Sistemas operacionais para Sistemas Embarcados

Aplicabilidade em sistemas embarcados em tempo real e não orientados a tempo real

Sistemas Embarcados (SE) vêm ganhando cada vez mais espaço no mercado mundial. Isso ocorre devido a criação de novos equipamentos, novas funções em equipamentos, concorrência do mercado e novos campos de aplicação (carros em geral, tvs, entre outros). O surgimento dos sistemas operacionais (SO) promoveram desenvolvimentos mais modulares e abstrações entre hardware e o código da aplicação.

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.

Os sistemas operacionais também são descritos por T Noergaard como sendo um conjunto de bibliotecas de software que atendem dois propósitos: (i) promover uma camada de abstração, o que faz com que a implementação de uma aplicação seja mais independente do hardware e facilitada por parte do desenvolvimento; e (ii) gerenciar os recursos disponíveis de hardware e software, de tal forma a garantir confiabilidade e eficiência na operação do sistema.

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. Esses tipos de sistemas são confundidos erroneamente com sistemas velozes, uma vez que em sistemas de tempo real, o mais importante é a previsibilidade do sistema e não a sua velocidade máxima.

Normalmente, os sistemas operacionais podem ser agrupados baseados no conceito de sistemas de tempo real, sendo dois grupos: SOs de propósito geral, ou GPOS (*General-purpose Operating System*), e os de tempo real, ou RTOS (*Real-Time Operating*

System). Ambos compartilham praticamente a mesma estrutura, conforme mostra a Figura 1, e possuem algumas semelhanças: proporcionam uma abstração do software da aplicação em relação ao hardware; gerenciamento de recursos de software e hardware e alguns níveis de multitarefas; além de oferecer serviços de baixo nível para aplicação. O uso dos GPOSs estão focados principalmente em computação de propósito geral e executam em sistemas como PCs, mainframes e microcontroladores de uso geral, enquanto o RTOS se adaptam melhor a sistemas embarcados com restrições de tempo real.

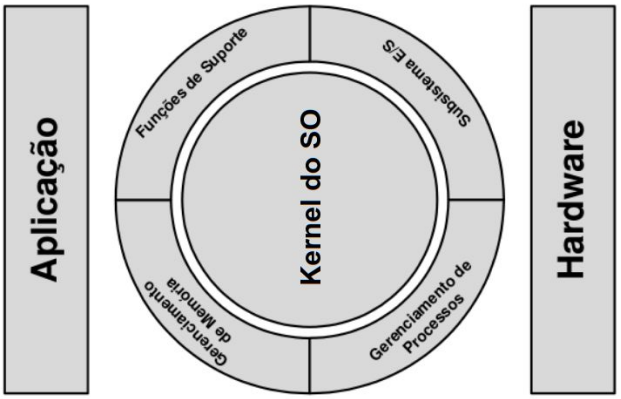

Figura 1: Relação entre a aplicação, hardware e o SO

Um sistema de controle de temperatura pode ser usado para mostrar a diferença entre os GPOS e RTOS. Este sistema hipotético possuirá algumas funções simples: (i) aquisição de dados da temperatura, (ii) manipulação dos dados (linearização/escala), (iii) processamento de cálculos de controle, e (iv) controlar o atuador. Por exemplo, aplicações envolvendo o controle da temperatura dentro de uma estufa podem ser feitas com GPOS, isso porque o máximo que ocorrerá se o sistema demorar a responder ou falhar 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 seja 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. Neste cenário hipotético, o uso de RTOS é necessário e imprescindível, pois eles apresentam uma confiabilidade maior para aplicações embarcadas, pois oferecem recursos de escalonamento de processos para atender requisitos de aplicação, políticas de escalonamento adaptados a sistemas de tempo real, uma portabilidade maior para diferentes plataformas e melhor desempenho quando comparado ao GPOS.

Sistemas digitais convencionais e os GPOS 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*. Existem até mais categorias, entretanto, as duas citadas são as mais gerais.

O sistema *hard real time* consegue comprovar teoricamente sua performance no pior caso. Em 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 e 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 populares 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 e vídeo-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.

