ECE 5550G: Advanced Real-Time Systems: Project 4 Report

1. Priority Inheritance Protocol Implementation

- **SchedTCB_t struct**
  - The SchedTCB_t struct extends the standard task control block to include fields relevant to managing resources and mutexes effectively, which is critical for implementing PIP and PCP.
    - These fields track the semaphores each task has acquired, crucial for managing priority inheritance when a task holds one or more mutexes.
    - The '`xDeadlineExceeded`' and '`xMaxExecTimeExceeded`' flags are used to indicate whenever the deadline is missed or the WCET is exceeded so that the respective tasks and free the mutexes they have acquire before being deleted and recreated.

```
#if ( configSUPPORT_DYNAMIC_ALLOCATION == 1 && configUSE_MUTEXES == 1 )
     // Array of semaphores the task currently holds.
     SemaphoreHandle_t xAcquiredSemaphores[MAX_SEMAPHORES_PER_TASK];
     // Count of currently held semaphores.
     BaseType_t xSemaphoreCount;
     BaseType_t xDeadlineExceeded;
     BaseType_t xMaxExecTimeExceeded;
#endif
```

- **static void prvDeleteAndRecreateTask(SchedTCB_t pxTCB)**
  - This function handles the deletion and recreation of tasks, which is crucial during timing errors or deadline misses. This function ensures system integrity by resetting tasks that fail to meet their time constraints.

```
vTaskDelete(*pxTCB->pxTaskHandle); // Delete the task
prvPeriodicTaskRecreate(pxTCB);  // Recreate the task
```

- **BaseType_t vTaskResourceTake(SemaphoreHandle_t xSemaphore)**
  - This function is a critical part of implementing PIP and PCP. It attempts to take a semaphore (mutex) and, upon success, adjusts the task's resource count and stores the semaphore handle.
  - Tracking the acquired semaphores helps manage priority inheritance (implemented inside `xSemaphoreTake`) should other tasks need the same resources.

```
// Attempt to acquire mutex
BaseType_t status = xSemaphoreTake(xSemaphore, portMAX_DELAY);
if (status == pdTRUE) {
     pxThisTask->xResourceAcquired = pdTRUE;
     BaseType_t xIndex;
     for ( xIndex = 0; xIndex < xTaskCounter; xIndex++ )
     {
          if ( pxThisTask->xAcquiredSemaphores[xIndex] == NULL )
          {
              break;
          }
     }
      pxThisTask->xAcquiredSemaphores[pxThisTask->xSemaphoreCount] =
xSemaphore;
      pxThisTask->xSemaphoreCount++;
}
```

- **BaseType_t vTaskResourceGive(SemaphoreHandle_t xSemaphore)**
    - This function complements `vTaskResourceTake()` by releasing a semaphore and updating the task's semaphore count and resource status, crucial for correctly reversing any priority inheritance (implemented inside `xSemaphoreGive()`) that may have been applied.

```
BaseType_t status = xSemaphoreGive(xSemaphore); // Release mutex
if (status == pdTRUE) {
     for (BaseType_t xIndex = 0; xIndex < pxThisTask->xSemaphoreCount;
xIndex++)
     {
          if (pxThisTask->xAcquiredSemaphores[xIndex] == xSemaphore)
         {
               pxThisTask->xAcquiredSemaphores[xIndex] = NULL;
               pxThisTask->xSemaphoreCount--;
               break;
          }
      }
      if (pxThisTask->xSemaphoreCount == 0)
     {
          pxThisTask->xResourceAcquired = pdFALSE;
      }
}
```

- **static void prvCheckDeadline(SchedTCB_t pxTCB, TickType_t xTickCount)**
    - This function checks if a task has missed its deadline, which is a critical part of managing task timing and ensuring system responsiveness.

```
if (pxTCB->xAbsoluteDeadline < xTickCount) {
     pxTCB->xDeadlineExceeded = pdTRUE;
     vTaskNotifyGiveFromISR(xSchedulerHandle, &xHigherPriorityTaskWoken);
}
```

- **static void prvExecTimeExceedHook(SchedTCB_t pxCurrentTask)**
    - Triggered when a task exceeds its execution time, this function marks the task for handling by the scheduler, potentially invoking task recreation.

```
pxCurrentTask->xMaxExecTimeExceeded = pdTRUE;
vTaskNotifyGiveFromISR(xSchedulerHandle, &xHigherPriorityTaskWoken);
```

