Tópico 11

Programação Concorrente

Autor: Wu Shin-Ting

DCA - FEEC - Unicamp

Outubro de 2019

Diferentemente dos microprocessadores de uso genérico, vimos no Capítulo 10 que os microcontroladores utilizam intensamente as suas portas de entrada e de saída para interagir com os outros componentes no sistema em que ele está embutido. Uma das características marcantes dos microcontroladores modernos é a existência de circuitos dedicados para processar os sinais de natureza distinta, como os sinais analógicos (Capítulo 8) e os sinais digitais em conformidade com algum protocolo de comunicação (Capítulo 9). Embora esses circuitos dedicados operam **em paralelo** e geram sinais no formato apropriado para o processador, é necessário ainda integrar estes sinais como dados num programa executado no processador. Uma integração perfeita em que os dados do mundo físico estejam sempre

disponíveis quando são requisitados por uma instrução do processador ou em que os dispositivos físicos estejam sempre prontos para serem atuados por uma instrução de um processador é um grande desafio.

Apresentamos na Seção 7.12 duas técnicas mais conhecidas para “sincronizar” a transferência de dados entre os periféricos e um processador: ***polling*** e **interrupção**. ***Polling*** é um procedimento de monitoramento simples, assumido totalmente por um processador. A CPU verifica periodicamente (1) a disponibilidade dos dados necessários para completar a execução de uma instrução, como comparar a temperatura do ambiente com um limiar, e/ou (2) as requisições do mundo físico a um processamento específico, como ligar uma lâmpada com o acionamento de uma botoeira. **Interrupção**, por sua vez, é um procedimento que requer a cooperação do processador e dos circuitos dedicados (Seção 10.4). Ao invés de fazer monitoramentos periódicos, o processador só dá atenção para os circuitos periféricos de entrada/saída e temporizadores quando interrompido. Figura 11.1 ilustra a diferença entre o mecanismo de interrupção e o de *polling* nas interações entre a CPU e os periféricos. No caso é um circuito responsável pelos sinais digitais dos pinos de uma porta GPIO (*General Purpose Input Output*).

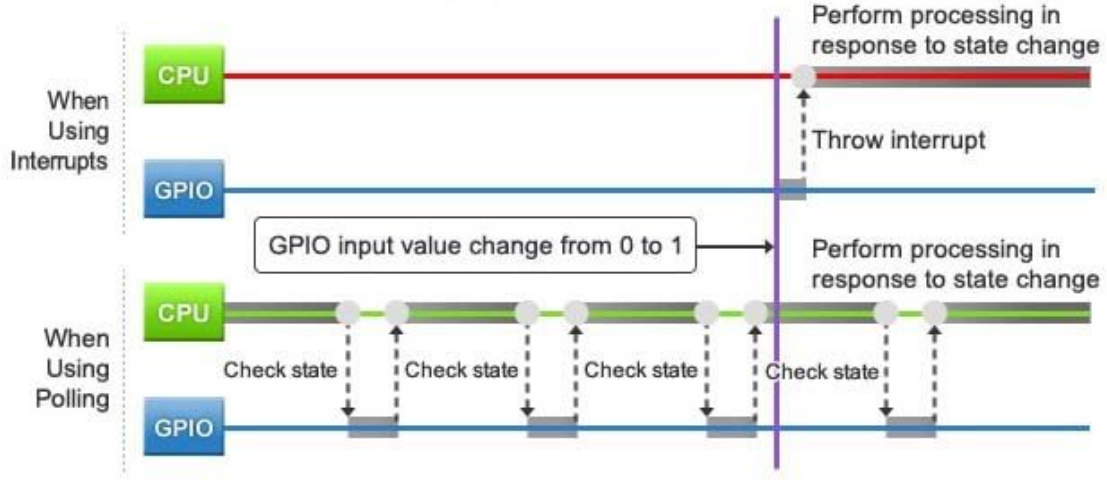


Figura 11.1: *Polling* e Interrupção (Fonte: [2]).

Vimos na Seção 10.4 que o mecanismo de interrupção é uma solução eficiente para compatibilizar um microcontrolador com o seu mundo externo. Ela provê uma forma eficiente tanto para capturar as variações nos estados dos periféricos quanto para respondê-las, sem ocupar a CPU com verificações desnecessárias. Note na Figura 11.1 que a fração de tempo *p* na linha azul (GPIO) diminuiu quando passou de *polling* para a interrupção. Por outro lado, a interrupção torna propensa a geração simultânea de eventos por circuitos dedicados distintos, que precisam ser tratados por um mesmo conjunto de recursos. Neste contexto, é interessante distinguirmos dois conceitos em termos de programação: paralelismo e concorrência. A execução de um programa é **paralela**, se as suas instruções são

executadas em paralelo **fisicamente**. Enquanto a execução de um programa é **concorrente**, se partes do programa são executadas **conceitualmente** em paralelo, competindo o uso de um mesmo conjunto de recursos físicos [1].

Já falamos sobre controladores de interrupção na Seção 4.5 no contexto de multiprogramação em que uma CPU pode ser compartilhada por vários processos alocando a cada processo uma fatia de tempo pré-especificada e na Seção 10.4 em que um mesmo processador pode interromper várias vezes o seu fluxo de execução corrente para executar trechos de códigos relacionados aos periféricos. Embora não sejam equivalentes os processamentos de interrupção num sistema operacional e numa interface com os periféricos, eles compartilham mesmos problemas de disputa e trava não-intencional. Ao invés de chavear entre os processos, o mecanismo de interrupção no contexto de atendimento dos periféricos consiste no chaveamento entre os trechos de instruções de um mesmo processo. Ao invés de alocar a cada processo uma fração de tempo da CPU, só é chaveado de um trecho de instruções para o outro na ocorrência de um evento, dividindo o fluxo de controle original em dois sub-fluxos de controle (um antes e outro depois do evento) com o(s) código(s) de serviço intercalado(s) entre eles. Observe que o fluxo só retorna para trechos de menor prioridade quando concluir a execução dos trechos de instruções de maior prioridade.