Os RTOS possuem algumas desvantagens e vantagens em relação a sistemas construídos diretamente sobre o hardware. Segundo Ganssle, as principais desvantagens estão relacionadas a falta de suporte, alto consumo de memória e falta de certificação. Já as vantagens, segundo Tan e Anh, estão nos pontos:

- Otimização do desenvolvimento de software: o uso de RTOS proporciona ciclos menores de desenvolvimento, uma vez que já possui um gerenciamento de complexidade e vários outros recursos implementados.
- Melhor sincronização e robustez: há a possibilidade de troca de mensagens e sincronização sem corrupção do sistema.
- Gerenciamento de recursos: oferece uma camada de abstração para gerenciar recursos do sistema por meio de APIs (interfaces para aplicações).
- Gerenciamento de tempo: permite que tarefas sejam adiadas ou disparadas sem conhecer profundamente o funcionamento do hardware.

Apesar das diversas vantagens dos SOs para microcontroladores, eles são de difícil implementação em sistemas com recursos mais restritos. Para esses casos, outros tipos de arquiteturas, como a Superloop, ou *Foreground-Background,* e *Pooled Loop* foram propostas. Essas arquiteturas mais simples apresentam vantagens sobre os SOs, uma vez que possuem baixo consumo de memória e processamento, todavia, traz uma complexidade maior para o projetista. O uso dos SOs somente é viável para sistemas que no mínimo tenham 4Kb de memória RAM e processos com limite de tempo máximo para execução (deadlines) superiores a 1 ms. Essas condições evitam a sobrecarga do microcontrolador. O *Pooled Loop* consiste na execução de um único loop repetidamente, o qual simplesmente verifica entradas e saídas ou flags que indicam a ocorrência de eventos. O Superloop consiste em um conjunto de rotinas de tratamento de interrupções (ISR - *Interrupt Service Routine*), ou processos de tempo real (*Foreground*), e uma coleção de chamadas de módulos do sistema em loop infinito para executar as operações desejadas (*Background*). Para sistemas menores, o mais usado é o Superloop.

Apesar da simplicidade envolvendo o Superloop, o seu uso não é aconselhável para sistemas hard real-time com tarefas com deadlines curtos, isso porque ele pode ser um gargalo. Isso ocorre devido ao tempo de resposta de interrupções que serão executadas no nível *Foreground*.

Segundo Tan e Ahn, as vantagens envolta ao uso de RTOS em microcontroladores de grande porte (32 bits), a qual é comum, incentivou a expansão para dispositivos menores, estes com 16 e 8 bits. Independente do microcontrolador, tanto os RTOS e GPOS possuem um kernel, o qual é o responsável pelo escalonamento de tarefas, gerenciamento de recursos, sistemas de arquivos, protocolos de comunicação e outros recursos específicos de cada distribuição.

O escalonamento de qual processo será executado primeiro é dependente da política adotada pelo RTOS. O kernel e o próprio SO pode ser classificado de acordo com essa política. 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). Os kernels não-preemptivos seguem a ideia de que cada tarefa execute e, explicitamente, libere o controle da CPU. Para que o conceito de concorrência exista, os kernels desse tipo precisam ser executados frequentemente. Os kernels preemptivos são usados quando a resposta do sistema é importante. Nesse contexto, a tarefa de maior prioridade no ponto de ser executada deve receber o controle de CPU.

A execução de uma tarefa de maior prioridade nos kernels preemptivos e de tempo real são determinísticos, ou seja, é possível determinar o momento em que ela receberá o controle da CPU. Isto faz com que o tempo de resposta das tarefas sejam minimizados. Esse ponto do determinismo é algo que difere os kernels preemptivos dos não-preemptivos. Em kernels não-preemptivos não é possível prever quando uma tarefa de alta prioridade irá ser executada, uma vez que a tarefa de alta prioridade pode esperar um longo período até que a tarefa em execução libere a CPU. Os RTOS tendem a ser preemptivos.

Os escalonadores não preemptivos, também chamados de sistemas multitarefas cooperativos, tratam os eventos assíncronos pela ISR. Ela pode também liberar uma tarefa de alta prioridade para execução. Todavia, sempre retorna para a tarefa interrompida. A nova tarefa somente pode usar a CPU quando a tarefa em execução dá a permissão. Eles não usam semáforos.

