Assignment 2: Scheduling Policy Demonstration Program

1. Describe how you implemented the program in detail. (20%)

Main Thread

Parse program arguments using getopt

```
while ((opt = getopt(argc, argv, "n:t:s:p:")) != -1) {
        switch (opt) {
            case 'n':
                num_threads = atoi(optarg);
                break;
            case 't':
                time_wait = atof(optarg);
                break;
            case 's':
                s1 = strdup(optarg);
                token = strtok(s1, ",");
                int i = 0;

                while(token!=NULL){
                    strcpy(pthread_policy[i],token);
                    //printf("policy : %s\n", pthread_policy[i]);
                    token =strtok(NULL,",");
                    i++;
                }
                break;
            case 'p':

                s2 = strdup(optarg);
                token = strtok(s2,",");
                int j = 0;
                while(token!=NULL){
                    priorities[j] = atoi(token);
                    //printf("priority : %d\n", priorities[i]);
                    token =strtok(NULL,",");
                    j++;
                }
                break;
            default:
                return 1;
        }
    }
```

`getopt` is a function that is used to parse command-line options and arguments. The third argument to `getopt` ("n:t:s:p:") specifies the expected options. If an option requires an argument, a colon (:) is placed after the option character.

This code snippet is using the `getopt` function to parse command-line arguments. The `getopt` function is a standard library function in C, used for parsing command-line options and arguments

Inside the loop, a `switch` statement is used to handle each option.

- For option 'n', it takes the argument following 'n' (specified by `optarg`), converts it to an integer using `atoi`, and assigns it to the variable `num_threads`.
- For option 't', it takes the argument following 't', converts it to a floating-point number using `atof`, and assigns it to the variable `time_wait`.
- For option 's', it takes the argument following 's', tokenizes it using `strtok` with ',' as the delimiter, and populates the array `pthread_policy` with the parsed values.
- For option 'p', it takes the argument following 'p', tokenizes it using `strtok` with ',' as the delimiter, and populates the array `priorities` with the parsed values.
- The `default` case is executed if an unknown option is encountered, and in this case, the program returns with an exit code of 1.

Create <num_threads> worker threads

```
thread_info_t thread_data_array[num_threads];
pthread_t child_thread_id[num_threads];
struct sched_param param[num_threads];
pthread_attr_t attr[num_threads];
```

1. `thread_info_t thread_data_array[num_threads];`
    - This declares an array named `thread_data_array` of type `thread_info_t` with a size of `num_threads`.
    - `thread_info_t` is presumably a user-defined type, which is likely a struct type used to hold information about each thread. The array is used to store information about each worker thread.
2. `pthread_t child_thread_id[num_threads];`
    - This declares an array named `child_thread_id` of type `pthread_t` with a size of `num_threads`.
    - `pthread_t` is the data type used to represent a thread identifier in the pthreads library. This array is used to store the thread identifiers of the worker threads created.
3. `struct sched_param param[num_threads];`
    - This declares an array named `param` of type `struct sched_param` with a size of `num_threads`.

    - `struct sched_param` is a structure used to store scheduling parameters for threads. Each element of the array is intended to hold the scheduling parameters for a corresponding thread.

4. `pthread_attr_t attr[num_threads];`
    - This declares an array named `attr` of type `pthread_attr_t` with a size of `num_threads`.
    - `pthread_attr_t` is the data type used to represent thread attributes in the pthreads library. Thread attributes include various settings such as the scheduling policy, stack size, etc. Each element of the array is intended to hold the attributes for a corresponding thread.

