Projeto e desenvolvimento de sistemas em tempo real sobre Sistemas Embarcados

Um sistema de computação, ou sistema digital, é um conjunto de vários componentes que coordenam os dispositivos de entrada e saída (I/O), além da memória para a realização do processamento de algum dado. Um sistema de computação pode ser entendido como a união do Hardware (processador principal, memória, dispositivos de I/O e barramentos) e do Software (programa que usa os componentes do hardware). Um sistema embarcado (SE) é um sistema de computação dedicado a uma tarefa específica que normalmente possui recursos limitados, como memória, autonomia reduzida (geralmente estão conectados a baterias) e baixo poder computacional.

De acordo com T Noergaard, um sistema embarcado é um sistema computacional aplicado. Um SE, ou sistema embutido, envolve o uso de um computador, o qual não é o de uso comum, em uma aplicação específica. De forma mais genérica, é qualquer sistema automatizado que envolva algum nível de inteligência usado para controlar sistemas de diferentes tipos, como por exemplo máquinas domésticas, microondas, televisão, roteadores. O primeiro SE registrado foi desenvolvido pelo MIT (década de 60) para equipar as naves do Projeto Apollo.

O tempo de resposta de um sistema digital pode ser entendido como o tempo gasto para, dado um conjunto de entradas, obter-se uma saída. É o tempo necessário para executar o fluxo de controle dentro de um sistema. Um sistema de tempo real é aquele onde o tempo de resposta para produzir uma saída é crucial.

De acordo com Burns & Wellings (2000), um sistema de tempo real é um sistema que deve responder a uma entrada externa ao sistema em um tempo finito e específico. O sistema deve ser determinístico. A corretude do sistema não depende exclusivamente da lógica resultante do processamento, mas também do tempo em que ela foi entregue. Falhar em responder no tempo esperado é tão ruim quanto uma resposta com uma lógica errada. Uma resposta do sistema depois do tempo esperado pode ser sem utilidade ou até representar uma ameaça.

O sistema de controle de temperatura apresentado na Figura 1 possui funções simples: (i) aquisição de dados, (ii) manipulação dos dados (linearização/escala), (iii) processamento de cálculos de controle, e (iv) controlar o atuador. Contudo, ele pode ser usado como exemplo para o entendimento da diferença entre um sistema comum e um de tempo real.

Se o sistema for usado para controlar a temperatura dentro de uma estufa, o máximo que ocorrerá se o sistema demorar a responder é que, ou a estufa vai ficar mais fria do que deveria, ou mais quente por um período de tempo diferente do planejado. A consequência está relacionada ao desenvolvimento das plantas dentro da estufa. Apesar do dano ocasionado, ele não é tão grande quando comparado caso o mesmo sistema for

implementado para controlar o resfriamento de uma usina nuclear, a qual se houver uma demora na resposta do sistema pode haver uma explosão e gerar grandes problemas.

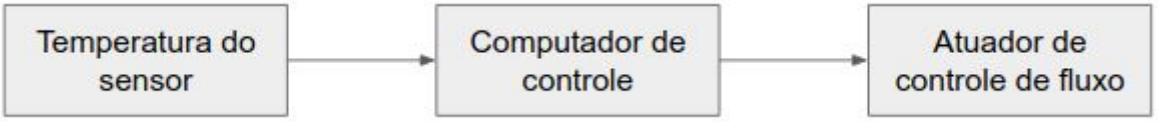


Figura 1: Esquema de um controlador de temperatura

Sistemas digitais normais não garantem as condições de tempo real para tarefas e processos, pois essa garantia é uma tarefa difícil e a maioria dos sistemas não consegue. Por causa disso, alguns Sistemas de Tempo Real, do inglês *Real Time Systems* (RTS), foram propostos e a definição de tempo real foi dividida em duas super categorias: *hard real time* e *soft real time*. Há até mais categorias, entretanto, as duas citadas são as mais gerais.

O sistema *hard real time* consegue comprovar teoricamente sua performance no pior caso. Eu sua maioria são projetados para aplicações especiais e sistemas onde um atraso pode comprometer toda a integridade do sistema ou o de quem o usa, por exemplo: acionamento do airbag, desfibrilador, marcapasso. É crucial que todos os componentes respondam no tempo esperado em qualquer caso. Exemplos de sistemas operacionais *hard real time*: RTLinux e RTAI, ambos de código aberto, e os mais usados na indústria, uC/OS (micro COS) e vxWorks.

