OS 2019

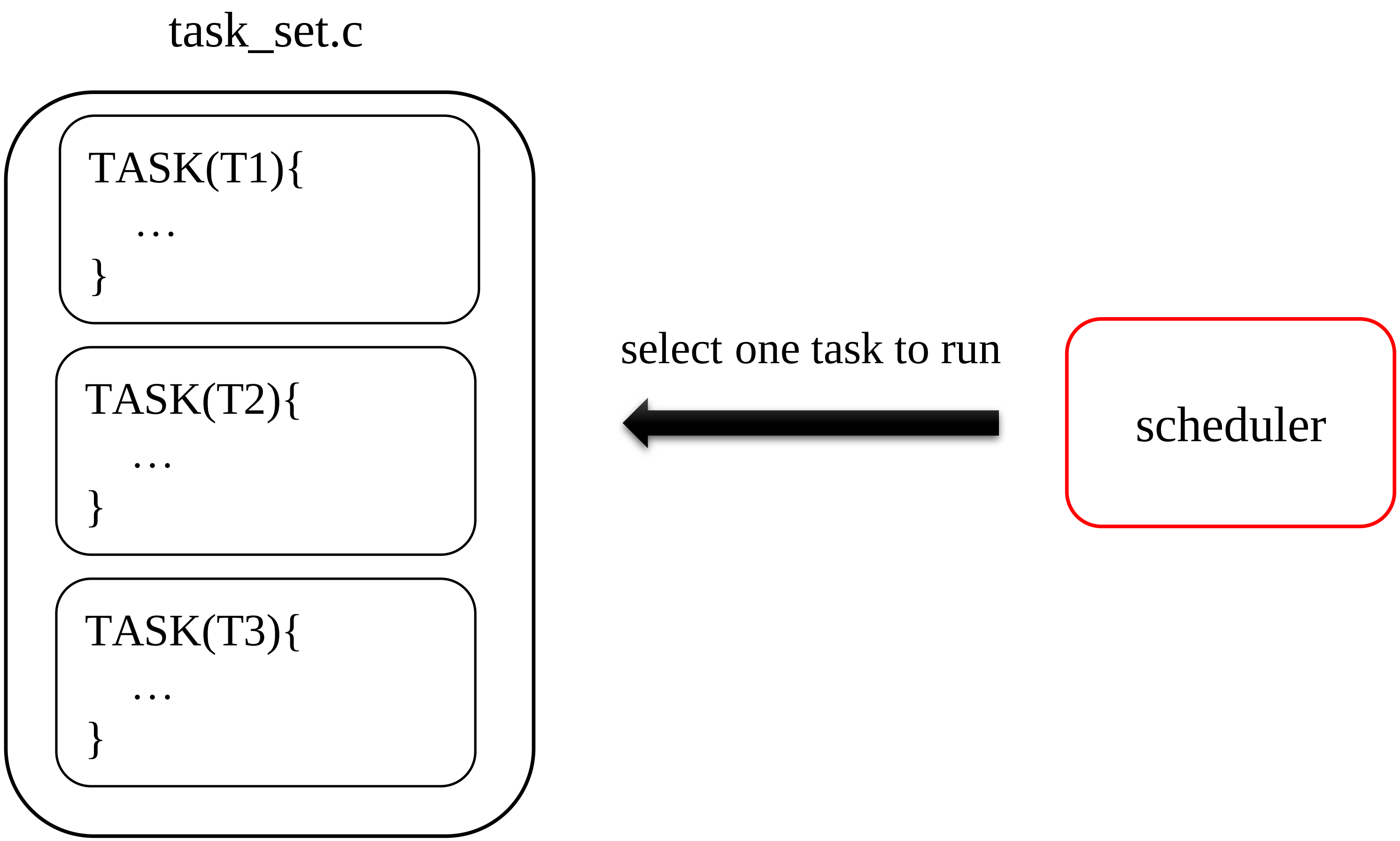
Homework3: scheduling simulation

(Due date 12/12 23:59:59)

Objectives

- Simulate task scheduling
- Understand priority ceiling protocol
- Understand how to implement context switch