Set CPU affinity

- `int cpu_id = 2;`: This line initializes an integer variable `cpu_id` and sets its value to 2. This value represents the CPU core to which the thread's affinity is going to be set.
- `cpu_set_t cpuset;`: Declares a variable `cpuset` of type `cpu_set_t`. This type is used to represent a set of CPU cores.
- `CPU_ZERO(&cpuset);`: Clears all the bits in the CPU set `cpuset`, effectively initializing it to an empty set.
- `CPU_SET(cpu_id, &cpuset);`: Sets the bit corresponding to the specified `cpu_id` in the CPU set `cpuset`, indicating that the thread should be allowed to run on CPU core 2.
- `pthread_t th = pthread_self();`: Retrieves the thread ID of the calling thread and assigns it to the variable `th`. This line gets the identifier of the current thread.
- `pthread_setaffinity_np(th, sizeof(cpuset), &cpuset);`: Sets the CPU affinity of the thread identified by `th` to the CPU set specified by `cpuset`. This means that the thread will be scheduled to run on the CPU core(s) included in `cpuset`. The function `pthread_setaffinity_np` is a non-portable pthreads extension function for setting CPU affinity.

```
int cpu_id = 2; // set thread to cpu2
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(cpu_id, &cpuset);
    //printf("pthread cpu%d\n",cpu_id);
    pthread_t th =pthread_self();
    pthread_setaffinity_np(th, sizeof(cpuset), &cpuset);
    //printf("sched_getcpu = %d\n",sched_getcpu());
```

Set the attributes to each thread

1. `pthread_barrier_init(&barrier, NULL, num_threads+1);`: Initializes a barrier (`barrier`) with a count of `num_threads + 1`. The `+1` is for the main thread or an

2. `thread_data_array[i].thread_id = i;`: Assigns a unique `thread_id` to each element in the `thread_data_array`.
3. `thread_data_array[i].time_wait = time_wait;`: Assigns the specified `time_wait` to each thread in the `thread_data_array`.
4. `pthread_attr_init(&attr[i]);`: Initializes the thread attribute for the current thread.
5. `pthread_attr_setinheritsched(&attr[i], PTHREAD_EXPLICIT_SCHED);`: Sets the thread inheritance attribute to `PTHREAD_EXPLICIT_SCHED`, indicating that the thread's scheduling attributes are explicitly set by the program.
6. `pthread_attr_getinheritsched(&attr[i], &policy);`: Gets the thread inheritance (scheduling policy) and stores it in the `policy` variable.
7. **Scheduling Policy Configuration**:
    - `if (strcmp(pthread_policy[i], "NORMAL") == 0)`: Checks if the specified policy for the current thread is "NORMAL."
        - `pthread_attr_setschedpolicy(&attr[i], SCHED_OTHER);`: Sets the scheduling policy to `SCHED_OTHER` (normal scheduling).
    - `else`: If the policy is not "NORMAL" (i.e., it's "FIFO"):
        - `pthread_attr_setschedpolicy(&attr[i], SCHED_FIFO);`: Sets the scheduling policy to `SCHED_FIFO` (FIFO scheduling).
8. **Priority Configuration**:
    - `param[i].sched_priority = priorities[i];`: Assigns the specified priority to the `param` structure for the current thread.
    - `if (priorities[i] != -1)`: Checks if a priority value other than -1 is specified.
        - `pthread_attr_setschedparam(&attr[i], &param[i]);`: Sets the scheduling parameters (priority) for the current thread.
9. `pthread_create(&child_thread_id[i], &attr[i], thread_func, (void *)&thread_data_array[i]);`: Creates a new thread with the specified attributes (`attr[i]`) and associates it with the function `thread_func`. Passes the corresponding `thread_data_array[i]` as an argument to the thread.

```
/* 4. Set the attributes to each thread */
  int policy;
  pthread_barrier_init(&barrier, NULL, num_threads+1);
  for (int i = 0; i < num_threads; i++) {
        thread_data_array[i].thread_id=i;
        thread_data_array[i].time_wait=time_wait;
        pthread_attr_init(&attr[i]); /*initialize thread attribute*/
              pthread_attr_setinheritsched(&attr[i],PTHREAD_EXPLICIT_SCHED); /*set thread inheritance*/
```

```
            pthread_attr_getinheritsched(&attr[i],&policy); /*get thread inheritance*/


            //other
        if(strcmp(pthread_policy[i],"NORMAL") == 0)
        {
            pthread_attr_setschedpolicy(&attr[i], SCHED_OTHER);

        }
        else //FIFO
        {
            pthread_attr_setschedpolicy(&attr[i], SCHED_FIFO);

        }

        param[i].sched_priority=priorities[i];
        if(priorities[i]!=-1)
        {
            pthread_attr_setschedparam(&attr[i],&param[i]);/*设置线程的调度参数*/
        }
        pthread_create(&child_thread_id[i], &attr[i], thread_func, (void *)
        &thread_data_array[i]);

    }
```

