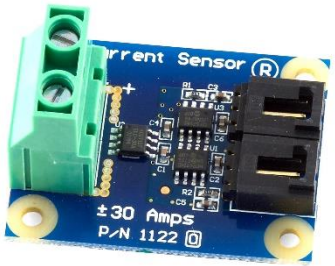
SENAI

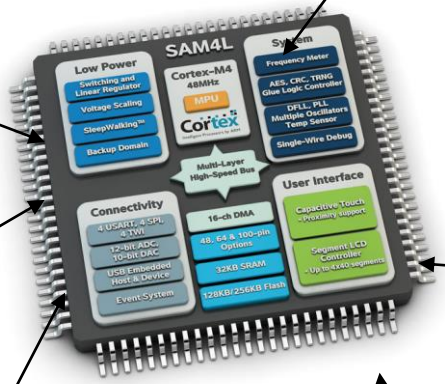
Disciplina

Sistemas Operacionais em Tempo Real

Prof° Fernando Simplicio

PROJETO com MCU



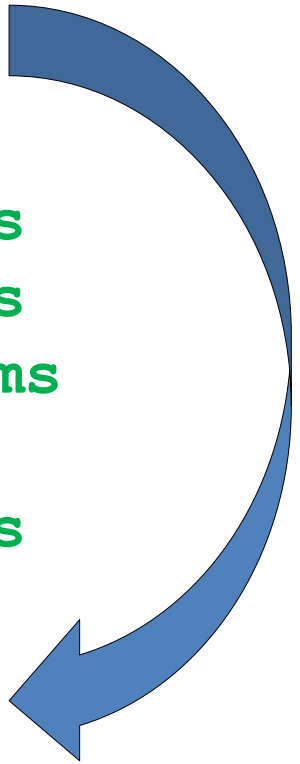


PROGRAMA em C

```
void main()
{
        InitSys();
        while(true)  {
          SensorCorrente();
          Termopar()
          Umidade();
          SensorPresenca();
          SendToUart();
          UpdateLcd();
    }
}
```

PROGRAMA em C

```
void main()
{
        InitSys();
        while(true) {
           SensorCorrente();      //2ms
           Termopar()             //10ms
           Umidade();             //15ms
           SensorPresenca();      //100ms
           SendToUart();          //1ms
           UpdateLcd();           //10ms
    }
}     //Tempo Gasto por loop: 138ms
```



Máquina de Estado

- **Break long tasks into a state machine**

```
while (1)
{
  Task_1(); //60 us
  Task_2(); //2 us
}
//max loop time = 62 us
```

```
while (1)
{
  switch(Task_1_state)
  {
   case a:
     Task_1_state_a(); //20 us
     break;
   case b:
     Task_1_state_b(); //20 us
     break;
   case c:
     Task_1_state_c(); //20 us
     break;
  }
  Task_2();  //2 us
} //max loop time = 22 us
```

Multitasking

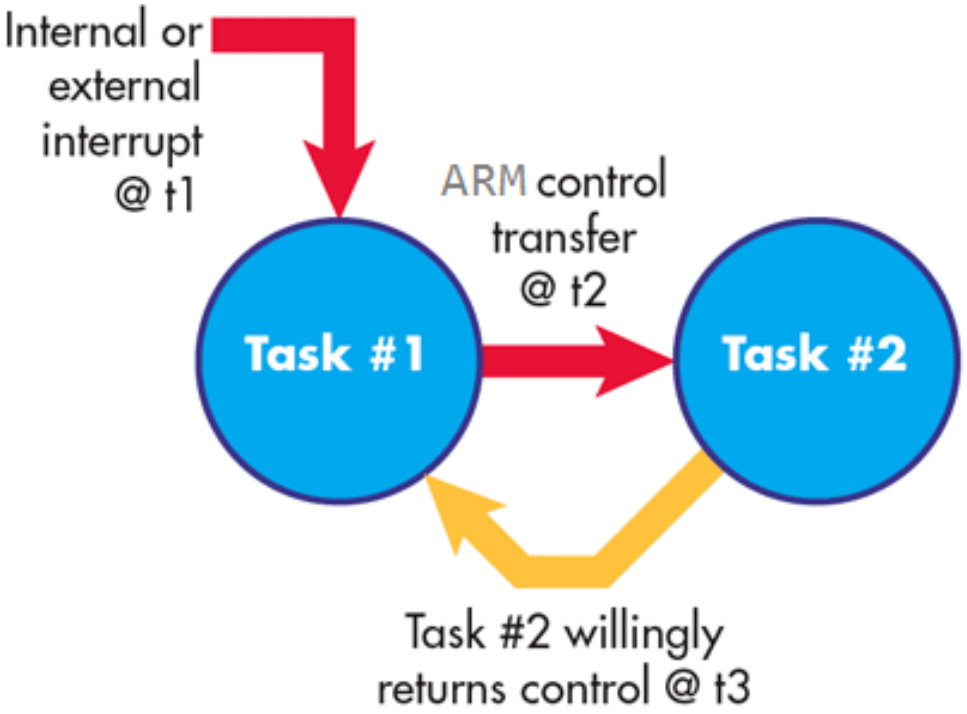
- **Quais os problemas de um programa Multitasking?**

Em multitask devemos considerar (...)

