Extensão do sistema operacional *xv6* com o suporte a escalonamento de processos em tempo real baseado no *Earliest Deadline First*

Lais Borin, Marco Aurélio Spohn

***Resumo*—Escalonamento de processos em tempo real determina um algoritmo a ser seguido onde, sob restrições de tempo, deve atender as exigências impostas pelo sistema. Determinar um algoritmo satisfatório exige um alto grau de conhecimento e controle sobre o sistema. A análise comportamental dos algoritmos, contudo, realiza-se a partir de simulações ou em plataformas de tempo real. Este trabalho descreve a implementação do algoritmo *Earliest Deadline First* para escalonamentos de processos em tempo real. A implementação é realizada como uma extensão no sistema operacional *xv6*, um sistema operacional educacional e convencional. O comportamento do algoritmo foi obtido através de métricas padrões de desempenho, aplicadas a partir da execução de um cenário na plataforma em um único processador. A partir da execução, tem-se que o *xv6* apresenta capacidade para modelagem em tempo real, de forma a viabilizar a inclusão e testes de novos algoritmos para escalonamento de processos em tempo real.**

Palavras-chave*—*Sistema operacional de tempo real, escalonamento, xv6, Earliest Deadline First.

I. Introdução

Sistemas operacionais de tempo real, diferentemente de sistemas operacionais convencionais, devem ser capazes de efetuar execução sob restrições de tempo. É necessário que o sistema produza, dentro das restrições de tempo, resultados corretos em relação à computação de dados [1]. Uma falha, relativa à restrição de tempo ou à computação de dados, pode gerar danos a nível do sistema ou até mesmo danos físicos ao ambiente onde é executado, danos esses que podem incluir vidas [2].

As restrições possuem, na visão do sistema operacional de tempo real, um nível de urgência respectivo. Conhecer o nível de urgência de cada restrição infere em um alto grau de conhecimento sobre o ambiente em que se aplica a abordagem de tempo real. Contudo, é necessário que o sistema adote uma política de seleção que contemple todas as restrições sem cometer erros de computação de dados, ou seja, o *escalonador* deve implementar um *algoritmo de escalonamento* que seja capaz de atender as restrições impostas e gerar resultados corretos [1].

Lais Borin é graduanda em Ciência da Computação pela Universidade Federal da Fronteira Sul (UFFS), campus Chapecó-SC. (borin.lais@gmail.com)

Marco Aurélio Spohn possui doutorado (*PhD*) em *Computer Science* pela *University of California at Santa Cruz* (UCSC) e é Professor no Curso de Ciência da Computação na Universidade Federal da Fronteira Sul (UFFS). (marco.spohn@uffs.edu.br)

Os esforços, dentro da área de escalonamento de processos em tempo real, voltam-se para a busca de *algoritmos de escalonamento*. Devido ao fato de ser necessário um amplo conhecimento acerca do ambiente, os estudos sobre a aplicação dos algoritmos de escalonamento acabam por serem restritivamente específicos ao cenário adotado. Como por exemplo, observa-se a aplicação de algoritmos de escalonamento em larga escala em um sistema operacional de tempo real e de código aberto [3]. Por outro lado, encontra-se avaliações de algoritmos de escalonamento de forma teórica e simulada [4].

Contudo, a avaliação de forma teórica ou simulada exclui custos e atrasos como, por exemplo, tempo de troca de contexto e acesso a dados. Assim, a avaliação dos algoritmos de escalonamento é realizada de maneira restrita, com foco apenas à definição do algoritmo em estudo. No que se refere às avaliações em ambientes reais, é notável um número reduzido de trabalhos na literatura. Por se tratar de uma área predominantemente industrial, de plataformas fechadas, não há o interesse de divulgação do comportamento e do desempenho de algoritmos em funcionamento.

Com o objetivo de estimular os estudos em arquiteturas reais e fornecer uma nova ferramenta de código aberto para a comunidade, este artigo apresenta uma extensão ao sistema operacional *xv6* com o suporte a escalonamento de processos em tempo real empregando o algoritmo *Earliest Deadline First*.

II. Sistemas de Tempo Real

Um sistema de tempo real é diferenciado dos sistemas convencionais por suas restrições de tempo. As restrições de tempo dependem diretamente do ambiente no qual o sistema é aplicado. Contudo, conhecer as restrições implica em prever como o ambiente funciona e a quais falhas e sobrecargas o sistema estará exposto. Deve-se analisar, assim, de forma clara e realista, todo e qualquer atraso que o sistema venha a sofrer quanto a questão de limite do hardware e da linguagem utilizada [5].

Sistemas de tempo real são classificados em *hard* ou *soft* [2]. Caso o não cumprimento dos prazos gerar erros fatais à aplicação, danos físicos e até mesmo a perda de vidas, o mesmo é classificado como *hard*. No caso onde a perda dos prazos gerar apenas atrasos, ou seja, se o resultado ainda é útil para aplicação e o atraso gerar apenas degradação do desempenho, o sistema é classificado como *soft*.

