Operating System HW3

Scheduler Simulator

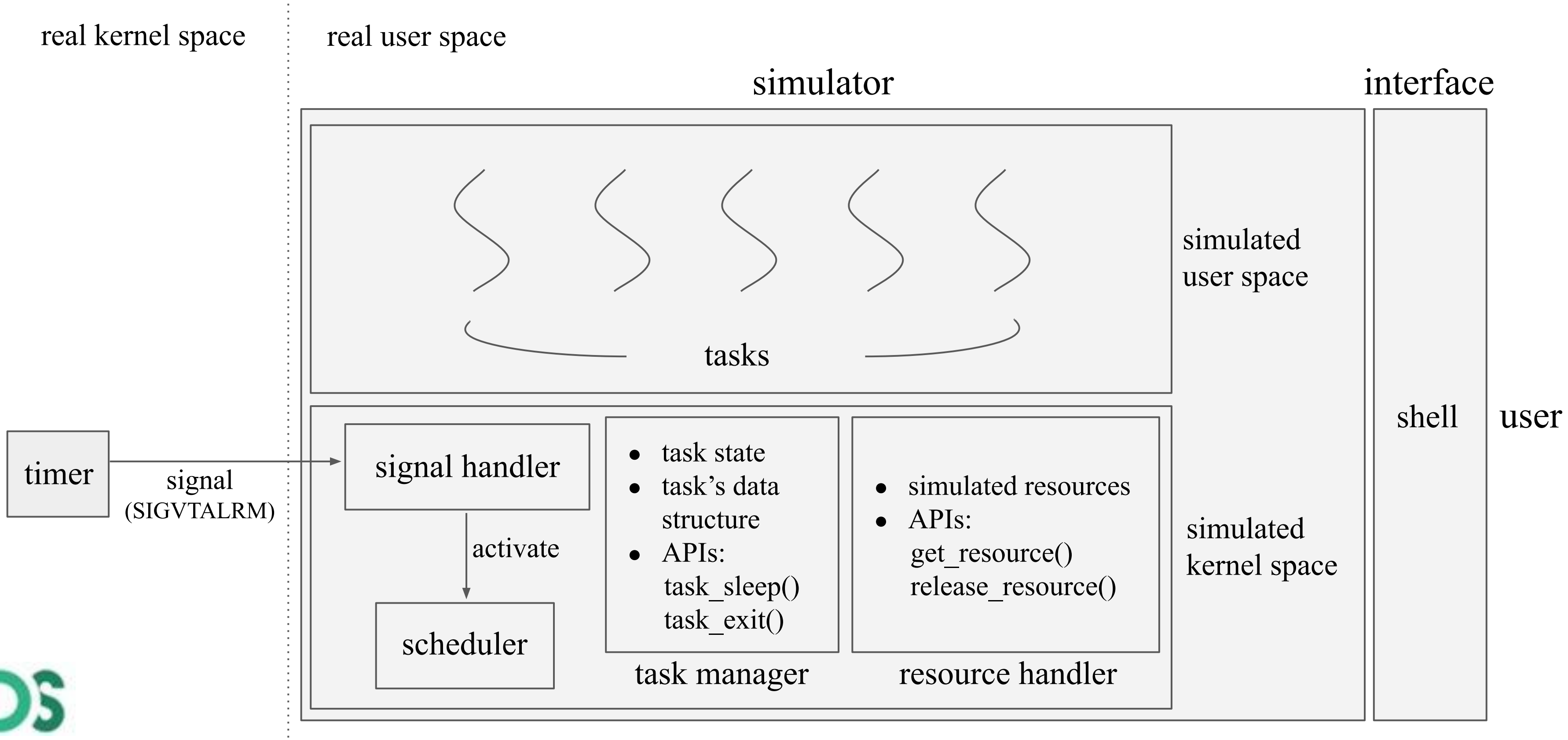
Due date: 12/16 23:59

HW3 TA email: p76101047@gs.ncku.edu.tw

Objective

- Understand how to implement user-level thread scheduling
- Understand how signal works in Linux
- Understand how scheduling algorithms affect results

Architecture



Requirement (1/5)

1. Tasks & task manager
   - Use ucontext and the related APIs to create tasks
   - Each task runs a function defined in 'function.c', where **all the functions are provided by TA** and **should not be modified**
   - Implement a task manager to manage tasks, including their state, data structures and so on
   - Implement task state-related APIs that can be used by the tasks (*described in slide 10-11*)
     1. void task_sleep(int *msec_10*);
     2. void task_exit();

Requirement (2/5)

2. Task scheduler
   - Use ucontext and the related APIs to do context switch
   - Implement three scheduling algorithms
     - FCFS
     - RR with time quantum = 30 (ms)
     - priority-based preemptive scheduling, smallest integer → highest priority
   - The algorithm is determined at execution: ***./scheduler_simulator {algorithm}***
     - *algorithm* = FCFS / RR / PP
   - Once the scheduler dispatches CPU to a task, print a message in the format:

     Task {task_name} is running.
   - If there are no tasks to be scheduled, but there are still tasks waiting, print a message in the format:

     CPU idle.