- **Tempo para troca de contexto.**
- **Consumo de memória para armazenamento do contexto.**
- **Prioridades as *tasks* a serem executadas.**
- ***Tasks* podem ser tolerantes a *loops* infinitos.**
- **Não podem conflitar com outros recursos.**
- **Permitir o compartilhamento de recursos (registradores internos, memória, unidades lógicas aritméticas da CPU, periféricos (LCD, Sensores e atuadores)) entre as outras *tasks* concorrentes.**
- **Devem suportar funções reentrantes.**

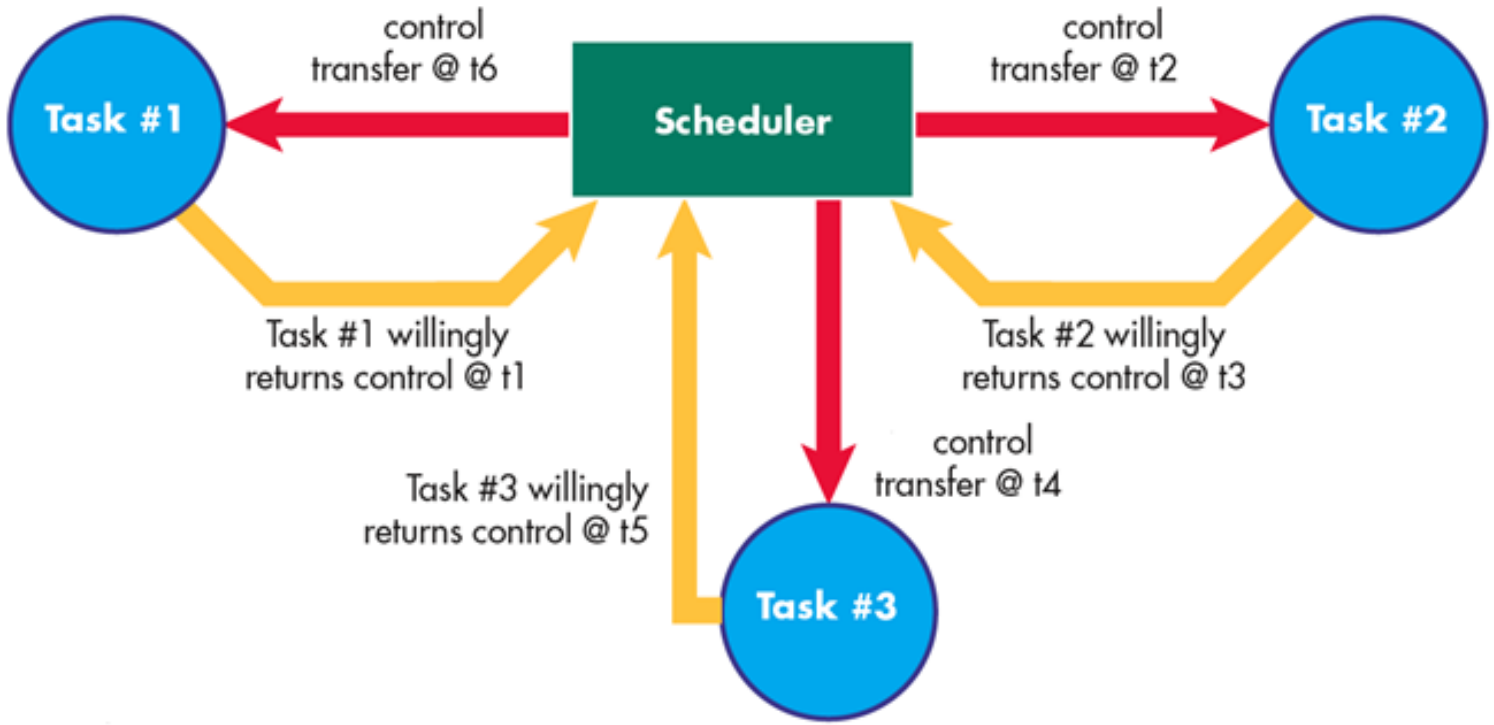
Exemplo de programa Multitask

- **Sistema não preemptivo.**



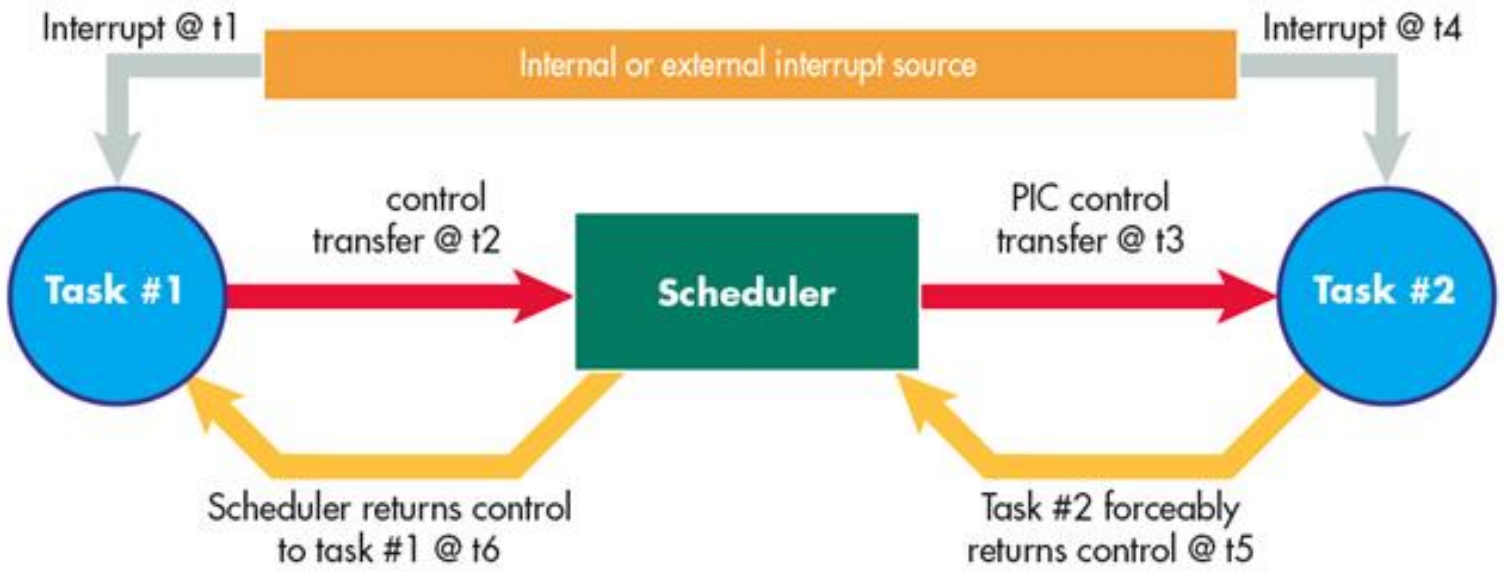
multitasking policy

- **As regras de chaveamento das tasks é determinada pelo *Scheduler*.**



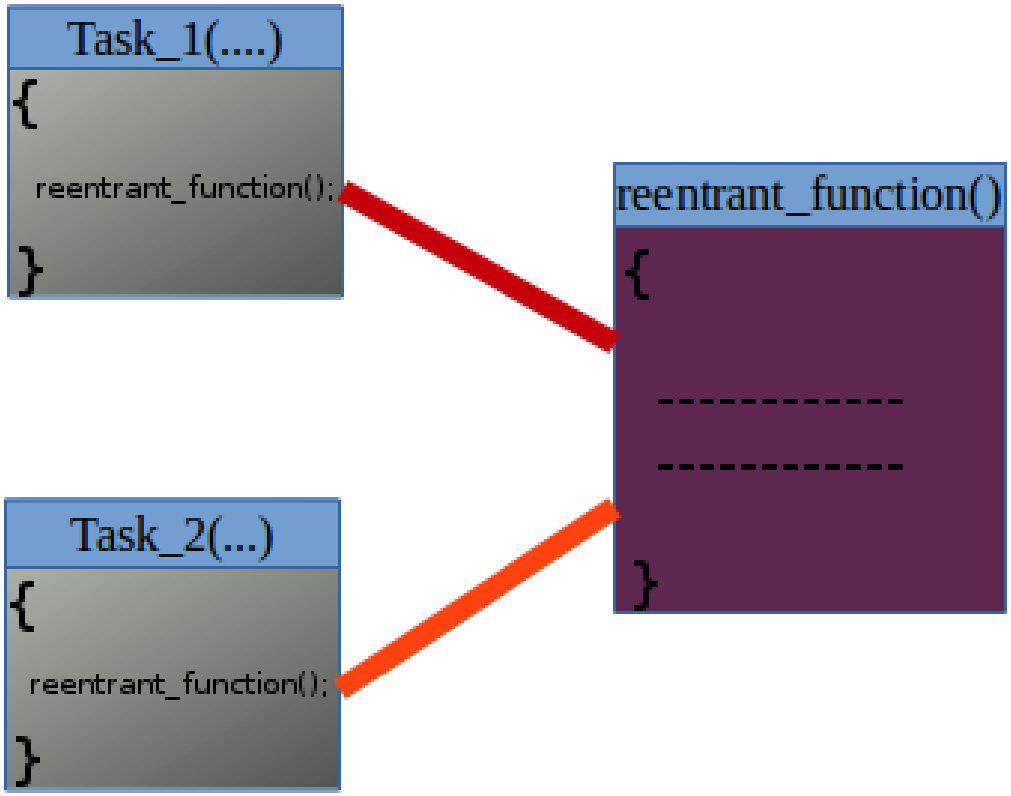
multitasking preemptive