- **static void prvSchedulerCheckTimingError(TickType_t xTickCount, SchedTCB_t pxTCB)**
    - This scheduler function checks for timing errors, including deadline misses and execution time overruns, and applies appropriate recovery actions.
    - This ensures that tasks adhering to their timing constraints are maintained correctly.
    - If the '`xWorkIsDone`' is set or '`xResourceAcquired`' is not set it means that the task has not acquired any resource and is free to be deleted and recreated.

```
if (pxTCB->xDeadlineExceeded == pdTRUE && (pxTCB->xWorkIsDone == pdTRUE ||
pxTCB->xResourceAcquired == pdFALSE)) {
     prvDeleteAndRecreateTask(pxTCB);
     pxTCB->xDeadlineExceeded = pdFALSE;
}
if (pxTCB->xMaxExecTimeExceeded == pdTRUE && (pxTCB->xWorkIsDone == pdTRUE ||
pxTCB->xResourceAcquired == pdFALSE)) {
     prvDeleteAndRecreateTask(pxTCB);
     pxTCB->xMaxExecTimeExceeded = pdFALSE;
}
```

- **void vApplicationTickHook(void)**
    - Invoked every tick, this function is crucial for monitoring tasks' execution times and deadlines, supporting the PIP and PCP.

```
#if( schedUSE_TIMING_ERROR_DETECTION_EXECUTION_TIME == 1 )
    if( pxCurrentTask->xMaxExecTime <= pxCurrentTask->xExecTime )
    {
        if( pdFALSE == pxCurrentTask->xMaxExecTimeExceeded )
        {
            prvExecTimeExceedHook( pxCurrentTask );
        }
    }
#endif /* schedUSE_TIMING_ERROR_DETECTION_EXECUTION_TIME */

#if( schedUSE_TIMING_ERROR_DETECTION_EXECUTION_TIME == 1 )
    if( pdFALSE == pxCurrentTask->xDeadlineExceeded )
    {
        prvCheckDeadline( pxCurrentTask, xTaskGetTickCountFromISR() );
    }
#endif /* schedUSE_TIMING_ERROR_DETECTION_EXECUTION_TIME */
```

Testing

1. Priority Inheritance Test: To check if basic priority inheritance is happening or not we take 2 tasks -

Tasks	Phase	Period	WCET	Relative Deadline
T1	7	24	6	24
T2	0	48	12	42

(C, D, and T values are in ticks)

Based on the RM scheduling algorithm, T1 has a higher priority (priority = 3) than T2 (priority = 2). Both tasks try to acquire the same semaphore, say S1.

Since T1 has a non-zero phase, T2 starts executing first and acquires the semaphore S1. While it holds the semaphore, T1 is activated and tries to acquire the semaphore. This should cause a priority inversion and we can verify the implementation of PIP is correct based on the following console output which shows priority inheritance -

```
11:03:25.290 -> t2: BP=2; CP=2
11:03:25.290 -> t2 acquired mutex!
11:03:25.390 -> t1: BP=3; CP=3
11:03:26.119 -> t2: BP=2; CP=3
```