O sistema *soft real time* é mais tolerante que o hard real time, uma vez que os deadlines são importantes, todavia o sistema funciona com ocasionais falhas, por exemplo: sistema de aquisição de dados, video-game. Aqui, a performance no pior caso é apenas observada e não provada. Ele é normalmente utilizado quando há a necessidade de concorrência exacerbada. Exemplos: Linux/RX,SMART-Linux, ART Linux.

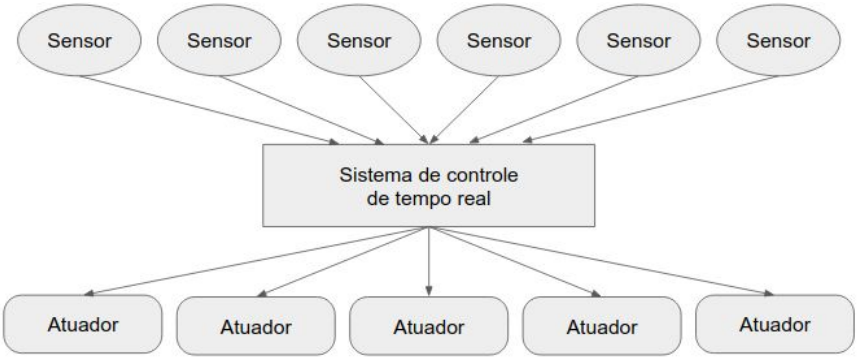
Um sistema embarcado de tempo real pode ser entendido como um software que executa em um computador e/ou controla outras máquinas. A todo momento ele está recebendo sinais de eventos gerados pelo hardware e, como resposta a cada evento, emite sinais de controle para o hardware Como é um sistema de tempo real, a resposta pode estar condicionada a restrições de tempo, uma vez que cada tarefa dentro de um sistema deve executar até um tempo predeterminado.

Todo o processo envolto no projeto e desenvolvimento do RTS deve considerar em detalhes o projeto e o desempenho do hardware do sistema como um todo. É necessário decidir que recursos devem ser implementados em software e quais devem vir do hardware, uma vez que custo e consumo de energia são críticos e precisam ser balanceados. Isso envolve a decisão se serão usados sistemas já prontos (CPUs ATMEL/PIC/AMD, por exemplo), se serão usados hardware descritos por software, ou se um novo hardware deverá ser projetado e construído a partir do zero.

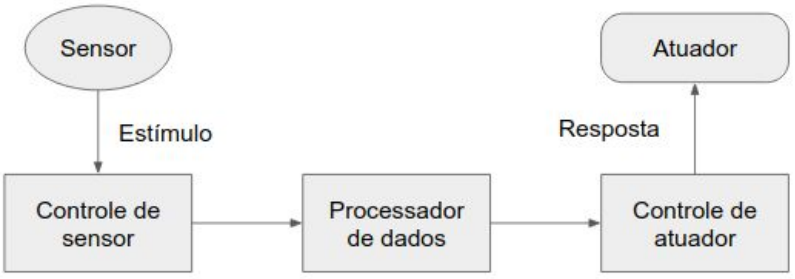
Há duas abordagens mais populares para projetar um SE: *top down* e *bottom up*. A primeira, *top down*, envolve a decomposição de todo o produto final em partes. Essa decomposição chega até certo nível, uma vez que faz-se uma abstração dos componentes mais baixo nível. Isso facilita no ponto em que a modelagem do sistema baseia-se em uma

abordagem de estímulo-resposta, onde o estímulo é a entrada que deve produzir uma resposta, a qual é dirigida a atuadores. Os estímulos podem tanto ser periódicos, quanto aperiódicos. Já os sistemas *bottom up* iniciam o processo dos componentes mais baixo nível relacionados ao hardware e vão construíndo cada parte do sistema final. Esse tipo de processo facilita bastante a reutilização de módulos e acaba por adaptar-se melhor a pequenos projetos. Em ambos os casos, *top down* e *bottom up*, as decisões de baixo nível relacionadas ao hardware precisam ser tomadas no início do projeto. Esse tipo de ação, diminui muito a flexibilidade do projeto, uma vez que o hardware é pensado para uma aplicação em específico. Em muitos casos, quando o processo *top-down* é escolhido, as definições de baixo nível do hardware que precisam ser tomadas não são práticas, uma vez que elas precisam se adaptar ao que foi planejado.

