Adding
Priority Ceiling Protocol
To FreeRTOS

By: Mohammed Al-Sanabani

Executive Summary

In this project, Priority Ceiling Protocol will be implemented as an added module to FreeRTOS. It will have public interfaces that can
1- Create a new “PCP_Mutex”
2- Lock the “PCP_Mutex”
3- Unlock the “PCP_Mutex”.

We will highlight
1- The algorithm implementing PCP
2- Implementation details
3- Implemented examples
4- The demonstration / test application
5- Performance of PCP v.s regular FreeRTOS mutexes / binary semaphores (in terms of which helps “schedule” tasks better”)
6- Timing performance of PCP_Mutex measured and analysed (time of executing lock/unlock)

Details will be kept to minimum (no fluff) to save time for the reader, but will be enough to give them an excellent idea of what the project does and achieve.

The Algorithm will be “summarized” in this report, and the details will be available in the comments of the code.

Introduction

Priority Ceiling Protocol is a scheduling algorithm that guarantees that a task will only be blocked once (at most), and no priority inversion / deadlocks /chained blocking will happen to higher priority tasks.

It is extremely useful in real time systems where high priority tasks need to run “asap” and be blocked for minimum amount of time.

This project will implement PCP as a “gate” or “mutex” type. It can also be implemented into the scheduler itself, but since we dont want to impact the original FreeRTOS code, this will be avoided.

The report will include
1- Overview of the project
2- Design (Algorithm)
3- Examples of implementation (That have run)
4- Timing analysis
5- Conclusions.

Hardware Overview

STI : STM32F429I-DISCO evaluation board was used. The development environment used was keil (trial, with 32KB executable size limit). Windows was used as keil is only available in windows.

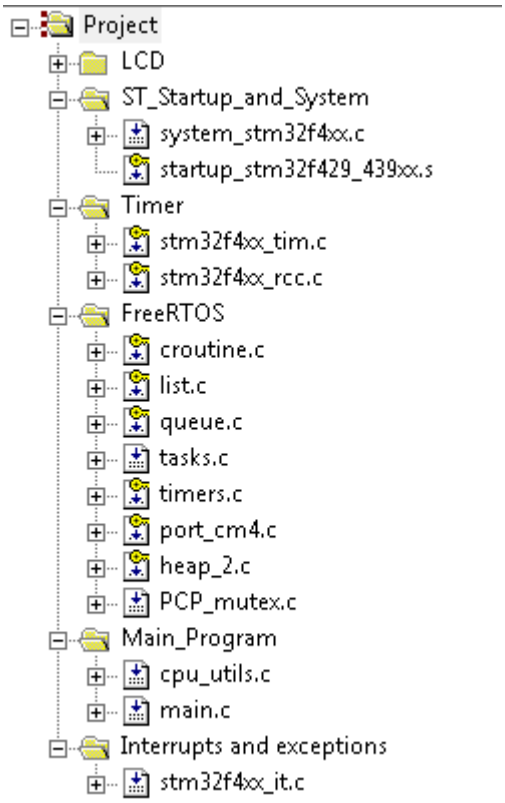
Key Features

- STM32F429ZIT6 microcontroller featuring 2 MB of Flash memory, 256 KB of RAM in an LQFP144 package
- On-board ST-LINK/V2 with selection mode switch to use the kit as a standalone
- ST-LINK/V2 (with SWD connector for programming and debugging)
- Board power supply: through the USB bus or from an external 3 V or 5 V supply voltage
- 2.4" QVGA TFT LCD
- SDRAM 64 Mbits

- L3GD20, ST MEMS motion sensor, 3-axis digital output gyroscope
- Six LEDs:
  - LD1 (red/green) for USB communication
  - LD2 (red) for 3.3 V power-on
  - Two user LEDs:LD3 (green), LD4 (red)
  - Two USB OTG LEDs:LD5 (green) VBUS and LD6 (red) OC (over-current)
- Two pushbuttons (user and reset)
- USB OTG with micro-AB connector
- Extension header for LQFP144 I/Os for a quick connection to the prototyping board and an easy probing

FreeRTOS Details:

FreeRTOS version V7.1.1 was used. The files (for the project) are the following:



Modifications to FreeRTOS were:

1- tasks.c: Added following APIs:
a- unsigned portBASE_TYE getTaskPriority(xTaskHandle* const taskHandle).
This is a public API to return the priority of a task

b- unsigned portBASE_TYPE getBaseTaskPriority(xTaskHandle * const taskHandle).
This is a public API that returns the Original priority of the Task: uxBasePCPPriority.

c- Added Original Task Priority (from PCP perspective): "uxBasePCPPriority" to tskTaskControlBlock structure. This is because we used the API: vTaskPrioritySet(task, newPriority) to set the priority of the tasks within PCP_mutex.c. This API (vTaskPrioritySet) sets both the uxBasePriority and the uxPriority (sets both !) and puts the task in the appropriate queue. so to remember the original priority we added this variable to the TCB and initialized it when the uxBasePriority and uxPriority were initialized during task creation.

These were the modifications. We then used binary semaphores and mutexes with the classic FreeRTOS APIs to compare their performance to PCP.