Nesse sentido, Buttazzo [1] considera que, ao implementar um sistema de tempo real e deparar-se com situações críticas,

uma aplicação deve ser capaz de garantir ao menos as seguintes propriedades:

- *Pontualidade*. O *kernel* do sistema operacional deve implementar mecanismos que sejam capazes de produzir resultados corretos não somente em questão de valores, mas também especificamente em questão de tempo;
- *Previsibilidade*. Espera-se que o sistema proporcione uma análise viável, para que assim seja possível identificar e prever as consequências de qualquer decisão gerada pelo escalonamento. As restrições de tempo deverão ser garantidas antes de colocar o sistema em funcionamento, para que, quando identificada uma exceção, o sistema possa traçar rotas alternativas de forma a contorná-la;
- *Eficiência*. É necessário que o sistema saiba utilizar todo e qualquer recurso disponível, visto que na maioria das aplicações apresentam-se de forma limitada;
- *Robustez*. Situações de sobrecarga deverão ser gerenciadas com mudanças de comportamento de modo que o sistema se mantenha estável;
- *Tolerância a falhas*. O sistema deve ser implementado considerando as falhas de software e hardware para que as mesmas não afetem a execução;
- *Manutenibilidade*. O desenvolvimento da arquitetura deve ser feito de forma a reduzir dependências, ou seja, recomenda-se o uso de estrutura modular para que seja de fácil manutenção;

Tem-se que a velocidade para a produção de uma resposta, bem como a capacidade em atender e lidar com todas as requisições dentro dos prazos estabelecidos, caracteriza a *capacidade de resposta* de um sistema [2]. Esta medida é obtida ao considerar quanto tempo o sistema leva para reconhecer que houve uma solicitação e, após isso, quanto tempo demorará para atendê-la.

A *capacidade de resposta* de um sistema de tempo real influencia diretamente no *deadline* de cada processo. Uma vez que é necessário obter esta medida, fica evidente o nível de conhecimento necessário sobre o sistema para torná-lo factível. Portanto, cabe ao usuário informar e designar as características de forma correta.

Contudo, o controle do usuário sobre um sistema de tempo real ressalta mais uma diferença quanto aos sistemas convencionais. Quando se trata de tempo real, o usuário deverá estar apto para realizar uma classificação acerca dos processos. Será necessário, entre outras ações, identificar e diferenciar por relevância cada processo, com a finalidade de informá-las ao escalonador [2].

A. Escalonamento de Processos em Tempo Real

O escalonador de processos de um sistema de tempo real é classificado como o responsável por implementar uma política de escalonamento [1], [5]. Política está que deverá, através de suas considerações, decidir qual processo receberá o direito ao uso de um determinado processador, além de definir em qual momento da execução e por quanto tempo o processo possuirá o direito ao recurso. Todo processo que solicita um recurso ao sistema deverá recebê-lo dentro do seu tempo limite pré definido, chamado de *deadline*. Ou seja, o escalonador de tempo real deve assegurar que todas as exigências feitas serão atendidas dentro dos prazos estabelecidos pelos processos.

Assim, dado um conjunto de processos S, este é dito ser *factível* para um dado sistema de tempo real se, e somente se, há ao menos um algoritmo de escalonamento capaz de cumprir, para todas as combinações possíveis de processos, o *deadline* imposto por cada elemento pertencente a S [6]. É necessário, portanto, moldar o ambiente com a finalidade de encontrar um algoritmo de escalonamento que melhor atenda o requisitado, ou vice-versa.

Compreende-se como modelo de processos o periódico e o esporádico [1]. O modelo periódico se caracteriza por processos que realizarão solicitações com intervalos de tempos iguais. Quando se trata do modelo esporádico, os processos solicitarão o uso de um processador em intervalos de tempo irregulares. Considera-se ainda que um processo pode ser interpretado como um *job*. Um *job* pode ser visto como uma divisão das funções de um processo, divisão essa também referenciada como *threads* pela literatura.

É significante a quantidade de abordagens sobre as classes de algoritmos de escalonamento. Visto que as classes representam, de maneira geral, o cenário onde os escalonadores serão aplicados, bem como particularidades adotadas na implementação. Contudo, conclui-se que não há consenso na literatura sobre uma classificação exata quanto a nomenclatura e ambientes [5].

Porém, de uma maneira ampla e generalista, Buttazzo [1] identifica as seguintes classes como principais:

- *Off-line ou online*.
  - Um escalonamento é dito *off-line* quando os processos são definidos antes do sistema entrar em execução, define-se assim um escalonamento *em tempo de projeto*;
  - Um escalonamento é dito *online* quando a todo início ou término de processos as decisões de escalonamento são tomadas, ou seja, o escalonamento é realizado *em tempo de execução* do sistema.
- *Estático ou dinâmico*.
  - Um escalonamento estático considera atributos fixos que são atribuídos aos processos antes da execução e permanecerão inalterados até seu fim;
  - Um escalonamento dinâmico atribui os parâmetros conforme computações que ocorrem durante a execução do sistema e mudam com o decorrer do tempo.
- *Preemptivo ou não preemptivo*.
  - O escalonamento é preemptivo se um processo que está usando um processador pode ser interrompido por outro processo em qualquer momento de sua execução, conforme a política de escalonamento em uso;
  - O escalonamento é dito não preemptivo quando as decisões do escalonador são postas em prática após o termino da execução de um processo. Assim, o

processo não perderá o direito ao uso do processador após tê-lo recebido e o liberará somente ao término de sua execução.

