Ecole Centrale de Nantes

EMARO1-CORO1 Master

R E T S Y

(**RE**al **T**ime **SY**stems)

Lab #3: SHARED OBJECT ACCESS PROTECTION

Report of:

Hryshaienko Olena

(Group C)

Tartari Michele

(Group C)

N A N T E S

2 0 1 7

INTRODUCTION

To show resources usage, we will use a bad program that allows to corrupt a shared global variable which is not protected against concurrent writes. This has been presented in the course. This lab will show different ways to prevent this wrong behavior by using resources (standard and internal) or other solutions (preemption and priority).

1) Application **REQUIREMENTS**

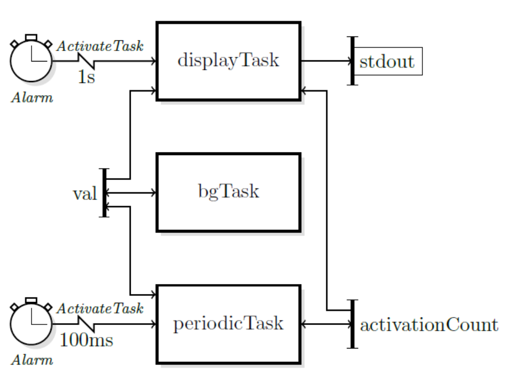


Figure 1: Application Diagram

The application has 2 global variables declared with the volatile keyword:

- *val*;
- *activationCount*;

and 3 tasks:

- ***bgTask*** - background task, activated at start (AUTOSTART = TRUE), never ends:
  - it increments/decrements the global variable val in an ∞ loop;
  - has a priority equal to 1.
- ***periodicTask*** - a periodic task, that runs every 100ms1:
  - it increments the global variable activationCount;
  - IF activationCount is odd, val is incremented;
  - IF NOT it is decremented.
- ***displayTask*** - a periodic task that runs every second and prints on the LCD val and activationCount.

Snippet of .cpp file

```
volatile int val=0;
volatile int activationCount=0;
TASK(displayTask)
{
    lcd.print("actCount= ");
    lcd.println(activationCount);
    lcd.print("val= ");
    lcd.println(val);
    TerminateTask();
}

TASK(periodicTask)
{
++activationCount;
if(activationCount%2)
    val++;
    else
        val--;
    TerminateTask();
}

TASK(bgTask)
{
    while(1){
        val++;
        val--;    }
    TerminateTask();
}
```

Snippet of .oil file

```
TASK bgTask {
    PRIORITY = 1;
AUTOSTART = TRUE {
  APPMODE = stdMode;
};
    ACTIVATION = 1;
    SCHEDULE = FULL;
};

TASK periodicTask {
    PRIORITY = 2;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
};

TASK displayTask {
    PRIORITY = 3;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
};

COUNTER SystemCounter{
  TICKSPERBASE=100;
  MAXALLOWEDVALUE=6565874;
  MINCYCLE=1;
};
ALARM alarmPeriodic{
    COUNTER = SystemCounter;
    ACTION = ACTIVATETASK{
      TASK = periodicTask;    };
    AUTOSTART = TRUE  {
       APPMODE = stdMode;
       ALARMTIME = 1;
       CYCLETIME = 1;
    };};

ALARM alarmDisplay{
    COUNTER = SystemCounter;
    ACTION = ACTIVATETASK{
      TASK = displayTask;
    };
    AUTOSTART = TRUE  {
       APPMODE = stdMode;
       ALARMTIME = 10;
       CYCLETIME = 10;
    }; };
```

Question 1.1 Give the values that should be displayed for val on the standard output.

Due to the behavior of the tasks we, therefore, expected *val* to be always = 0 and *activationCount* to be increased at each iteration of *periodicTask*. On the screen, we should see val always equal to 0 and *activationCount* increasing of 10 each time.

Question 1.2 Does the behavior correspond to what you expect? Why?

The application behaved differently from what expected: *val*'s value kept changing on the screen when printed. This was caused by the lack of protection of the shared variables that allowed data corruption.

2) Global variable **PROTECTION**

Question 2.1 To protect the access to the global variable *val* - update the OIL file and the C program. Use a resource[1] to do it.

Since *activationCount* is always used in task together with *val* we can define a single resource *S1* to protect both variables.