Demonstration Application

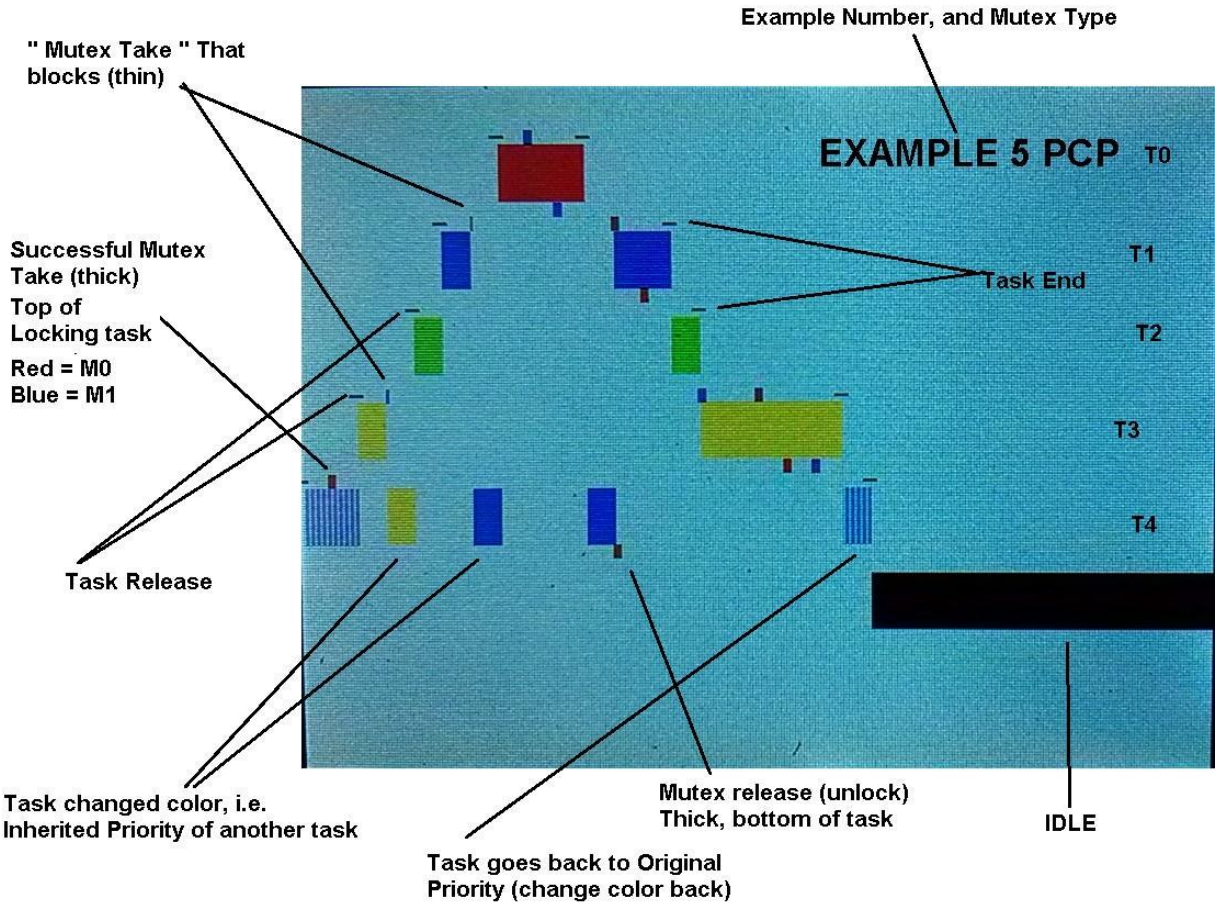
The demonstration application made use of the LCD on the STM discovery board.

Summary of the demo application design:

1- Display that the task is running. Give every task running a color based on its current priority.
2- Only display current running Task. This is done by periodically (every ~100ms) running an interrupt that will "getCurrentRunningTask" from FreeRTOS which simply returns pxCurrentTCB. Based on this information, print to a specific location on the LCD (detailed below). Note that it does not have any impact when we disable the interrupts, since after "locking/unlocking" we enable them right away. So "sampling" by the interrupt is still pretty accurate.
3- highest priority task is displayed in top (t0), and lowest (t4) is displayed in the bottom.
4- IDLE is printed as black (after lowest priority task)
5- Release tasks and mark the release of the task and end tasks (i.e. suspend indefinitly in this case) and mark the end of the task.
6- We have 5 tasks running and create (based on example running) up to 5 mutexes (PCP / Mutex / Binary semaphore). When all tasks are suspended indefinitly (i.e. finished their "role"), IDLE (black) will be printed (i.e. when interrupt fired and "currentRunningTask" is not t0-t5, it assumes idle is running.
7- Mark when PCP_Mutexes are taken. Mark when they are released. Mark the mutex take by a mark on the top of the task, and the mutex release by a mark on the bottom of the task (a small square like tick).

8- Mark when there is a “failed” or “Blocked” Mutex take attempt. Mark it by a thinner stripe on top of when the task was running (and task should get blocked after that)
9- Differentiate between the different types of mutexes taken by giving each mutex mark a different color.
10- Create a for loop that takes ~ 1 second to execute. This will be a simple way of making it look like the “Task is running or doing something” . Make that re-entrant for all tasks.
11- Make examples and conditionally compile them, and conditionally chose which mutex type (PCP / Mutex / Binary semaphore) to use for the example.
12- Design the examples so that they would run in “eye pleasing easy to follow” fashion. For Example, make a task t2 run for 1 second, then lock mutex then run for another second. After that “release” another task (higher priority) and run it for a second and then the higher priority task tries to take the mutex”. etc. This makes “visually simulating” what the tasks are doing much easier (will be shown below).
13- Design “Scenarios” that highlight the advantages of PCP over regular priority inheritance (e.g. no chained blocking, no priority inversion, no deadlocks).
14- Start by testing simple examples and go to more complicated ones.
15- Create a random example (not pre-calculated) and see its behaviour if it will result in proper analysis / confirm to PCP.
16- when having multiple “Mutexs” (PCP / Mutex / Binary Semaphore), mix unlocking one of them then the other (e.g. lock M0, lock M1 then: 1- unlock M0, unlock M1. 2- try unlock M1 then M0).
17- Implement the two server example we had in the slides and see if you get the same results.
18- For simplicity reasons, make examples “run once” in a similar way to how class examples were done
19- Design examples such that, Always, the lower priority task starts first, since if it does not, it will be “naturally” be blocked due to presence of higher priority tasks.
20- An example of “example” implementation is
    a- delay t0 (highest) until all lower priority tasks got to run (i.e. point d)
    b- start with lowest priority for 1 second, make it take a mutex then run for 1 second
    c- then start an intermediate priority task, make it run for 1 second, then make it , for eg. required the mutex taken by lowest priority task. Medium priority should be blocked and low priority run
    d- now since high priority is running, make it run for 1 second (it should pre-empty lowest priority), and then after the 1 second, make the highest priority task take the mutex. It should be blocked and the lowest priority task should now be promoted to highest priority.
    e- sequentially unlock the mutex from lowest task, then highest task, then medium task.
21- The average time for PCP/Mutex/Semaphore lock/unlock will be measured using two seperate timers (one for lock and one for unlock). They will be started when we enter, and stopped when the task yields or when the lock /unlock succeeds.