Requirement (2/5)

function.h

```c
#ifndef FUNCTION_H
#define FUNCTION_H

void test_exit();
void test_sleep();
void test_resource1();
void test_resource2();
void idle();

void task1();
void task2();
void task3();
void task4();
void task5();
void task6();
void task7();
void task8();
void task9();

#endif
```

function.c

```c
void test_exit()
{
    task_exit();
    while (1);
}

void test_sleep()
{
    task_sleep(20);
    task_exit();
    while (1);
}

void test_resource1()
{
    int resource_list[3] = {1, 3, 7};
    get_resources(3, resource_list);
    task_sleep(5);
    release_resources(3, resource_list);
    task_exit();
    while (1);
}

void test_resource2()
{
    int resource_list[2] = {0, 3};
    get_resources(2, resource_list);
    release_resources(2, resource_list);
    task_exit();
    while (1);
}

void idle()
{
    while (1);
}
```

You may need this when CPU is idle.

Requirement (3/5)

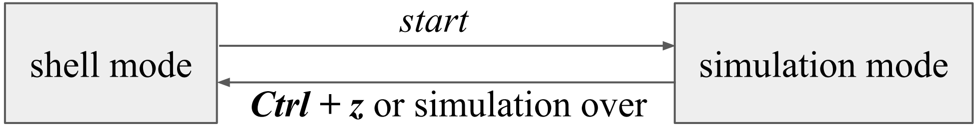
3. Resource handler
   - Implement resource-related APIs that can be used by the task (*described in slide 12-13*)
     i. void get_resource(int *count*, int **resource_list*);
     ii. void release_resource(int *count*, int **resource_list*);
   - There should be 8 resources with id 0-7 in the simulation.
   - How to simulate resources is up to your design. For example, you can use a boolean array,

     resource_available = { true, false, true, ... }

Requirement (4/5)

4. Timer & signal handler
    - Use related system calls to set a timer that should send a signal (SIGVTALRM) every 10 ms
    - The signal handler should do the followings:
        i. Calculate all task-related time (granularity: 10ms)
        ii. Check if any tasks' state needs to be switched
        iii. Decide whether re-scheduling is needed

Requirement (5/5)



5. Command line interface
    - Use HW1’s shell as the simulator’s CLI (HW1’s code is provided by TA, you can also use your own code)
    - Should support four more commands (*details are described in slide 14-17*)
        1. ***add***: Add a new task
        2. ***del***: Delete a existing task
        3. ***ps***: Show the information of all tasks, including TID, task name, task state, running time, waiting time, turnaround time, resources occupied and priority (if any)
        4. ***start***: Start or resume simulation
    - ***Ctrl + z*** should pause the simulation and switch to shell mode
    - Timer should stop in the shell mode and resume when the simulation resumes
    - When the simulation is over, switch back to shell mode after printing a message in the format:

      Simulation over.

API Description (1/4)

- void task_sleep(int *msec_10*);
  - Print a message in the format: ***Task {task_name} goes to sleep.***
  - This task will be switched to **WAITING** state
  - After 10 * *msec_10* ms, this task will be switched to **READY** state

API Description (2/4)

- void task_exit();
  - Print a message in the format: ***Task {task_name} has terminated.***
  - This task will be switched to **TERMINATED** state

API Description (3/4)

- void get_resource(int *count*, int **resource_list*);
  - Check if all resources in the list are available
    - If yes
      - Get the resource(s)
      - Print a message for each resource in the list in the format:

        Task {task_name} gets resource {resource_id}.
    - If no
      - This task will be switched to **WAITING** state
      - Print a message in the format: ***Task {task_name} is waiting resource.***
      - When all resources in the list are available, this task will be switched to **READY** state
      - Check again when CPU is dispatched to this task

API Description (4/4)

- void release_resource(int *count*, int **resource_list*);
  - Release the resource(s)
  - Print a message for each resource in the list in the format:

    Task {task_name} releases resource {resource_id}.

