TÓPICOS ESPECIAIS – SISTEMAS EMBARCADOS
PROGRAMAÇÃO PARALELA E TEMPO REAL

PROF. JOSENALDE OLIVEIRA

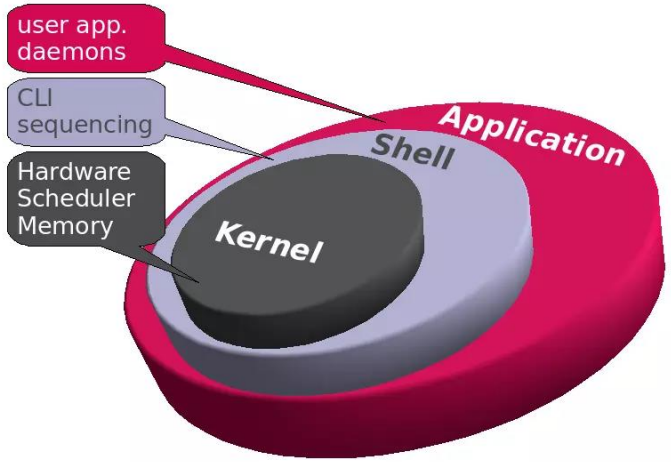
josenalde.oliveira@ufrn.br

ANÁLISE E DESENVOLVIMENTO DE SISTEMAS - UFRN

DIH – análise de código no ESP32 – conceitos base

- RTOS: ambiente multitarefas para sistemas embarcados
  - Um **Sistema Operacional** é um conjunto de ferramentas (softwares) que criam um ambiente multitarefas e também é uma abstração entre software e hardware, incluindo gerenciamento de recursos internos

https://www.freertos.org/index.html



- **Kernel**: camada de abstração entre sw e hw para gerência de recursos, como memória, processos, I/O etc.

- **Scheduler**: software que "decide" quais tarefas executar, seja por maior prioridade ou, se iguais, por tentativa de divisão igualitária de tempo. No caso pthread_t as threads **bounded** (padrão) são gerenciadas pelo scheduler.

- **Task** (tarefa): como se fossem 'miniprogramas', podem ser totalmente independentes ou compartilhar recursos. Assim um código pode ser dividido entre tarefas gerenciadas pelo *scheduler*. *No FreeRTOS uma thread em execução é chamada TASK*

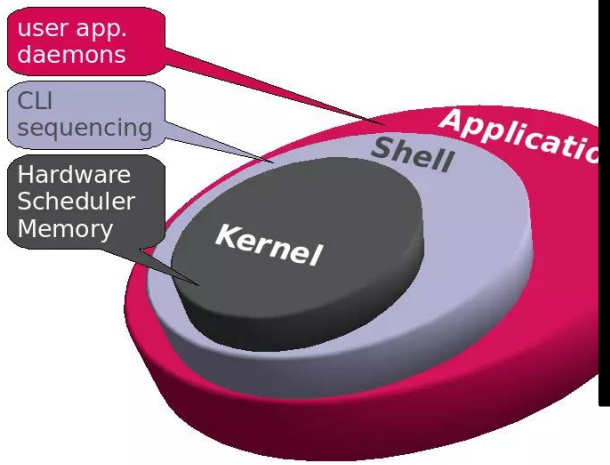
https://www.zephyrproject.org/

Exemplo de código de sincronização de tasks com semáforos no Zephyr

DIH – análise de código no ESP32 – conceitos base

- RTOS: ambiente multitarefas para sistemas embarcados
  - Um **Sistema Operacional** é um conjunto de ferramentas (softwares) que criam um ambiente multitarefas e também é uma abstração entre

https://www.freertos.org/index.html




```
pthread_attr_t meuAtributo;
pthread_t minhaThread;
pthread_attr_init(&meuAtributo);
pthread_attr_setscope(&meuAtributo, PTHREAD_SCOPE_SYSTEM); //ou _PROCESS
pthread_attr_setdetachstate(&meuAtributo, PTHREAD_CREATE_JOINABLE); //ou
_DETACHED
pthread_create(&minhaThread, &meuAtributo, <func>, NULL);
```