- *Ótimo ou heurístico*.
  - Um escalonamento é ótimo quando, para um dado conjunto de processos escalonável, for capaz de minimizar algum custo que está relacionado ao conjunto de processos. Ou ainda, se a política implementa um escalonamento factível para o conjunto de processos;
  - Um escalonamento é dito heurístico quando suas decisões de escalonamento são baseadas em uma função heurística, o que faz o escalonamento tender ao ideal, porém sem garantias.
- *Melhor esforço*.
  - Um escalonamento é dito ser de melhor esforço quando alcança o melhor possível para uma dada abordagem.

Estabelecido um conjunto de processos, um conjunto de processadores e um conjunto de recursos, determinar um escalonamento dentro desse contexto é um problema NP-completo [7]. Formalmente, determinar um escalonamento significa então, distribuir o conjunto de processadores e o de recursos de forma que o conjunto de processos possa ser atendido dentro de todas as suas restrições impostas. Portanto, caracteriza-se assim o que é chamado de *problema de escalonamento* [2].

III. *Earliest Deadline First*

Proposto por Liu e Layland [8], o algoritmo *Earliest Deadline First* (EDF) é amplamente aplicado e abordado pela literatura ([9], [10], [11], [12], [13], [14]). A escolha do EDF para a implementação neste trabalho parte do princípio de que o mesmo está correto a nível teórico e prático, dado os estudos apresentados pela literatura.

A proposta original do EDF consiste em um algoritmo de prioridade dinâmica, capaz de obter uma total utilização dos recursos em um único processador. Segundo os autores, não havia um número significativo de pesquisas voltadas para sistemas *hard* e, as que possuíam bons resultados, funcionavam apenas em situações irreais. Portanto, a criação do algoritmo justifica-se, segundo Liu e Layland [8], pela *"necessidade de abordagens mais sistemáticas para o desenvolvimento de software"*.

Segundo o que é apresentado, o algoritmo para o EDF assume cinco suposições:

1) Os processos possuem um *deadline* e são periódicos, onde o intervalo de requisição se mantêm constante;
2) Cada processo deve ser executado até o fim, antes que uma próxima requisição possa ocorrer;
3) Os processos não dependerão da inicialização ou da finalização de outros processos para serem postos em execução; ou seja, são independentes entre si;
4) O tempo que o processador leva para executar um dado processo será o mesmo em todas as vezes que for necessário executá-lo (desconsiderando interrupções);
5) Rotinas de recuperação de falhas ou de inicialização do sistema são consideras aperiódicas. Quando suas execuções forem necessárias, o processador será entregue às rotinas e os processos periódicos serão deslocados.

Contudo, o EDF atribuirá prioridade a partir do *deadline* de cada processo. A maior prioridade será concedida ao processo que possuir o *deadline* mais próximo do tempo atual de execução; ou seja, ao processo que possuir o *deadline* mais próximo do fim. Por sua vez, a menor prioridade será dada ao processo que possuir o *deadline* mais distante do tempo atual de execução [12]. Em outras palavras, o EDF busca atender primeiramente os processos que estão para perder o tempo limite de execução.

Segundo a proposta, tem-se que um conjunto S com n processos será escalonável pelo EDF se, e somente se, a Inequação 1 for verdadeira:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{1}$$

e se S não pode ser escalonado pelo EDF, então não será escalonável por qualquer outro algoritmo [8]. A razão entre o tempo de computação e período representa a utilização da CPU para um determinado processo. Portanto, o somatório de utilização do conjunto de processos representa a utilização total. Se a utilização total ultrapassar o valor 1 significa que a capacidade da CPU foi exaurida, pois os processos requisitam mais do se pode processar. Consequentemente, para ser escalonável, um conjunto de processos deve possuir a utilização menor ou igual a 1, independentemente do algoritmo de escalonamento utilizado.

Quando um escalonamento com EDF é implementado, o sistema não possuirá conhecimento prévio sobre os processos, bem como de quantos serão executados. Assim, a atribuição das prioridades será feita de forma dinâmica com o decorrer do tempo, na medida em que os processos fazem requisições. Fato esse caracterizado como uma desvantagem [10], dado que a atribuição de prioridade será realizada sempre que um processo requisitar o processador. Portanto, é notável que o cálculo acontecerá muitas vezes no decorrer da execução.

Segundo Buttazzo [10], apesar de não apresentar um bom desempenho em situações de sobrecarga, o EDF comporta-se de maneira eficiente em sistemas embarcados que trabalham com recursos limitados. Considera ainda que demonstra um bom desempenho em aplicações multimídia, onde há reserva de recursos, pois é necessário gerar garantias sobre a qualidade do serviço prestado.

IV. SOBRE O *xv6*

Segundo a página oficial [15], o *xv6* foi desenvolvido em 2006 pela turma de engenharia de sistemas operacionais do MIT. Derivado do *Unix V6*, o *xv6*, apesar de ser educacional, é um sistema operacional convencional real. Sua implementação se deu pelo fato da arquitetura e abordagens utilizadas pelo *Unix* V6 terem se tornado obsoletas para a época. Com o intuito de possuir uma arquitetura moderna como objeto de estudo, implementou-se o *xv6*. Desenvolvido em padrão ANSI C e baseado no *Unix V6*, foi modelado para executar em multiprocessadores baseados na arquitetura *x86* da *Intel*.