Figura 11.2 ilustra um sistema computacional em que os periféricos de entrada e de saída podem ser controlados separada e **paralelamente** por módulos dedicados, como conversores AD, DA, interfaces de comunicação serial, representados pelos planos "*Peripheral*". Quando o processador, "*Processor Core*", precisa de dados para a execução das suas instruções, ele os solicita aos respectivos circuitos dedicados. Assim que os circuitos estiverem com os dados requisitados pelo *core*, eles geram assíncronamente os eventos para **concorrerem** o atendimento do processador "*Processor Core*". O controlador de interrupções, "NVIC", é responsável pelo gerenciamento desses eventos "solicitantes" e pela decisão da prioridade de atendimento desses eventos, como vimos na Seção 10.4.1. Para que as interações de um circuito sequencial com o mundo físico sejam bem "casadas", **o tempo de resposta aos eventos deve ser compatível com os diferentes fenômenos físicos que podem ocorrer paralelamente**. Por exemplo, o tempo de resposta a um pressionamento de uma tecla em um teclado por um usuário deve ser consistente com a resposta humana e o tempo de resposta à detecção de um curto-circuito por um sensor deve ser consistente com a velocidade de alastramento do efeito devastador de um incêndio.

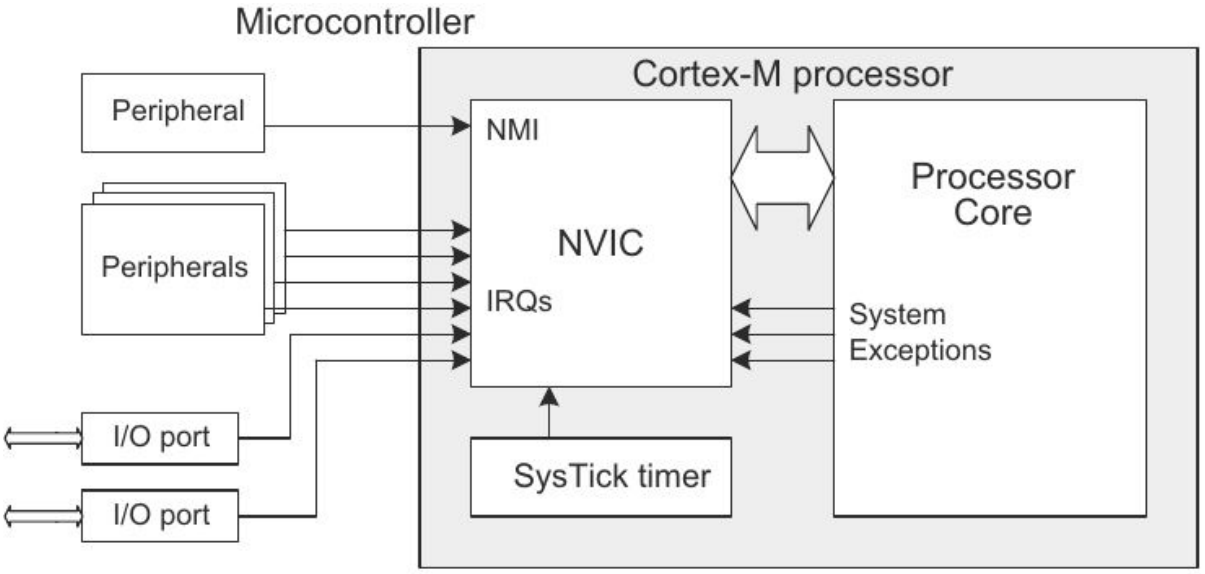


Figura 11.2: Organização de um sistema computacional com interrupções.

Neste capítulo vamos apresentar as diferentes técnicas de exclusão mútuas dos trechos de instruções concorrentes que possam amenizar possíveis condições de disputa e de trava involuntária ao longo da execução dos serviços de tratamento de eventos gerados pelos periféricos de um sistema microcontrolado.

11.1 Processos e *Threads*

Processos e *threads* são dois conceitos básicos de um sistema operacional [3]. Embora discussões sobre sistemas operacionais fogem do escopo desta disciplina, decidimos fazer uma breve introdução a estes dois conceitos, já mencionados na Seção 4.5, antes de apresentarmos algumas técnicas básicas de programação concorrente elaboradas em cima destes dois conceitos. Algumas dessas técnicas de programação são hoje amplamente aplicadas em sistemas embarcados.

Um processo é uma abstração de um programa em execução, junto com os valores atuais do contador de programa, dos registradores e das variáveis. **Um mesmo programa pode ser executado em diversos processos**, acessando ou não um mesmo espaço de memória. Conceitualmente, cada processo tem a sua própria CPU e o seu próprio espaço de memória. Cabe ao sistema operacional escalonar os processos em execução, atribuindo ciclicamente uma fatia de tempo da CPU a cada processo. Esta estratégia de compartilhamento do tempo da CPU é conhecida por **tempo compartilhado**, em inglês *time sharing* (Figura 11.3). Quando se carrega um sistema operacional, vários processos são criados simultaneamente para gerenciar os recursos computacionais. Alguns processos são denominados em primeiro plano, em inglês *foreground*, e outros em segundo plano, em inglês *background*. Os processos em *foreground* são tipicamente processos que requerem interações com

usuários como um editor de texto, enquanto os processos em *background* são aqueles que realizam alguma atividade específica, como gerenciamento de mensagens eletrônicas. Tradicionalmente, esses processos em segundo plano são denominados ***daemon*** e os seus nomes terminam com a letra d. Após a inicialização, novos processos podem ser criados, executados e terminados. Para gerenciar os processos, o sistema operacional mantém uma **tabela de processos** contendo todas as informações relevantes para a sua execução.

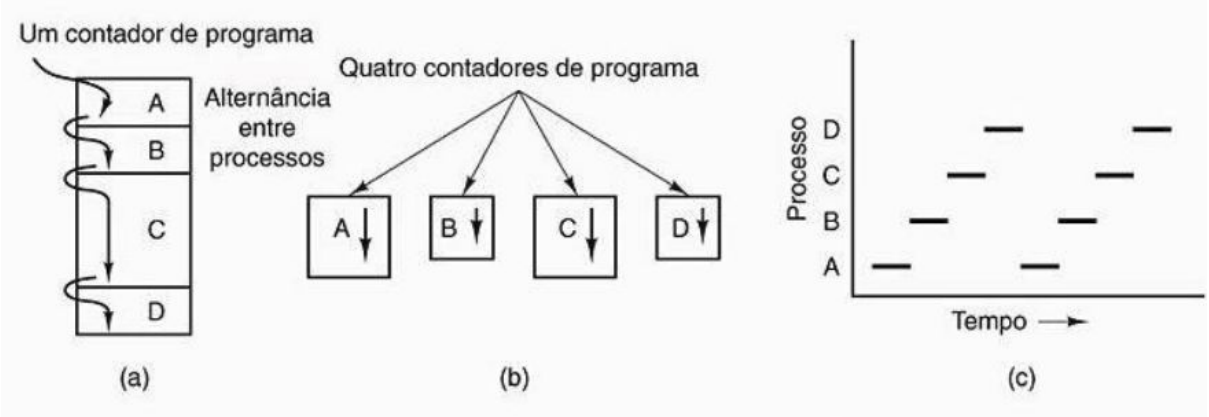


Figura 11.3: Tempo compartilhado no uso da CPU.