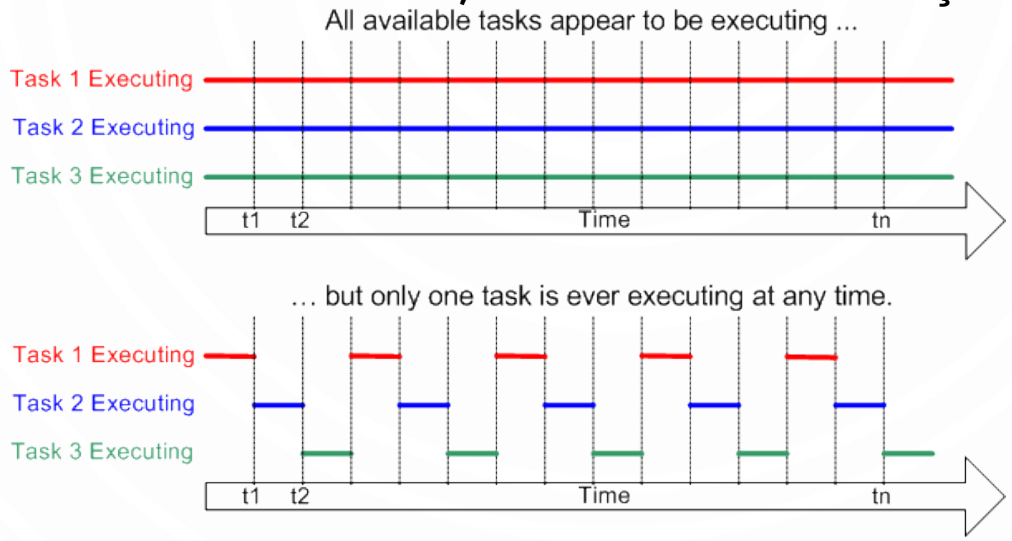
- **Task** (tarefa): como se fossem 'miniprogramas', podem ser totalmente independentes ou compartilhar recursos. Assim um código pode ser dividido entre tarefas gerenciadas pelo *scheduler*

https://www.zephyrproject.org/

Exemplo de código de sincronização de tasks com semáforos no Zephyr

DIH – análise de código no ESP32 – conceitos base

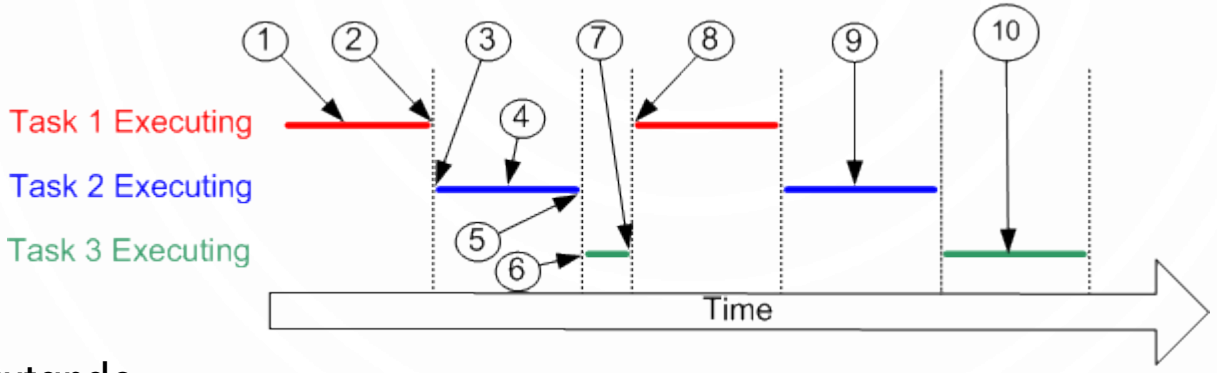
- Na prática, em GPOS (General Purpose OS), cada núcleo só executa uma thread por vez, mas o scheduler chaveia rapidamente entre várias threads, dando 'ilusão' de execução simultânea



- O scheduler no RTOS é projetado para prover um padrão de execução previsível (determinístico). Isto é particularmente útil em sistemas embarcados, que geralmente possuem restrições de tempo real, ou seja, que uma tarefa deva responder estritamente dentro de um período de tempo. Isto só pode ocorrer se o comportamento do scheduler do SO puder ser predito. No freeRTOS isto é controlado pelo programador por meio da definição de PRIORIDADES para as TASKS.
- O freeRTOS é uma classe de RTOS. Disponibiliza primitivas para scheduler, comunicação entre processos (IPC), temporização e sincronismo

DIH – análise de código no ESP32 – conceitos base

- Ideia geral de escalonamento



- (1) task1 executando
- Em (2) kernel suspende task1 e assume task2 em (3)
- Quando task2 está executando (4) bloqueia um periférico para seu próprio uso
- Em (5) kernel suspende task2 e em (6) assume task3
- task3 tenta usar o mesmo periférico que task2, não consegue, e se auto suspende em (7)
- Em (8) task1 assume etc.
- Da próxima vez que task2 executar em (9) concluiu o uso do periférico libera-o
- E agora task3 em (10) acessa o periférico e executa até ser suspensa pelo kernel