Snippet of .cpp file

```
DeclareResource(S1);

TASK(bgTask)
{
    while(1){
        GetResource(S1);
        val++;
```

Snippet of .oil file

```
RESOURCE S1 {
  RESOURCEPROPERTY = STANDARD;
};

TASK bgTask {
    PRIORITY = 1;
AUTOSTART = TRUE {
```

[1] The resource priority is automatically computed by goil according to the priorities of the tasks which use it.

```
        val--;
        ReleaseResource(S1);
    }
    TerminateTask();
}

TASK(periodicTask)
{
    GetResource(S1);
    ++activationCount;
    if(activationCount%2)
        val++;
    else
        val--;
    ReleaseResource(S1);
    TerminateTask();
}

TASK(displayTask)
{
    GetResource(S1);
    lcd.print("actCount= ");
    lcd.println(activationCount);
    lcd.print("val= ");
    lcd.println(val);
    ReleaseResource(S1);
    TerminateTask();
}
```

```
  APPMODE = stdMode;
};
    ACTIVATION = 1;
    SCHEDULE = FULL;
    RESOURCE = S1;
};

TASK periodicTask {
    PRIORITY = 2;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    RESOURCE = S1;
};

TASK displayTask {
    PRIORITY = 5;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    RESOURCE = S1;
};
```

Question 2.2 What priority will be given to the resource?

The resource priority was supposed to be put ≥ to the value of the highest priority task (5 in our case) accessing the variable val, since we used a Priority Ceiling Protocol (PCP) to protect it.

Question 2.3 Find the priority computed by goil and the identifier for each task. Is it the same as defined in the OIL file? If not, is it a problem?

The OIL compiler (goil) generates many files in the directory bearing the same name as the oil file (less the .oil sufix). Among them 3 are interesting: tpl_app_define.h, tpl_app_config.h, and tpl_app_config.c. The latter contains the tasks' descriptors as long as all other data structures. If we check the content of this files we can notice the actual priority given to each task and variable.

Snippet of tpl_app_config.c

```
 * Resource descriptor of resource S1
 *
 * Tasks which use this resource :
 * bgTask
 * displayTask
 * periodicTask
 */
VAR(tpl_resource, OS_VAR) S1_rez_desc = {
/* ceiling priority of the resource */   4,
```

Snippet of .oil file

```
TASK bgTask {
    PRIORITY = 1;
AUTOSTART= TRUE {
 APPMODE = stdMode;
};
    ACTIVATION = 1;
    SCHEDULE = FULL;
    RESOURCE = S1;
};
```

```
[…]

/*
 * Static descriptor of task bgTask
 */
[…]
/* task base priority       */  1,

[…]

/*
 * Static descriptor of task periodicTask
 */
[…]
/* task base priority       */  2,

[…]

/*
 * Static descriptor of task displayTask
 */
[…]
/* task base priority       */  3,
```

```
TASK periodicTask {
    PRIORITY = 2;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    RESOURCE = S1;
};

TASK displayTask {
    PRIORITY = 5;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    RESOURCE = S1;
};
```

As visible the priority is not equal to the one defined in the .oil, though the task order by priority is the same.

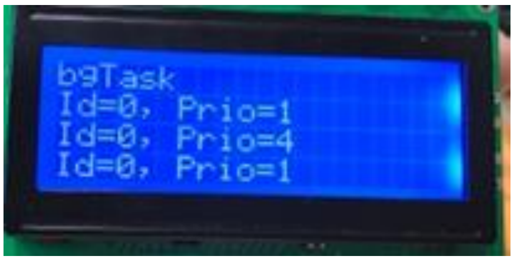
Question 2.4 What is the priority of the resource? Is it consistent?

PCP rule requires the task priority is raised to the resource priority when the resource is taken. We will show this behavior by displaying the priority of the currently running task. Since OSEK does not have a function to do that, we use the given *displayIdAndCurrentPriority ()* function. The code can be modified as follow to print on screen the priority of each task:

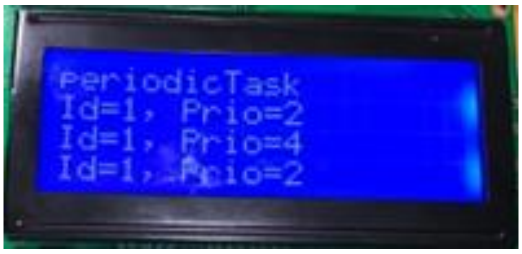
Snippet of .cpp file

```
TASK(bgTask)
{
    lcd.println("bgtask");
    displayIdAndCurrentPriority();
    while(1){
        GetResource(S1);

displayIdAndCurrentPriority();
        val++;
        val--;
        ReleaseResource(S1);

displayIdAndCurrentPriority();
    }
    TerminateTask();
}

TASK(periodicTask)
```