O multiprocessamento numa única CPU pode aumentar a ocupação da CPU quando se trata de vários processos envolvendo acessos aos dados do mundo externo. Supondo que a fração de tempo de espera seja *p* para cada processo, a probabilidade da ocupação da CPU com *n* processos aumenta:

$$ocupação\ CPU = 1 - p^n.$$

Da mesma forma como existem microinstruções dentro de uma instrução, como vimos nas Seções 4.1.1 e 4.2.2, distingue-se também mini-processos num processo. Os mini-processos são programados como atividades distintas que se cooperam na realização de uma tarefa. Denominamos estes mini-processos de ***threads*** (de execução). Diferentes dos processos, os *threads* de um mesmo processo compartilham o mesmo espaço de endereçamento[1], como ilustra o espaço comum em verde na Figura 11.4. Somente o contador de programa, registradores, a pilha e o registrador de estado são individualizados para cada *thread*. Quando há mais de um *thread* num mesmo processo, dizemos que o processo é ***multithread***. como mostra a imagem direita da Figura 11.4. Neste caso, os *threads* podem ser executados simultaneamente num multiprocessador ou em tempo compartilhado num mono-processador.

[1] Em alguns sistemas operacionais, processos que trabalham juntos podem também compartilhar o uso de recursos [3].

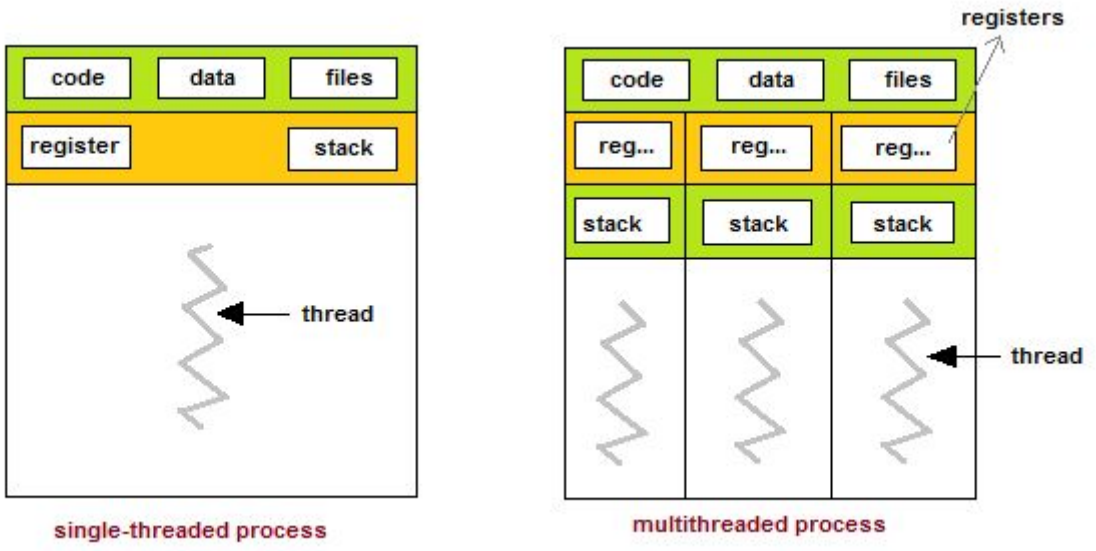


Figura 11.4: *Threads* num processo (Fonte: [4]).

11.1.1 Regiões Críticas

Como os *threads* de um mesmo processo compartilham o mesmo conjunto de recursos, tais como a memória principal e os arquivos (Figura 11.4), eles podem acessar concorrentemente uma mesma posição da memória de forma que a ordem de acessos possa gerar resultados finais diferentes. Situações como essas são chamadas **condições de disputa**, em inglês *race condition*. A depuração de programas que contenham condições de disputa não é nada trivial, pois os resultados da maioria dos testes são usualmente corretos [3]. Os trechos de instruções de um *thread* que possa gerar disputas a recursos compartilhados, como uma variável ou um arquivo, são chamados de **região crítica**, em inglês *critical region*, ou **seção crítica**, em inglês *critical section*.

Vamos ilustrar com um exemplo concreto adaptado de [11]. Dada a seguinte rotina de desconto de um cheque num banco:

```
doCheck (payer, recipient, amount) {
   if (amount < 0) return DIAF;
   balance = getBalance( payer );
   if (balance < amount) return NSF;
   setBalance( recipient, getBalance(recipient) + amount );
   setBalance( payer, balance - amount );
}
```

Imagine que um cliente C tenha na sua conta R$ 11.000,00 e emitiu dois cheques no valor de R$ 10,00 para A e R$ 10.000,00 para B. Vamos agora considerar que um analista do sistema do banco decidiu paralelizar o processamento dos dois cheques por 2 *threads* para melhorar o desempenho. Ao iniciar o processamento do primeiro cheque pelo *thread* 1, foram executadas as seguintes instruções:

```
if (amount < 0) return DIAF;                    // (R$ 10,00 < 0)?  OK
balance = getBalance(C);                        // R$ 11.000,00
if (balance < amount) return NSF;               // (11.000,00 < 10,00?)  OK
setBalance(A, getBalance(A) + amount); // A recebeu o crédito
```

e o sistema operacional chaveou para o *thread* 2 e começou processar o segundo cheque:

```
if (amount < 0) return DIAF;                    // (R$ 10.000,00 < 0)?  OK
balance = getBalance(C);                        // R$ 11.000,00
if (balance < amount) return NSF;               // (11.000,00 < 10.000,00?)  OK
setBalance(B, getBalance(B) + amount); // B recebeu o crédito
 setBalance(A, 11.000,00 - 10.000,00 );    //A conta foi atualizada para 1.000,00
```

