Université Djilali BOUNAAMA, Khemis Miliana
مليانة خميس بونعامة جيلالي جامعة
Faculté des Sciences et de la Technologie
والتكنولوجيا العلوم كلية

Série d'exercices 8: Partie B : Noyau et services

Exercice 1 : Sémaphores binaires pour signalisation inter-tâches

Deux tâches partagent une ressource matérielle (un capteur). La tâche A lit les données du capteur toutes les 100 ms. La tâche B traite ces données. On veut que B ne traite que lorsqu'une nouvelle donnée est disponible (pas de polling actif). Utilisez un sémaphore binaire pour synchroniser A (producteur) et B (consommateur).

Exercice 2 : Mutex avec gestion d'inversion de priorité

Trois tâches de priorités différentes (Haute=3, Moyenne=2, Basse=1) partagent une ressource critique protégée par un mutex. Simulez un scénario d'inversion de priorité classique et montrez comment un mutex avec héritage de priorité (priority inheritance) le résout.

Exercice 3 : File de messages (queue) pour communication

Une tâche produit des relevés de température (structure { uint32_t timestamp; float value; }). Une autre tâche les affiche sur un écran. Utilisez une file de messages pour transmettre ces structures de manière déterministe (taille fixe).

Exercice 4 : Memory pool (allocation fixe) vs heap dynamique

Comparez les deux approches pour allouer 5 blocs de 256 octets. Montrez comment un memory pool évite la fragmentation, contrairement au heap. Écrivez le code pour créer et utiliser un pool de mémoire fixe.

Exercice 5: Timer logiciel périodique et watchdog logiciel

Créez un timer logiciel qui s'exécute toutes les 1 seconde pour surveiller une tâche "travailleuse". Si la tâche ne signale pas son activité dans un délai de 2 secondes, le timer déclenche une action corrective (redémarrage de la tâche). Implémentez un watchdog logiciel simple.

Exercice 6: Exercice 6 : Débogage en conditions réelles – Tâche de supervision

Créez une tâche de supervision qui affiche toutes les 5 secondes l'état des autres tâches : nom,

pourcentage d'utilisation CPU (via des hooks de trace), nombre de fois qu'elles ont manqué leur deadline. Utilisez des compteurs incrémentés dans chaque tâche.

Solution Série 8

Exercice 1 :

Solution (pseudo-code RTOS – type FreeRTOS) :

```
SemaphoreHandle_t binSem;

void TaskCapteur(void *pvParameters) {
   while(1) {
      LireCapteur(&donnee);
      // Signaler que la donnée est prête
      xSemaphoreGive(binSem);
      vTaskDelay(pdMS_TO_TICKS(100));
   }
}

void TaskTraitement(void *pvParameters) {
   while(1) {
      // Attendre la donnée (bloquant)
      xSemaphoreTake(binSem, portMAX_DELAY);
      TraiterDonnee();
   }
}

void main() {
   binSem = xSemaphoreCreateBinary();
   xTaskCreate(TaskCapteur, "Capteur", 200, NULL, 1, NULL);
   xTaskCreate(TaskTraitement, "Traitement", 200, NULL, 1, NULL);
   vTaskStartScheduler();
}
```

Exercice 2 :

```
MutexHandle_t mutex;

void TaskBasse(void *pvParameters) {
   xSemaphoreTake(mutex, portMAX_DELAY);
   // Longue section critique
   vTaskDelay(pdMS_TO_TICKS(500));
   xSemaphoreGive(mutex);
}

void TaskMoyenne(void *pvParameters) {
   while(1) {
      vTaskDelay(pdMS_TO_TICKS(100));
      // Tâche CPU-bound sans le mutex
      for(int i=0; i<100000; i++);
   }
}

void TaskHaute(void *pvParameters) {
   vTaskDelay(pdMS_TO_TICKS(200));
   xSemaphoreTake(mutex, portMAX_DELAY);
```

```
    // Sans héritage de priorité, attendrait longtemps
    xSemaphoreGive(mutex);
}

// Dans main
mutex = xSemaphoreCreateMutex(); // Héritage de priorité activé par défaut
```

Explication :

- Sans héritage : TaskBasse prend le mutex, TaskHaute préempte, bloque sur mutex → TaskMoyenne (prio 2) tourne et retarde TaskHaute.
- Avec héritage : TaskBasse hérite temporairement de la priorité 3, donc TaskMoyenne ne peut pas préempter → inversion résolue.

