Operating Systems
Assignment 2 – *Hard*

Instructions:

1. The assignment has to be done individually.
2. You can use Piazza for any queries related to the assignment and avoid asking queries on the last day.

1 Scheduler for HPC Workloads

High-performance computing (HPC) workloads are typically parallel applications that require multiple threads or processes to execute simultaneously while frequently communicating with each other. These workloads are commonly found in scientific simulations, machine learning, large-scale data processing, and computational fluid dynamics.

Traditional scheduling mechanisms in the Linux kernel, such as the completely fair scheduler (CFS), struggle to efficiently manage tightly coupled parallel applications. This is because it does not account for interdependent scheduling needs in HPC workloads. Since CFS schedules tasks independently based on CPU shares, it can cause communication delays when one thread of a parallel job is scheduled while others are waiting. This increases the overall latency and disrupts the synchronization between threads. To resolve this, we can use the gang scheduling mechanism.

1.1 Gang Scheduling Mechanism

Gang scheduling ensures that all threads of a parallel job execute simultaneously on different processors or cores. This approach significantly reduces communication delays and synchronization overheads. As a result, it is suitable for HPC workloads. Let us elaborate on the workings of this mechanism:

1. Assume we have a system with m physical CPU cores. We want to execute an HPC workload i that consists of n preemptable threads, where $n \leq m$.
2. The threads of the HPC workload must register themselves for gang scheduling under the same gang ID using the *sys_register_gang* system call. You have to create a new scheduling class, `SCHED_GANG` so that the kernel will be able to recognize these threads.

```
int sys_register_gang(int pid, int gangid, int exec_time)
```

- *pid:* pid of the process.
- *gangid:* unique id to represent threads of a gang.
- *exec_time (**unit**: seconds):* The execution time of the thread (th_i). You need to ensure that the work that you are doing will finish by *exec_time* seconds. You can ensure this by running the function of interest multiple times in isolation and calculating the maximum time. This assumption will hold since the demo will be on your machine.

If the total number of threads of an HPC workload exceeds the number of physical CPU cores, the corresponding system call should return -22; otherwise, its should return 0.

3. Each thread is pinned to a unique CPU core using the *sched_setaffinity* system call to prevent thread migration.

4. One CPU core is designated as the *gang governor core*, which is responsible for scheduling the gang. When a gang task is ready, it sends an inter-processor interrupt (IPI) to other CPU cores to wake up and execute the waiting gang threads.

5. Upon receiving an IPI signal, the core triggers *need_resched*, preempts any lower-priority task, and executes its assigned gang thread.

6. To notify the kernel that a gang thread has finished execution, it uses the *sys_exit_gang* system call. Subsequently, we remove the thread from a gang. If all threads in the gang exit, the gang is considered complete, and the scheduler selects the next job in the queue.

   `int sys_exit_gang(pid)`

   - *pid:* pid of the process.

   If the system call was successful, it should return 0; otherwise, it should return -22.

7. To list all the registered processes under a gang, the *sys_list* system call must be invoked with the following signature:

   `void sys_list(int gangid, int* pids)`

   - *gangid:* id of the gang.

   This system call should return the PIDs of the threads that belong to the given gang ID.

8. You must execute multiple threads of an HPC workload to test the scheduler. The pseudo-code for one of the threads is shown below:

```
int main(int argc, char *argv[]) {
  exec_time = argv[1];
  gangid = argv[2];
```

```
  pid = getpid();
  syscall(sys_register_gang(pid, gangid, exec_time));

  perform_job(exec_time); // you must ensure that this
      function finishes within the exec_time

  syscall(sys_exit_gang(pid));
}
```

2 Report

Page limit: 10
The report should mention the implementation methodology for the gang scheduling class. Showing small, representative code snippets or the pseudo-code in the report is sufficient.

- Implementation of the gang scheduling class.
- Challenges faced and the novelties introduced (if any).
- Submit a PDF file containing all the relevant details.

3 Submission Instructions

1. There will be a demo for assignment 2 in which you must demonstrate how the gang scheduler works with a dummy HPC workload. A viva will follow for testing your theoretical and practical understanding within the context of the assignment. Note that regardless of the code that you submit, the viva performance is vitally important.

2. Create a patch file with the gang scheduler.

   ```
   sudo diff -rupN  linux-change/ linux-base/ > res_usage.patch
   # linux-base refers to the vanilla version of the Linux
   kernel, and linux-change refers to the modified Linux kernel.
   ```

3. Create a zip file that contains the report and gang_sched.patch files and then name the zip file as, $assignment2\_hard\_\langle entryNumber\rangle.zip$. Submit this zip file on Moodle. Entry number format: 2020CSZ2445. *Note that all English letters are in capitals.*