An Example of implemented “example” of demo application is below. We highlight the meaning of different things drawn on the LCD.



// This is commented out /uncommented out to run a specific example with a specific mutex type

// for Taking mutex
#define PCP
//#define FREE_RTOS_MUTEX

#ifdef FREE_RTOS_MUTEX
//#define REG_MUTEX
#define BIN_MUTEX
#endif

// Example Running
//#define EXAMPLE1
//#define EXAMPLE2
//#define EXAMPLE3

```
//#define EXAMPLE4
//#define EXAMPLE5
#define EXAMPLE6
```

The Algorithm:

Mutexes, and the processes they “Block” are linked list. Every Mutex has a linked list of “Processes” it blocked

```
// Blocked PRocess
typedef struct BlockedProcesses
 {
        struct BlockedProcesses    *      next;
        xTaskHandle                       process;
 }BlockedProcesses;

// Semaphore Control Block
typedef struct PCP_Mutex
    {
        struct PCP_Mutex *              next;
        PCP_MutexID                     id;
        bool                            locked;
        unsigned portBASE_TYPE  priorityCeling;
        xTaskHandle                     taskHoldingPCP_Mutex;
        signed char      *              nameTaskHoldingPCP_Mutex; //used for debug
        BlockedProcesses  *             rootQueueTasksBlocked;
    }PCP_Mutex;
```

Summary of the algorithm is in the following points
For Lock

1- If current task already locked the semaphore, return (no recurrent locking in scope of project)
2- Disable interrupts
3- if priority of the current task wanting lock is <= system ceiling:

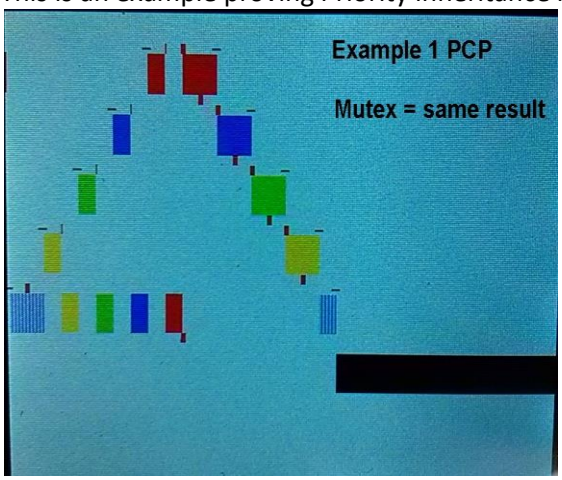
1) Enter an infinite loop to try and lock the mutex. insert this task in the queue of blocked processes by this mutex
   a) if mutex is already locked: give the locker the priority of this task (inherit) if this task has higher priority.
   b) if mutex is not locked, give the process that's causing the "systemCeiling" the priority of this task (inherit) if this task has higher priority.
   c) enable interrupts, and suspend the current task (Let scheduler determine which task gets to run next).
4- This is the "else" of 3-: lock the mutex and update structures properly. Then enable interrupts.

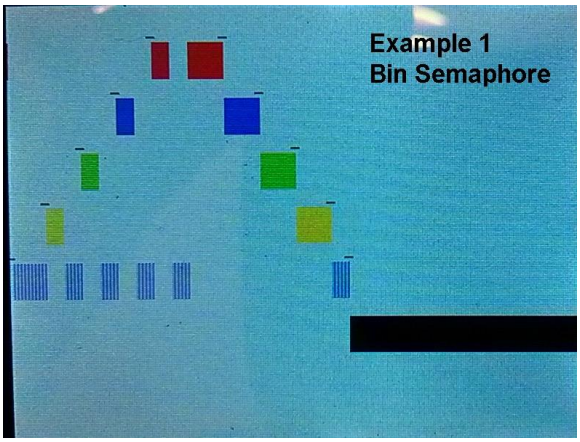
For Unlock

1- if the task is not the task holding the mutex: enter an infinite loop (fail!)
2- Go through list of all other mutexes still locked by this task, and give this unlocking task the highest priority of a task blocked by the mutexes its still holding.
3- If there are tasks that are only blocked by this mutex being unlocked, resume them (but before that disable the scheduler, so that we still have some time to "cleanup" the mutex structure, i.e. set it to unlocked, etc.)
4- Cleanup the mutex structure. "Free" space allocated to "tasks blocked by this mutex"..
5- Enable interrupts.

Example 1

This is an example proving Priority inheritance in PCP.





PCP / Mutex:

In this example, the mutex is taken by lowest priority task, and every time a higher priority task tries to take the mutex, the lowest priority task (t4) gets bumped up in terms of priority (i.e. Priority inhertiance)

Binary Semaphore

priority is not inherited. simply blocking.