Shell Command (1/4)

- **add**
  - Command format: ***add {task_name} {function_name} {priority}***
  - Create a task named *task_name* that runs a function named *function_name*
  - *priority* is ignored if the scheduling algorithm is not priority-based preemptive scheduling
  - This task should be set as **READY** state
  - Print a message in the format: ***Task {task_name} is ready.***

Shell Command (2/4)

- **del**
  - Command format: ***del {task_name}***
  - The task named *task_name* should be switched to **TERMINATED** state
  - Print a message in the format: ***Task {task_name} is killed.***

Shell Command (3/4)

- **ps**
  - Command format: ***ps***
  - Show the information of all tasks, including TID, task name, task state, running time, waiting time, turnaround time, resources occupied and priority (if any)
  - Example

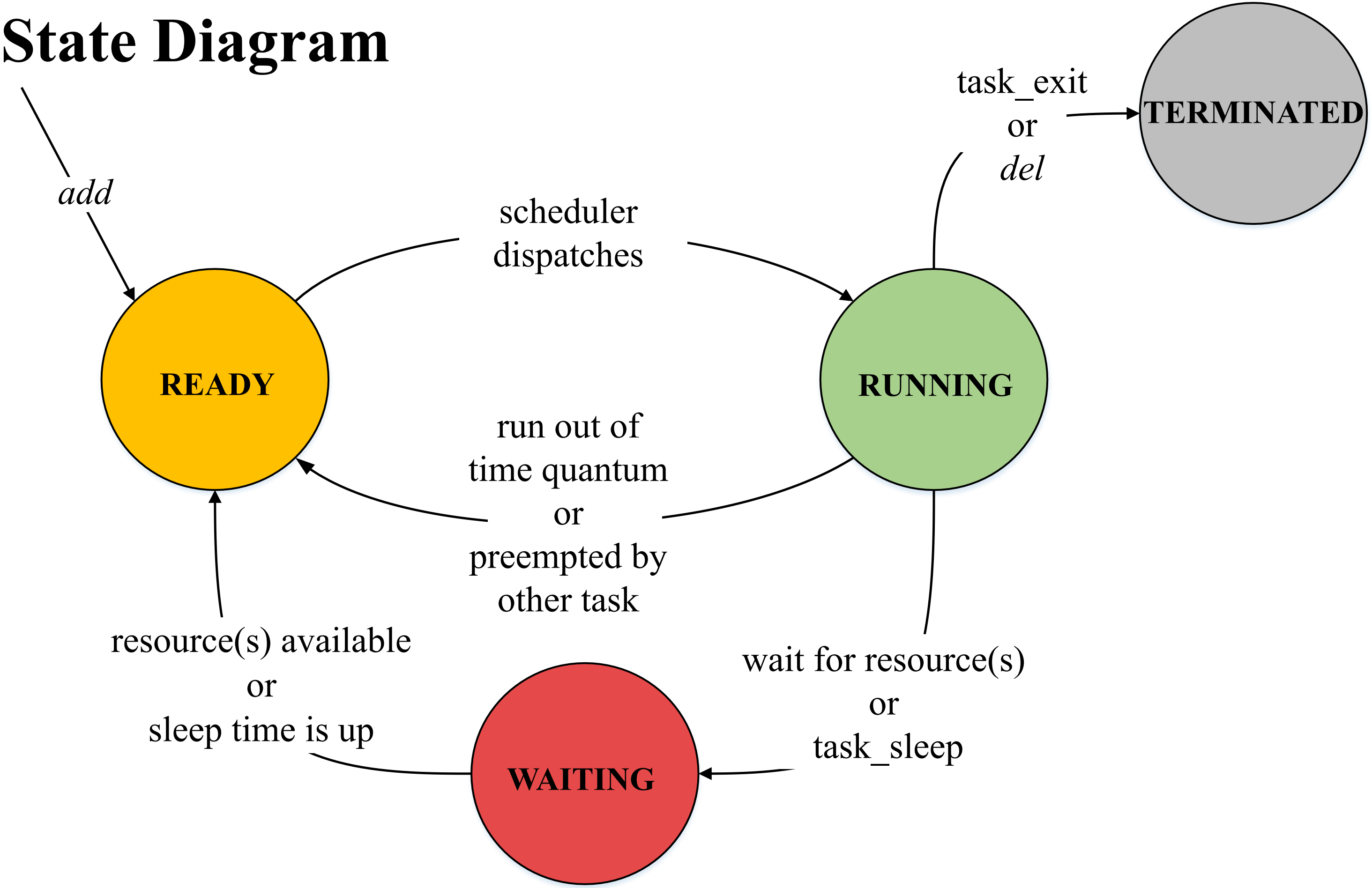
TID	name	state	running	waiting	turnaround	resources	priority
1	T1	TERMINATED	1	0	1	none	1
2	T2	WAITING	1	1	none	1 3 7	2
3	T3	READY	0	2	none	none	4
4	T4	RUNNING	0	2	none	none	3