Exercice 3 :

```
typedef struct {
    uint32_t timestamp;
    float value;
} TempData;

QueueHandle_t tempQueue;

void TaskProducteur(void *pvParameters) {
    TempData data;
    while(1) {
        data.timestamp = xTaskGetTickCount();
        data.value = LireTemperature();
        // Envoi bloquant si pleine (timeout 10 ms)
        xQueueSend(tempQueue, &data, pdMS_TO_TICKS(10));
        vTaskDelay(pdMS_TO_TICKS(50));
    }
}

void TaskAfficheur(void *pvParameters) {
    TempData received;
    while(1) {
```

```
        if(xQueueReceive(tempQueue, &received, portMAX_DELAY)) {
            printf("T=%d -> %.2f°C\n", received.timestamp, received.value);
        }
    }
}

// Création : queue de 10 éléments
tempQueue = xQueueCreate(10, sizeof(TempData));
```

Exercice 4

```
// Memory pool
static uint8_t poolBuffer[5 * 256];
StaticPool_t poolCtrl;
PoolHandle_t pool = xCreateStaticPool(poolBuffer, 5, 256, &poolCtrl);

void* bloc1 = xPoolAlloc(pool, portMAX_DELAY);
void* bloc2 = xPoolAlloc(pool, portMAX_DELAY);
xPoolFree(pool, bloc1); // Pas de fragmentation
void* bloc3 = xPoolAlloc(pool, portMAX_DELAY); // Réutilise l'emplacement exact

// Versus heap dynamique (ex: pvPortMalloc)
void* a = pvPortMalloc(256);
void* b = pvPortMalloc(256);
vPortFree(a);
void* c = pvPortMalloc(300); // Si heap fragmenté, peut échouer alors qu'il y a assez de mémoire
```

Avantage pool : temps d'allocation constant, pas de fragmentation, mais pas de taille variable.

Exercice 5

```
TaskHandle_t taskWorker;
TimerHandle_t watchdogTimer;
```

```
volatile bool taskAlive = true;

void WorkerTask(void *pvParameters) {
   while(1) {
      // Travail réel
      vTaskDelay(pdMS_TO_TICKS(800));
      taskAlive = true; // Reset watchdog
   }
}

void WatchdogCallback(TimerHandle_t xTimer) {
   if(!taskAlive) {
      // Détection de blocage
      vTaskDelete(taskWorker);
      xTaskCreate(WorkerTask, "Worker", 200, NULL, 2, &taskWorker);
      printf("Watchdog: Tâche relancée\n");
   }
   taskAlive = false;
}

void main() {
   xTaskCreate(WorkerTask, "Worker", 200, NULL, 2, &taskWorker);
   watchdogTimer = xTimerCreate("Wdog", pdMS_TO_TICKS(2000), pdTRUE,
NULL, WatchdogCallback);
   xTimerStart(watchdogTimer, 0);
}
```

Exercice 6

```
// Dans chaque tâche
volatile uint32_t taskMissedDeadline[TASK_COUNT];

void TaskCritical(void *pvParameters) {
```

```
    TickType_t start = xTaskGetTickCount();
    while(1) {
        // Travail avec deadline 50 ms
        if((xTaskGetTickCount() - start) > pdMS_TO_TICKS(50))
            taskMissedDeadline[0]++;
        start = xTaskGetTickCount();
        vTaskDelay(pdMS_TO_TICKS(40));
    }
}

void SupervisorTask(void *pvParameters) {
    char taskName[20];
    UBaseType_t uxHighWaterMark;
    while(1) {
        for(int i=0; i<TASK_COUNT; i++) {
            vTaskGetTaskInfo(taskHandles[i], &taskInfo, pdTRUE, &uxHighWaterMark);
            printf("%s: Stack free=%d, Missed=%lu\n",
                taskInfo.pcTaskName, uxHighWaterMark, taskMissedDeadline[i]);
        }
        vTaskDelay(pdMS_TO_TICKS(5000));
    }
}
```

Pour le débogage réel : on peut aussi utiliser configUSE_TRACE_FACILITY et vTaskGetRunTimeStats() pour obtenir le pourcentage CPU.