Overall

Same performace from all locking mechanisms.

Average PCP Lock time: 13 microseconds

Average PCP Unlock Time: 10 microseconds

Average Bin Semaphore Lock: 4.94 microseconds

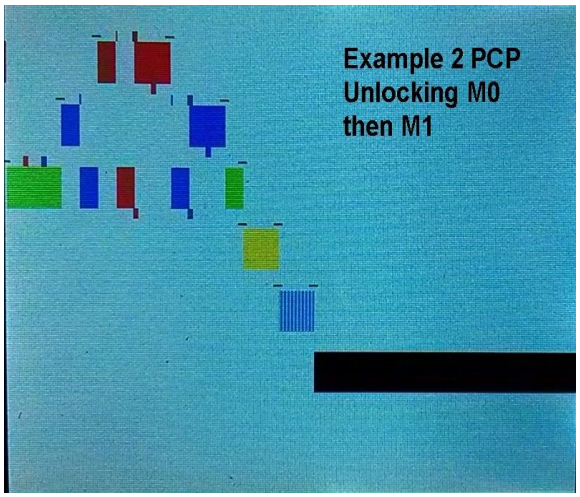
Average Bin Semaphore Unlock: 13.8 microseconds

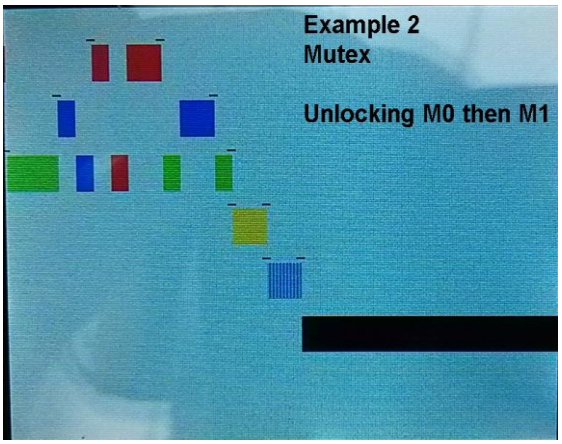
Average Mute Lock: 3.96 microseconds

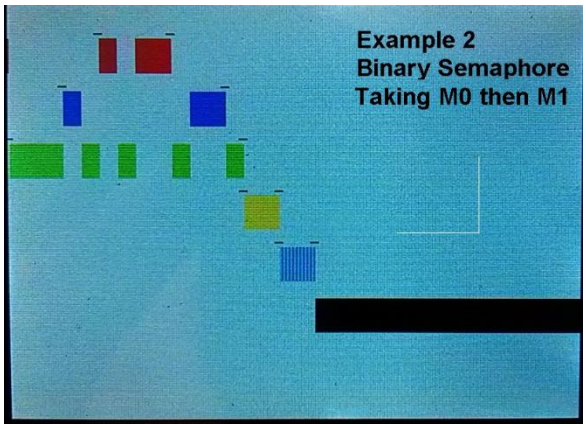
Average Mutex Unlock: 13.8 microseconds

Example 2

1- Unlocking M0 then M1







PCP / Mutex: same Performance. As soon as the M0 is unlocked, the high priority task thats waiting for M0 takes it and commences. same for M1. Note that for Mutex, as soon as we unlock the Mutex that unblocks the highest priority task, the priority goes directly to green where as PCP goes to blue, then unlocks the second one, then green.

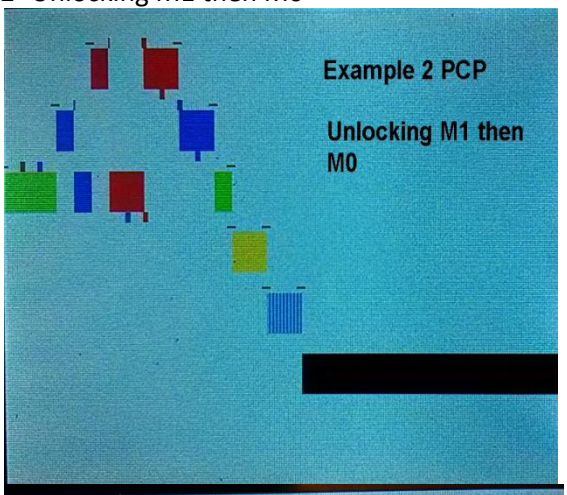
Binary semaphore: same performance. Just no priority inheritance.

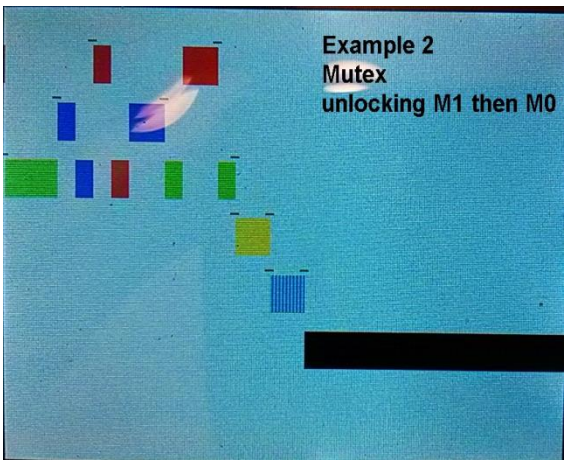
Overall: same performance for all mechanisms. Note that PCP and Mutex examples would have been more efficient than semaphore if the t3 (yellow) and t4 (striped) were released in the middle and given priority higher than t2 (green task). This is where priority inversion would have been prevented by PCP/Mutex, but not by semaphore.