Neste ponto, o sistema operacional chaveou de volta para o *thread* 1 para finalizar o processamento do primeiro cheque interrompido usando o valor que estava armazenado na variável balance:

```
 setBalance(A, 11.000,00 - 10,00 );        //!
```

Observe que o valor da conta foi equivocadamente atualizada para 10.990,00. Certamente, é um erro grave decorrente das condições de disputa do acesso a um mesmo valor da conta do cliente por dois *threads* (regiões críticas). Para evitar isso, a execução de *multithreads* deve satisfazer seguintes condições [3]:

- **exclusão mútua**: duas sequências de execução não podem estar simultaneamente na sua região crítica.
- **imprevisibilidade**: não se deve fazer nenhuma suposição acerca a velocidade e a quantidade de processadores.
- **bloqueio condicionado**: nenhuma sequência de execução fora da região crítica pode bloquear a entrada de outra sequência na região crítica.
- **espera limitada**: nenhuma sequência de execução deve esperar eternamente para entrar na região crítica

Figura 11.5 ilustra a execução de duas sequências de instruções contendo regiões críticas. Quando a sequência A entra na região crítica, ela não deixa a sequência B entrar na região crítica (exclusão mútua). Este bloqueio termina quando A sai da

região crítica (bloqueio condicionado) e B entra na região crítica em algum momento após a requisição à entrada (espera limitada).

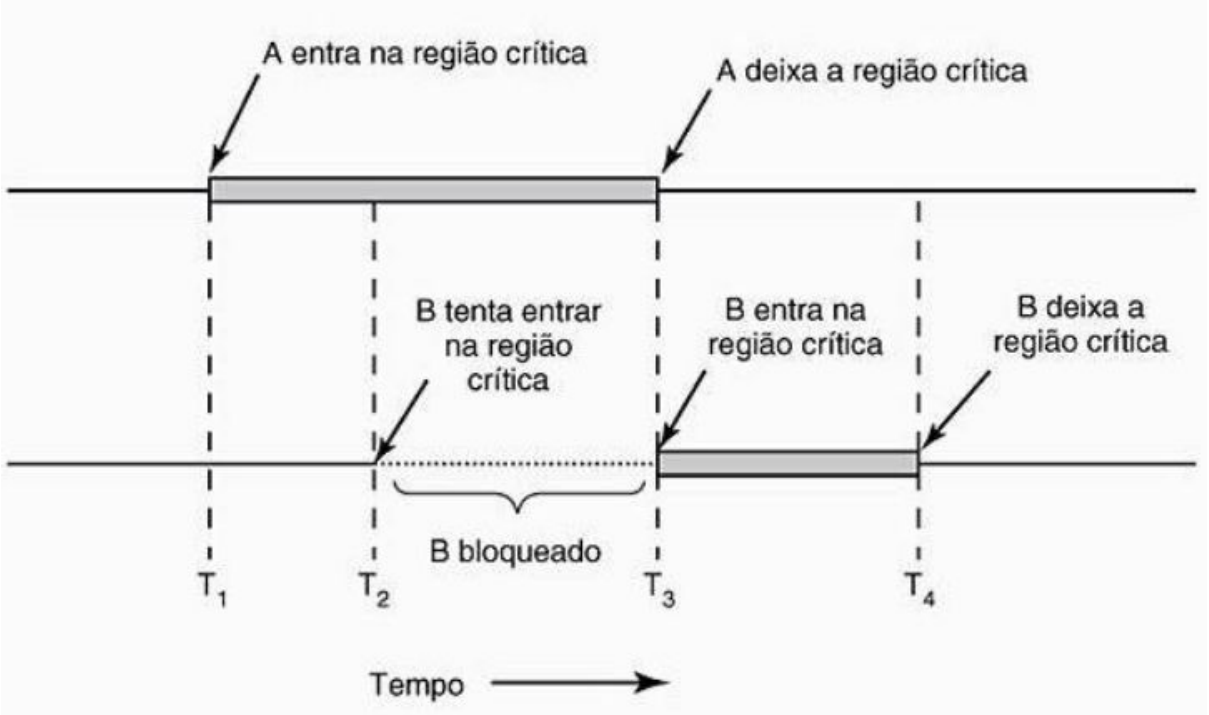


Figura 11.5: Exclusão mútuas de trechos de instruções usando regiões críticas (Fonte: [3]).

11.1.2 *Deadlock*

O bloqueio por região crítica pode causar um outro problema conhecido como ***deadlock***, que trava completamente o fluxo de execução. Isso acontece quando dois *threads* acessam dois recursos na ordem trocada, como mostra a Figura 11.5. Na Figura 11.6, o *thread* X precisa acessar o recurso A e o recurso B, enquanto o *thread* Y, o recurso B e o recurso A. O *thread* X ficará travado aguardando a liberação do B pelo *thread* Y, e este não consegue liberar B porque precisa de A.

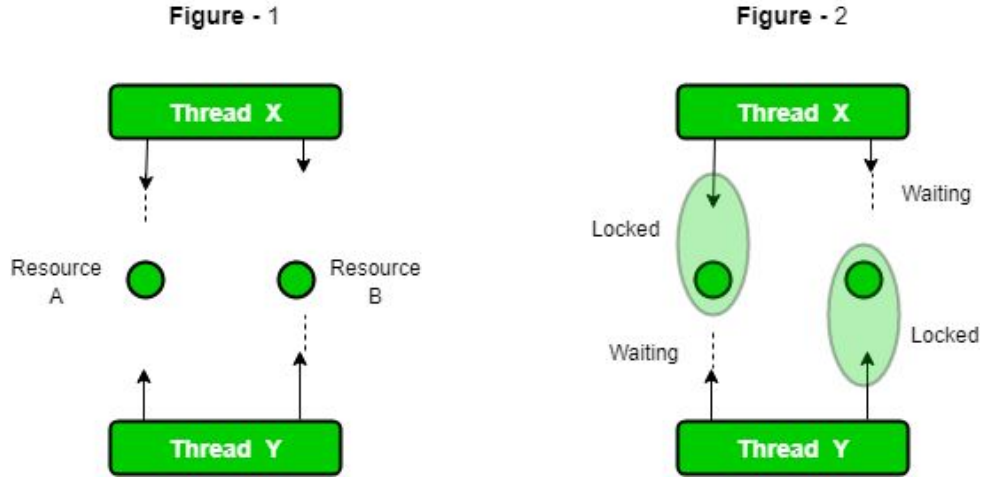


Figura 11.6: *Deadlock* (Fonte:[8])