- **É quando uma *task* em execução é interrompida em algum ponto por um evento interno ou externo do sistema.**



multitasking

- **Quais os problemas que podem ocorrer em programas *multitasking?***



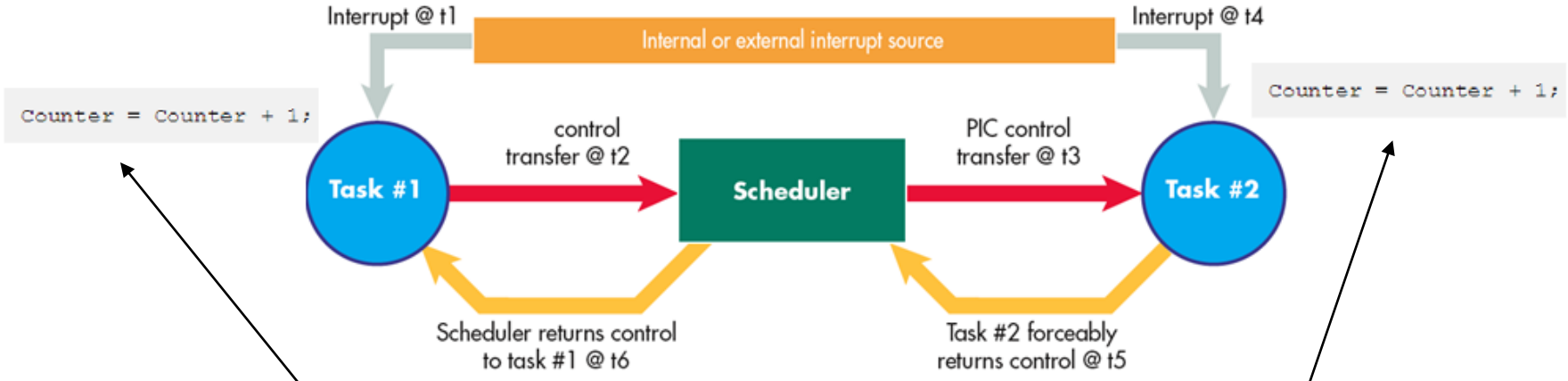
Reentrância ("re-enter")

- **Denota os problemas que podem ocorrer quando o mesmo código é executado simultaneamente por várias tarefas ou quando os dados globais são acessados simultaneamente por várias tarefas.**

- **Em um ambiente *multitasking*, deve-se ter cuidado para que tanto código quanto possível seja reentrante para que ele possa ser usado por várias tarefas simultaneamente.**

Exemplo: Reentrância ("re-enter")

- **Por uma questão de simplicidade, suponha que ambas as tarefas tenham a mesma prioridade, o agendamento preventivo e o** *time slicing* **está ativo.**




```
Counter = Counter + 1;
```