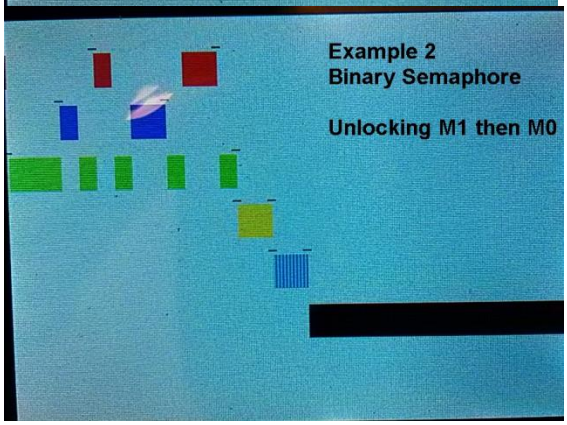
Average PCP Lock time: 13.2 microseconds

Average PCP Unlock Time: 10.43 microseconds

2- Unlocking M1 then M0







PCP: This is an example of Both priority inversion and chained blocking prevention, when using PCP. t2 locks M0 (red) and M1(blue). t1 (blue) is prevented from running (and locking mutex M1) when M1 became available, since the priority of t2 was higher (inherited t0 priority) since M0 was still locked (and t0 wants M0). PCP is a clear winner

Mutex / Binary semaphore: The same performace. Although, for the mutex, it gives the t3 its priority (which has no efficiency effect in this scenario).Note that Mutex would have been more efficient than semaphore if the t3 (yellow) and t4 (striped) were released in the middle and given priority higher than t2 (green task). This is where priority inversion would have been prevented by Mutex, but not by semaphore.

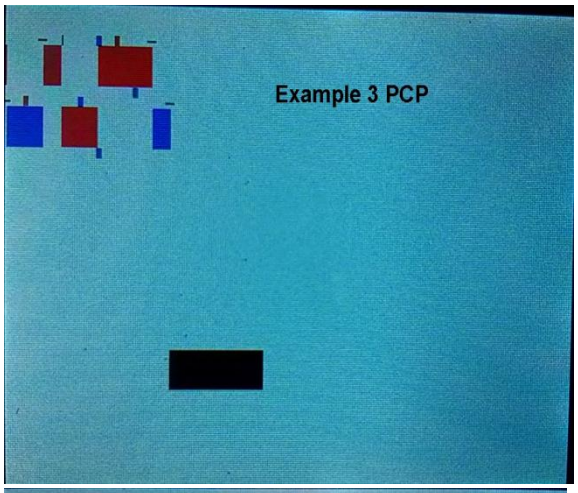
Overall: PCP clearly outperforms other scenarios. Priority inheritance if there were tasks released in the middle

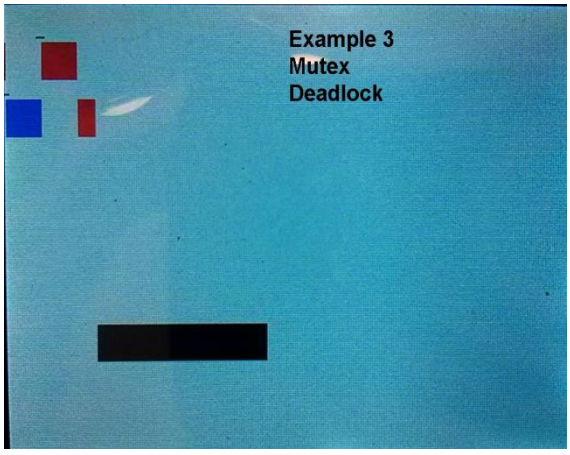
Average PCP Lock time: 11.75 microseconds

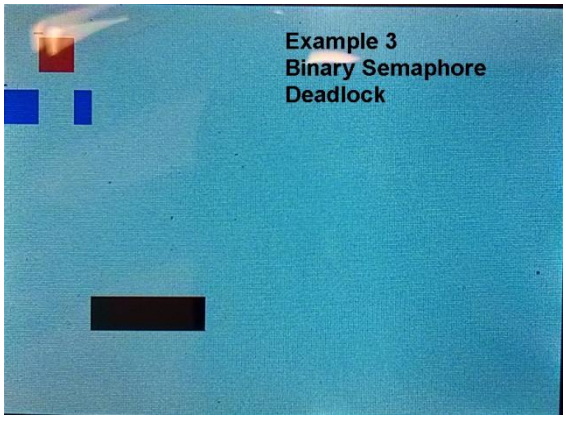
Average PCP Unlock Time: 10.4 microseconds

Example 3

t1 Locks M0(red), and then t0 runs and locks M1(blue) . Then t0 wants M0 (red) and t1 wants M1 (blue).







PCP: PCP outperforms other scenarios. in this case, there is no deadlock since t1 inherited priority of t0 as soon as t0 tried to lock M1.

Mutex/Binary Semaphore: Deadlock. Although in the mutex example, t1 inherits t0 until it needs the resource held by t0 where it deadlocks.

Overall: PCP prevented deadlock. Mutex/Semaphore did not.