Dois famosos exemplos de *deadlock* são o de cruzamento de 4 carros (Figura 11.7) e o problema do jantar de filósofos (Figura 11.8). No primeiro, se os carros chegarem simultaneamente no cruzamento, nenhum conseguiria prosseguir pelo intertravamento. No segundo, se todos pegarem simultaneamente o garfo do seu lado esquerdo, nenhum deles conseguiria comer por ter que ficar aguardando a liberação do garfo do seu lado direito.

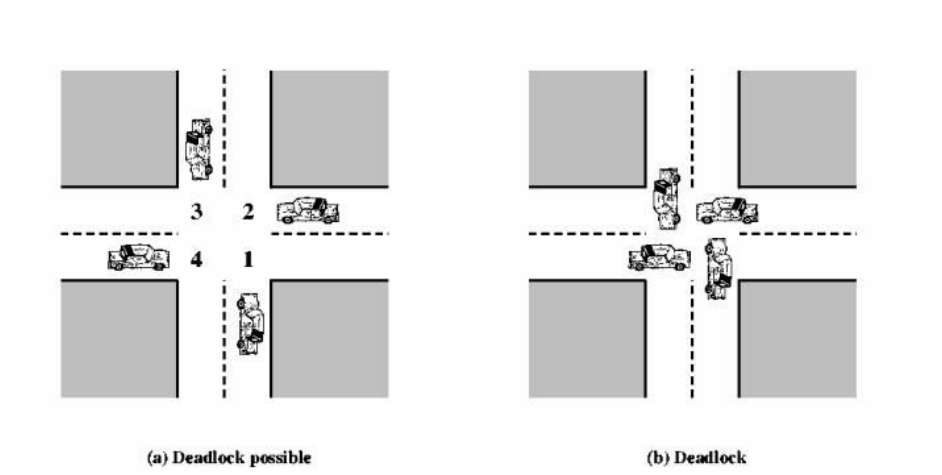


Figura 11.7: 4 carros num cruzamento (Fonte: [12]).

Figura 11.8: Problema do jantar de filósofos (Fonte: [13]).

Deadlock é um processo complexo, difícil de ser detectado e tratado. Essencialmente, ele pode acontecer em situações em que

1. cada *thread* faz acessos a recursos compartilhados,
2. cada *thread* pode ocupar algum recurso compartilhado e solicitar, ao mesmo tempo, o acesso a um outro recurso compartilhado, e
3. uma cadeia de espera circular é potencialmente possível.

Estratégias para evitar *deadlocks* consistem essencialmente em remover uma das condições acima, removendo os bloqueios desnecessários e/ou os bloqueios gerados por múltiplas fontes (de controle). Uma estratégia seria impor uma ordem nos acessos aos recursos compartilhados, como os semáforos nos cruzamentos para ordenar o acesso ao cruzamento pelos carros. Outra estratégia seria assegurar a alocação de todos os recursos compartilhados a um processo antes da sua execução. Uma terceira estratégia é redesenhar o *software* para minimizar ou remover todas as possíveis fontes de *deadlock*. Há ainda casos em que o temporizador *watchdog* (Seção 10.3) é usado para resetar o processo quando ele entra em *deadlock*.

11.2 Exceções e Interrupções

Ao longo da execução de um fluxo de controle programado, um microprocessador pode se deparar com situações de erros conhecidos como **eventos de *software***:

- ***faults***: antes do processador iniciar a execução, como detecção de uma instrução inválida ou de operandos inválidos (divisão por zero);
- ***traps***: durante ou após a execução de uma instrução pelo processador, como a ocorrência de *overflow*, e
- ***abort***: quando ocorre uma falha grave no processador e que o sistema não sabe como prosseguir, como falha num acesso à memória por uso de endereços inválidos.

Estas situações são detectadas com base nas condições pré-definidas. Porisso, elas são consideradas eventos síncronos (em relação ao relógio do sistema) e são usualmente chamadas **exceções**. As interrupções, por sua vez, são eventos assíncronos gerados fora do microprocessador como vimos na Seção 10.4. Eles são tipicamente denominados **eventos de *hardware***.

Num microcontrolador moderno o controlador de interrupção consegue controlar centenas de interrupções. Um exemplo é o controlador de interrupção aninhada, em inglês *Nested Vector Interrupt Controller* (NVIC), mostrado na Figura 11.2. Este controlado centraliza todas as exceções e interrupções uniformizando o processamento de prioridade de atendimento das solicitações. Como a cada exceção/interrupção é associado um número de exceção (Seção 10.4.2), o NVIC seleciona, dentre as solicitações recebidas, o evento que deve ser atendido e passa para o processador o número de exceção correspondente. Com base nesse número, busca-se na tabela de vetor de interrupção o endereço da rotina de serviço.

Sob o ponto de vista de processamento de um evento (1 na Figura 11.9), distinguem-se 6 passos: (1) avisar a ocorrência de uma interrupção ao processador (2 na Figura 11.9) que vai interromper o fluxo corrente de execução (3 na Figura 11.9) e aguardar que este solicite a identificação do evento solicitante (4 na Figura 11.9), (2) buscar o endereço da rotina de serviço (ISR), numa **tabela de vetores de interrupção**, em inglês *interrupt vector table*, de acordo com a identificação do evento (5 na Figura 11.9), (3) salvar o estado atual (SR) das instruções (7 na Figura 11.9), e (4) carregar no contador de programa (PCR) esse endereço (8 na Figura 11.9), de forma que a primeira instrução da ISR seja a próxima instrução a ser executada. (5) Após a execução da rotina de tratamento (9 na Figura 11.9), o estado do trecho de instruções interrompido é restaurado (10 na Figura 11.9) e (6) a bandeira de interrupção é baixada (6 na Figura 11.9), tornando o mecanismo disponível para tratamento de um próximo evento de interrupção.