DIH – análise de código no ESP32 – conceitos base

- No código executado em setup() é utilizada a função

  `xTaskCreatePinnedToCore()` que é uma especialização de `xTaskCreate()`

Cria nova task e a inclui na lista de tarefas READY_TO_RUN

- RAM automaticamente alocada pelo heap do freeRTOS
- **pvTaskCode**: nome da função que implementa a task
- **pcName**: nome descritivo para a task. Pode ser usado, se necessário, para resgatar o handle da task, com xTaskGetHandle(*pcName*)
- **usStackDepth**: número de WORDS (grupos de 16bits/2bytes) para alocar à pilha da task (task stack). Contudo no ESP-IDF está em bytes mesmo.
- **pvParameters**: parâmetros para passar a task (tal como no pthread_create)
- **uxPriority**: prioridade da task
- pxCreatedTask: handle externo da task. Em geral NULL.

xTaskCreate

[Task Creation]

task. h

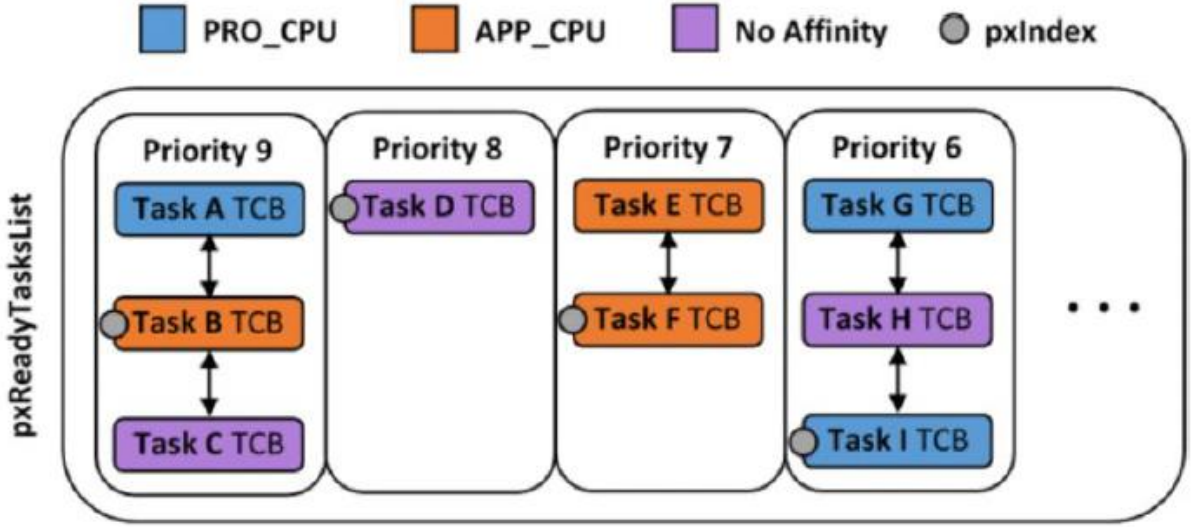
```
BaseType_t xTaskCreate(    TaskFunction_t pvTaskCode,
                           const char * const pcName,
                           configSTACK_DEPTH_TYPE usStackDepth,
                           void *pvParameters,
                           UBaseType_t uxPriority,
                           TaskHandle_t *pxCreatedTask
                       );
```

Sucesso: pdPASS
Falha: errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY

DIH – análise de código no ESP32 – conceitos base

- No código executado em setup() é utilizada a função

  `xTaskCreatePinnedToCore()`



Afinidade por núcleo: introduz parâmetro extra xCoreID, que pode ser:

- 0 (CPU0): Protocol CPU (PRO_CPU)
- 1 (CPU1): Application CPU (APP_CPU)

Obs: em geral comunicações (wireless etc.) no PRO_CPU e o resto da aplicação no APP_CPU (por exemplo setup, loop())

TaskHandle_t xTaskGetHandle(const char *pcNameToQuery);

DIH – análise de código no ESP32 – conceitos base

- Na função t1 é utilizado vTaskDelay(ticks) – exemplo de uso:

```
void vTaskFunction( void * pvParameters ) {
          /* Block for 500ms. */
          const TickType_t xDelay = 500 / portTICK_PERIOD_MS;
          for( ;; ) {
          /* Simply toggle the LED every 500ms, blocking between each toggle. */
          vToggleLED();
          vTaskDelay( xDelay );
          }
}
```

Mas para processos real time com necessidade precisão temporal, indicado vTaskDelayUntil() vTaskDelay() dependerá do momento em que foi chamada em relação ao tick Count. No código em questão, é usado pdMS_TO_TICKS para obter o número de ticks associado ao tempo em milissegundos.

```
vTaskDelay(pdMS_TO_TICKS(400));
```