Start all threads at once

```
pthread_barrier_wait(&barrier);
```

Wait for all threads to finish

```
for (int i = 0; i < num_threads; i++) {
    pthread_join(child_thread_id[i], NULL);
}
pthread_barrier_destroy(&barrier);
```

This code is responsible for waiting for each worker thread to finish its execution using `pthread_join` and then destroying the barrier created earlier using `pthread_barrier_destroy`. Let's break down the code:

Use “typedef” struct to build the information that will be used by the worker thread first.

```
typedef struct {
    pthread_t thread_id;
    int thread_num;
    int sched_policy;
    int sched_priority;
    float time_wait;
} thread_info_t;
```

Worker Thread

```
void *thread_func(void *arg)
{

  thread_info_t *data = (thread_info_t*)arg;
  pthread_barrier_wait(&barrier);
  float time_waiting = data->time_wait;
  int taskid = data->thread_id;

    /* 1. Wait until all threads are ready */

    //pthread_barrier_wait(&barrier);
    /* 2. Do the task */
  for (int i = 0; i < 3; i++) {

        // Busy for <time_wait> seconds
    printf("Thread %d is running\n", taskid);
    struct timeval start;
    struct timeval end;
    double start_time, end_time;
    gettimeofday(&start,NULL);
    start_time = (start.tv_sec*1000000+(double)start.tv_usec)/1000000;
    //printf("start_time: %lf\n", start_time);
    while(1)
    {
    gettimeofday(&end,NULL);
    end_time = (end.tv_sec*1000000+(double)end.tv_usec)/1000000;
    //printf("start_time: %lf\n", start_time);
    if (end_time > start_time+time_waiting);
        break;
  }
  sched_yield();
    }
    /* 3. Exit the function  */
  pthread_exit(NULL);
}
```

1. `thread_info_t *data = (thread_info_t*)arg;`: Casts the void pointer `arg` to the correct type (`thread_info_t*`). This allows the function to access the thread-specific information passed as an argument.

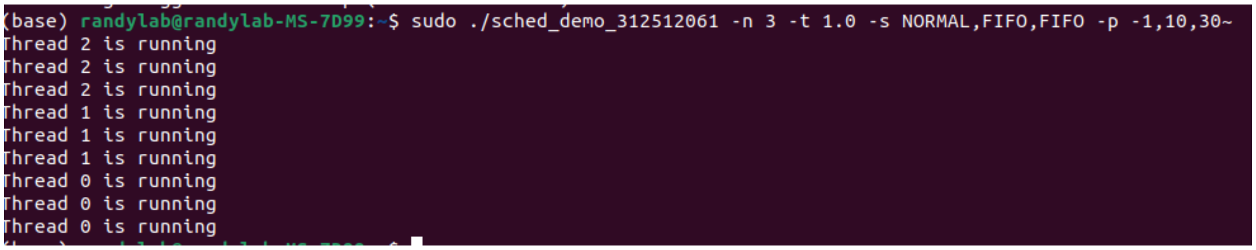
2. `pthread_barrier_wait(&barrier);`: Waits for all threads to reach this point before proceeding. This is used to synchronize the threads at the beginning of the function.

3. `float time_waiting = data->time_wait;` and `int taskid = data->thread_id;`: Retrieves thread-specific information from the `thread_info_t` structure.

4. `for (int i = 0; i < 3; i++) { ... }`: This loop simulates some computational work that the thread is doing. It runs three times, each time printing a message indicating that the thread is running.
5. **Time Measurement**: Measures the start time and checks if the specified waiting time has passed using a loop that continuously checks the current time.
6. `sched_yield();`: Yields the CPU to allow other threads to run. This is a system call that suggests to the scheduler that the current thread is willing to give up the CPU, allowing other threads to run.
7. `pthread_exit(NULL);`: Exits the thread, returning `NULL` as the exit status.