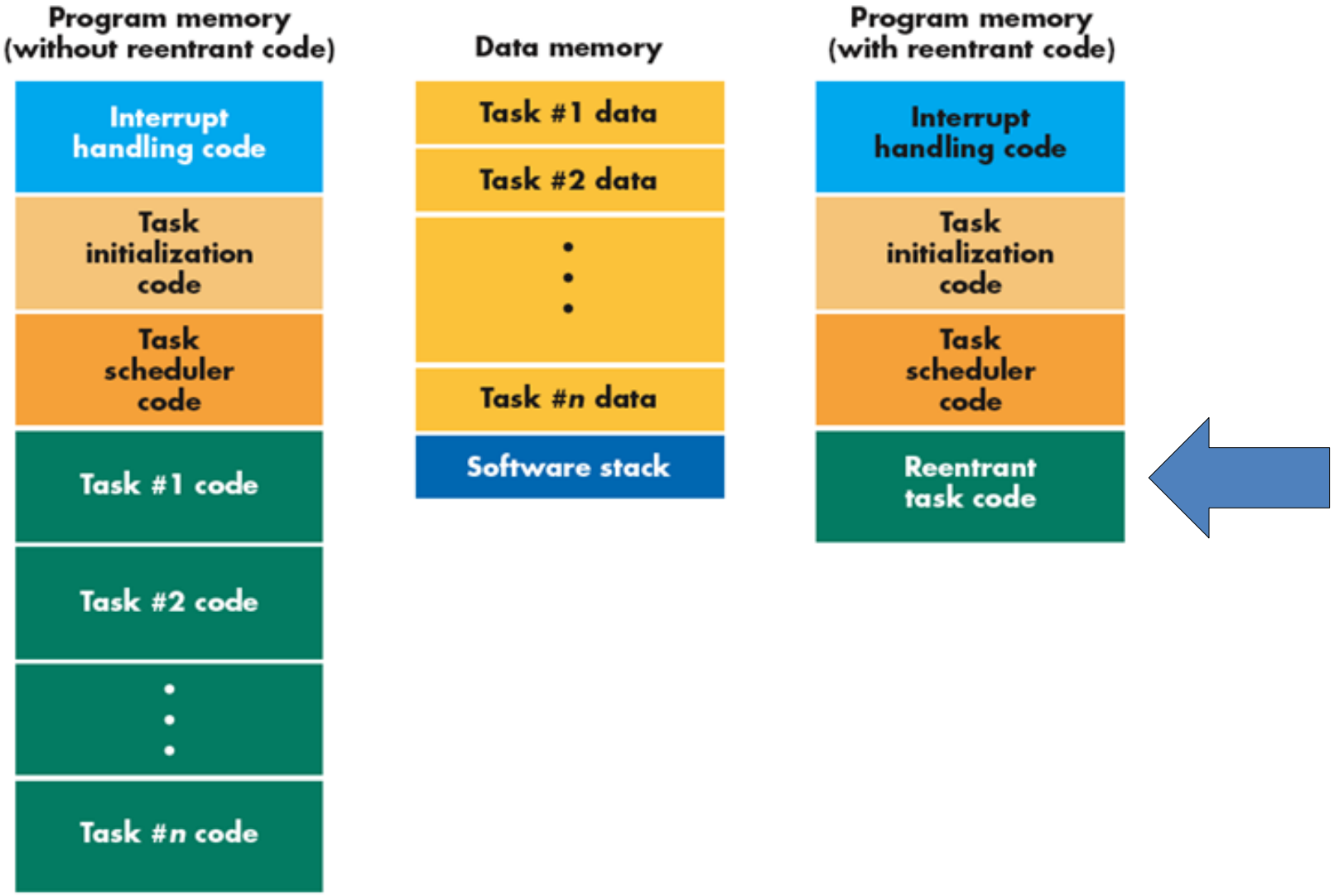
```
MOV   EAX, [Counter]     ; line 1
ADD   EAX, 1             ; line 2
MOV   [Counter], EAX     ; line 3
```

Exemplo: Reentrância ("re-enter")

- **Este exemplo mostra que duas ou mais tarefas nunca devem acessar dados globais simultaneamente se pelo menos uma das tarefas puder modificar os dados.**

- **Como consequência, os dados globais compartilhados devem ser evitados. Se for impossível, o acesso a dados globais deve ser protegido por semáforos.**

Reentrância



Reentrância (Portanto...)

- **Uma função é reentrante somente se em cada chamada função for usado seus próprios dados.**
- **Normalmente causada por variáveis globais.**
- **Muitas funções escritas em C são não reentrantes.**
- **Um código executado por muitas tarefas é conhecida como código de função compartilhado e deve obedecer a propriedade reentrante.**
- **Uma função reentrante deve manipular dados: não *static*, não globais e não constantes.**
- **Não deve retornar endereços para dados estáticos (ou globais), não constantes.**
- **Não deve chamar funções não reentrantes.**

Benefícios de um prog. *Multitasking*

- **Aplicações complexas podem ser divididas em um conjunto de tarefas menores e gerenciáveis.**
- **Permite testar mais facilmente cada tarefa do programa.**
- **Temporizações e sequenciamento do código podem ser removidos do programa e delegados ao SO.**

Benefícios de um prog. *Multitasking*

- **Um processador convencional só pode executar uma única tarefa por vez. Porém um sistema operacional *multitasking* pode fazer aparecer como se cada tarefa estivesse sendo executada simultaneamente.**