A programação concorrente faz uso de técnicas de escalonamento. 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.

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 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.

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. Existem 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). 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 RTOS 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).

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.

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 e confiáveis, isto porque quanto menor for o grau de gerência por parte do ser humano, melhor.

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).

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.

Exemplos de algoritmos de escalonamento são o *round-robin* e algoritmos orientados à prioridade. O *round-robin* é utilizada normalmente para sistemas de tempo-compartilhado. Neste tipo de política, todas as tarefas prontas para serem executadas são alocadas em uma fila FIFO (*first-in-first-out*). Cada processo recebe uma porção de tempo igual para serem executadas. Caso a execução não termine no tempo alocado, ela é parada (liberando o processador) e colocada no fim da fila.

Já o s algoritmo orientados à prioridade não utilizam escalas pré-computadas. As decisões de escalonamento são feitas quando um processo é terminado ou liberado. Há uma atualização da listas de processos prontos para executar. Geralmente, as prioridades atribuídas aos processos são fixas. Os exemplos de algoritmos que implementam essa forma de prioridade estão:

- *Rate-Monotonic*: As prioridades são atribuídas de acordo com o período da tarefas. Quanto menor o período, maior a prioridade. Funciona somente com uso de CPU até 70%, o que ocasiona que alguns slots de processamento não sejam usados.
- *Deadline-Monotonic*: Atribui as prioridades de acordo com os deadlines das tarefas. Quanto menor o *deadline*, maior a prioridade. Prioridade fixa das tarefas. Quando o *Deadline-Monotonic* falha, o *Rate-Monotonic* também falha.
- *Early Deadline First*: Funciona semelhante ao *Deadline-Monotonic*, a diferença está no ponto em que usa prioridade dinâmica das tarefas.

- *Least Slack Time First*: Esse é uma exceção aos anteriores, uma vez que apresenta prioridade dinâmica para os processos. Ele atribui a prioridade baseado no tempo ocioso. O tempo ocioso é calculado a partir da diferença entre o *deadline* e o tempo de execução, este chamado de *slack*. Quanto menor o *slack*, maior a prioridade.

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 objetos (bits ou palavras) do kernel 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 binário 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, utiliza 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. Eles são semáforos binários com protocolos de exclusão mútua. 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 aumenta a eficiência do sistema como um todo, já 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 estados dos 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).

T Noergaard descreve que os sistemas operacionais podem ser divididos em três arquiteturas de desenvolvimento. Em geral, a diferença desses modelos está no ponto de desenvolvimento do kernel, além dos componentes incorporados ao SO, como drivers e serviços. Os modelos são:

1. Monolítico: Normalmente, os serviços e drivers são integrados aos SO, formando um único arquivo executável. Tipicamente, eles são mais difíceis de serem escalonados para pequenas aplicações, modificados e depurados devido a sua natureza grande, integrada e dependente. Devido a grande integração, esses kernels tendem a ser mais velozes, uma vez que apresentam menores tempos de comunicação e no chaveamento de componentes. Ex: Linux embarcado, o Jbed RTOS e o PDOS.
2. Em camadas: O SO é dividido em camadas hierárquicas, onde as camadas superiores são dependentes das funcionalidades implementadas nas camadas inferiores. O kernel inclui os drivers e serviços que formam um único arquivo. Ex: DOS-C, DOS/eRTOS e VTRX.
3. Microkernel: São SOs reduzidos ao mínimo de funcionalidades. Ele é composto somente pelo gerenciamento de processos e memórias. Eles não apresentam

serviços, somente mecanismo para o seu desenvolvimento. O conceito surgiu em 1970, todavia caiu em certo desuso para computação de propósito geral na década de 1990. Esse conceito sobreviveu em algumas áreas específicas, principalmente na indústria de sistemas embarcados. Normalmente, esse tipo de arquitetura tende a ser mais escalonáveis e depurável que as demais arquiteturas. Isso só é possível porque é possível adicionar dinamicamente alguns componentes. Além disso, essa arquitetura tende a ser mia seguro, já que a maioria das suas funcionalidades são independentes do SO e há espaços de memória separados para kernel (servidor) e aplicação (cliente). Apesar dessas vantagens, eles tendem a ser mais lentos que os demais. Ex: C Executive, VxWorks, CMX-RTX, Nucleus Plus, QNX e uc/OS.