Apesar de ser um sistema operacional real, o *xv6* não implementa todas as funcionalidades de um sistema usual como, por exemplo, o *Linux* ou o *Windows*. Desenvolveu-se somente o essencial para tornar possível a instanciação do sistema. Portanto, por não possuir interface gráfica, o *xv6* é executado a partir de um ambiente de emulação, frequentemente, sob a recomendação da documentação, pelo *Qemu*. A execução de uma aplicação é feita através do *prompt* de comando do *xv6*.

O *kernel* do *xv6* é implementado na forma tradicional, baseado em *Unix*. Ou seja, através de chamadas de sistema, os processos podem acionar um serviço no *kernel* quando necessário. Assim, a execução de um processo altera entre executar no espaço de usuário e executar no espaço de *kernel*.

Para o *xv6*, um processo é interpretado como um espaço de memória que contém instruções, dados e pilha. Um processo pode ser criado através da chamada de sistema *fork*. Toda vez que uma chamada *fork* é feita, um novo processo filho, idêntico ao pai, é criado. Assim, o processo criado conterá exatamente o mesmo conteúdo de memória que o processo que o criou. Na instanciação do sistema, o primeiro processo criado é o *init* que será o processo com posição de maior hierarquia no *xv6* e, consequentemente, pai de todos os processo que venham a ser criados a partir dele.

É apresentado, no diagrama da Fig. 1, os estados que um processo pode assumir no *xv6* durante a execução, bem como as funções responsáveis por tais mudanças. Um processo, inicialmente, tem seu estado em *UNUSED*. Através da chamada da função *allocproc*, uma busca por processos inutilizados é iniciada. Ao encontrar um processo em *UNUSED*, um espaço de memória é alocado e, a partir disso, o estado do processo passa a ser *EMBRYO*, indicando aptidão para ser utilizado. Com um espaço de memória alocado para um processo, pode ser realizada com sucesso uma chamada de *fork*. O *fork* é o responsável em sinalizar para o escalonador que o processo foi criado e está apto para ser posto em execução, portanto, o qual tem seu estado alterado para *RUNNABLE*.

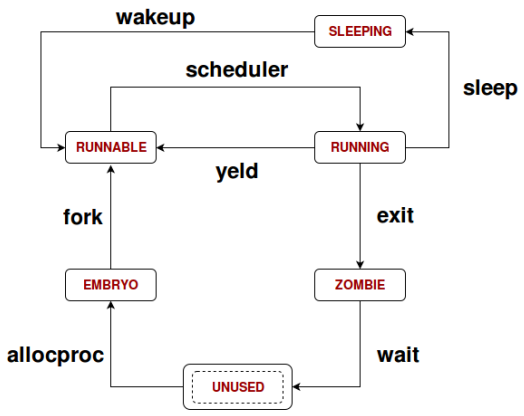

Fig. 1. Diagrama de estados dos processos no *xv6*

Para ser posto em execução, o processo deverá ser escolhido pelo escalonador. Assim, receberá o uso do processador e passará a ter seu estado igual a *RUNNING*. Se o processo que está em execução sofrer uma interrupção de relógio, a função *yield* é acionada para modificar o estado do processo de *RUNNING* para *RUNNABLE* novamente, de modo a conceder o uso do processador a outro processo. Porém, se o processo que está com o estado em *RUNNING* realizar uma chamada de *exit*, finalizará sua execução e terá seu estado alterado para *ZOMBIE* e, se finalizar antes da conclusão de seus processos filhos, seus filhos permanecerão órfãos até serem entregues ao processo *init*. Os processos com o estado em *ZOMBIE* voltarão para o estado de *UNUSED* através da chamada de *wait*, que é feita por um processo pai que espera a conclusão de um processo filho para prosseguir. Por fim, um processo que está em execução pode também assumir o estado de *SLEEPING*, alterado pela chamada de *sleep*. Com a chamada de *wakeup* o processo que está dormindo passa a estar apto a ser executado, portanto seu estado será alterado para *RUNNABLE*.

O *xv6* utiliza o mecanismo de tabelas de páginas para endereçamento de memória. A memória física é indexada em endereços físicos, portanto, a realização de mapeamento entre endereços físicos e lógicos é atribuída como responsabilidade ao *hardware* do *x86*. Apesar de não ser tão sofisticado, o hardware de paginação do *xv6* proporciona mecanismos para proteção de acesso à memória.

O sistema de arquivo implementado pelo *xv6* considera duas abordagens: uma por *pipes* e outra por *inodes*. O esquema implementado por *inodes* se mantém idêntica a abordagem utilizada ao longo dos anos em sistemas *Unix*. Porém, o sistema de arquivos do *xv6* não lida de forma satisfatória a falhas de disco. Caso ocorra uma falha durante alguma operação em disco, o sistema simplesmente entra em modo *panic*.

A. *Escalonamento*

O *xv6* utiliza a abordagem de multiplexação. Portanto, cada processo terá a impressão que possui o processador somente para si. O escalonador faz uso de interrupções de relógio para forçar um processo a suspender sua execução. A cada 100 ms, o escalonador do *xv6* irá interromper o processo em execução e escalonar um novo processo para ser posto em seu lugar. Ao realizar a interrupção, o contexto do processo é salvo. Ao entrar execução novamente, o processo continuará a executar e levará em conta tudo o que já havia feito, como se não tivesse sofrido uma interrupção.

