LAB 4
PRIORITY INHERITANCE PROTOCOL/PRIORITY CEILING PROTOCOL

Introduction:

In real time systems, the problems due to priority inversion can be solved by using resource access protocols like Priority Inheritance Protocol (PIP) or Priority Ceiling Protocol (PCP). In the Xilkernel, these protocols are not implemented as part of the kernel. However, we have APIs that allows us to dynamically change the priority of the threads during run-time. These APIs can be used to implement the priority inheritance mechanism for threads.

Objectives:

- To develop code which perform Priority Inheritance Protocol
- To develop code which perform Priority Ceiling Protocol

Inputs:

- The dual core system that was developed in Lab 3.
- The operating systems for both the Microblaze cores will be the Xilkernel. Ensure that all settings required for Xilkernel are done for both the processors through the Board Support Package Settings. There are additional pointers to take note:
  - From "Board Support Package Settings", enable pthreads, timers and mutexes via the Xilkernel configuration page if you require these facilities.
  - Modify the linker script so that the code memory spaces of both processors reside in non-overlapping address ranges.
  - Ensure that main_prog is set as a priority 0 static thread to be run on the Xilkernel.
  - If you prefer to debug through the JTAG interface, ensure that stdin, stdout and xmdstub_peripheral under "Board Support Package Settings" are set to mdm_0 (if your Microblaze Debug Module has the identifier mdm_0), and under "Run Configurations", set STDIO Connection to use JTAG UART port.

Priority Inheritance Protocol:

- The source code to be used for Priority Inheritance Protocol is pip.c

- Create a new software application project using this C code as source. You may use microblaze_0 as the associated processor for this.

Source code Description:

- Three threads are created by the program, with priorities specified as 1, 2 and 3 respectively.

- Here, the threads are such that thread 1 has highest priority while thread 3 has lowest priority.

Also, thread 1 and 3 are allowed mutually exclusive access to the UART terminal while thread 2 can access the UART anytime. Such a scenario is created to demonstrate the priority inversion problem.

• When thread 3 locks the UART resource, thread 1 will be directly blocked by thread 3 during any access to the UART. However, thread 2 will continue to preempt thread 3 and execute as shown in the screenshot below. This is the problem of priority inversion where a lower priority thread 2 is allowed to execute over a higher priority thread 1 which remains blocked by thread 3.

• We can dynamically alter the priority of the threads during runtime so as implement a priority inheritance mechanism through which the thread 3 inherits the priority of the blocked thread 1 during the execution of its critical section. Thus, we can ensure that thread 2 will not be able to preempt thread 3 during its execution. Once the execution of thread 3 is complete, thread 1 is automatically allowed to run since it has higher priority over thread 2 and also the resource requested by thread 1 is now unlocked.

•Function calls which can dynamically alter thread priorities are shown below: (documentation is **xilkernel_v5_00_a.pdf, page 10**)

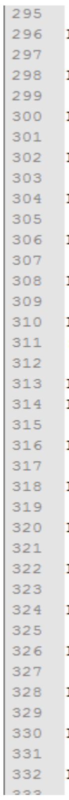
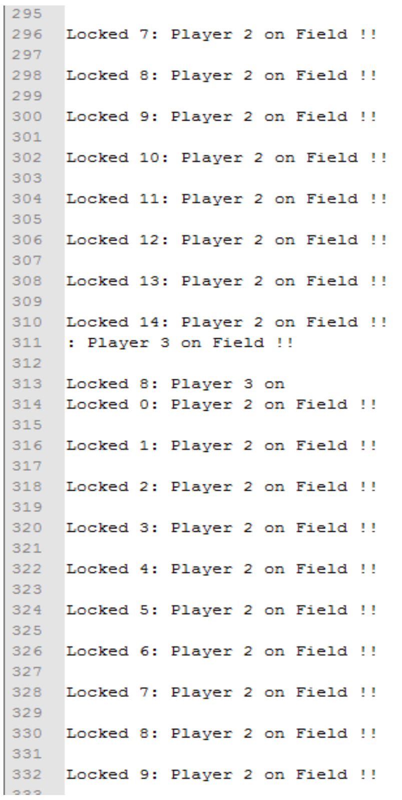
```
1. int pthread_getschedparam(pthread_t thread, int *policy,
                            struct sched_param *param)

2. int pthread_setschedparam(pthread_t thread, int policy,

                            const struct sched_param *param)
```

✍ **Lab Assessment**

1. You are required to implement the Priority Inheritance Protocol using function calls `pip_mutex_lock()/pip_mutex_unlock()`.
2. Accompany your code with an annotated diagram, either handwritten or computer-drawn, a timeline which shows the threads that are currently controlling the resource.

```

Locked 7: Player 2 on Field !!

Locked 8: Player 2 on Field !!

Locked 9: Player 2 on Field !!

Locked 10: Player 2 on Field !!

Locked 11: Player 2 on Field !!

Locked 12: Player 2 on Field !!

Locked 13: Player 2 on Field !!

Locked 14: Player 2 on Field !!
: Player 3 on Field !!

Locked 8: Player 3 on
Locked 0: Player 2 on Field !!

Locked 1: Player 2 on Field !!

Locked 2: Player 2 on Field !!

Locked 3: Player 2 on Field !!

Locked 4: Player 2 on Field !!

Locked 5: Player 2 on Field !!

Locked 6: Player 2 on Field !!

Locked 7: Player 2 on Field !!

Locked 8: Player 2 on Field !!

Locked 9: Player 2 on Field !!
```

Priority Ceiling Protocol:

• Priority Ceiling Protocol is used to solve the issue of deadlocks in scheduling and also to reduce the blocking time of a process to duration of utmost one critical section of a lower priority thread.

• Use the source code `pcp.c` to implement the priority ceiling protocol. You may use microblaze_0 for its implementation.

Source Code Description:

• Three threads are created by the program, in order of decreasing priority. There are three resources A, B and C. Thread 1 uses only the resource B, while Threads 2 and 3 uses the resource A and C. Each resource is a structure object with two elements: value and ceiling. The value is used in the calculations and the ceiling refers to the priority ceiling of the resource.

• There are three functions, `sub()` called by Thread 1, `add2` called by thread 2 and `add3` called by thread 3.

• When the code is executed, at some point there occurs a deadlock between thread 2 and 3. At this point, only the thread 1 will continue to execute. The deadlock occurs due to the fact that thread 3 locks the resource C first and then goes to lock the resource A, while thread 2 locks the resource A first and then goes to lock the resource C. At some point, the thread 2 starts executing just before thread 3 gets resource A locked. At this point, the dead lock occurs.

Lab Assessment

3. You are required to implement the Priority Ceiling Protocol, making use of the Priority Inheritance Protocol and additional ceiling parameters for the resources. Name the two functions as `pcp_mutex_lock()/pcp_mutex_unlock()`.
4. Accompany your code with an annotated diagram, either handwritten or computer-drawn, a timeline which shows the threads that are currently controlling the resource.