A arquitetura genérica abstrata de um RTS, possui três tipos de processos (Figura 2b): (i) um processo responsável por gerenciar cada atuador; (ii) um processo para gerenciar os sinais de cada sensor; e (iii) os processos que controlam e processam os dados do RTS. Essa arquitetura genérica, a qual pode ser vista na Figura 2a, pode ser instanciada para qualquer arquitetura específica, dadas suas adaptações, como por exemplo, sistemas de monitoração e de aquisição de dados



(a) Arquitetura genérica abstrata



(b) Processos de sensores e atuadores

Figura 2: Diretriz geral: processos separados para cada tipo de sensor e atuadores

As atividades principais envolvendo o projeto de um RTS são:

1. Seleção das plataformas que serão usadas para o hardware e para o software;

2. Decisão de quais partes serão implantadas como hardware e quais serão implementadas como software;
3. Identificar quais são os estímulos e respostas possíveis do sistema;
4. Analisar as restrições temporais (*timing*) de cada estímulo;
5. Alocar receptor de estímulo e processamento a processos concorrentes;
6. Projeto de processos concorrentes de acordo com que as arquiteturas escolhidas possibilitam;
7. Projetar os algoritmos para fazer o processamento necessário;
8. Avaliar e projetar as estruturas necessárias dos dados;
9. Projetar um sistema de alocação (*scheduling*) que permite atender as várias restrições de tempo “simultaneamente”.

Apesar dos Sistemas Operacionais funcionarem de forma sequencial, isso não se aplica ao mundo real e ao contexto de RTS. Eventos ocorrem simultaneamente no mundo real a todo instante e acabam por concorrer pela atenção do sistema digital. A criação de um Sistema Operacional de Tempo Real, ou do inglês, *Real Time Operating System* (RTOS) envolve a modelagem de entidades de forma a funcionarem em paralelo, estas que podem estar sendo processadas em threads. O conceito de thread será descrito mais à frente, mas pode ser entendida como uma instância de uma tarefa em um sistema operacional. Cada evento é tratado por uma tarefa distinta, e tarefas distintas podem ser necessárias simultaneamente. Esse cenário recebe o nome de sistemas concorrentes e a programação de tais, de programação concorrente.

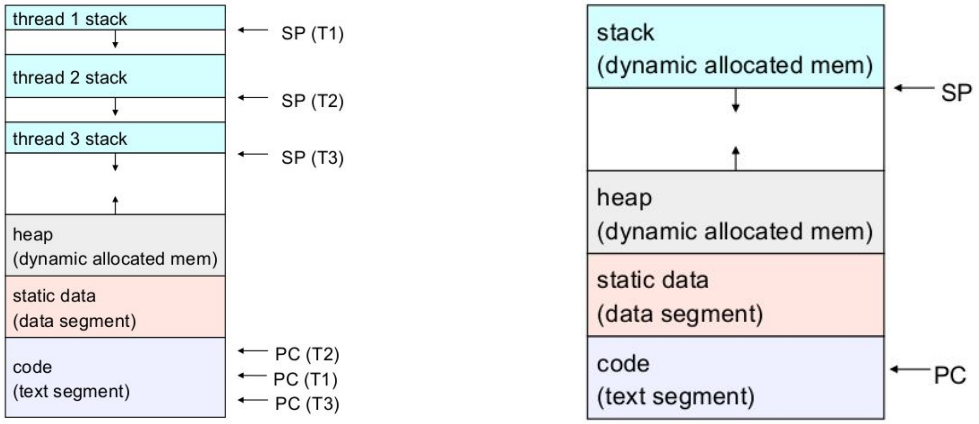
Segundo Seixas Filho e Szuster (2003), a programação concorrente é o estudo da execução alternada de instruções atômicas de processos sequenciais (escalonamento). Um SO é composto por diversos processos concorrentes. Cada processo, ou até mesmo programa, pode ser composto por uma ou mais tarefas. Cada tarefa é um conjunto de instruções executadas por uma thread.

Assim como a escolha dos componentes do hardware são importantes, a escolha por qual linguagem de programação também o é. A escolha deve envolver linguagens que incluem recursos de baixo nível, como o C. O código gerado por essas linguagens tendem a ser mais próximos ao Assembly (linguagem próxima a das instruções dos processadores). Esse código será substituído por instruções, variáveis pelos códigos binários e endereços de memória correspondentes. Algumas linguagens vêm sendo atualizadas para incluir mecanismos que permitam o seu uso em sistema de tempo real, como o Java. O mecanismo mais básico usado são as threads.