Os processos estão dispostos em um vetor *proc*, de tamanho fixo igual a *NPROC*, que é interpretado pelo escalonador como uma fila circular. Ressalta-se que o *xv6* assume, também com o mesmo nome, *proc* como o processo corrente, o que não deve ser confundido com o vetor *proc* de processos, declarado dentro da estrutura de *ptable* e que contém todos os processos possíveis a serem instanciados durante a execução.

O *xv6* implementa *round-robin* como a política de escalonamento. Assim, a partir do vetor *proc* de processos, a função do escalonador será percorrer a fila até encontrar um processo p que esteja pronto para ser executado, ou seja, com o estado igual a *RUNNABLE*. Como pode ser observado na Fig. 2, a função *scheduler* seleciona um processo p *RUNNABLE*. Posteriormente fará de p o processo corrente, atualizará seu estado

pra *RUNNING* e, finalmente, realizará a troca de contexto, colocando-o de fato em execução.

```
1  scheduler(void){
2    for(;;){
3      ...
4      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
5        if(p->state != RUNNABLE)
6          continue;
7        ...
8        proc = p;
9        p->state = RUNNING;
10       swtch(&cpu->scheduler, proc->context);
11       ...
12     }
13     ...
14   }
15 }
```

Fig. 2. Implementação do escalonamento por *round-robin* no *xv6*

Caso um processo fique dependente da execução de seus filhos ou de uma entrada de dados, por exemplo, é possível que o mesmo desista do processador. O *xv6* permite que o processo durma enquanto espera por suas dependências, assim é possível escalonar outros processos para execução, sem desperdiçar o tempo do processador com um processo inativo.

V. Modificações gerais realizadas no XV6

Para ser possível a representação de um comportamento em tempo real foi necessária a definição de um mecanismo para medição do *tempo*. O *tempo* na versão em tempo real do *xv6* é representado e mensurado via um contador externo. Se o contador de tempo fosse de responsabilidade do *xv6*, haveria a necessidade de criação de um processo encarregado por contar. Pelo fato dessa opção acarretar em uma execução contínua desse processo por parte do sistema, optou-se por fazer uso da instrução *rdtsc*, disponível na linguagem *assembly* do *x86*.

A instrução *rdtsc* retorna, em uma variável de 64 bits, o *Time Stamp Counter* (TSC), que é a quantidade de ciclos de *clock* que o processador acumulou desde sua inicialização. Por ser um contador incrementado na granularidade de ciclos de *clock*, foi necessário converter o valor do contador para trabalhar em proximidade à faixa de milissegundos. Portanto, a conversão é relativa e dependente da quantidade de ciclos de *clock* da CPU onde o sistema é executado. O tempo, já em ms, é obtido no *xv6* através da função *tick*. Dessa forma, é possível verificar o tempo corrente da execução do sistema.

Além do mecanismo para controle do tempo, foi necessário desenvolver mecanismos para o gerenciamento dos processos que estão aptos a entrar em execução. Uma lista foi criada a fim de armazenar somente os processos que possuem seus estados iguais a *RUNNABLE*. A lista, que no *xv6* é um vetor *queue* de processos com tamanho *NPROC*, é gerenciada de maneira específica ao algoritmo de escalonamento adotado. Para a implementação, foi criada uma biblioteca adicional, chamada de *heap.h*, que contém todas as funções necessárias para a manipulação da lista.

Para a abordagem em tempo real, foi essencial adicionar novos atributos aos processos do *xv6*. Além dos já existentes, o *deadline* e o tempo de computação foram adicionados à estrutura do processo. Assim como, para realizar a validação e a coleta de dados na execução dos algoritmos, o tempo de chegada, o tempo de início de atendimento e o tempo de finalização são coletados e, portanto, sua inserção na estrutura de processos foi necessária.

Contudo, a chamada de sistema *fork* foi alterada para ser possível informar o *deadline* e o tempo de computação do processo para o sistema. A cada criação de processo é necessário saber, de antemão, os valores dos atributos e informá-los através do *fork*. Portanto, torna-se intrínseco estimar o tempo de computação para cada processo. O método utilizado para estimar os tempos de computação dos processos será abordado nas próximas seções.

Todas as modificações para a implementação do algoritmo foram realizadas dentro do escopo de diretivas de compilação. Assim, todo o código está presente em uma única versão, o que torna possível a compilação em versões separadas contendo ou EDF ou o escalonamento em *round-robin*, nas configurações originais da versão convencional do *xv6*. Para executar com o algoritmo de tempo real, a constante *RT* deve ser definida no arquivo *types.h* com valor igual a um. A não definição da constante resulta em comportamento convencional. O código fonte da versão em tempo real do *xv6* está disponível no repositório *GitHub* com o nome de *RT-xv6*[1].

VI. Implementação do EDF

Para a implementação do EDF foi necessário interpretar a lista de processos prontos como uma *heap-min*. Uma *heap-min* realiza a ordenação colocando na posição inicial o menor valor dentre os demais. Na versão com o EDF, a ordenação da lista é feita a partir da quantidade de tempo restante para a execução de um determinado processo. Assim, o processo que possuir o menor tempo restante para ser executado será adicionado ao início da lista.