Em sistemas embarcados, é muito comum o uso de SOs micro-kernel. Isso porque eles tem *footprints* pequenos (quantidade de memória principal que um programa usa ou referência enquanto está executando).

A construção dos RTOS devem seguir alguns padrões de construção, isso porque eles auxiliam no ataque a problemas de qualidade, custo e cronogramas, além de trazer uma confiabilidade aos sistemas construídos. Há padrões focados para diferentes objetivos. Entre os principais padrões, destacam-se:

- POSIX (*Portable Operating System Interface*): Padrão criado pelo IEEE baseado no Unix, com o objetivo de melhorar a probabilidade, a manutenção e o reuso de códigos de aplicação que utilizam serviços de SOs. Ex: QNX, VxWorks e o LynxOS.
- OSEK (Sistemas e interfaces abertas para eletrônica automotiva): Os seus objetivos são a redução de custos, do tempo de desenvolvimento e simplificação da integração de unidade individuais do sistema. Ela define um modelo de SO, comunicação, gerenciamento de rede e configuração. Esse tipo de padrão prevê a portabilidade entre processadores de 8 a 32 bits, além de configuração facilitada e alocação estática de tarefas e memória.
- APEX (*Avionics Application Software Standard Interface*): Criado pela ARINC (*Aeronautical Radio Incorporated*). O seu objetivo é estabelecer os detalhes de sistemas aviônicos, como segurança, robustez e técnicas padronizadas. Ele permite analisar softwares com requisitos de tempo real críticos em segurança, certificá-los e executá-los. Sua descrição é bastante detalhada, completa e formal, pois ela descreve detalhes desde como os sistemas computacionais devem trocar mensagens, até como o SO deve tratar, separar e escalonar tarefas.
- µITRON: Foi desenvolvida no Japão para padronizar sistemas operacionais de tempo real utilizados na construção de SE. As suas especificações foram implementadas em vários microcontroladores, de 8, 16 e 32 bits. Esse padrão visa descrever pontos como: (i) evitar excessiva virtualização do hardware (define os aspectos de diferentes processados e aspectos dependentes de aplicação); (ii) permitir otimização de aplicação (alteração específicas no kernel e a implementação interna para atender requisitos da aplicação); (iii) permitir otimização baseado no hardware (alteração específica do kernel e da implementação interna para otimizar o desempenho do sistema e implementações específicas); (iii) ênfase em facilitar o treinamento dos engenheiros de software; (iv) criar séries de especificações e

divisão em níveis; e (v) fornecer um grande número de funções (ao invés de limitar as funções primitivas fornecidas, esse padrão fornece outras diferentes funções).

Apesar das diversas vantagens dos SOs, o desenvolvimento de SE de pequeno porte tendem a usar a arquitetura *Superloop*, o qual é eficiente em sistemas simples e com restrições de memória. Porém quando há tarefas de sistemas *hard real-time*, o uso de RTOS mostra-se altamente necessário. O atendimento de requisitos de tempo-real são independentes da arquitetura utilizada (*Superloop*, RTOS), todavia, pode ser facilitado através do uso de RTOS. O processo da escolha de qual OS usar nos sistemas de tempo real é uma das etapas mais importantes do processo de desenvolvimento de SE. Não deve-se considerar apenas requisitos técnicos da aplicação no processo de design do hardware e software, mas também fatores comerciais como suporte, custos de curto e longo prazo, documentação, além da credibilidade e reputação do fornecedor.

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

https://www.oficinadanet.com.br/post/12781-sistemas-operacionais-o-que-e-escalonamento-de-processos

https://pt.wikipedia.org/wiki/Escalonamento_de_processos

http://ctd.ifsp.edu.br/~marcio.andrey/images/Escalonamento-Processos-IFSP.pdf

http://www.teses.usp.br/teses/disponiveis/18/18152/tde-09082011-081631/pt-br.php

file:///home/pedro/Desktop/Aplicabilidade%20SO%20em%20RTS.pdf