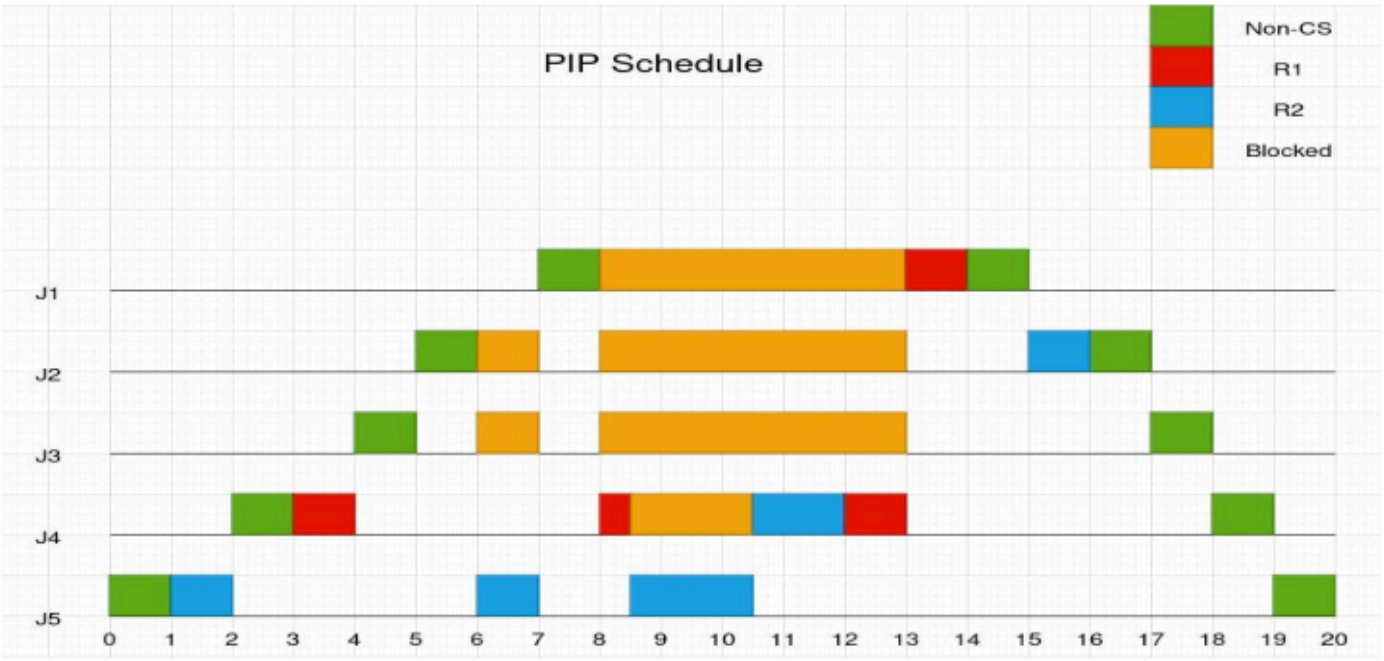
Here BP is the base priority and CP is the current priority. We can see that T2 inherits the priority from T1 while T1 is blocked on semaphore S1.

2. Complex Task Set: To check a more complicated scenario, we use the following task set from HW4 Part 1.

Job	r	C	π	Resources
J_1	7	3	5	R_1 for 1 time unit
J_2	5	3	4	R_2 for 1 time unit
J_3	4	2	3	None
J_4	2	6	2	R_1 for 4 time units, R_2 for 1.5 time units (nested)
J_5	0	6	1	R_2 for 4 time units

Tasks	Phase	Period	WCET	Relative Deadline
T1	70	200	40	130
T2	50	200	40	150
T3	40	200	30	160
T4	20	200	70	180
T5	0	200	70	200

Theoretically, we should expect the following schedule based on the PIP algorithm -



Based on the above task set we get the following console output -

The console output matches the expected sequence of execution and priority inheritance values to the theoretic schedule.

```
20:38:01.404 -> T5
20:38:01.404 -> P5=1
20:38:02.325 -> T5
20:38:02.325 -> P5=1
20:38:02.690 -> T4
20:38:02.690 -> P4=2
20:38:03.021 -> T3
20:38:03.021 -> P3=3
20:38:03.220 -> T2
20:38:03.220 -> P2=4
20:38:03.353 -> T5
20:38:03.353 -> P5=4
20:38:03.551 -> T1
20:38:03.551 -> P1=5
20:38:03.684 -> T4
20:38:03.684 -> P4=5
20:38:03.750 -> T5
20:38:03.750 -> P5=5
20:38:04.082 -> T4
20:38:04.082 -> P4=5
20:38:04.481 -> T1
20:38:04.481 -> P1=5
20:38:04.780 -> T2
20:38:04.780 -> P2=4
20:38:05.080 -> T3
20:38:05.080 -> P3=3
20:38:05.180 -> T4
20:38:05.213 -> P4=2
20:38:05.313 -> T5
20:38:05.313 -> P5=1
```

2. Priority Ceiling Protocol Implementation

The FreeRTOS kernel implements semaphores and mutexes via queues. FreeRTOS already implements the PIP algorithm in the native xSemaphoreTake() and xSemaphoreGive() functions by inheriting and de-inheriting priorities appropriately. To implement PCP, I have changed the '**queue.c**' file to build the PCP functionality on the already existing PIP implementation.