O algoritmo de escalonamento consiste, então, em verificar se há processos na lista de prontos e, caso haja, realizar a troca de contexto do processo corrente com o processo que está no início da lista. O processo escalonado é removido da lista e o processo que estava executando tem seu estado alterado para *RUNNABLE* e, consequentemente, é adicionado à lista de prontos. As inserções são feitas sempre ao fim da lista e o algoritmo da *heap* encarrega-se em ordenar os processos a partir da modificação.

Como o processo considerado para o escalonamento sempre estará na posição inicial, dada a remoção, o algoritmo da *heap* fará do último o primeiro processo da lista e a reorganizará até que o processo com menor prazo esteja na posição inicial. Um exemplo da manipulação da lista é dado na Fig. 3. Dado quatro processos prontos em (1), o escalonador seleciona o processo τ_0, na posição inicial da lista. No passo (2), o processo τ_0 é selecionado para a execução, assim sua posição na lista é substituída pelo processo τ_3, conforme o passo (3). Após isso, a lista é reordenada conforme os menores prazos, a partir do três processos restantes.

Contudo, o escalonamento com o EDF ocorre sempre que houver uma inserção na lista. Portanto, a cada novo processo

[1]Disponível através do endereço https://github.com/laisborin/RT-xv6.git

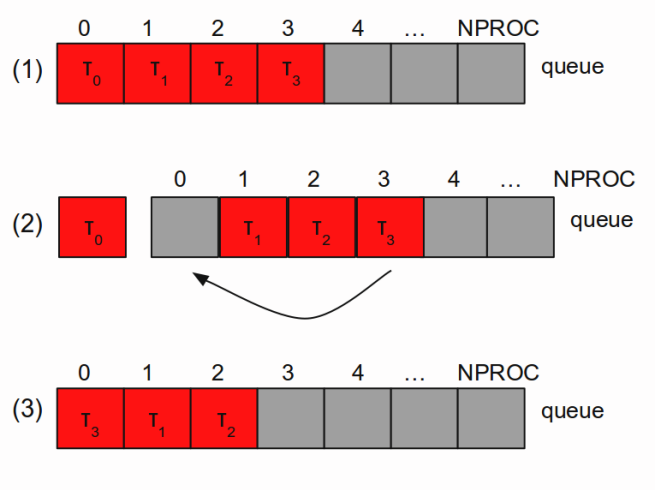

Fig. 3. Manipulação da lista de prontos.

criado a partir da chamada de sistema *fork*, uma interrupção é gerada e o processo na posição inicial da lista é escalonado para a execução. O mesmo ocorre quando o processo assume estado igual a *RUNNABLE*, a partir da chamada de *wakeup*.

Para representar a execução do algoritmo de forma fiel, conforme suas definições, adotou-se o método *dormir-acordar*. Os processos são criados uma única vez e, com o decorrer do tempo de execução, após realizarem seu processamento, são postos para dormir com a chamada de sistema *sleep*. Portanto, o tempo que o processo permanece dormindo é igual ao seu período. Com isso, a implementação de um mecanismo para controle da periodicidade foi suprida com o uso dos recursos já existentes no *xv6*.

Para estimar o tempo de computação de um processo no *xv6*, um único processo foi criado e posto em execução. A computação a ser realizada pelo processo criado consiste em um laço que é executado n vezes. Ao ser posto em execução, o tempo de início foi registrado e, ao fim, computou-se a diferença entre os instantes de forma a obter o tempo gasto para executar o laço n vezes. Portanto, ajustou-se n até que o tempo exigido para a execução ficasse próximo de $1\ ms$. Assim, quando um processo é criado, seu valor de computação informado será multiplicado por n para assumir um tempo de computação semelhante ao que é esperado.

Apresenta-se na Fig. 4 a função que é atribuída a todos os processos criados. A função *work* recebe um período t e um tempo de computação c. Da linha 4 à 7, executa-se a computação e o procedimento de dormir. Observa-se que na linha 5 a instrução *for* será executada $c * n$ vezes, de forma a representar o tempo (em ms) estimado para a computação do processo. O procedimento é realizado 1000 vezes, ou seja, o processo requisitará 1000 vezes, conforme seu período, o uso do processador. Executa-se este procedimento com o intuito de capturar o comportamento total do conjunto de processos durante a execução.

```
1  void work(int t, int c){
2    int i, j;
3
4    for(j = 0; j < 1000; j++){
5      for (i = 0; i < n*c; i++);
6      print(2);
7      sleep(t);
8    }
9
10   exit();
11 }
```

Fig. 4. Função atribuída aos processos.

VII. Coleta de dados

Para realizar a validação bem como a descrição do comportamento, tornou-se imprescindível a coleta de determinados dados durante a execução dos algoritmos. Inicialmente, além dos atributos já acrescentados à estrutura do processo, foi inserido também um contador de troca de contextos. Os dados do processo a serem coletados são inseridos na estrutura de dados *statistic*, adicionada ao arquivo *proc.h*. Um vetor de *statistic* com tamanho *TAM* é criado em *proc.c* e é alimentado durante a execução do sistema.