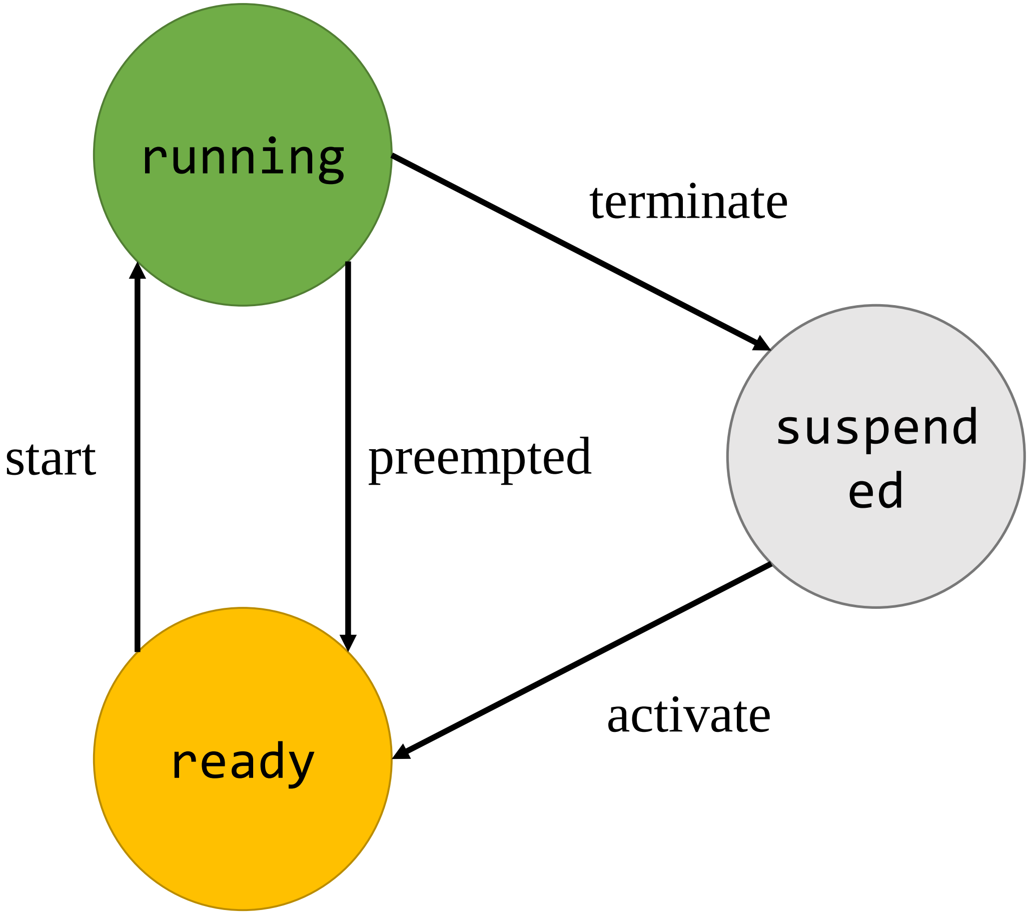
Overview


task_set.c
TASK(T1){
…
}
TASK(T2){
…
}
TASK(T3){
…
}
select one task to run
scheduler

Tasks

- The state of each task is shown in *slide 5*.
- A task is a function in '*task_set.[ch]*'.
- All the task functions are provided by TA and can not be modified.
- A task may be activated by *activate_task()* API calls or automatically be activated before the scheduler starts to choose the first task to run (auto start task, described in *slide 15*).
- Each task is assigned a priority statically in configuration files (*config.[ch]*).
  - A larger priority value indicates a higher priority, i.e., the value 0 is the lowest priority, and the value 7 is the highest priority.

Task State



- suspended: The task is not active and can be activated.
- ready: The task is preempted or activated. The task should be put into the ready queue, waiting for allocation of the processor.
- running: The processor is assigned to the task, so that its instructions can be executed.