A thread é um pequeno processo, ou subprocesso, que trabalha como um subsistema dentro do SO. É a forma mais simples do processo se dividir em duas ou mais tarefas. Ela é basicamente uma unidade de utilização da CPU e possui um ID, um código (PC), um conjunto de registradores e uma pilha. Ela compartilha com outras threads do mesmo processo o código e as seções de memória, além de outros recursos do Sistema Operacional (arquivos abertos e outros sinais). As threads do mesmo processo não possuem proteção entre si. As várias threads de um mesmo processo utilizam a pilha já alocada para o processo, conforme mostrado na Figura 3. Há as threads de usuário que são visíveis exclusivamente pelo programador, onde todo o controle é feito pelas bibliotecas de

thread do nível do usuário sem nenhum auxílio do kernel, e há também as threads do sistema/kernel, onde cada thread processa uma tarefa específica.

O uso de várias threads trás alguns benefícios como responsividade (parte do programa pode estar bloqueado esperando um recurso, enquanto o restante funciona normalmente), divisão de recursos (as threads compartilham memória e recursos de um mesmo processo, o que é mais barato do que dividir entre processos) e economia (criar e destruir threads é mais barato do que quando comparado com processos, além de ser mais barato trocar de contexto entre threads e a comunicação entre elas ser mais fácil).



(a) Várias threads em execução (b) Um processo

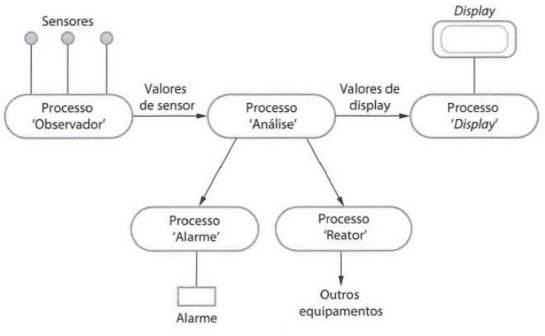
Figura 3: Exemplo do uso da pilha e do heap por um processo com uma e várias threads.

Apesar dos inúmeros benefícios envolvendo as threads, surge um problema com o seu uso: definir qual processo/thread deve executar enquanto há vários para serem processados e como impedir que várias threads/processos alterem porções de memória compartilhada sem gerar estados inválidos. A solução é a coordenação de processos, com semáforos e regiões críticas, e técnicas de escalonamento. O escalonamento em RTS necessita a análise teórica para avaliar se as restrições temporais serão atendidas, tarefa a qual pode ser bastante complexa. Como os processos que serão escalonados possuem diferentes instruções e acabam por executar com diferentes velocidades, torna-se necessário a criação de “buffers” compartilhados, além do uso de exclusão mútua para acessá-lo (enquanto um processo estiver usando, todos os demais devem esperar) realizado por meio de semáforos. Os buffers auxiliam nas trocas de informações e nas trocas de contexto (um processo deixa de ser executado e outro começa a ser executado).

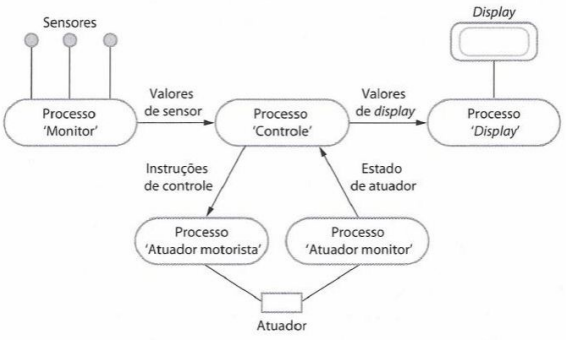
A modelagem de qualquer RTS passa pelo planejamento de como o sistema irá responder a cada estímulo, já que a resposta a um estímulo muitas vezes depende do estado corrente do sistema. O diagrama de UML é usado para representar cada estado do RTS, uma vez que mapeia-se cada um em um diagrama de estados. O modelo de estados mapeia todos os estados possíveis de um sistema e qual a transição de estado para estado deve ser feita após um estímulo ser recebido.