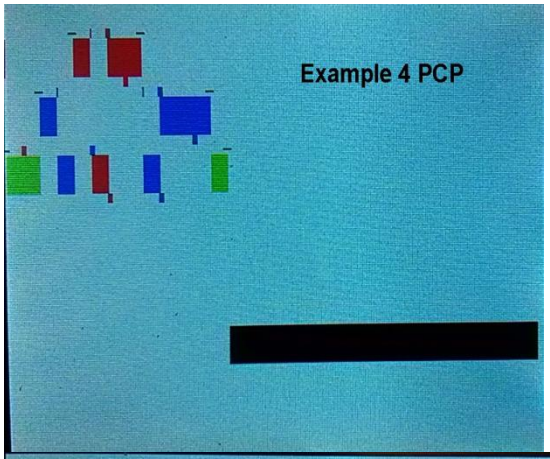
Average PCP lock /unlock time is within range of above examples

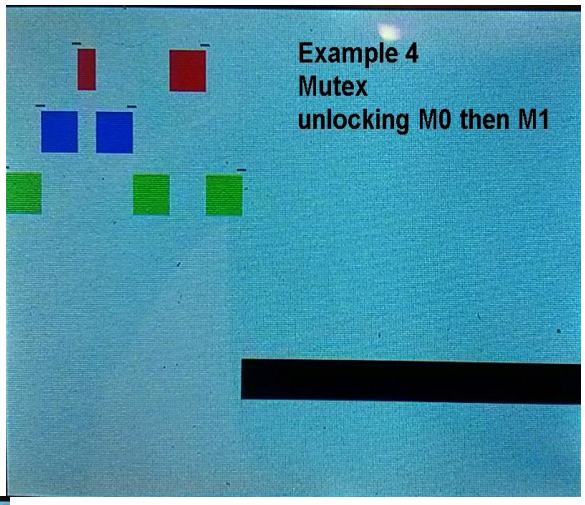
Example 4

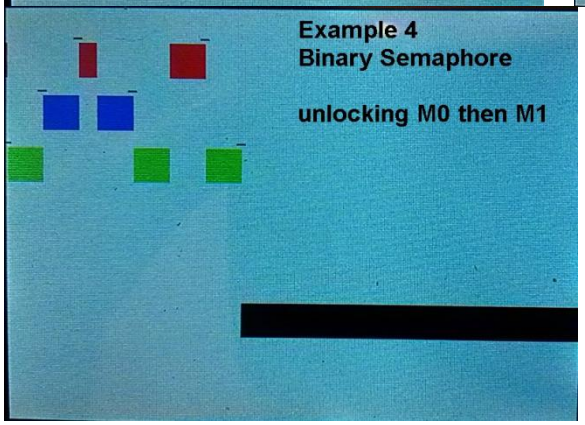
This is similar to example 2. The differnece here is that t2 locks M1 (blue) after t1 wanted to lock M1(blue). Note that t1 was "blocked" from getting M1(blue) because the ceiling was M0 which was locked to t2. (PCP)

Average PCP lock/unlock time is within range of above examples

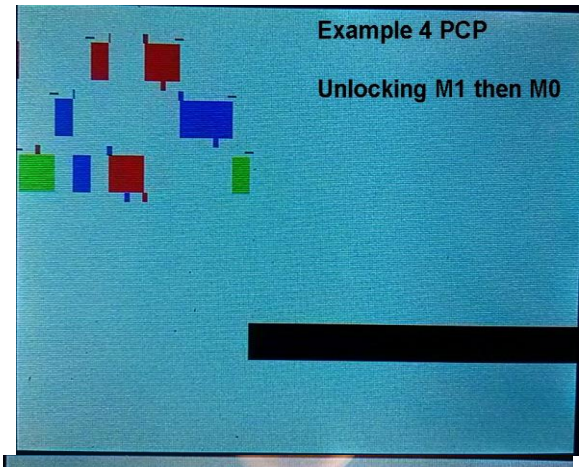
1- Unlocking M0 then M1

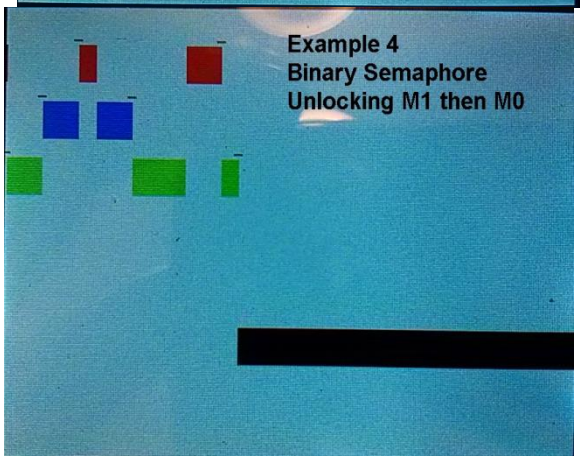






2- Unlocking M1 then M0





Overall: Here, PCP outperforms both when unlocking M0 then M1 and when unlocking M1 then M1 examples. Note that in first part (Unlocking M0 then M1), mutex is identical to semaphore (in performance). When unlocking M1 then M0, its interesting to see that binary semaphore had slightly better performance than mutex, since it allowed t1 (blue) to finish.