1) The TID of each task is unique, and TID starts from 1.
2) There is no turnaround time for unterminated tasks.
3) Time unit: 10ms

Shell Command (4/4)

- **start**
  - Command format: ***start***
  - Start or resume the simulation
  - Print a message in the format: ***Start simulation.***

Task State Diagram



Grading

- For each part of the requirements, TA will do some tests to check whether the simulator is working properly.
- You will need to explain to TA how you implemented your simulator according to these requirements.
- TA will ask some questions about the simulation results for each test case, so you need to understand exactly what happened during the simulation.
- If you cannot explain smoothly, you won’t get points.

Precautions

- All purple texts are prescribed formats and must be followed.
- You should implement hw3 with **C** language.
- You will get a template from hw3 github classroom (*see Appendix for details*).
- You can modify makefile as you want, but make sure your makefile can compile your codes and generate the executable file correctly.
- The executable file should be named ***scheduler_simulator***.
- Make sure your codes can be compiled and run in the DEMO environment introduced in the hw0 slide.

GitHub Classroom

- GitHub classroom

  Click [Here]() to start your assignment.

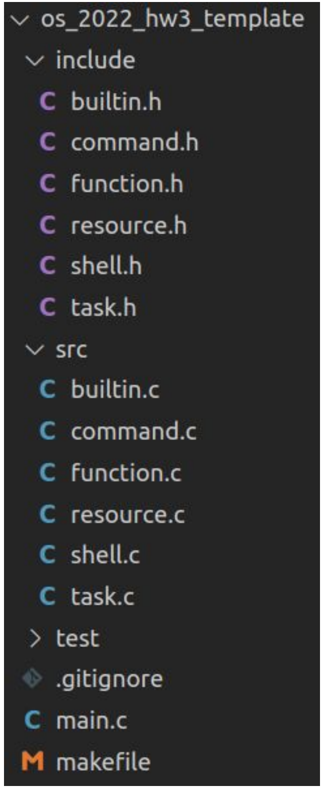
- Due date

  2022/12/16 (Fri.) 23:59 (以 github 上傳時間為準)

Reference

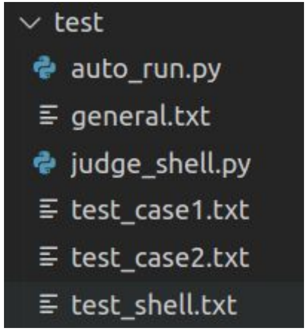
- ucontext
  - The Open Group Library
  - Linux manual page
    - getcontext()
    - setcontext()
    - makecontext()
    - swapcontext()
- signal
  - Gitbook
  - Linux manual page
- timer
  - Linux manual page
  - IBM® IBM Knowledge Center

Appendix - file structure of the template



- *shell.h/.c*, *command.h/.c*, *builtin.h/.c* are shell-related files, and *builtin.c* contains the four commands need to be implemented in this assignment
- *resource.h/.c* contains resource-related APIs that need to be implemented
- *task.h/.c* contains task state-related APIs that need to be implemented
- *function.h/.c* contains functions run by tasks. These 2 files **cannot be modified**
- *test* folder contains all test cases, an auto-judge script for the shell and an auto-run script for the simulation

Appendix - how to use auto-judge and auto-run



- If you need to check whether the shell is broken due to your modification
  - ➢ *python3 test/judge_shell.py*
- Auto-run the scheduler simulator
  - ➢ *python3 test/auto_run.py {algorithm} {test_case}*
  - ➢ *algorithm* can be *FCFS*, *RR*, *PP* or *all*, where *all* will perform three rounds of simulation for all scheduling algorithms
  - ➢ **test_case** can be *test/general.txt*, *test/test_case1.txt* or *test/test_case2.txt*
    - ■ Test case files contain a list of commands seperated by newlines. You can also write your own test cases, just follow the format
    - ■ *test/general.txt* tests all requirements except pausing
    - ■ *test/test_case1.txt* and *test/test_case2.txt* are test cases that need to observe the results
  - ➢ The auto-run script will generate a file to store the simulation result, and the file name is *{test_case's file name}_{algorithm}.txt*, for example, *general_FCFS.txt*
  - ➢ Auto-run script does not support pausing with ***Ctrl + z***