Outputs

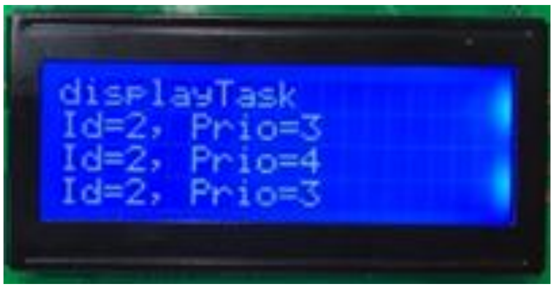


```
bgTask
Id=0, Prio=1
Id=0, Prio=4
Id=0, Prio=1
```



```
periodicTask
Id=1, Prio=2
Id=1, Prio=4
Id=1, Prio=2
```

```
{
    lcd.println("periodicTask");
    displayIdAndCurrentPriority();
    GetResource(S1);
    ++activationCount;
        if(activationCount%2)
            val++;
        else
            val--;
    displayIdAndCurrentPriority();
    ReleaseResource(S1);
    displayIdAndCurrentPriority();
    TerminateTask();
}

TASK(displayTask)
{
    lcd.println("displayTask");
    displayIdAndCurrentPriority();
    GetResource(S1);
    displayIdAndCurrentPriority();
    ReleaseResource(S1);
    displayIdAndCurrentPriority();
    TerminateTask(); }
```



```
displayTask
Id=2, Prio=3
Id=2, Prio=4
Id=2, Prio=3
```

As visible when *GetResource(S1)* is called the task priority will rise to 4 and once *ReleaseResource(S1)* is called it will return to its original value.

Question 2.5 Is the behavior ok? Explain.

Yes, such behavior desired because it will allow the task not to be preempted during critical parts of the code.

3) Protection with an **INTERNAL RESOURCE**

An internal resource is automatically taken when the task gets the CPU. Replace the standard resource by an internal resource in the OIL file.

Snippet of .cpp file

```
TASK(bgTask)
{
    while(1){
        val++;
        val--;
    }
    TerminateTask();
}

TASK(periodicTask)
{
    ++activationCount;
    if(activationCount%2)
        val++;
    else
        val--;
    TerminateTask(); }
```

Snippet of .oil file

```
TASK bgTask {
    PRIORITY = 1;
AUTOSTART = TRUE {
  APPMODE = stdMode;
};
    ACTIVATION = 1;
    SCHEDULE = FULL;
    RESOURCE = S1;
};

TASK periodicTask {
    PRIORITY = 2;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    RESOURCE = S1;
};
```

```
TASK(displayTask)
{
    lcd.print("actCount= ");
    lcd.println(activationCount);
    lcd.print("val= ");
    lcd.println(val);
    TerminateTask();
}
```

```
TASK displayTask {
    PRIORITY = 3;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    RESOURCE = S1; };

RESOURCE S1 {
  RESOURCEPROPERTY = INTERNAL; };
```

Question 3.1 Remove the GetResource and ReleaseResource in the C file. What happens? Why?

Nothing will be displayed: *bgTask* will not be preempted by the other tasks until it terminates but the infinite while loop will never make it possible. Thus *periodicTask* and *displayTask* will never run nor display anything.

Question 3.2 Modify the task bgTask: instead of infinite loop, use a ChainTask to the bgTask. What happens? Why?

Snippet of .cpp file

```
TASK(bgTask)
{
    val++;
    val--;
    ChainTask(bgTask);}
```

After increasing and decreasing the *val*, *bgTask* will terminate. The application will then execute the next tasks according to priority order.

4) Protection using a **SINGLE PRIORITY LEVEL**

Question 4.1 Remove the resource and set the priorities so that no task can be preempted. Keep the version with the ChainTask instead of the infinite loop. Modify the OIL file. What happens? Why?

Snippet of .oil file

```
TASK periodicTask {
    PRIORITY = 3;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    RESOURCE = S1; };

TASK displayTask {
    PRIORITY = 3;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    RESOURCE = S1; };
```

```
TASK bgTask {
    PRIORITY = 3;
AUTOSTART = TRUE {
  APPMODE = stdMode;
};
    ACTIVATION = 1;
    SCHEDULE = FULL;
    RESOURCE = S1;
};
```

Giving to all tasks the same priority will prevent them from preempting each other, solving the problem of data corruption of shared variables.