- **Queue_t structure**
  - This structure has been augmented to include a priority ceiling for each queue (mutex).
  - The uxCeilingPriority member holds the priority ceiling of the mutex, crucial for PCP operations.

```
typedef struct QueueDefinition {
      UBaseType_t uxMessagesWaiting;
      UBaseType_t uxLength;
      UBaseType_t uxCeilingPriority;  // Added for PCP
      ...
} Queue_t;
```

- **xMutexesPCP array**
  - This array stores all mutexes (queues) that are being managed under PCP, including their mutex holders and ceiling priorities.

```
typedef struct {
      ...
      UBaseType_t uxCeilingPriority;
      ...
} Mutex_t;

Mutex_t xMutexesPCP[MAX_MUTEXES];
```

- **void vInitializePriorityCeiling(QueueHandle_t xQueue, UBaseType_t uxCeilingPriority)**
  - This function initializes a mutex (queue) with a specified priority ceiling. This ceiling represents the highest priority that any task holding this mutex can assume, crucial in PCP to prevent priority inversions.
  - The priority ceiling is set upon mutex initialization and used in subsequent operations to adjust task priorities when the mutex is acquired or released.
  - This function has to be called by the user after creating mutexes and before starting the scheduler.

```
void vInitializePriorityCeiling( QueueHandle_t xQueue, UBaseType_t
uxCeilingPriority ) {
      if (xQueue != NULL) {
      xQueue->uxCeilingPriority = uxCeilingPriority;
     xMutexesPCP[uxIndex++] = xQueue;
      }
}
```

- **UBaseType_t xGetHighestCeilingPriority(TaskHandle_t xCurrentTaskHandle)**
  - This function scans all mutexes to find the highest priority ceiling among them. This is used to decide whether a task is eligible to acquire a mutex.

```
UBaseType_t xGetHighestCeilingPriority( TaskHandle_t xCurrentTaskHandle ) {
      UBaseType_t uxCeiling = 0;
      for (int i = 0; i < MAX_MUTEXES; i++) {
          if (xMutexesPCP[i].xOwner != xCurrentTaskHandle &&
xMutexesPCP[i].uxCeilingPriority > uxCeiling) {
           uxCeiling = xMutexesPCP[i].uxCeilingPriority;
          }
      }
      return uxCeiling;
}
```

- **QueueHandle_t xGetHighestCeilingPriorityMutex(QueueHandle_t xQueue, TaskHandle_t xCurrentTaskHandle)**
  - This function determines which mutex has the highest priority ceiling. It's used to manage task priorities dynamically when multiple mutexes are involved.

```
QueueHandle_t xGetHighestCeilingPriorityMutex( QueueHandle_t xQueue,
TaskHandle_t xCurrentTaskHandle ) {
      for(UBaseType_t uxIndex = 0; uxIndex < queueMAX_MUTEXES_PCP; uxIndex++)
      {
           if ( xMutexesPCP[uxIndex] != NULL )
           {
                Queue_t *pxQueue = xMutexesPCP[uxIndex];
                if ( pxQueue->u.xSemaphore.xMutexHolder != NULL &&
pxQueue->u.xSemaphore.xMutexHolder != xCurrentTaskHandle )
                {
                if ( pxQueue->uxCeilingPriority > uxHighestCeilingPriority )
                {
                     uxHighestCeilingPriority = pxQueue->uxCeilingPriority;
                     xHighestCeilingPrioritySemaphore = pxQueue;
```

```
                }
            }
        }
    }
}
```

- **BaseType_t xQueueSemaphoreTake(QueueHandle_t xQueue, TickType_t xTicksToWait)**
  - This function is a modified version of the standard semaphore acquisition function, integrating PCP to manage task priorities when a semaphore is taken.
  - It checks if the priority of the calling task is greater than the ceiling priority of mutexes held by other tasks.
  - If yes, the calling task goes ahead and acquires the mutex.
  - If not, priority inheritance occurs and the task is blocked on the mutex with the highest ceiling priority (using `xGetHighestCeilingPriorityMutex()`) and is added to its waiting queue until the timeout expires or the mutex becomes available.

```
if( (xTaskDetails.uxCurrentPriority > xGetHighestCeilingPriority(
xTaskDetails.xHandle )) && (uxSemaphoreCount > ( UBaseType_t ) 0) )
{
    /* Semaphores are queues with a data size of zero and where the
     * messages waiting is the semaphore's count.  Reduce the count. */
    pxQueue->uxMessagesWaiting = uxSemaphoreCount - ( UBaseType_t ) 1;
#if ( configUSE_MUTEXES == 1 )
{
    if( pxQueue->uxQueueType == queueQUEUE_IS_MUTEX )
    {
        /* Record the information required to implement
         * priority inheritance should it become necessary. */
        pxQueue->u.xSemaphore.xMutexHolder = pvTaskIncrementMutexHeldCount();
    }
}
#endif /* configUSE_MUTEXES */
}

...
...

if( prvIsQueueEmpty( pxQueue ) != pdFALSE )
{
#if ( configUSE_MUTEXES == 1 )
{
    if( pxQueue->uxQueueType == queueQUEUE_IS_MUTEX )
    {
```

```
        taskENTER_CRITICAL();
        {
            xInheritanceOccurred =
xTaskPriorityInherit(pxQueue->u.xSemaphore.xMutexHolder);
        }
        taskEXIT_CRITICAL();
    }

}
```