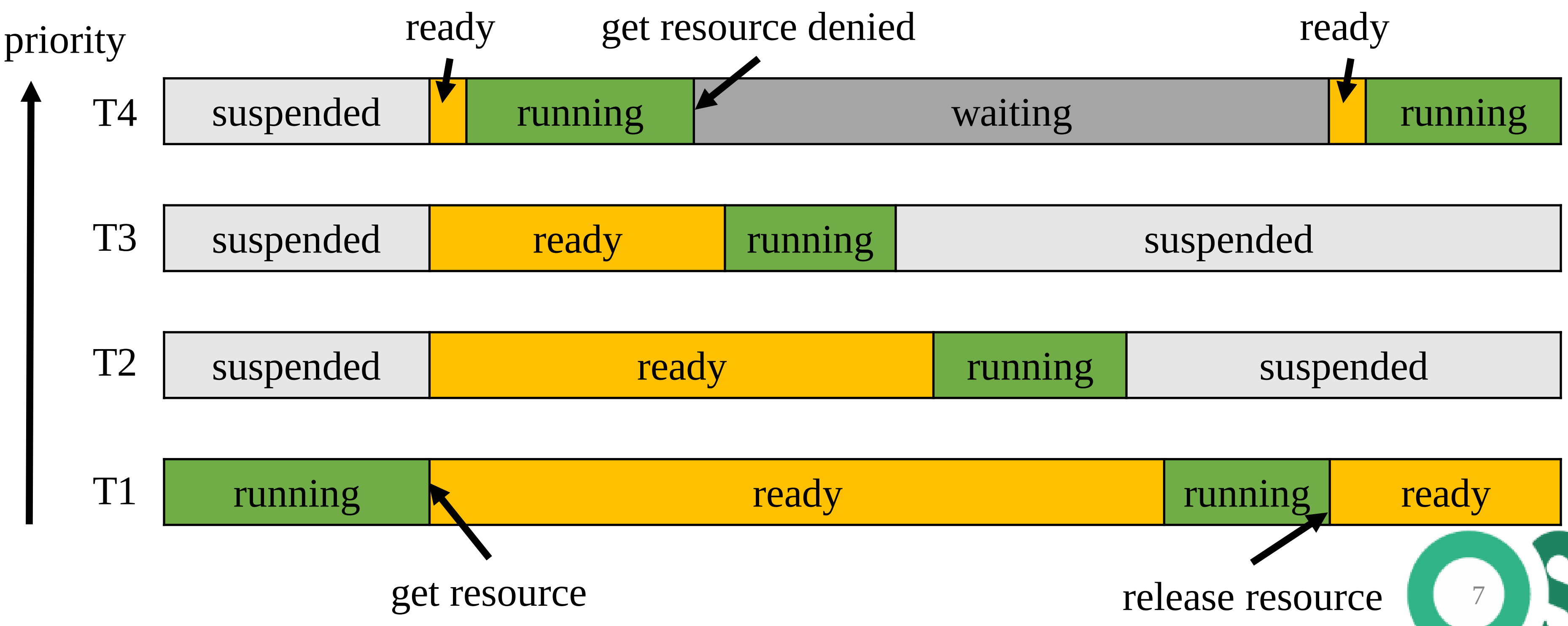
Resource

- In real world
  - resources might be I/O devices or shared data
- In this homework
  - resources are just abstract objects
  - resource related APIs (*get_resource*, *release_resource*) are used to protect a resource from race condition
  - tasks would get and release resources in LIFO order, like following example

```
get_resource(res_a);
get_resource(res_b);
…
release_resource(res_b);
release_resource(res_a);
```