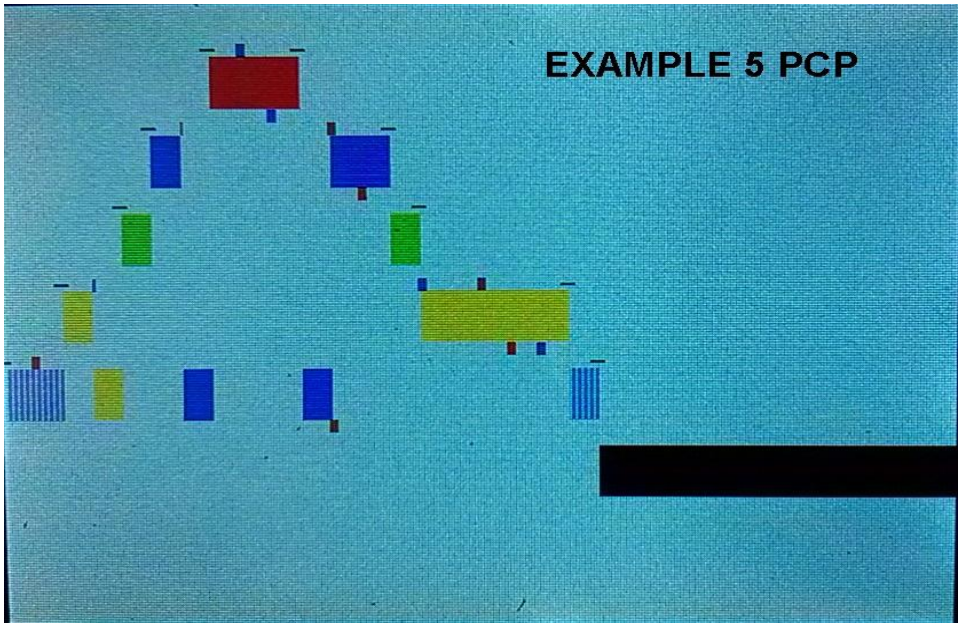
Example 5

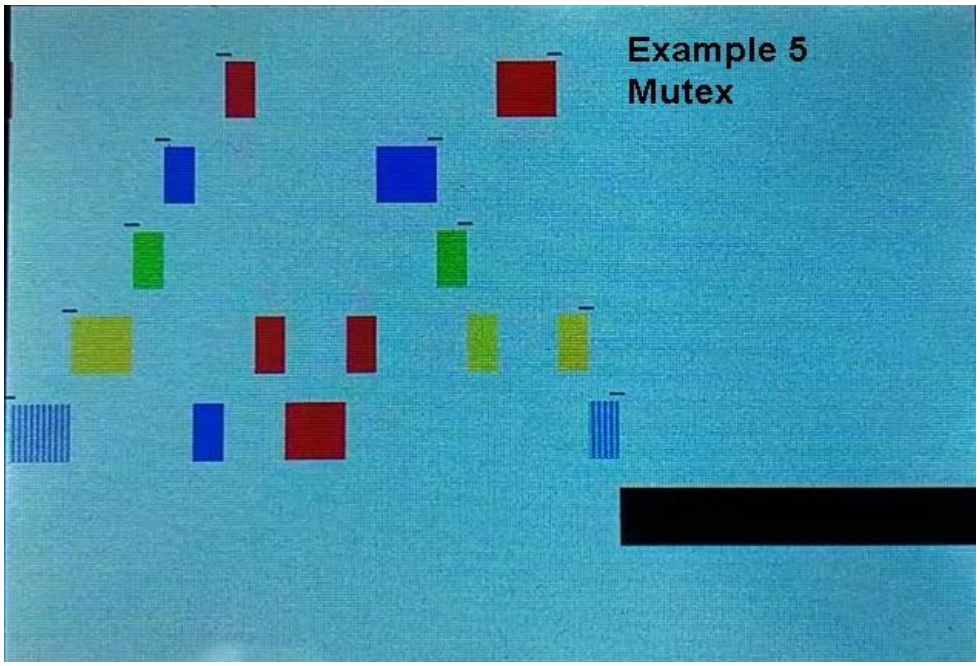
This example is taken from the slides, but it implemented full PCP instead of PCP emulation. It is in the lecture: "Access Control: PCP", Feb 11/14, slide 44.

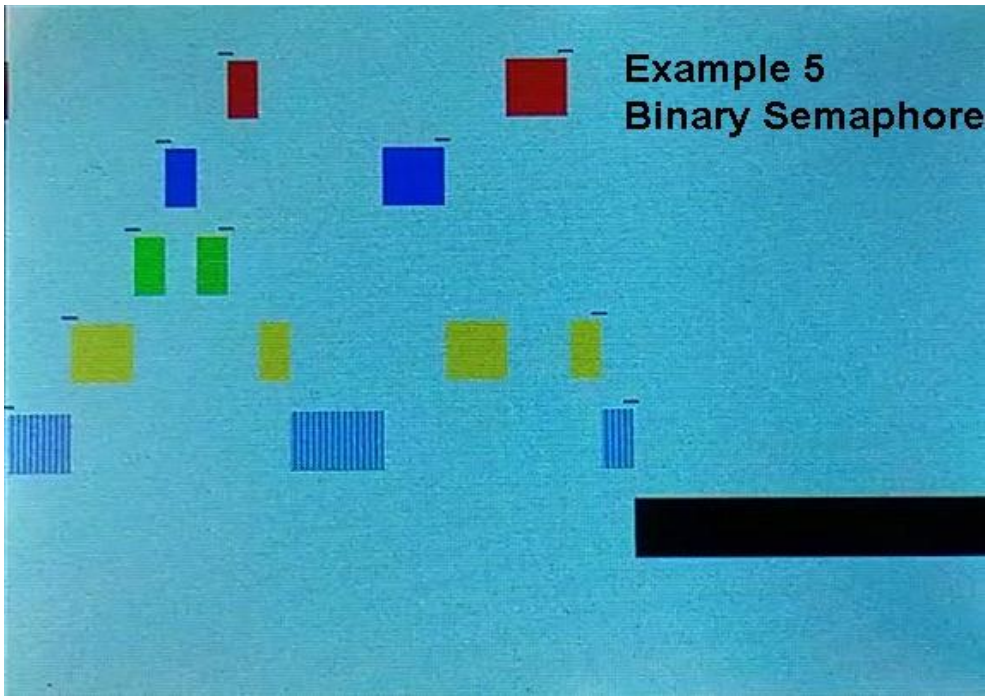
The PCP is clearly the winner here, but as an interesting note, the binary semaphore in this case is actually more efficient that the mutex. This can be seen in t2 (the green task) which does not take any mutexes, where it got to execute and finish before lower priority tasks. The overall delay of t1 (blue) in binary semaphore and in mutex is the same. Binary semaphore favoured t2 (green), and mutex slightly favoured t3 (yellow) as it got to run a little bit in the middle.