- **BaseType_t xQueueGenericSend(QueueHandle_t xQueue, const void * const pvItemToQueue, TickType_t xTicksToWait, const BaseType_t xCopyPosition)**
    - This function is used to release a mutex.
    - In the context of PCP, when a task releases a semaphore (which may be represented as a queue in FreeRTOS), its priority should be restored to its original level if it was previously elevated due to acquiring a semaphore with a higher priority ceiling.
    - If the operation results in unblocking a higher-priority task waiting on the queue, the current task yields immediately. This mechanism is vital for maintaining system responsiveness and is aligned with PCP practices, where a task must relinquish control promptly when a higher-priority task becomes runnable.

```
xYieldRequired = prvCopyDataToQueue( pxQueue, pvItemToQueue, xCopyPosition );

/* If there was a task waiting for data to arrive on the
 * queue then unblock it now. */
if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToReceive ) ) == pdFALSE )
{
  if( xTaskRemoveFromEventList( &( pxQueue->xTasksWaitingToReceive ) ) !=
pdFALSE )
  {
     queueYIELD_IF_USING_PREEMPTION();
  }

...
...

static BaseType_t prvCopyDataToQueue( Queue_t * const pxQueue,
                                      const void * pvItemToQueue,
                                      const BaseType_t xPosition )
{
     BaseType_t xReturn = pdFALSE;
     UBaseType_t uxMessagesWaiting;
```

```
    uxMessagesWaiting = pxQueue->uxMessagesWaiting;
    if( pxQueue->uxItemSize == ( UBaseType_t ) 0 )
    {
        #if ( configUSE_MUTEXES == 1 )
        {
            if( pxQueue->uxQueueType == queueQUEUE_IS_MUTEX )
            {
              /* The mutex is no longer being held. */
              xReturn = xTaskPriorityDisinherit(
pxQueue->u.xSemaphore.xMutexHolder );
              pxQueue->u.xSemaphore.xMutexHolder = NULL;
            }
        }
    }
```

- **configUSE_PRIORITY_CEILING_PROTOCOL**
    - Use the above macro in FreeRTOSConfig.h to use PCP instead of PIP.

```
#define configUSE_PRIORITY_CEILING_PROTOCOL 1
```

Testing

1. Priority Inheritance Test: To check if basic priority inheritance is happening or not we take 2 tasks -

Tasks	Phase	Period	WCET	Relative Deadline
T1	7	24	6	24
T2	0	48	12	42

(C, D, and T values are in ticks)

Based on the RM scheduling algorithm, T1 has a higher priority (priority = 3) than T2 (priority = 2). Both tasks try to acquire the same semaphore, say S1.

Since T1 has a non-zero phase, T2 starts executing first and acquires the semaphore S1. While it holds the semaphore, T1 is activated and tries to acquire the semaphore. This should cause a priority inversion and we can verify the implementation of PIP is correct based on the following console output which shows priority inheritance -

```
11:03:25.290 -> t2: BP=2; CP=2
11:03:25.290 -> t2 acquired mutex!
11:03:25.390 -> t1: BP=3; CP=3
11:03:26.119 -> t2: BP=2; CP=3
```