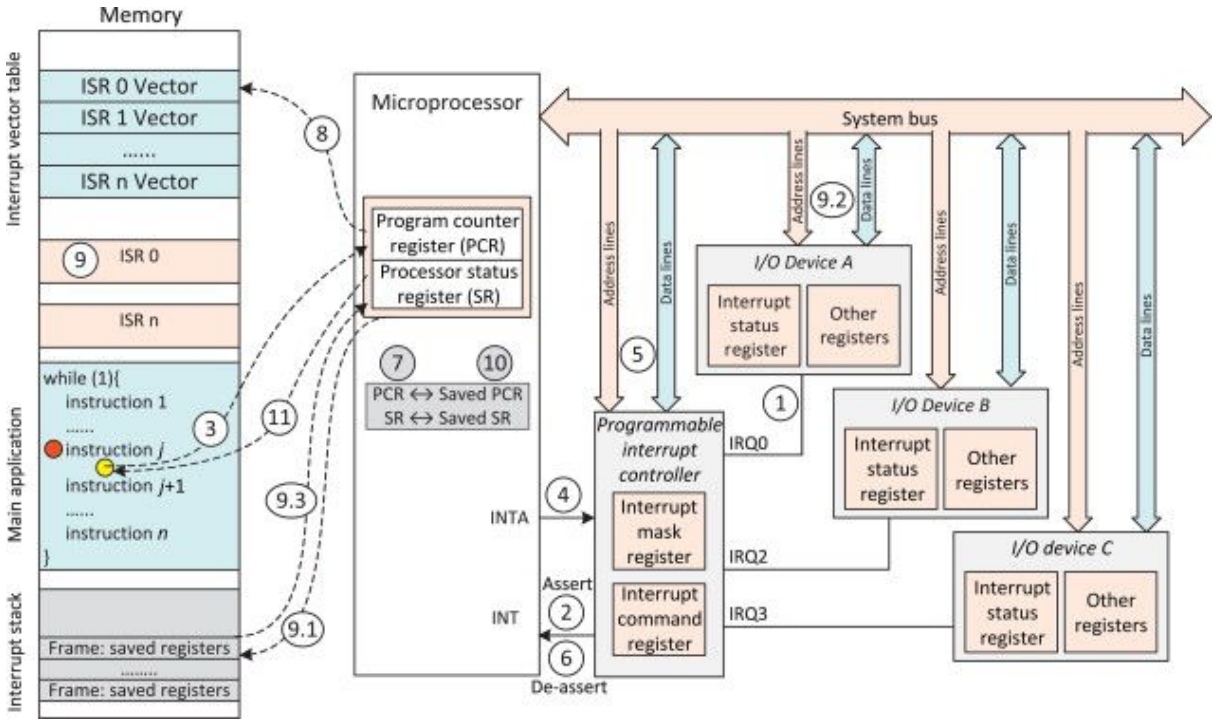


Figura 11.9: Processamento de uma interrupção (Fonte: [10]).

11.3 Regiões Críticas em Interrupções

Figura 11.10 mostra a execução de 2 rotinas de serviço de prioridades diferentes (prioridade de Interrupt 2 > prioridade de Interrupt 1) na linha de tempo. Observe a similaridade entre a disposição dos trechos de códigos das rotinas de serviço (Figura 11.10) e a disposição das regiões críticas dos *threads* (Figura 11.5) na linha de tempo.

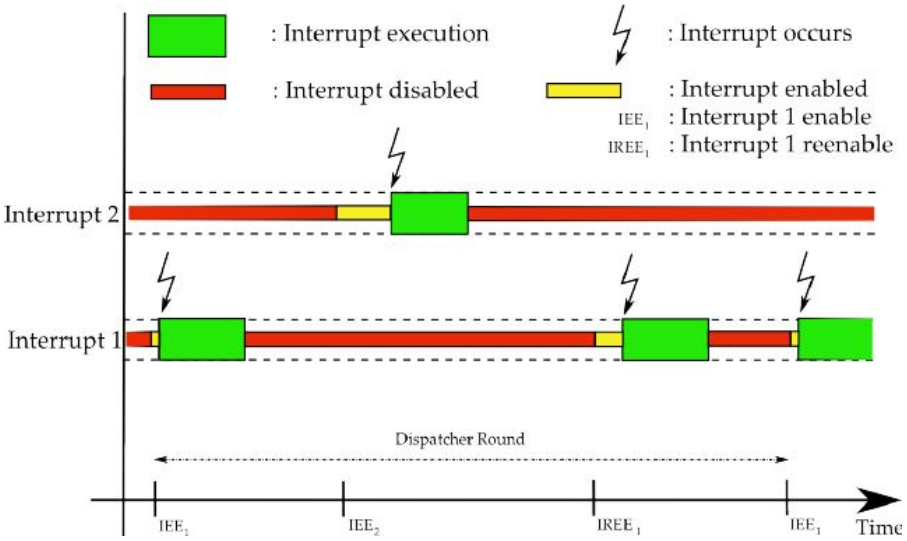
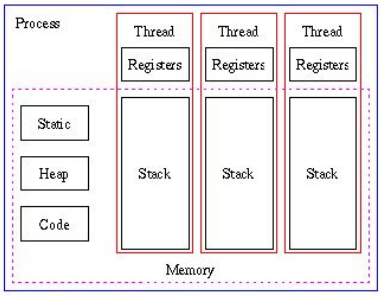
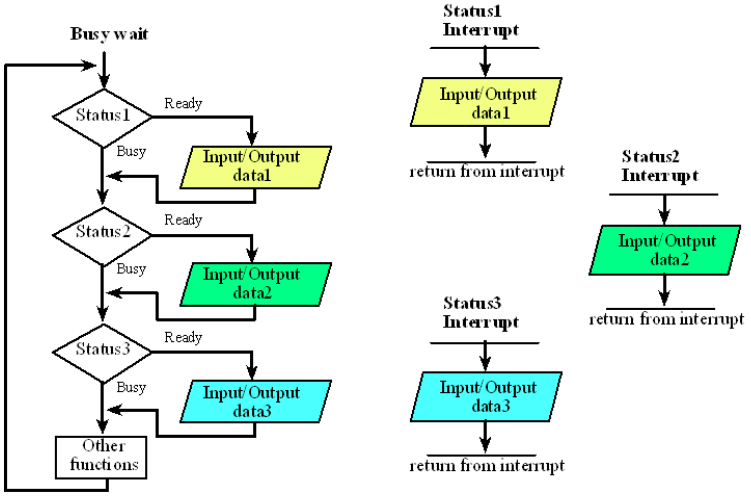


Figura 11.10: Execução de uma interrupção (Fonte: [6]).

