Mentoring Operating System (MentOS)

Exercise 1 - Deadlock prevention

Created by
Mirco De Marchi
Enrico Fraccaroli
enrico.fraccaroli@gmail.com

Table of Contents

Preparation

Preparation

Switch to Exercise branch

1. Save your work!!!
   - *e.g.*, **mentos/src/process/scheduler_algorithm.c**
2. `git reset -hard`
3. `git pull`
4. `git checkout -track origin/feature/Feature-DeadlockExercise`

Deadlock: theoretical aspects

Deadlock: theoretical aspects

Definitions

Deadlock

Deadlock

State of a concurrent system with shared resources between tasks, in which at least a single task is waiting for a resource acquisition that can be released by another task without resolution.

If you want to avoid deadlock you have to **prevent** that at least one of the following conditions hold:

- Mutual exclusion;
- Hold and wait;
- No preemption;
- Circular waiting;

Safe state

Safe state

The system state is safe if you can find a sequence of resource allocations that satisfy the tasks resource requirements, otherwise is unsafe.

!!! You need to know tasks resource requirements. Not so simple to do.

Methodologies that use the concept of unsafe state:

- Dynamic Prevention: check each allocation request if leads to an unsafe state;
- Detection: only detect when happens;

For example: **Banker's Algorithm**.

Banker's Algorithm and alternatives

Banker's Algorithm main idea: I will satisfy your request only if I am sure to satisfy the requests that others can ask.
Not so generous because he considers the upper bound of the resource requests $\Rightarrow$ Drawback: tasks starvation.

Alternative methodologies:

- Static prevention: design constraints to falsify deadlock conditions;
- Detect and Recovery: rollback or, at worst, system restart;
- Not handled: programmers have to write good code (e.g. Linux);

Deadlock: theoretical aspects

Banker's Algorithm

Banker's Algorithm: notations

- **n**: Current number of tasks in the system.
- **m**: Current number of resource types in the system.
- **req_task**: Process that perform the resource request.
- **req_vec[m]**: Resource instances requested by *req_task*.
- **available[m]**: Number of resource instances available for each resource type.
- **max[n][m]**: Maximum number of resources instances that each task **may** require;
- **alloc[n][m]**: Current resource instances allocation for each task.
- **need[n][m]**: Current resources instances need for each task.
  `need[i][j] = max[i][j] - alloc[i][j]`.

Banker's Algorithm: resource request

```
Require: req_task, req_vec[m], available[m], max[n][m], alloc[n][n],
    need[n][m]
if req_vec > need[req_task] then
    error()
end if
if req_vec > available then
    wait()
end if
available = available - req_vec
alloc[req_task] = alloc[req_task] + req_vec
need[req_task] = need[req_task] - req_vec
if !safe_state() then
    available = available + req_vec
    alloc[req_task] = alloc[req_task] - req_vec
    need[req_task] = need[req_task] + req_vec
end if
```

Algorithm 1: Resource request performed by a requesting task

Banker's Algorithm: check state safe

```
Require: available[m], max[n][m], alloc[n][m], need[n][m]
work[m] = available; finish[n] = (0,...,0)
while finish[] != (1,...,1) do
    for i=0 to n do
        if !finish[i] and work >= need[i] then
            break
        end if
    end for
    if i == N then
        return false // UNSAFE
    else
        work = work + alloc[i]
        finish[i] = 1
    end if
end while
return true // SAFE
```

Algorithm 2: Check if the allocation leads an unsafe state

MentOS: Deadlock Prevention

MentOS: Deadlock Prevention

How to

How to do in MentOS

Deadlock prevention is not easy to perform, because we need to know in advance information about tasks execution. In particular, we need to fill the **available**, **max**, **alloc**, **need** matrices.

What to do to get the matrices?

- available: need for a list of created resources;
- max: need to know for each task which are the resources that they are interested for.
- alloc: need to know which process a resource has been assigned.
- need: need for a library to manage arrays (also for the algorithm itself).

How to do in MentOS

Assumptions made:

- Each semaphore created belongs to a existing resource.
- Each resource can be used by the process that created it and by the child processes.

What has been implemented:

- Definition of **resource_t** with task reference that own it.
- Creation of global created resources list.
- List of resources that tasks are interested for, in `task_struct`.
- Copy of this list in child **task_struct** during syscall `fork`.
- Resource creation during semaphore creation in kernel-side syscall.
- Implementation of `arr_math` library.

Resource definition and task_struct improvements

```
typedef struct resource {
    /// Resource index. The resources indexes has to be continuous: 0, 1, ... M.
    size_t rid;
    /// List head for resources list.
    list_head resources_list;
    /// Number of instances of this resource. For now, always 1.
    size_t n_instances;
    /// If the resource has been assigned, it points to the task assigned,
    /// otherwise NULL.
    task_struct *assigned_task;
    /// Number of instances assigned to assigned task.
    size_t assigned_instances;
} resource_t;
```

```
typedef struct task_struct {
    ...
  /// Array of resource pointers that task need for.
  struct resource *resources[TASK_RESOURCE_MAX_AMOUNT];
    ...
} task_struct;
```

MentOS: Deadlock Prevention

Library `arr_math`

Library `arr_math` 1

The implementation of *Banker's Algorithm* needs to manage matrices and arrays. You can find `arr_math` definition in **mentos/inc/experimental/math/arr_math.h**.
The following is a summary of the definitions:

- `uint32_t *all(uint32_t *dst, uint32_t value, size_t length);`
  Initialize the destination array with a value.
- `uint32_t *arr_sub(uint32_t *left, const uint32_t *right, size_t length);`
  Array element-wise subtraction, saved in left pointer.
- `uint32_t *arr_add(uint32_t *left, const uint32_t *right, size_t length);`
  Array element-wise addition, saved in left pointer.

Library arr_math 2

- `bool_t arr_g_any(const uint32_t *left, const uint32_t *right, size_t length);`
  Check that at least one array element is greater than the respective other. E.g. $[1, 1, 6] g\_any [1, 2, 3] = true$
- `bool_t arr_g(const uint32_t *left, const uint32_t *right, size_t length);`
  Check that all array elements are greater than the respective other. E.g. $[2, 3, 4] g\_all [1, 2, 3] = true$
- `arr_ge_any`: greater or equals at least one.
- `arr_ge`: greater or equals all elements.
- `arr_l_any`, `arr_le_any`: less (and less or equals) at least one.
- `arr_l`, `arr_le`: less and less or equals all elements.
- `arr_e`, `arr_ne`: equals and not equals all elements.

MentOS: Deadlock Prevention

Exercise

Requirement: semaphore syscalls and scheduler algorithm.

1. cd <mentos-main-dir>
2. git checkout -track origin/feature/Feature-DeadlockExercise
3. git pull
4. Prepare MentOS with semaphore syscalls implementation and at least one scheduler algorithm.
   - **mentos/src/process/scheduler_algorithm.c**
   - **src/experimental/smart_sem_user.c**

Deadlock Prevention in MentOS

Implement *Banker's Algorithm* in MentOS starting from the template file given in **mentos/src/experimental/deadlock_prevention.c**.

Check for the results:

- *Build project*
  1. `cd <mentos-main-dir>`
  2. `mkdir build && cd build`
  3. `cmake -DENABLE_DEADLOCK_PREVENTION=ON ..`
  4. Build: `make`
  5. Run: `make qemu`
- Check in debug console for deadlock prevention deterministic simulation.
- Try the shell command line `deadlock [-i <iterations>]` to test deadlock prevention in real tasks.

See you in laboratory for more info about.