Average PCP Lock time: 10.8 microseconds

Average PCP Unlock Time: 9.5 microseconds


EXAMPLE 5 PCP


Example 5
Mutex

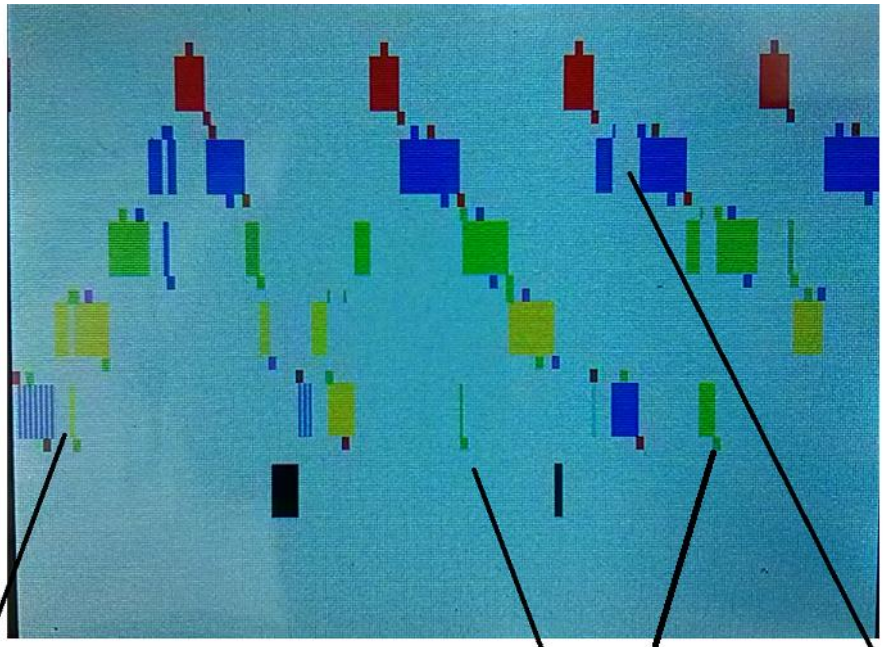


Example 6 (random example)

Below is PCP mutex. Seems to be working ok. Things seem to "form" into a "Pyramid". Maybe this is why government / military is also in a pyramid structure, where things on top get to execute and take precedence to things on bottom. Total tasks used = 5, total mutexes used = 5. M0 ceiling = t0, M1 ceiling = t1, M2 ceiling = t2, M3 ceiling = t3, M4 ceiling = t4.

Average PCP Lock time: approx 13.75 microseconds

Average PCP Unlock Time: approx 11.4 microseconds



Task4 inherits Task2 until it unlocks

Task4 inherits Task3 until it unlocks

Task1 blocked by Task4, and Task4 will be promoted to blue when this happens

Put a breakpoint here and caulculate average lock/unlock time.
This will be worst since we have to traverse a lot of mutex / locked processes linked lists

Conclusions

1- PCP is much more oriented towards running the higher priority tasks than the lower priority tasks (“ lower priority tasks: please get out of the way). This is clearly apparent from the random example we implemented (example 6) where things sort of “formed” into a pyramid.
2- At times, performance of PCP was equivallent to Mutex (simply inheritance) and Binary semaphores. This was in simple examples, mostly with single Mutexes.
3- Overall, Mutex was more “efficient” than simply binary semaphore (gave attention to higher priority tasks), but there were times where this was the opposite. The examples explained which scenario is which.
4- Our implementation of PCP seemed to work 100% in all examples. Efficiency could have been improved greatly if the PCP itself was implemented more with data structure design instead of running through “algorithms” or if else and while, etc.
5- In PCP, average Lock time and Unlock time increases with the increase in number of mutexes created and are locked, and with the number of processes getting blocked on each of those mutexes. This is because traversing the linked lists take more time (we have more processes blocked for a task, and more mutexes to traverse).
This is reflected in the values of lock / unlock times in each example.

Average PCP Locking time is about 13 microseconds or 2340 clock cycles.
Average PCP Unlocking time is about 10 microseconds or 1800 clock cycles

Binary semaphore and Mutex locking were calculated for example 1 only.
Modified queue.c (added functions: startTimerx() stopTimerx() below defined in main.h)

Average Binary Semaphore Locking is 4.94 microseconds or 890 clock cycles
Average Binary Semaphore unlocking time is 13.8 microseconds or 2484 cycles

Average Mutex Locking is 3.96 microseconds or 712 clock cycles
Average Mutex unlocking is 13.8 micro seconds or 2484 cycles.

Average time for Mutex/Bin semaphore Lock / unlock will not change since it does not run through linked lists etc. for that measurement, the functions

```
startTimer2();
startTimer3();

stopTimer2();
stopTimer2();
```

from main.h were used to measure the time. (same functions were used in PCP_mutex.c)

Our Code is therefore considered moderately to highly efficient.

Appendix

Contents of Zip File:

1- This document
2- main.c (demo application and main program) and main.h
3- PCP_mutex.c / PCP_mutex.h (file implementing the PCP Mutex)
4- tasks.c changes attached below (no need to attach whole file)

```
unsigned portBASE_TYPE getTaskPriority( xTaskHandle * const taskHandle )
{
	// Note that this does not get the task's Original (uxBasePriority)
	// but rather the current priority
  tskTCB * const pxTCB = ( tskTCB * ) taskHandle;
	return pxTCB->uxPriority;
}

unsigned portBASE_TYPE getBaseTaskPriority( xTaskHandle * const taskHandle )
{
	// Note that this gets the task's Original (uxBasePriority)
	// rather than the current priority
	tskTCB * const pxTCB = ( tskTCB * ) taskHandle;
	return pxTCB->uxBasePCPPriority;
}
```

Added uxBasePCPPriority to task control block.

The other files were not changed.