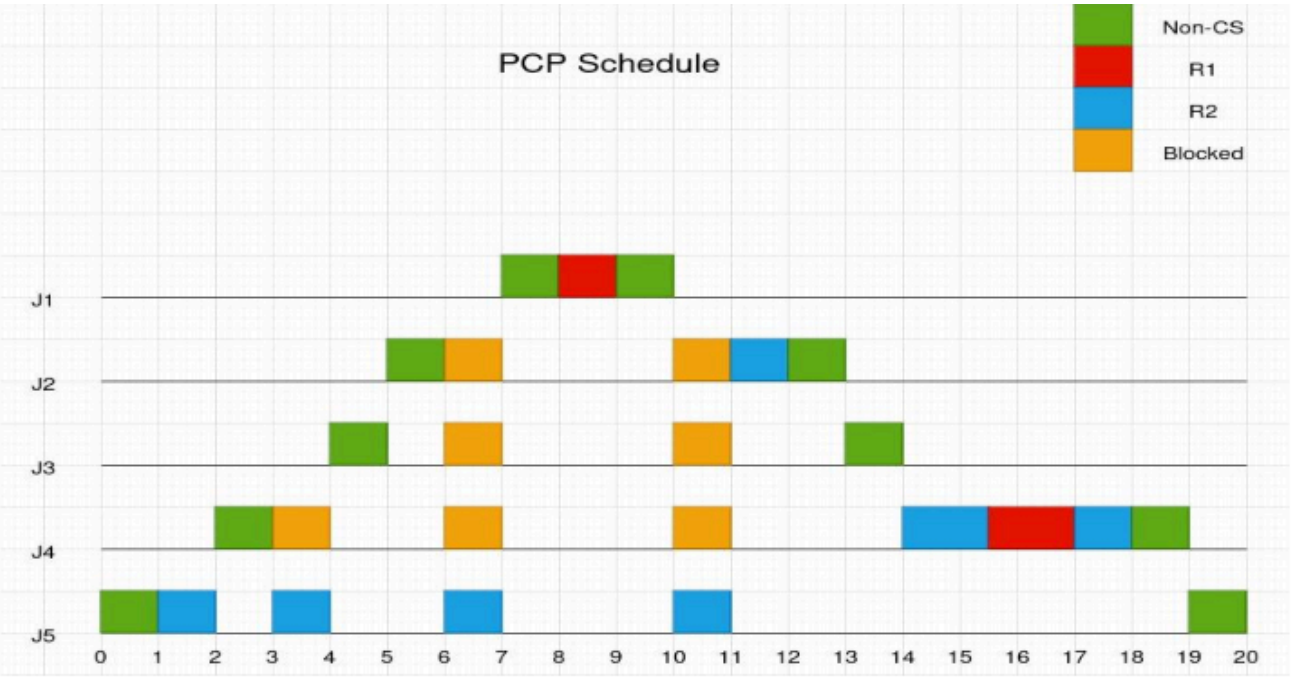
Here BP is the base priority and CP is the current priority. We can see that T2 inherits the priority from T1 while T1 is blocked on semaphore S1.

2. Complex Task Set: To check a more complicated scenario, we use the following task set from HW4 Part 1.

Job	r	C	π	Resources
J_1	7	3	5	R_1 for 1 time unit
J_2	5	3	4	R_2 for 1 time unit
J_3	4	2	3	None
J_4	2	6	2	R_1 for 4 time units, R_2 for 1.5 time units (nested)
J_5	0	6	1	R_2 for 4 time units

Tasks	Phase	Period	WCET	Relative Deadline
T1	70	200	40	130
T2	50	200	40	150
T3	40	200	30	160
T4	20	200	70	180
T5	0	200	70	200

Theoretically, we should expect the following schedule based on the PCP algorithm -



Based on the above task set we get the following console output -

The console output matches the expected sequence of execution and priority inheritance values to the theoretic schedule.

```
20:29:39.179 -> T5
20:29:39.179 -> P5=1
20:29:40.106 -> T5
20:29:40.106 -> P5=1
20:29:40.437 -> T4
20:29:40.437 -> P4=2
20:29:40.571 -> T5
20:29:40.571 -> P5=2
20:29:40.770 -> T3
20:29:40.803 -> P3=3
20:29:40.970 -> T2
20:29:40.970 -> P2=4
20:29:41.104 -> T5
20:29:41.104 -> P5=4
20:29:41.303 -> T1
20:29:41.303 -> P1=5
20:29:41.767 -> T5
20:29:41.767 -> P5=4
20:29:41.867 -> T2
20:29:41.900 -> P2=4
20:29:42.166 -> T3
20:29:42.166 -> P3=3
20:29:42.299 -> T4
20:29:42.299 -> P4=2
20:29:42.531 -> T5
20:29:42.531 -> P5=1
```