De forma a realizar a coleta de dados sem ocasionar interferências significativas na execução dos algoritmos, criou-se a função *getData(p)* e uma chamada de sistemas *print(int)*. A função *getData(p)* recebe como parâmetro um processo p e coleta os valores contidos nos atributos de p, armazenando-os em *statistic*. A chamada de sistema *print(int)* recebe como parâmetro um inteiro de valor igual zero, um ou dois. Com *int* igual a zero, *print(int)* remove todos os dados contidos no vetor *statistic*; Com valor igual a um, *print(int)* exibe em tela todas as informações contidas em *statistic* e com valor igual a dois, força a chamada da função *getData(p)* para a coleta de dados do processo corrente.

Com a implementação desses recursos, torna-se possível capturar o comportamento do sistema em momentos específicos da execução. A coleta de dados é realizada antes da chamada de *sleep* dentro da função *work*, que é o ponto fim da execução do processo. Como faz-se uso do mesmo processo, quando o mesmo entrar em execução novamente, os valores de seus atributos serão sobrescritos pelos da nova execução. Portanto, ao fim da execução de todos os processos criados, realiza-se a chamada de *print(1)* para obter acesso aos dados coletados.

VIII. Execução do EDF no xv6

A partir da implementação, tem-se o EDF *estáticos* nos atributos dos processos, *online* e *preemptivo* no escalonamento, onde, para o conjunto de processo, o EDF é *dinâmico*. Com a finalidade de apresentar os dados de forma coerente e, a partir disso, obter as características comportamentais reais do algoritmo, a execução foi realizada a partir da execução de 3 processos. O conjunto de processos adotado é escalonável pelo EDF a partir da inequação 1 e apresentado na Tabela I. Segundo a Tabela I, cada processo τ_i é definido por um

deadline D_i, um tempo de computação C_i e um período T_i. Assume-se T_i como o tempo mínimo entre as solicitações de uso do processador por parte do processo.

Tabela I. PROCESSOS UTILIZADOS PARA AVALIAÇÃO DO EDF.

τ_i	C_i	T_i	D_i
τ_1	4	14	10
τ_2	4	16	16
τ_3	7	40	20

Os dados utilizados para a avaliação foram obtidos a partir da execução do EDF sob a ótica de um único processador. A avaliação, por sua vez, considera todos os custos e os atrasos acarretados pela execução no *xv6*. Portanto, todo tempo gasto para realizar a troca de contexto e a criação de processos, por exemplo, é considerado dentro das métricas utilizadas.

Conforme o apresentado na Fig. 5, tem-se os primeiros 25 atendimentos referente à execução do conjunto de processos da Tabela I. A curva *Conclusão* no gráfico da Fig. 5 representa o tempo que cada processo levou para ser concluído, dado o tempo de chegada do mesmo. Observa-se que a curva está dentro dos limites de *deadline* impostos (curva *D*). Portanto, evidencia-se que o *xv6*, dotado de escalonamento em tempo real a partir do EDF, é capaz de realizar a computação (curva *C*) do conjunto de processos sem perda de prazos.

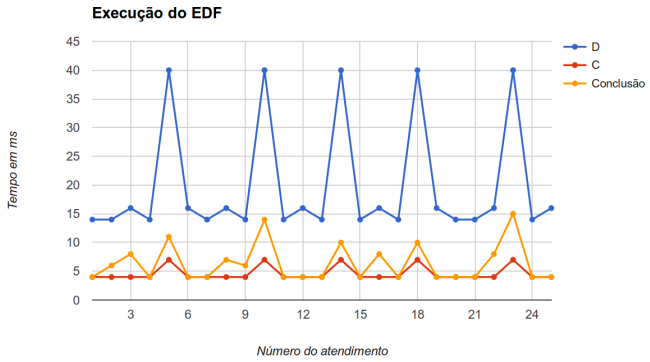

Fig. 5. Gráfico da execução do conjunto de processo com o EDF.

Em questão de capacidade de resposta, o *xv6* apresentou atrasos nulos para o conjunto de processos em questão. Mas segundo a execução apresentada na Fig. 5, nota-se que houve atrasos no que se refere à latência. O cálculo da latência consiste em determinar, dado o tempo do primeiro escalonamento do processo, qual o respectivo intervalo para que o mesmo seja concluído. Assim, no EDF, as latências serão menores nos casos onde os processos que possuírem o menor tempo restante antes de perder o prazo, com isso os menores *deadlines* darão maiores prioridades aos processos. Este fato é observado na Fig. 6, onde o alto valor de *deadline* de τ_3 faz com que em média o mesmo demore mais para ser executado por completo que os demais processos.

Na prática, a latência sofrida por τ_3 é ocasionada pelas trocas de contextos com os demais processos. Portanto, quando τ_3 está em execução e ocorre a chegada de τ_1 ou τ_2, pelo alto valor de *deadline* associado, as chances de τ_3 ser suspendido

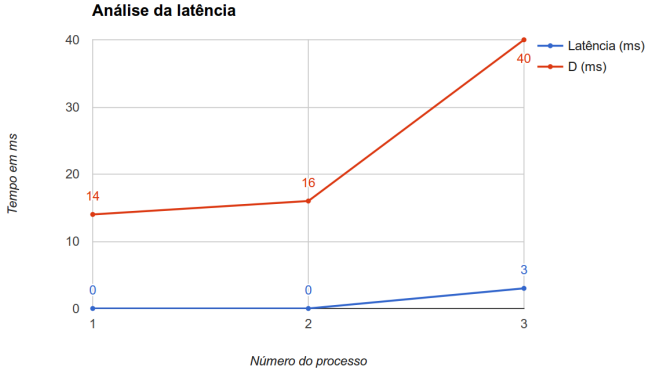

Fig. 6. Gráfico de latência por processo com o EDF.

são consideravelmente altas. Assim, o processo é posto em espera, o que ocasiona a latência e maiores trocas de contexto em relação aos outros dois elementos do conjunto de processos. O número de trocas de contexto pode ser observado na Fig. 7.