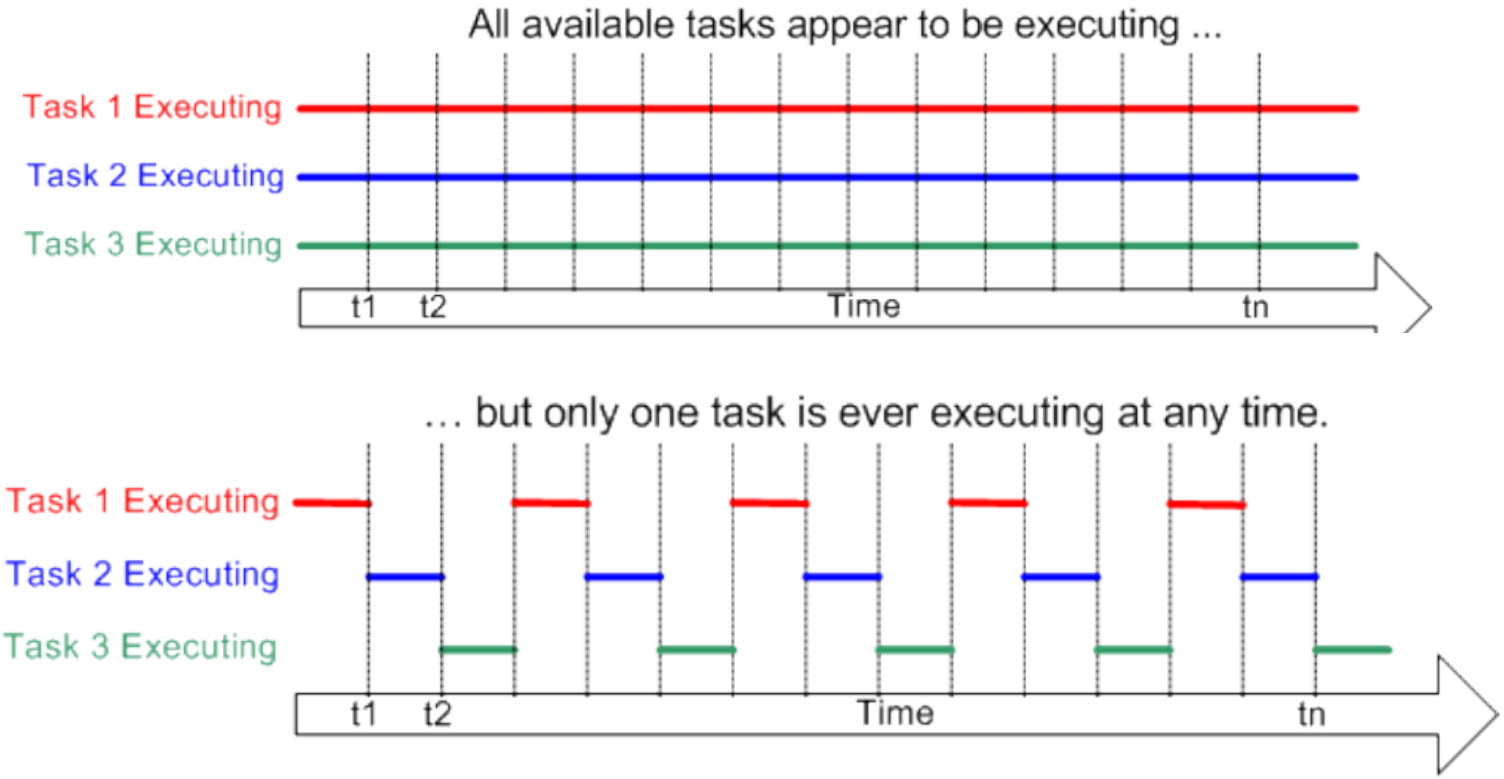
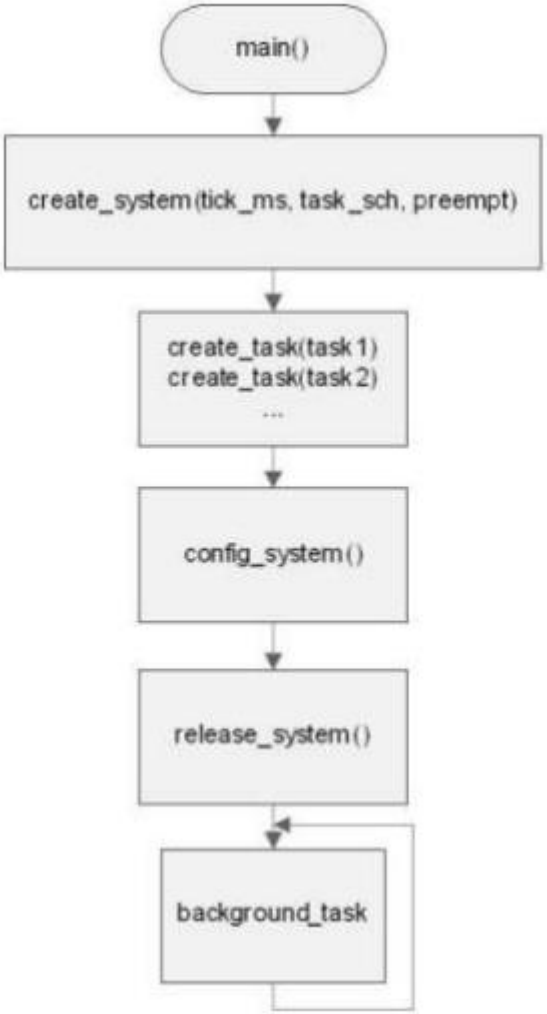


Diagrama de um prog. Multitask



Análise de um programa *Multitasking em C*

Do Baremetal ao RTOS

```c
#define TIMEOUT_50MS 50

void task_led(void)
{
    GPIOB_PTOR|= (1 << 19);
}


int main(void)
{
    task_init();

    add_task(&task_led, TIMEOUT_50MS);

    for (;;) {

        task_run();
    }

    return 0;
}
```

Do Baremetal ao RTOS

```
define TIMEOUT_50MS 50

oid task_led(void)

   GPIOB_PTOR|= (1 << 19);


nt main(void)

   task_init();

   add_task(&task_led, TIMEOUT_50MS);

   for (;;) {

       task_run();

   }

   return 0;
```

```
void SysTick_Handler (void) {
    systimer_tick();
}

/**
 * Inicialização da biblioteca.
 * Deve ser chamado antes de qualquer rotina da bib
 */
void task_init(void)
{
   /*
    * Limpa a lista de tarefas.
    */
   _tasks = NULL;
   ticks = 0;

   SysTick_Config(SystemCoreClock/1000);       /* Ge

}
```