Priority Inversion

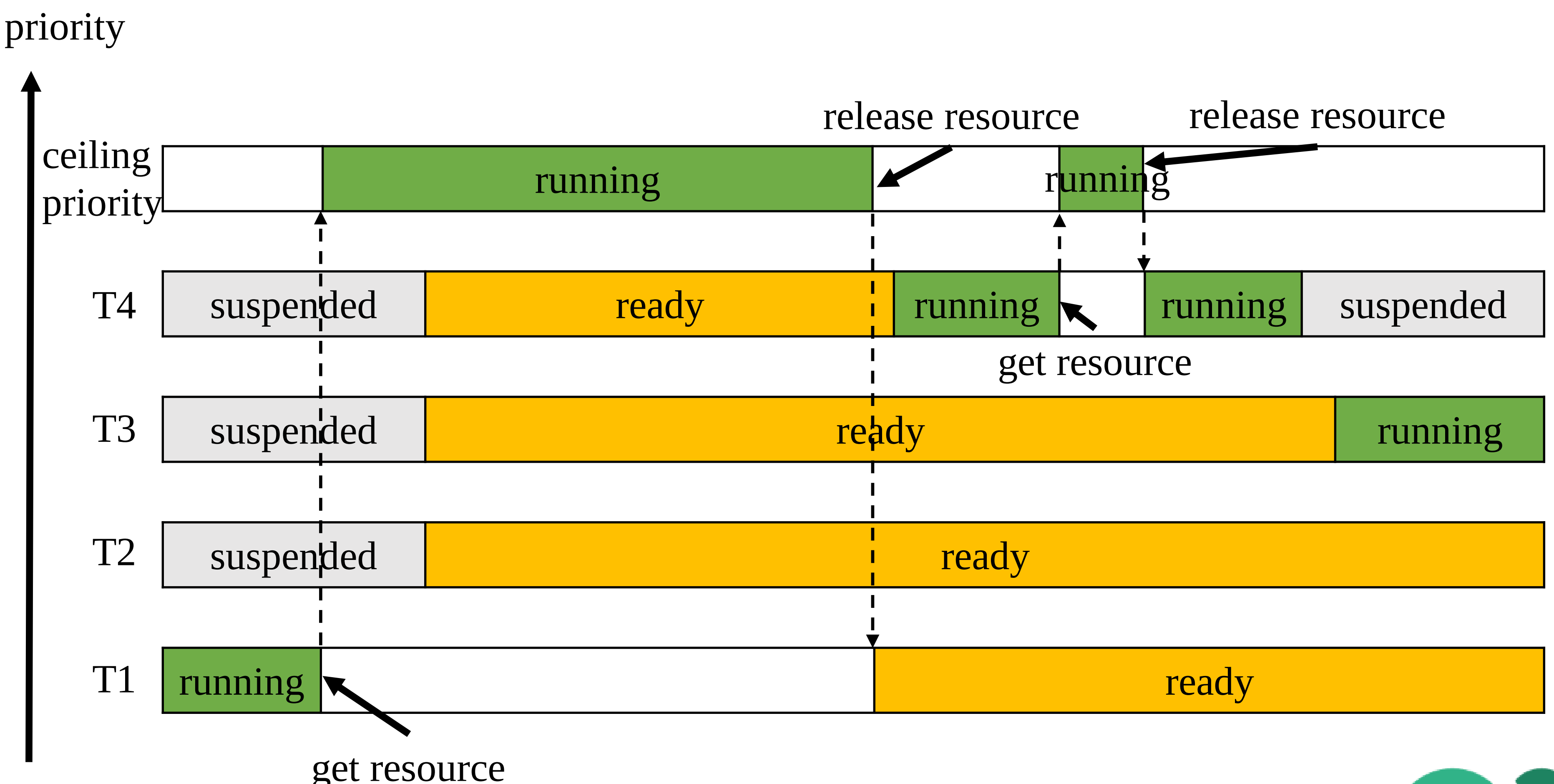
- A high priority task is indirectly preempted by a lower priority task effectively inverting the relative priorities of the two tasks.



Priority Ceiling Protocol

- To solve the problem of priority inversion, each resource is assigned a **ceiling priority**.
- The ceiling priority would be statically set to the highest priority of all tasks that would access the resource.
- If a task got a resource and its current priority is lower than the ceiling priority of the resource, the priority of the task would raise to the ceiling priority of the resource.
- If a task released a resource, the priority of the task would be reset to the priority before getting that resource.