Figura 11.11 faz um paralelo entre *multithreading* e processamentos de interrupções em termos do uso de recursos compartilhados de forma explícita. Em ambos os casos, instruções em trechos distintos que fazem uso de um mesmo conjunto de recursos, como usar uma mesma variável ou acessar um mesmo arquivo, podem ter diferentes ordens de execução e gerar condições de disputa. Portanto, de forma análoga ao processamento de *multithreads*, tomamos emprestado aqui o termo de **região crítica** para caracterizar trechos de instruções do fluxo principal de um processo (losangos em branco na Figura 11.12(b)) que fazem acessos a um mesmo conjunto de recursos usados pelas rotinas de serviço (paralelepípedos em amarelo, em verde e em azul na Figura 11.12(b)).



(a) multithread



(b) Interrupções

Figura 11.11: Compartilhamento de recursos por trechos de instruções independentes.

Vamos ilustrar a ideia com um projeto de sistemas embarcados. Considere um alarme sonoro que deve ser acionado quando a temperatura de uma caldeira ultrapasse de uma temperatura T estabelecida pelas normas de segurança. Foi configurado um temporizador para amostrar periodicamente as medidas de um sensor de temperatura. Ou seja, na rotina de tratamento das interrupções periódicas do temporizador, temos:

```
ISR_Timer() {
        amostra (temperatura);
}
```

E o projetista inseriu no laço do fluxo de controle principal o teste do valor de temperatura para controlar o sinalizador visual (led) e auditivo (alarme) do estado da temperatura da caldeira:

```
1 main () {

2       for (;;) {
                :
3               visualiza (temperatura);
4               if (temperatura <= T) {
5                 alarme = desliga;
6                 led_vermelho = desliga;
7                 led_verde = liga;
8               } else {
9                 alarme = liga;
10                led_verde = desliga;
11                led_vermelho = liga;
12              }
                :
        }
}
```

O que aconteceria se a interrupção ocorresse exatamente na linha 6 do fluxo principal? O sistema poderia ter amostrado um valor de temperatura maior do que o limite T, mas ao retornar para a linha 7, depois de ter completada a execução da rotina ISR_Timer, ele ligaria o led_verde. Isso aconteceu porque a variável temperatura (armazenada na memória) é acessada concorrentemente por dois trechos de códigos, linhas 4 a 12 da rotina main e a rotina ISR_Timer. Esses trechos de código são chamados **regiões críticas** que disputam o acesso a um espaço de memória onde está armazenada a variável temperatura.

Ressaltamos que, em muitos microcontroladores modernos, **é possível interromper o fluxo de execução de uma rotina de serviço somente por um evento de prioridade maior, mas o retorno ao fluxo de controle principal interrompido só acontece depois de concluída a execução da rotina de serviço do evento original que o interrompeu**. Pois, diferentes do processamento de *multithreads*, as **interrupções nos microcontroladores** são disparadas por eventos assíncronos e atendidas de acordo com a sua prioridade de atendimento definida pelo projetista, e não por um algoritmo de escalonamento sofisticado disponível em sistemas operacionais.

Rotinas de tratamento de interrupções com regiões críticas podem virar uma grande fonte de ***deadlocks****,* caso não analisarmos os possíveis sequenciamentos de bloqueios dentro das rotinas de serviço aninhadas. É bem provável que, durante a execução de uma região crítica, ocorra uma interrupção por um evento cuja rotina de serviço seja também uma região crítica. Vamos analisar duas possíveis soluções:

- Se bloquearmos a entrada à região crítica numa rotina de serviço, causaremos um *deadlock*. Pois, o fluxo de controle não retornará à sequência de execução interrompida enquanto não concluir o tratamento da rotina de serviço (Figura 10.28) e a rotina de serviço estará aguardando o desbloqueio do fluxo interrompido para poder concluir a sua execução.
- Se continuarmos o fluxo de execução desviando a região crítica, poderemos perder atualizações dos dados do mundo físico.

A pergunta que se faz no contexto de projetos de sistemas embarcados é como podemos evitar esses impasses na programação concorrente das rotinas de serviço e do fluxo de controle principal esquematizada na Figura 11.11(b).

11.4 Programação Concorrente

Vimos na Seção 11.1 que o *deadlock* pode ser evitado numa programação concorrente, se **minimizarmos a quantidade de bloqueios** e, quando estes são imprescindíveis, devemos evitar que os bloqueios sejam feitos por múltiplas fontes. Na Seção 11.2 vimos que quando o fluxo de controle principal é interrompido, ele só retorna ao ponto interrompido depois de executar integralmente a rotina de serviço correspondente. Portanto, as **rotinas de serviço devem ser mais simples possíveis**, adiando estrategicamente processamentos mais complexos para o fluxo de controle principal.

Vamos discutir nesta seção a aplicação de duas alternativas propostas para sistemas operacionais [3], a fim de realizar a exclusão mútua em microcontroladores que não suportam sistemas operacionais.

11.3.1 Desabilitação de Interrupções

A solução mais simples é aquela em que o fluxo de controle principal desabilita todas as interrupções logo depois de entrar em sua região crítica e as reabilita imediatamente antes de sair dela, como o seguinte esboço do código:

```
regiao_critica () {
desabilita_interrupcoes();
processamento_instrucoes();
habilita_interrupcoes();
}
```

Com as interrupções desabilitadas, é garantida que dentro da região crítica a sequência de execução não será interrompida e o fluxo de controle principal consegue processar os dados dos recursos compartilhados sem correr o risco de disputas com outras sequências de execução, Embora seja considerada uma técnica obsoleta para sistemas operacionais modernos com multinúcleos, esta técnica é muito aplicada na programação de microcontroladores com um único microprocessador.