Os padrões de arquitetura de SE projetado são orientados a processos, ao invés de orientados a objetos e componentes. Existem três padrões, os quais podem ser combinados, além de mais de um estarem presentes em um sistemas:

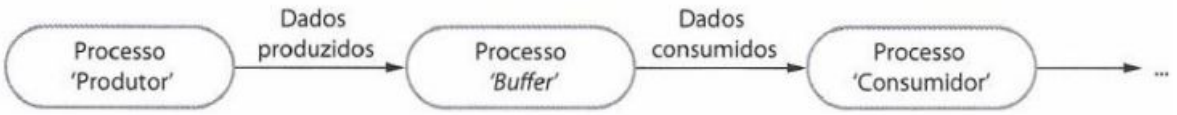
1. Observar e reagir: A cada específicos intervalos de tempo, o conjunto de sensores do sistema é monitorado e exibido. Se caso ocorrer algo fora do esperado (ocorreu um evento), o sistema reage e trata o evento, por exemplo, ligar um alarme. Não há uma alteração no ambiente. O seu uso se concentra em sistemas de monitoração e sistemas de alarme. Exemplo da estrutura na Figura 4a.
2. Controle de ambiente: O sistema faz a análise as informações de um conjunto de sensores que coletam dados do ambiente do sistema, além das informações sobre os atuadores. Com bases nesses dados coletados, sinais de controle são enviados aos atuadores, forçando assim, alterações no ambiente do sistema. O seu uso em geral está ligado a sistemas de controle, e talvez seja o caso mais comum de SE. Exemplo da estrutura na Figura 4b.
3. Pipeline de processo: A partir de estímulos dos sensores de entrada do ambiente ou de outro processo, o processamento move-se dentro de um pipeline. Esse pipeline é criado como uma sequência de dados. Na maioria das vezes, os processos são ligados por *buffers* sincronizados, os quais permitem a execução de um esquema produtor-consumidor (produtor produz recursos que devem ser processados e guardam em um buffer, o consumidor consome os recursos e processam de acordo com o que for necessário - produtor e consumidor possuem velocidades diferentes, com o último sendo muito mais lento devido ao processamento e apresentar um número de instâncias maior do que o primeiro). O seu uso geralmente está ligado a quando dados devem ser transformados de uma representação para outra antes de serem processados. Isso ocasiona uma sequência de processamentos que podem eventualmente serem feitos em paralelo por diferentes processadores. Os maiores exemplos são sistemas de aquisição de dados e sistemas multimídia, onde o auge está relacionado ao armazenamento e/ou exibição de dados. Exemplo da estrutura na Figura 4c.



(a) Padrão observar e reagir



(b) Padrão controle de ambiente



(c) Padrão de pipeline de processo

Figura 4: Exemplos de cada padrão

O RTOS necessita da execução de vários processos e threads simultaneamente (multitarefa). Para isso, é necessário analisar a frequência de execução de cada processo para garantir que todas as entradas sejam processadas e todas as saídas sejam produzidas no tempo esperado. Essa análise é mais complexa quando, dentro do mesmo sistema, há eventos periódicos e sem periodicidade.

Os processadores usados em sistemas digitais como um todo, inclusive SE, são sequenciais e acabam por executar tarefas em loop contínuo. Toda tarefa tem um tempo para se executada (Te). As tarefas podem ocorrer a cada intervalo TP. Em sistemas de tempo real, usualmente as tarefas possuem um tempo máximo para serem realizadas (Td). Algumas tarefas ficam ociosas (Ts) até serem executadas novamente. A Figura 5 mostra a relação entre esses tempos citados e a execução de uma tarefa.