Do Baremetal ao RTOS

```
define TIMEOUT_50MS 50

oid task_led(void)

   GPIOB_PTOR|= (1 << 19);


nt main(void)

   task_init();

   add_task(&task_led, TIMEOUT_50MS);

   for (;;) {

       task_run();

   }

   return 0;
```

```
bool_t add_task  (uint32_t f,  uint16_t p,  uint16_t t)
{
   task_t *novo;

   if(f == 0) return FALSE;

   /*
    * Aloca espaco para uma nova tarefa.
    */
   novo = malloc(sizeof(task_t));
   if(novo == NULL) return FALSE;

   /*
    * Informacões iniciais da tarefa.
    */
   novo->func = f;
   novo->par = p;
   novo->tempo = t;

   /*
    * Adiciona à lista de tarefas para execução.
    */
   disable();
   list_add(&_tasks, novo);
   enable();

   return TRUE;
}
```

Do Baremetal ao RTOS

```
define TIMEOUT_50MS 50

oid task_led(void)

   GPIOB_PTOR|= (1 << 19);


nt main(void)

   task_init();

   add_task(&task_led, TIMEOUT_50MS);

   for (;;) {

       task_run();

   }

   return 0;
```

```
void task_run (void)
{
   register task_t *task;
   static void (*fn)(uint16_t);

   disable();

   /*
    * Varre a lista de tarefas.
    */
tenta:
   list_for_each(_tasks, task) {
      if(task->tempo == 0) {
         /*
          * Tarefa pronta.
          * Chama rotina definida.
          */
         fn = task->func;
         enable();
         fn(task->par);
         disable();

         /*
          * Remove a tarefa da fila.
          */
         free((void*)task);
         list_remove(&_tasks, task);
         goto tenta;
      }
   }

   enable();
}
```

Do Baremetal ao RTOS

```
bool_t cancel_task (uint32_t f)
{
    register task_t *p;
    bool_t ok;

    if(f == 0) return FALSE;

    ok = FALSE;
    disable();

    /*
     * Procura na lista de tarefas.
     */
tenta:
    list_for_each(_tasks, p) {
        if(p->func == f) {
            /*
             * Tarefa encontrada: remove.
             */
            list_remove(&_tasks, p);
            free((void*)p);
            ok = TRUE;
            goto tenta;
        }
    }

    enable();
    return ok;
}
```

Do Baremetal ao RTOS

```
void task_led(void)
{
    GPIOB_PTOR|= (1 << 18);
    add_task(&task_led, TIMEOUT_50MS);
}
void led_init(void)
{
    SIM_SCGC5 |= SIM_SCGC5_PORTB_MASK;
    PORTB_PCR18 = PORT_PCR_MUX(1) | PORT_PCR_DSE_MASK | PORT_PCR_PE_MASK   | PORT_PCR_PS_MASK;
    //Configura como saída.
    GPIOB_PDDR |= (1<<18);
    //Seta o pino;
    GPIOB_PSOR = (1<<18);
}

int main(void)
{
    task_init();
    led_init();
    add_task(&task_led,TIMEOUT_50MS);
    for (;;)
    {
        task_run();
    }
    return 0;
}
```

Do Baremetal ao RTOS

```
static volatile void (*_call)(uint16_t) = NULL;
void callback(void (*f)(uint16_t))
{
   _call = f;
}

void task_button(void)
{
  //...
}

void task_led(void)
{
    GPIOB_PTOR|= (1 << 18);
    /*dispara função de callback previamente assinada: Ex: callback(&task_button);*/
    /*
      if(event_trigger == TRUE)
         add_task(&_call, TIMEOUT_0MS);

      if(event_trigger == FALSE)
        add_task(&task_button,TIMEOUT_0MS);
    */
    add_task(&task_led, TIMEOUT_50MS);
}
```

SENAI

Exercício Prático:

Sistema Multitask com ARM Cortex M0+

Prof° Fernando Simplicio

SENAI

Exercício Prático: (call-back)

Sistema Multitask com ARM Cortex M0+

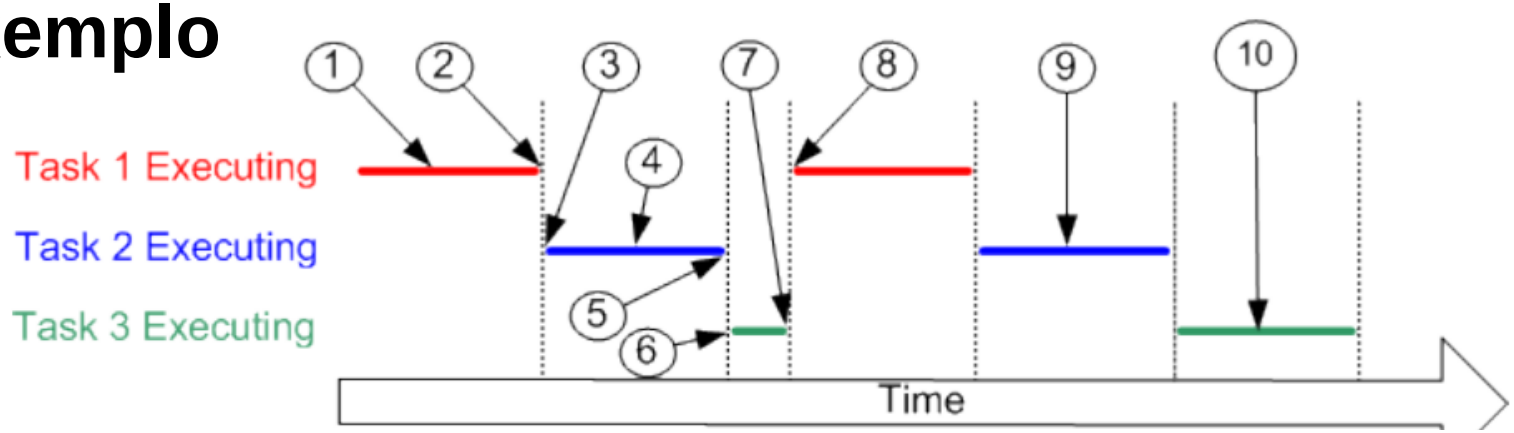
Prof° Fernando Simplicio

Scheduler

- O ***Scheduler*** é a parte do kernel responsável por decidir qual tarefa deve ser executada na unidade de tempo.
- O kernel pode suspender e depois retomar uma tarefa muitas vezes durante a vida útil da tarefa.