11.3.2 Mutexes ou Travas

Em sistemas operacionais, uma alternativa mais simples por *software* é o uso de uma variável compartilhada denominada **trava de exclusão mútua**. A trava é iniciada com 0 indicando que todas as sequências de execução estão fora da região crítica. Antes de uma sequência entrar na região crítica, ela verifica o estado da trava. Se a trava estiver em 1, ela aguarda até que a trava fique em 0. Se a trava estiver em 0, ela altera a trava para 1, entra na região crítica e reseta a trava para 0 antes de sair da região crítica. A ideia é transcrita no seguinte pseudocódigo em que a região crítica se encontra entre dois trechos de códigos processamentoA() e processamentoB():

```
processamentoA();
while (mutex != 1);
mutex ← 1;
processamento_regiao_critica();
mutex ←0;
precessamentoB();
```

Como a interrupção pode ocorrer em qualquer ponto. A rotina de serviço lerá 0 na variável mutex se a interrupção ocorrer entre a linha “while (mutex != 1)” e

“mutex=1”. E lerá 1, se ela ocorrer depois da linha “mutex=1” e antes de “mutex=0”. Tipicamente, isso resultará dois diferentes comportamentos:

- ler 0: o fluxo principal e a rotina de serviço estarão em suas regiões críticas ao mesmo tempo, podendo ocorrer disputas nos acessos aos recursos compartilhados.
- ler 1: a rotina de serviço fica bloqueado aguardando pelo valor 0 no mutex que não acontecerá.

Além desses problemas, temos ainda o laço de **espera ociosa**, em inglês *busy waiting*, por um valor do mutex.

Em sistemas embarcados, quando

- os dados são atualizados em alta frequência pelos periféricos e ocupam pouco espaço da memória, e
- a atualização do conteúdo da memória só é de responsabilidade de um componente (processador ou periférico),

tenho adotada a estratégia de **cópia dos dados** do recurso compartilhado, de forma que as rotinas de serviço fiquem livres para atualizar os dados do mundo físico enquanto o fluxo de controle principal processa uma cópia da última versão estável destes dados. Neste caso, uso um *bit* de estado *flag* para controlar somente o fluxo de controle principal, como mostra os seguintes trechos de pseudocódigos:

1) Fluxo de controle principal

```
flag ← 0;
while (TRUE) {
    if (flag == 1) {
      copia ←dados;
      flag ←0;
      processamento_iteracao ();
    }
}
```

2) Rotina de serviço

```
{
      atualizacao_memoria_compartilhada (dados);
      flag ← 1;
}
```

Observe que no código acima a consistência do conteúdo copia é mantida na região considerada antes como crítica e o potencial de concorrência entre os trechos de códigos (1) e (2) foi removido. Pois, o conteúdo de dados é sempre atualizado em (2) e, para evitar processamentos desnecessários, o flag em (1) é somente aplicado a fim de verificar se há de fato atualização de dados em cada iteração. Note que

processamento de uma cópia dos dados defasados no tempo pode resultar em um atraso na atuação de um dispositivo, que precisa ser ponderado criteriosamente pelo projetista.

Outra situação muito comum em sistemas embarcados é o sincronismo das tarefas de um produtor e de um consumidor, por exemplo de uma mensagem, numa comunicação. Neste caso, não devemos perder nenhum caractere de informação. Portanto, não podemos ignorar nenhuma interrupção de recepção ou de transmissão. Como então programar um código que evite concorrência sem ignorar a característica peculiar de processamento de interrupções e sem desabilitar as interrupções? Uma estratégia muito aplicada é armazenar os dados produzidos num *buffer*[2] de dados para serem processados na velocidade no consumidor. Veremos no Capítulo 12 algumas técnicas de estruturação desses dados em linguagem C que possam evitar concorrência e *deadlocks*.

11.5 Exercícios

1. Qual é a diferença entre *threads* e *processos*?
2. O que você entende por condições de disputa e por regiões críticas?
3. O que você entende por *deadlocks*?
4. Por quê o gerenciamento de concorrência a nível de tarefas é importante em projetos de sistemas embarcados? Em qual estágio de um projeto de sistemas embarcados se deve ocupar com a identificação de tarefas concorrentes? Justifique.
5. Dê um exemplo de possíveis condições de disputa no processamento de amostras analógicas de um sensor de fumaça para monitoramento do nível de gás carbônico num ambiente. Quais seriam as possíveis soluções?
6. Qual é a diferença entre uma exceção e uma interrupção?
7. Dê um exemplo de possíveis *deadlocks* no processamento de interrupções aninhadas (*nested interrupts*). Quais seria as possíveis soluções?
8. Por quê se deve minimizar a quantidade de instruções de uma rotina de serviço?
9. Uma das estratégias para “proteger” regiões críticas é desabilitar as interrupções nesta região. Ao aplicar esta estratégia, relevantes interrupções serão perdidas num controlador de interrupções moderno como NVIC? Justifique.
10. O que são travas mutexes? Qual problema podemos ter ao aplicá-los em processamento de interrupções?

[2] *Buffer* é um espaço da memória reservada para armazenar temporariamente os dados enquanto eles estão sendo movidos de um lugar para outro.

11.6 Referências

[1] Edward A. Lee and Sanjit A. Seshia. Introduction to Embedded Systems. A Cyber-Physical Systems Approach. Second Edition, MIT Press, ISBN 978-0-262-53381-2, 2017.
[2] Danielle Collins. What is nested vector interrupt control (NVIC)? https://www.motioncontroltips.com/what-is-nested-vector-interrupt-control-nvic/
[3] Tanenbaum, A. S. Sistemas Operacionais Modernos. ISBN: 978-8543005676. Pearson Universidade. 2015.
[4] https://www.studytonight.com/operating-system/multithreading
[5] Microchip Technology. ATmega328P Datasheet. http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf
[6] https://www.researchgate.net/figure/An-example-of-interrupts-in-OSEKtime-OS_fig47_261989601
[7] Mike Silva. Introduction to Microcontrollers - Interrupts. https://www.embeddedrelated.com/showarticle/469.php
[8] GeeksForGeeks. Deadlock in Java Multithreading. https://www.geeksforgeeks.org/deadlock-in-java-multithreading/
[10] https://www.sciencedirect.com/topics/engineering/interrupt-vector-table
[11] https://stackoverflow.com/questions/2482957/what-is-critical-section-in-threading
[12] https://stackoverflow.com/questions/1385843/simple-deadlock-examples
[13] https://en.wikipedia.org/wiki/Dining_philosophers_problem