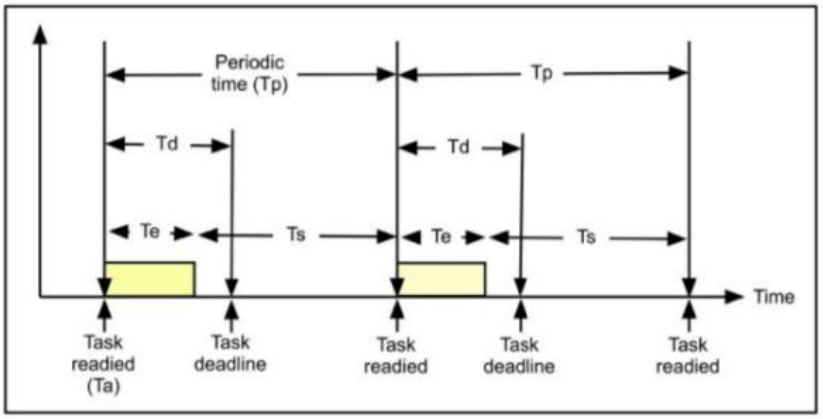


Figura 5: Execução de uma tarefa e os diferentes tempos relacionados a ela

O escalonamento no contexto de uma ou poucas tarefas é simples, todavia, em sistemas mais complexos como o de um avião, organizar todas as tasks para que terminem antes do deadline (prazo máximo) é um desafio. Se uma tarefa tiver que esperar outra terminar completamente para começar, pode ser que não termine a tempo. Para contornar esse problema, uma primeira abordagem simples, pode ser uso de interrupções e tarefas que permitem a programação de temporizadores e de *timeout*, o que para vários projetistas, já é o suficiente. Entretanto, essas práticas têm permitido, de forma aceitável, garantir exigências de sistemas com restrições temporais que não são tão restritivas (sistemas *soft real time*). Quando passa para sistemas de tempo real mais complexos (sistemas *hard real time*), torna-se necessário abordagens mais complexas.

Muitos RTOS estão presentes em sistemas embutidos ou podem ser executados em um sistema operacional (µc/OS). Eles podem ser muito simples ou grandes e complexos, como: Windows/CE, Vxworks e RTLinux. Em geral, todos ele possuem um relógio de tempo real, tratador de interrupções, um escalonador (*scheduler*), gerenciador de recursos e um despachador (inicia a execução no processador disponível).

Os sistemas de tempo real podem ser divididos em duas categorias: sistemas síncronos (lidam com eventos periódicos) e assíncronos (lidam com eventos sem periodicidade). Os sistemas síncronos, ou *clock-driven*, são caracterizados por um temporizador físico que gera interrupções a intervalos fixos de tempo. Neste cenário, pode-se usar somente uma interrupção que é ligada ao timer. Esse tipo de sistema apresenta o problema de que toda a gerência das tarefas fica encargo do desenvolvedor. Para sistemas simples isso é resolvido, contudo, isso não ocorre quando o número de tarefas aumenta e o sistema fica mais complexo. Os sistemas assíncronos vieram como solução para esse problema. Eles também são conhecidos como sistemas baseados em tarefas. O seu desenvolvimento envolve seis passos:

1. Definição clara do problema.
2. Subsistemas, ou subfunções, concorrentes.
3. Temporização e interface entre os subsistemas.
4. Projeto e implementação de cada subsistema (code unit).
5. Associar cada subsistema a um “processo executivo”.
6. Rodar todas as tarefas em hardware simultaneamente por meio de um processador “abstrato” (escalonar as tarefas, a qual dá impressão de vários tarefas executando simultaneamente no hardware).

No projeto de RTS, é necessário tratar eventos que não possuem periodicidade. Como os eventos sem periodicidade de sistemas são imprevisíveis, uma abordagem para tratá-los é estabelecer algumas suposições sobre sua probabilidade de ocorrência. As suposições podem não corresponder à realidade quando o sistema for implementado. Como os computadores estão ficando cada vez mais velozes, os eventos sem periodicidade podem ser tratado como eventos periódicos. Por exemplo:

- Situação: falha de alimentação em um sistema embarcado pode resultar em um desligamento do sistema físico de forma controlada.
- Solução:
    - Por evento não periódico: implementar uma interrupção de “falha de energia”.
    - Por evento periódico: processo que é executado com alta frequência, o qual verifica se houve alguma falha na alimentação, com tempo suficiente para desligar o sistema antes que o SE seja desligado forçadamente (falta de energia).

Para que seja possível rodar vários processos simultaneamente e escaloná-los, é preciso dar prioridades a cada um. Deve haver pelo menos dois níveis de prioridade: nível de interrupção (processos que precisam de resposta muito rápido) e nível de relógio (processos periódicos). Além desses dois níveis, um terceiro nível é desejável para os processos que não precisam satisfazer deadlines.

O escalonamento de qual processo será executado primeiro é dependente da política adotada pelo RTOS. Essas políticas podem ser definidas em dois tipos: programação não preemptiva (espera um processo terminar para começar outro) e preemptiva (tira de execução um processo em detrimento de outro.). Exemplos de algoritmos de escalonamento são o *round-robin* (aloca-se uma porção de tempo para cada processo), *rate monotonic* (processo mais rápidos possuem maior prioridade, funciona somente com uso de CPU até 70%, o que ocasiona que alguns slots de processamento não sejam usados), shortest deadline first (quanto menor o deadline, maior a prioridade) e *first-in first-out*.