Scheduler

- **Exemplo**



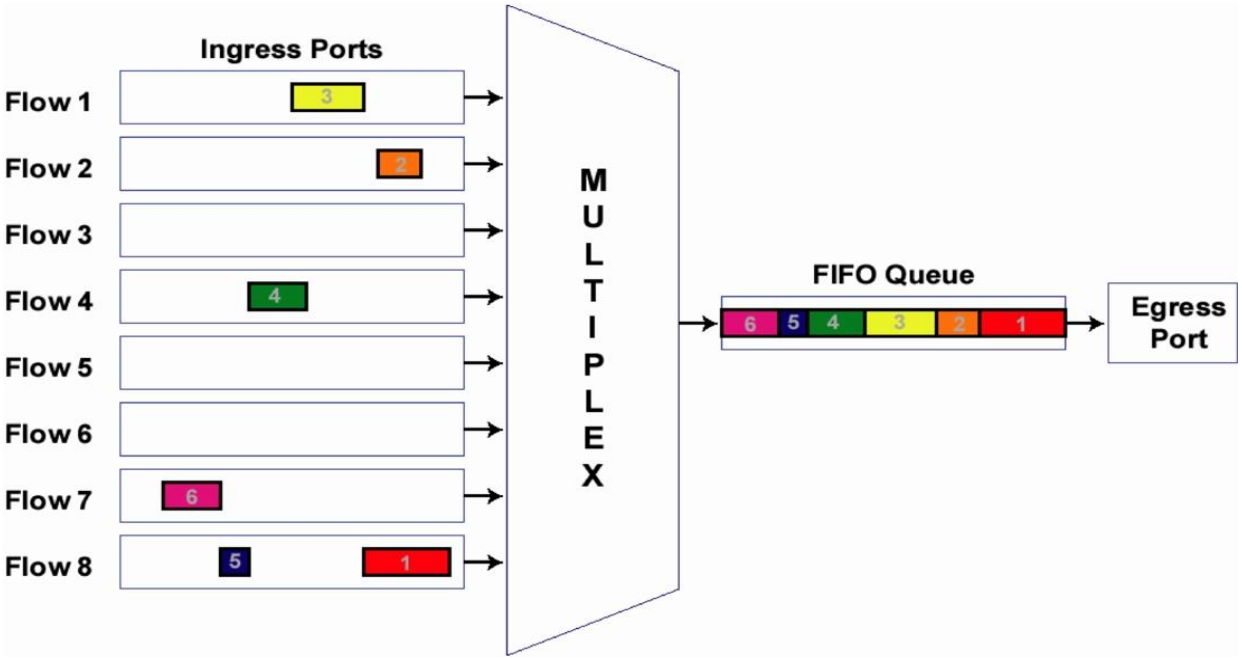
Referring to the numbers in the diagram above:

- At (1) task 1 is executing.
- At (2) the kernel suspends task 1 ...
- ... and at (3) resumes task 2.
- While task 2 is executing (4), it locks a processor peripheral for its own exclusive access.
- At (5) the kernel suspends task 2 ...
- ... and at (6) resumes task 3.
- Task 3 tries to access the same processor peripheral, finding it locked task 3 cannot continue so suspends itself at (7).
- At (8) the kernel resumes task 1.
- Etc.
- The next time task 2 is executing (9) it finishes with the processor peripheral and unlocks it.
- The next time task 3 is executing (10) it finds it can now access the processor peripheral and this time executes until suspended by the kernel.

