OS HW04 GROUP 21

Part 1:Implementation Detail

1. **threads/kernel.h**

   Since we want to execute processes with priority scheduling, we need to store both the filenames and their corresponding priorities. To achieve this, we declare an array, execpriority, to store the priority for each executable.

```
char*   execfile[10];
int execpriority[10];
```

   In addition, to enable priority scheduling, the Exec() function in the class Kernel needs to be modified to include the ‘priority’ argument, which specifies the execution priority.

```
int Exec(char* name, int priority);
```

2. **threads/kernel.cc**

   Since the ‘-ep’ command flag is intended to execute the process with priority scheduling, we added a condition for ‘-ep’ in Kernel::Kernel to handle argument parsing. First, we use assert to verify that there are two input arguments for the process: the filename and its priority. Next, we retrieve the process name and priority, and ensure that the priority is within the valid range of 0 to 149. Finally, we store the filename and priority in their respective arrays for further processing.

```
else if (strcmp(argv[i], "-ep") == 0) {
    ASSERT(i + 2 < argc);
    char* processName = argv[i + 1];
    int priority = atoi(argv[i + 2]);
    ASSERT(priority >= 0 && priority <= 149);
    execfile[++execfileNum] = processName;
    execpriority[execfileNum] = priority;
    i += 2;
}
```

   In the ExecAll() function, we pass the priority argument when invoking the Exec() function to specify the priority for the corresponding process.

```
void Kernel::ExecAll()
{
    for (int i = 1;i <= execfileNum;i++) {
        int a = Exec(execfile[i],execpriority[i]);
    }
    currentThread->Finish();
}
```

   As with the modification in kernel.h, we add priority as an input argument for the Exec() function. After creating a new thread, we set its priority using the parsed value. The SetPriority() function, implemented in thread.cc, assigns the priority to the thread, allowing the scheduler to access and use it when managing the thread's execution.

```
int Kernel::Exec(char* name, int priority)
{
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->SetPriority(priority);
```

3. **threads/thread.h**
   To allow the scheduler to access the thread's priority, we first declare a ‘priority’ variable in the Thread class to represent its execution priority. We also define two functions: SetPriority() and GetPriority(), which are implemented in the thread.cc.

```
int priority;
void SetPriority(int p);
int GetPriority();
```

4. **threads/thread.cc**
   The SetPriority() function stores the parsed priority value in the ‘priority’ variable during the thread's initialization. The GetPriority() function retrieves the thread's priority by returning the ‘priority’ variable.

```
// For the new thread, set its priority.
void Thread::SetPriority(int p) {
    priority = p;
}

// For comparison, we need to retrieve the priority.
int Thread::GetPriority(){
    return priority;
}
```

5. **threads/scheduler.h**
   Since we want to implement a priority queue scheduler, we replace the original implementation using a List with a SortedList. We declare readyList as a SortedList to maintain threads in order of their priority.

```
private:
    SortedList<Thread*> *readyList;
```

6. **threads/scheduler.cc**
   As declared in scheduler.h, in Scheduler::Scheduler(), we use ‘readyList’ as a SortedList to maintain threads in order based on their priority. However, the original implementation of SortedList sorts elements in ascending order, whereas we want higher-priority threads (with larger priority values) to appear at the front of the list. To achieve this, the compare function for x and y is defined to return “y's priority - x's priority”. This ensures that when x has a higher priority, the return value is negative, causing x to be inserted at the front of the list.

```
int CompareThreads(Thread* x, Thread* y) {
    return y->GetPriority() - x->GetPriority();
}
Scheduler::Scheduler()
{
    readyList = new SortedList<Thread*>(CompareThreads);
    toBeDestroyed = NULL;
}
```

   Since we are using SortedList instead of List, we modify the code in ReadyToRun() to use Insert() instead of Append() for adding threads to the ‘readyList’. The Insert() method places the thread in the

list according to its priority, ensuring the correct order. Additionally, we use DEBUG to print a type [A] message, indicating the insertion of a process into the ready queue.

```
void Scheduler::ReadyToRun(Thread* thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    thread->setStatus(READY);

    // Todo -----
    readyList->Insert(thread);
    DEBUG(dbgPriority, "[A] Tick [" << kernel->stats->totalTicks
          << "]: Process [" << thread->getName()
          << "] is inserted into queue.");
    // -------
}
```

In FindNextToRun(), we use DEBUG to print a type [B] message indicating that a process has been removed from the list.

```
Thread* Scheduler::FindNextToRun()
{
    if (readyList->IsEmpty()) {
        return NULL;
    }
    else {
        // Todo -----
        Thread *nextThread = readyList->RemoveFront();
        DEBUG(dbgPriority, "[B] Tick [" << kernel->stats->totalTicks
              << "]: Process [" << nextThread->getName()
              << "] is removed from queue.");
        return nextThread;
        // ---------
    }
}
```

In Scheduler::Run(), before switching to the new thread, we use DEBUG to print a type [C] message, indicating that a process has changed its scheduling priority.

```
void Scheduler::Run(Thread* nextThread, bool finishing)
{
    // Todo -----
    DEBUG(dbgPriority, "[C] Tick [" << kernel->stats->totalTicks
          << "]: Process [" << nextThread->getName()
          << "] is now selected for execution, Process ["
          << oldThread->getName() << "] is replaced.");
    // --------
    SWITCH(oldThread, nextThread);
```

7. **threads/alarm.cc**

   To implement non-preemptive priority scheduling, we modify Alarm::CallBack() so that preemption only occurs when the currently running thread transitions to the waiting state. That is, preemption occurs only if the thread's status becomes BLOCKED, at which point YieldOnReturn() is called to perform a context switch and preempt the thread.

```
void Alarm::CallBack()
{
    Interrupt* interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();
    Thread *currentThread = kernel->currentThread;

     if (status != IdleMode) {
        if (currentThread->getStatus() == BLOCKED) {
            // If running -> waiting => preempted
            interrupt->YieldOnReturn();
        } else {
            // If not => non-preemptive
        }
    }
}
```

8. **lib/debug.h**

   We add ‘dbgPriority’ as the debugging flag to log and monitor changes in the priority queue, with its value set to 'z'. This flag controls whether the message will be printed.

```
const char dbgPriority = 'z';
```

Part 2:Contribution

1. **Describe details and percentage of each member’s contribution.**

   313551047 陳以瑄, contribution (50%):

   Implementation

   111550150 俞祺譯, contribution (50%):

   Report