As tarefas de um RTOS tendem a ser geralmente especializadas e curtas, visto as limitações de SEs em geral. Isso ocorre pois todas as tarefas devem ser executadas no menor tempo possível. Tende-se a ter um grande paralelismo na execução de atividades, além das prioridades já descritas, onde as tarefas mais prioritárias interrompem as menos prioritárias e assumem o controle do processador. Todo esse cenário exige uma forma de sincronismo ou troca de informação entre todos os processamentos, sendo que as formas mais comuns são:

- Semáforos ou *flags*: são bits ou palavras para sinalização que uma tarefa está em uma seção crítica e não pode ser parada.
- Seção crítica: parte da execução do processo que está alterando dados e que se vários programas acessassem simultaneamente pode resultar em um estado inválido. Por exemplo, um processo tenta buscar um dado enquanto estou escrevendo e salva outro no lugar (estado inválido).
- Áreas de troca de mensagens, filas ou *buffers*: são as memórias temporárias que com auxílio dos semáforos, permitem a transferência de estruturas de dados maiores entre tarefas.

Em sua essência, o semáforo possui dois estados: bloqueado (alguém está executando algo) e desbloqueado (o processamento está liberado). O seu principal uso é para partilhar recursos, e para comunicação entre tarefas. Dado que uma tarefa está usando um recurso, ela pode bloquear um semáforo. Outra tarefa antes de acessar o mesmo recurso, verifica o estado do semáforo, se estiver bloqueado espera até que esteja desbloqueado ou executa outras subrotinas, caso contrário, pode utilizar o recurso normalmente.

Em sistemas reais, pode ocorrer o uso de vários semáforos, um para cada seção crítica do programa. Quando se trabalha com vários semáforos é importante que uma tarefa evite acessar várias seções críticas simultaneamente, pois pode-se criar um *deadlock* (um entrave que ocasiona uma espera infinita entre tarefas, que acabam por nunca terminem sua execução). Por exemplo, dada que uma tarefa A bloqueia um semáforo S1 e depois espera o desbloqueio do semáforo S2, enquanto uma tarefa B estiver bloqueada no semáforo S1 para desbloquear o semáforo S2, ambas as tarefas estarão paradas, pois a tarefa A espera a tarefa B terminar, que por sua vez espera a tarefa A terminar.

Pode ocorrer que uma tarefa com uma prioridade baixa bloqueou o semáforo, e uma outra tarefa com uma alta prioridade tente executar. Nesse caso, a tarefa com alta prioridade terá que esperar. Esse cenário recebe o nome de prioridade invertida. Um mecanismo usado para solucionar esse problema é aumentar a prioridade da tarefa que está bloqueando o semáforo, para que ela termine o mais rápido possível. Esse é o funcionamento básico de

um Mutex. Mutexes são uma adaptação dos semáforos. Eles são semáforos que incluem um mecanismo de herança de prioridade que ativa o mecanismo citado. Esse mecanismo é projetado para assegurar que tarefas com prioridades altas são mantidas no estado de bloqueio pelo menor tempo possível, além de minimizar a inversão de prioridades que ocorreu. *Hard RTS* devem ser projetadas de tal forma a evitar fortemente inversões de prioridade.

Enquanto os semáforos são melhores para sincronização entre tarefas e tarefas-interrupção, os mutexes são melhores para implementação de exclusão mútua (do inglês *MUTual EXclusion*). O uso de mutexes deve ser evitado no caso de interrupções, uma vez que eles são projetados para a comunicação de tarefas e não interrupções e que uma interrupção não pode ficar bloqueada esperando por uma tarefa que bloqueou o mutex.

Além do conceito de multitarefa (várias tarefas executando no mesmo processador), existe o conceito da técnica de multiprocessamento, onde as tarefas são repartidas entre múltiplos processadores, algo que é comum hoje em dia. Esse conceito é muito relevante dentro do contexto de RTS, uma vez que os possibilita executar mais tarefas em um espaço de tempo menor.

A multitarefa é importante para RTS uma vez que aumentar a eficiência do sistema como um todo, uma vez que um usuário não pode manter a CPU (processador) e os dispositivos de I/O ocupados o tempo todo. Alguns programas estão na memória concorrentemente e o sistema troca entre eles para maximizar a eficiência de processamento e diminuir o tempo ocioso da CPU. A multitarefa organiza as tarefas (código + dados) de uma maneira que a CPU sempre tenha algo para executar. Para aumentar ainda mais a eficiência, um subconjunto do total de tarefas do sistema é mantido em memória. Sempre que é necessário esperar algo, como de I/O por exemplo, o sistema operacional troca de tarefa.