http://www.freertos.org/implementation/a00005.html

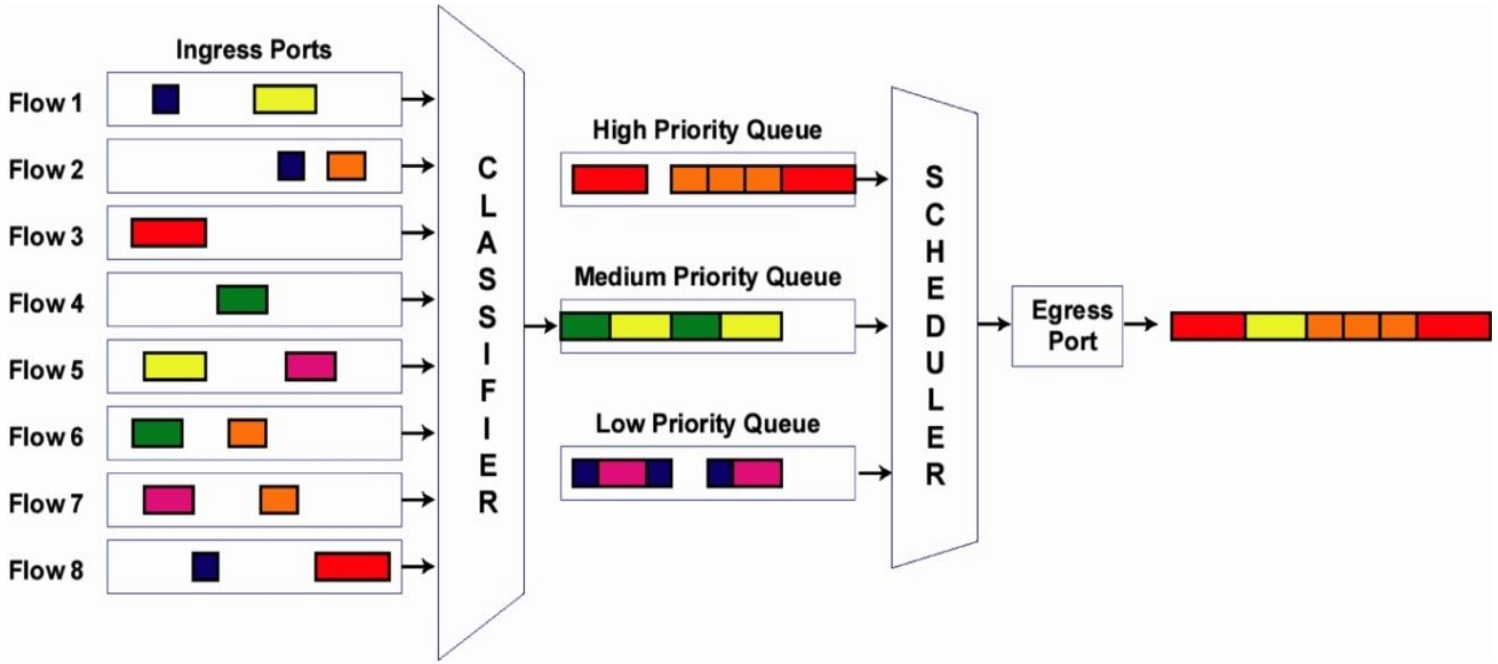
Filas e Scheduler

- **Em um sistema de fila única, todos os pacotes são colocados no *link* de saída obedecendo a uma ordem de distribuição definida no *Schedule* (buffer FIFO de saída).**



Filas e Scheduler (c/ Prioridade)

- **Frequentemente o SO diferenciam as tasks de acordo com suas prioridades, e para isso implementam um sistema de prioridades nas regras do *scheduler*.**



Scheduler (Prioridade)

Um sistema de filas c/ prioridades é composto por:

- **FIFO(s) separados em classes/prioridades.**
- **Pacotes de prioridade inferior iniciam a transmissão somente se nenhum pacote de prioridade mais alta estiver aguardando.**

Scheduler (tipos básicos)

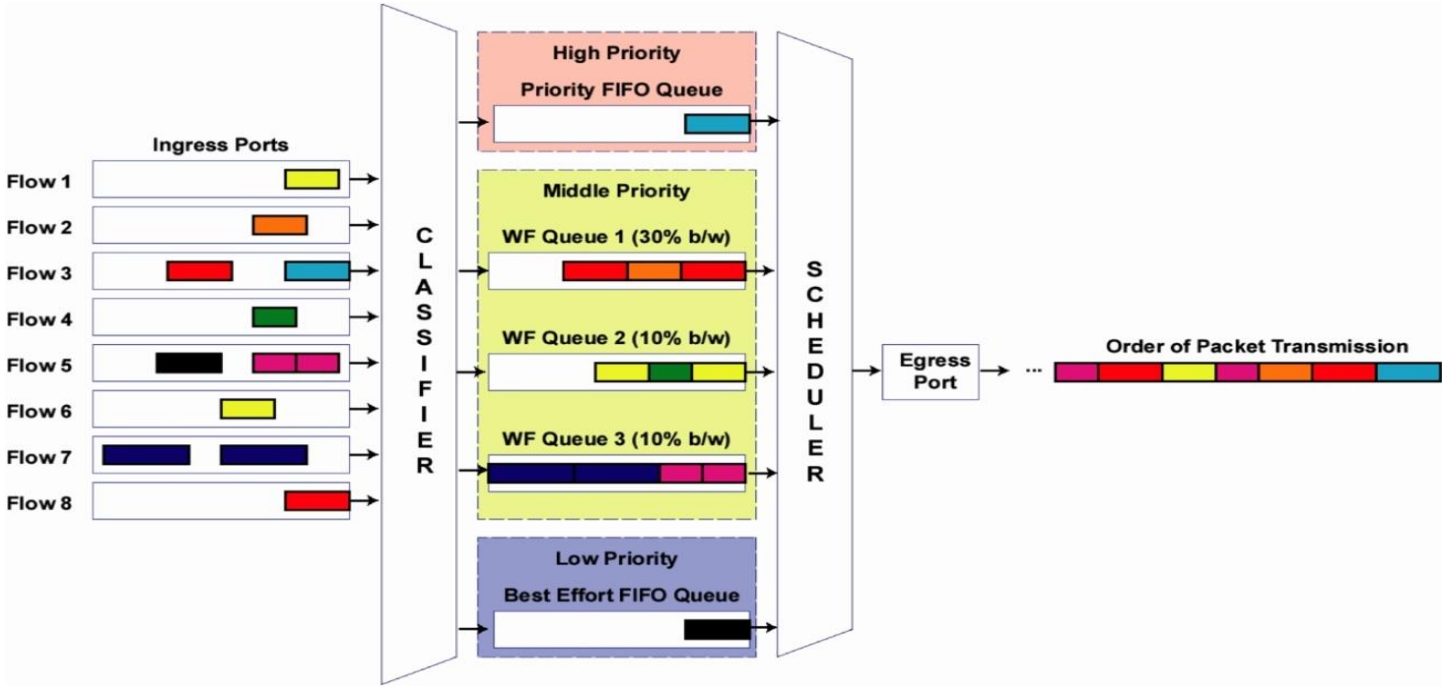
Um sistema de filas de prioridades é composto por:

- **Não preemptiva.** **Quando um pacote de alta prioridade aguardar o término da transmissão de um pacote de baixa prioridade.**

- **Preemptiva.** **Quando um pacote de alta prioridade não precisar esperar, ou seja, mesmo que um pacote de baixa prioridade estiver em processo de transmissão, este será interrompido e colocado na condição de espera, enquanto o pacote de alta prioridade é tratado.**

Filas + Scheduler (tipos básicos)

- **Alguns algorítmos para *scheduler* mais complexos utilizam a combinação de vários esquemas de filas (Exemplo: Sistema LLQ (*Low Latency Queueing with Priority Percentage Support).***



TRABALHO

Trabalho para Entregar.

Pontuação: 2.

Trabalho individual

Entregue em formato PDF (impresso) e códigos no formato digital (*.zip).

- **1° Explique as diferenças entre funções *Thread-Safe* e funções *Reentrancy*.**
- **2° Desenvolva programas em C com funções *Thread-Safe* e Reentrantes. *Explique os problemas de cada função e apresente uma possível solução para o problema.***

SENAI

Exercício Prático:

Integração do FreeRTOS no Kinetis Studio

Prof° Fernando Simplicio