount of computation and exits. The value of `NCHILD` is passed as a **command line argument**, and it takes a default value of 2. Just to test if your implementation of **`wait2`** works, you can set the number of children to 1.

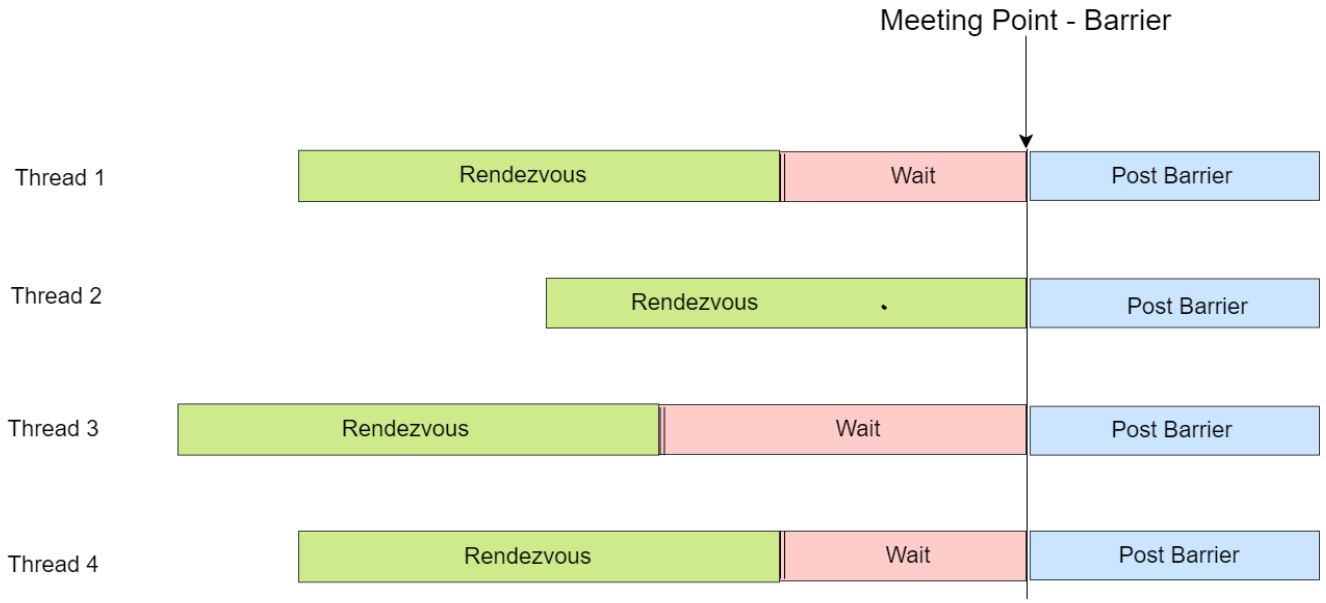
Here is a sample output with five children:

```
$ q1tc 5
Parent process started
Forking child processes
pid: 4, arrival time: 523, first scheduled time: 524, exit time: 532
pid: 5, arrival time: 523, first scheduled time: 532, exit time: 540
pid: 6, arrival time: 523, first scheduled time: 541, exit time: 550
pid: 7, arrival time: 523, first scheduled time: 551, exit time: 559
pid: 8, arrival time: 524, first scheduled time: 559, exit time: 567
Parent process has reaped all the child processes
```

If your implementation is correct, then you should expect the first scheduled time of process to increase with an increase in the arrival time. Also, since all the children in the test program are non-blocking, the first scheduled time of one child should never be less than the exit time of another child.

Q2. The Reusable Barrier (10 marks)

A barrier is a type of synchronization method that forces multiple worker threads to wait until all threads have reached a particular execution point (barrier) before any thread continues.



You are required to implement a **reusable** barrier using **pthreads** and **semaphores** on Linux, where the workers perform a series of steps in a loop, and use the same barrier code to synchronize for each iteration of the loop. All threads should wait for each other at the start of each iteration, and carry out steps for an iteration only after all threads have completed the previous iteration.

The `auxiliary_files` folder has `reusable_barrier.c` . Use it as a starting point for your code. It maintains a global C initialized to 0. The main parent thread reads two integers **N**, **K** from STDIN.

The `worker()` function runs a loop K times, increments C at the start of every iteration and prints C and thread number at the end of the iteration.

You will have to add functionality, e.g., creating worker threads, synchronization for shared variables etc.

In your code:

- Create N worker threads and number them from 0 to N-1. Pass this thread number as an argument to the thread routine - `worker()`.
- Add proper synchronization such that numbers are printed as follows:
  - Only **N**, **2*N**, **3*N**, … **K*N** numbers are allowed since each worker executes K loops
  - Number is never less than number printed on line above
  - Each number should be displayed exactly n times, once for each thread number

These functions will be useful to implement the solution.

```
sem_t barrier_sem;
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

```
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                           void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
```

You can use man pages for more details on the above functions. e.g., `man sem_post`

Note:

1. All threads should be joined by the main thread.
2. You **must only use** semaphores to achieve the synchronization between the worker threads.
3. The thread number ordering may differ from that provided in sample outputs, i.e., for example in example 1 you may get thread_num 1 printed before thread_num 2.

Sample Outputs:

1. With N=3, K=1. This is similar to the standard barrier synchronization case.