Como os programas concorrentes rodam paralelamente, ou o escalonador pode interromper o processo a qualquer momento, surge então o problema dos recursos compartilhados por eles. Não basta que os programas estejam logicamente corretos, mas também é necessário considerar os outros processos. O desafio é como compartilhar recursos (mesmo programa acessando os mesmos dados) e como evitar que recursos sejam perdidos (programas diferentes acessarem a mesma porção de memória).

Como já dito, os SEs são sistemas de processamento de informação embutidos em um produto maior. Ele é um software integrado com processos físicos. O seu maior enfoque é gerenciar tempo e concorrência em sistemas computacionais. Apesar de estarem em grande parte do nosso dia, o sistema ciber-físico (Cyber Physical Systems - CPF) vem ganhando espaço. Eles são a integração da computação com processos físicos. Acaba-se por utilizar computação e embarcados para interagir com processos físicos, os quais adicionam novas propriedades ao sistema.

De forma grosseira, o CPF é a evolução dos sistemas embarcados, uma vez que é a convergência entre computação, comunicação e controle. Está um nível abaixo da Internet-das-coisas. Apesar de ser uma evolução, os sistemas embarcados e principalmente

os sistemas embarcados de tempo real são cruciais dentro do CPF, como mostra a Figura 6.

No contexto de Internet-das-coisas, os CPFs possuem um papel de destaque, uma vez que colaboram para construir sistemas inteligentes, como por exemplo, as cidades inteligentes, as quais podem ser vistas como sistemas ciber-físicos em ampla escala. As cidades inteligentes possuem sensores monitorando indicadores virtuais e físicos e por meio de atuadores, muda dinamicamente o ambiente urbano complexo de alguma maneira. Os CPFs podem ser usados na produção a âmbito industrial para construir arquiteturas baseadas na internet que facilitam o controle remoto de sistemas de produção *stand-alone*. Os CPFs vêm se destacando em vários campos, desde governos, organizações até indústrias de tecnologia, os quais direcionam seus esforços para integrar diversos sistemas em redes inteligentes, a fim de, por exemplo, controlar uma rede de energia ou o tráfego de trânsito para torná-lo mais seguro.

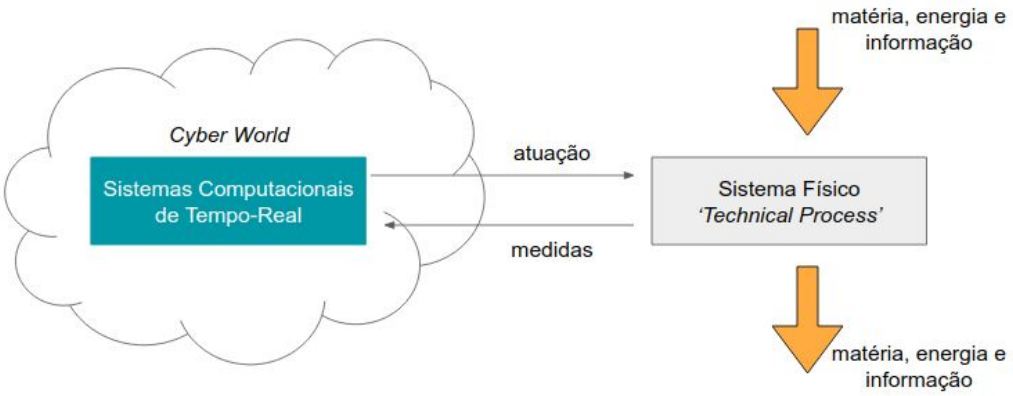


Figura 6: Exemplo de um CPF

Presente nas mais variadas aplicações do nosso dia-a-dia, os SE possuem aplicações onde o tempo de resposta deve ser cumprido, as quais recebem o nome de Sistemas de tempo real. O projeto destes sistemas de tempo real incluem a escolha pelo tipo de hardware e software. Ambos são dependentes do tipo da aplicação: *hard real time* a qual envolve um estudo mais detalhado e a comprovação no pior; ou *soft real time*, onde atrasos na resposta são aceitos.

Disponível em: https://github.com/plsilva/Files_DigitalSystems