Priority Ceiling Protocol



Requirements

- Write a user application (scheduling_simulator)
  - Use ucontext and related APIs to implement context switch.
  - Implement a scheduler with priority ceiling protocol (*described in slide 8~9*).
- Implement the APIs that can be used by tasks (*described in slide 16~19*)
  - task.[ch]
    - status_type activate_task(task_type id);
    - status_type terminate_task(void);
  - resource.[ch]
    - status_type get_resource(resource_type id);
    - status_type release_resource(resource_type id);

Requirements

- Scheduling Policy
  - Preemptive scheduling: The task in *running* state would be preempted and be transferred into *ready* state, as soon as a higher-priority task has got ready.
  - The context of the preempted task should be saved so that the task could be continued from where it was preempted.
  - Tasks with equal priority are started depending on their order of activation.
- Ready Queue
  - Maintain a ready queue only for tasks in *ready* state.
  - There is no requirement of data structure for the ready queue.

Requirements

- Basic types, macros, and data structure of tasks are defined by TA (*typedefine.h*) and can not be modified. Students can define other types, macros, or data structure by their needs.
- In addition to files in directory '*testcase1*' and '*testcase2*', there are unreleased test cases may be used to verify correctness of students' code.
- Make sure your program can run correctly with following commands.
    - $ make test1
    - $ ./scheduling_simulator
    - $ make test2
    - $ ./scheduling_simulator

Configuration Files

- Configuration files (*config.[ch])* are provided by TA and can not be modified.
- Entry point, id, and priority of each task are statically defined in configuration files by TA.

```
const task_const_type task_const[TASKS_COUNT] = {
    /* idle_task */
    {
        TASK_idle_task,      /* task entry point */
        idle_task,           /* task id */
        0,                   /* task priority */
    },
    /* TASK T1 */
    {
        TASK_T1,             /* task entry point */
        T1,                  /* task id */
        1,                   /* task priority */
    },
```

Configuration Files

- Ceiling priority and id of each resource are statically defined in configuration files by TA.

```
/* Brief Resources ID */
const resource_type resources_id[RESOURCES_COUNT] = {
    RESOURCE_1      /* may be accessed by T1, T4 */
};

/* Brief Resources Priorities */
const task_priority_type resources_priority[RESOURCES_COUNT] = {
    4   /* ceiling priority of RESOURCE_1 */
};
```

Configuration Files

- Auto start tasks: Tasks listed in *auto_start_task_list* in ‘*config.c*’ should be activated automatically before the scheduler starts to choose the first task to execute.

```
const task_type auto_start_tasks_list[AUTO_START_TASKS_COUNT] = {
    idle_task,
    T1
};
```

- Configuration files will be changed for different test cases.
- Do not write your code in configuration files.

API Description

- status_type activate_task(task_type id);
  - The task with <id> is transferred from *suspended* state into *ready* state.
  - Tasks transferred from *suspended* state into *ready* state should start from entry point of the task.
  - If the task with <id> is not in *suspended* state, the activation is ignored.
  - Reschedule if needed.
  - Return *STATUS_OK* if no error.
  - Return *STATUS_ERROR* if the task with <id> is not in *suspended* state.

API Description

- status_type terminate_task(void);
  - The calling task is transferred from *running* state into *suspended* state.
  - If the calling task still occupies any resource, the termination is ignored.
  - Reschedule if needed.
  - Return *STATUS_OK* if no error.
  - Return *STATUS_ERROR* if the calling task still occupies any resource.

API Description

- status_type get_resource(resource_type id);
  - The calling task occupies the resource with <id>.
  - If a task got a resource and its current priority is lower than the ceiling priority of the resource, the priority of the task would raise to the ceiling priority of the resource.
  - If the resource with <id> is already occupied, the above operations are ignored.
  - Return *STATUS_OK* if no error.
  - Return *STATUS_ERROR* if the resource with <id> is already occupied.

API Description

- status_type release_resource(resource_type id);
  - The calling task releases the resource with <id>.
  - If a task released a resource, the priority of the task would be reset to the priority before getting that resource.
  - If the calling task attempts to release a resource that is not occupied by the calling task, the above operations are ignored.
  - Reschedule if needed.
  - Return *STATUS_OK* if no error.
  - Return *STATUS_ERROR* if the calling task attempt to release a resource that is not occupied by the calling task.

References

- ucontext
  - The Open Group Library
  - IBM® IBM Knowledge Center
    - getcontext()
    - setcontext()
    - makecontext()
    - swapcontext()