2. Describe the results of `./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30` and what causes that. (10%)

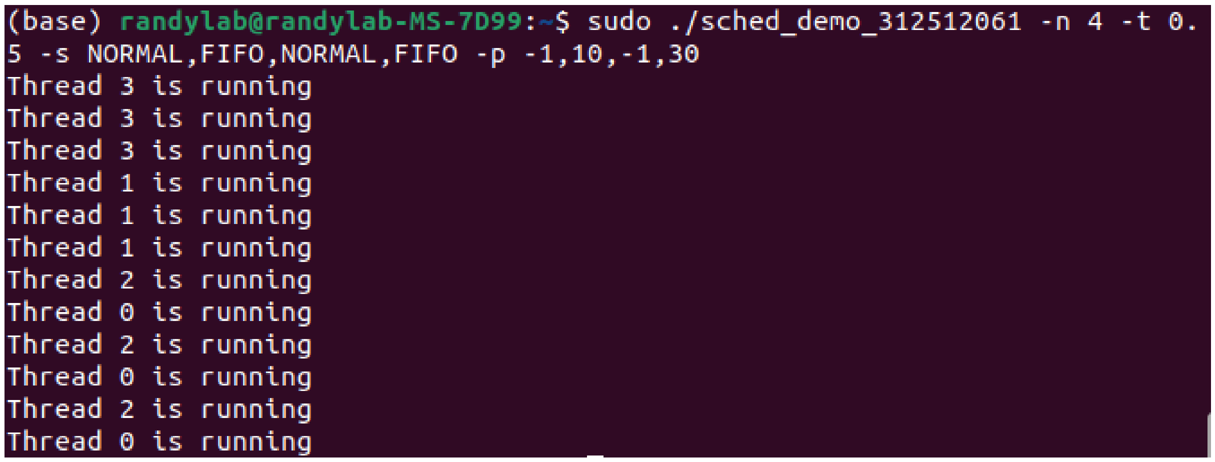
This code creates three threads, each performing a task that takes 1 second to complete. The thread using the First-In-First-Out (FIFO) scheduling policy starts first. Thread 2 has a priority of 30, making it the highest priority and thus executing first. Following that, Thread 1 with a priority of 10 executes, and finally, the thread with a normal priority (priority 0) executes.



```
(base) randylab@randylab-MS-7D99:~$ sudo ./sched_demo_312512061 -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30~
Thread 2 is running
Thread 2 is running
Thread 2 is running
Thread 1 is running
Thread 1 is running
Thread 1 is running
Thread 0 is running
Thread 0 is running
Thread 0 is running
```

3. Describe the results of `./sched_demo -n 4 -t 0.5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,10,-1,30`, and what causes that. (10%)

Creates four threads, each performing a task that takes 0.5 seconds to complete. Similarly, the thread using the First-In-First-Out (FIFO) scheduling policy starts first. Thread 3 has a priority of 30, making it the highest priority and thus executing first. Following that, Thread 1 with a priority of 10 executes. Finally, the threads with normal priority (priority 0), Thread 0 and Thread 2, execute. Since the NORMAL scheduling policy is fair-time, these two threads are evenly distributed, resulting in Thread 2 and Thread 0 taking turns to execute.



```
(base) randylab@randylab-MS-7D99:~$ sudo ./sched_demo_312512061 -n 4 -t 0.
5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,10,-1,30
Thread 3 is running
Thread 3 is running
Thread 3 is running
Thread 1 is running
Thread 1 is running
Thread 1 is running
Thread 2 is running
Thread 0 is running
Thread 2 is running
Thread 0 is running
Thread 2 is running
Thread 0 is running
```

4. Describe how did you implement n-second-busy-waiting? (10%)

Utilizing `gettimeofday` for this task offers precision down to microseconds. The `timeval` data structure encompasses `tv_sec` for seconds and `tv_usec` for microseconds. The function `gettimeofday(&start, NULL);` is employed to retrieve the current time, and the process waits until `end_time` surpasses `start_time + time_waiting`.

```
for (int i = 0; i < 3; i++) {

        // Busy for <time_wait> seconds
        printf("Thread %d is running\n", taskid);
  struct timeval start;
  struct timeval end;
  double start_time, end_time;
  gettimeofday(&start,NULL);
  start_time = (start.tv_sec*1000000+(double)start.tv_usec)/1000000;
  //printf("start_time: %lf\n", start_time);
  while(1)
  {
  gettimeofday(&end,NULL);
  end_time = (end.tv_sec*1000000+(double)end.tv_usec)/1000000;
  //printf("start_time: %lf\n", start_time);
  if (end_time > start_time+time_waiting);
      break;
  }
  sched_yield();
    }
```