```
purvi2001@purvi:~$ gcc -o reusable_barrier reusable_barrier.c -lpthread
purvi2001@purvi:~$ ./reusable_barrier
N: 3
K: 1
thread_num: 2, C = 3
thread_num: 1, C = 3
thread_num: 0, C = 3
purvi2001@purvi:~$
```

2. N=2, K=3

```
purvi2001@purvi:~$ ./reusable_barrier
N: 2
K: 3
thread_num: 1, C = 2
thread_num: 0, C = 2
thread_num: 1, C = 4
thread_num: 0, C = 4
thread_num: 1, C = 6
thread_num: 0, C = 6
purvi2001@purvi:~$
```

3. N=4, K=3

```
purvi2001@purvi:~$ ./reusable_barrier
N: 4
K: 3
thread_num: 3, C = 4
thread_num: 2, C = 4
thread_num: 1, C = 4
thread_num: 0, C = 4
thread_num: 2, C = 8
thread_num: 3, C = 8
thread_num: 0, C = 8
thread_num: 1, C = 8
thread_num: 0, C = 12
thread_num: 3, C = 12
thread_num: 1, C = 12
thread_num: 2, C = 12
purvi2001@purvi:~$
```

Q3. `return of the clone` (10 marks)

Implement a `clone()` system call, like you did in Lab 8. 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 quiz, like in Lab8, we will use the **`struct proc`** itself as the PCB-cum-TCB for all the threads. We have provided you with a **`proc.h`** file to patch, which has a modified **`struct proc`** adhering to our design of the TCB. **You must not modify it any further**. We have already added some thread-relevant fields to the PCB, such as:

- 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)

We will be using the **`pid`** field of the **`struct proc`** as the thread-ID for this lab.

The signature of the clone system call is as follows:

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

This system call will effectively create a kernel thread and the thread will start execution at the function **fn** with **arg1** and **arg2** as the function's arguments using the given **stack** argument as its stack. Note that we are designing the system call to work only with functions that take two arguments. Also, **the order in which the two arguments are pushed onto the stack is critical, and will be tested in the testcases**. This is where this question differs from the one in Lab 8.

The arguments of the system call are:
- **`fn`** -> a target function, which indicates the start point of execution of the thread
- **`arg1`** -> pointer to the first argument of function fn
- **`arg2`** -> pointer to the second 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.

Notes:

1. The stack pointer is passed from a 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.

We have implemented a wrapper function **create_thread** in the file **ulib.c**.

2. Following are the expectations for 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. The base pointer also needs to be intiliazed.
   - A new thread once created and initialized, when scheduled, should start executing the user specified function.
   - Each thread function MUST end execution with an explicit call to **exit**(). We have ensured this in the testcases that we have provided you with.
   - We have updated the **allocproc**, **exit**, **wait** and **kill** implementations based on the requirements. Make sure that your implementation of **clone** is consistent with the changes that we have made. **DO NOT CHANGE ANY EXISTING CODE.** Only add code which is relevant to the implementation of the **clone** system call.

We have implemented a **join** system call to reap threads, which has been used in the test cases. All that is left for you to do is to implement the **clone** system call.

Testing: You have been provided with two test programs:

1. `q3-tc-var.c`, in which multiple threads update a common global variable, irrespective of their arguments. A sample output is given below:

```
$ q3-tc-var
Calling Process Print VAR value: 0
Sum of args: 0, VAR: 1
Sum of args: 1, VAR: 2
Sum of args: 2, VAR: 3
Sum of args: 3, VAR: 4
Sum of args: 4, VAR: 5
All threads joined, VAR value: 5
```

2. `q3-tc-var.c`, in which multiple threads compute the sums of different parts of two arrays. Each thread identifies the relevant parts of the arrays which it has to sum up, using the two arguments that it has been provided with. A sample output is given below:

```
$ q3-tc-array
Creating threads to sum up two arrays
Argument 1: 0, Argument 2: 2 Sum Value: 29
Argument 1: 0, Argument 2: 3 Sum Value: 29
Argument 1: 1, Argument 2: 2 Sum Value: 55
Argument 1: 1, Argument 2: 3 Sum Value: 46
All threads joined
Sum of thread calls is equal to that of both array sums, value: 159
```

Note that in the above testcases, the domains of the two arguments are disjoint. This means that you have to set-up your stack properly for your code to pass this testcase.

Submission Instructions

- All submissions are to be done on moodle only.
- Name your submission as cs333_<roll_number>.tar.gz
- The tar should contain the following files in the following directory structure:
  <rollnumber>_labquiz3/

  |__q1/

  |__< 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, . . . >

  |__q2/

  |__reusable_barrier.c

  |__q3/

  |__< 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, . . . >

  Please adhere to it strictly.

- Your modified code/added code should be well commented on and readable.
- `tar -czvf cs333_<rollnumber>.tar.gz <rollnumber>_labquiz3`

Deadline: Monday, 17th October 2022, 5:00 PM. Leave your submission tar file in a directory to be specified during the quiz. A script will pick this up (only if it is named in the exact format specified).