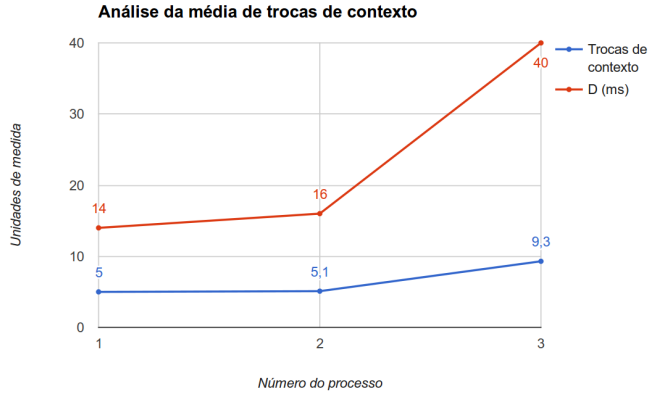

Fig. 7. Gráfico de trocas de contexto por processo com o EDF.

IX. DISCUSSÃO

Durante a implementação, notou-se que a chamada de sistema *sleep* não se portava na granularidade de segundos, conforme descrito no livro do *xv6* [16]. Foi necessário, portanto, ajustar o temporizador *TICR* (em *lapic.c*) de maneira que o *sleep* passou a considerar uma granularidade de tempo em milissegundos. O ajuste, por sua vez, foi realizado com base na máquina utilizada para a execução do sistema. Caso o *sleep* porte-se em uma granularidade que não corresponde a milissegundo, ao ser executado em outra máquina, evidencia-se que o temporizador *TICR* deverá ser ajustado para tal.

Verificou-se que o *xv6* comporta-se de maneira diferente do modo uniprocessado quando executado no modo multiprocessado, devido ao fato de considerar a quantidade de processadores para a instanciação do sistema. Contudo, a implementação da versão em tempo real, realizada a partir

deste trabalho, não é suportada ao definir o modo multiprocessado no *xv6*. A identificação e as alterações necessárias para a execução no modo multiprocessado demandam de um estudo mais aprofundado e minucioso. Portanto, a implementação do EDF é executada com sucesso somente no modo uniprocessado do *xv6*.

X. Conclusão

A partir do que foi apresentado, conclui-se que é possível moldar o *xv6* para uma abordagem de tempo real, assim como, realizar a coleta e a reprodução do comportamento de forma fiel à definição do algoritmo adotado.

O resultado da implementação do algoritmo aponta para a capacidade de aceitação de modelagem da plataforma, o que viabiliza a inclusão de novos escalonadores de tempo real. Apesar do cenário limitado de avaliação, identifica-se um desempenho satisfatório em relação à capacidade de resposta e resultados que coincidem com as especificações do EDF.

Com o intuito de aprimoramento e expansão, como trabalhos futuros, pode-se enumerar os seguintes pontos: implementar o suporte a múltiplos processadores, implementar demais algoritmos, sistema híbrido a partir da implementação de outros algoritmos e a realização de testes de execução em mais cenários.

Este trabalho está incluso como primeira etapa de um projeto de inovação tecnológica, que visa realizar um estudo aprofundado dos algoritmos de escalonamento de processos em tempo real implementados no *xv6*. Para tanto, será necessário expandir os cenários abordados, de forma a contemplar diferentes situações para que seja possível realizar uma avaliação precisa sobre o desempenho de algoritmos de tempo real na plataforma.

Referências

[1] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications*. Real-Time Systems Series. Springer, Spring Street, New York, NY 10013, USA, third edition, 2011.

[2] William Stallings. *Operating Systems. Internals and Design Principles*. Pearson Education, Inc., 1 Lake Street, Upper Saddle River, New Jersey, 07458, seventh edition, 2012.

[3] Matthew Dellinger, Piyush Garyali, and Binoy Ravindran. Chronos linux: A best-effort real-time multiprocessor linux kernel. *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, June 2011.

[4] Gang Yao, Giorgio C. Buttazzo, and Marko Bertogna and. Feasibility analysis under fixed priority scheduling with fixed preemption points. *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*, pages 71–80, Aug. 2010.

[5] Jean-Marie Farines, Joni da Silva Fraga, and Rômulo Silva de Oliveira. *Sistemas de Tempo Real*. 12ª Escola de Computação, IME-USP, São Paulo-SP, 24 a 28 de julho de 2000.

[6] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43, 4(Article 35), October 2011.

[7] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, NY, 1979.

[8] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM*, 20(1):46–61, January 1973.

[9] Robert I. Davis. A review of fixed priority and edf scheduling for hard real-time uniprocessor systems. *ACM SIGBED Review - Special Issue on the 3rd Embedded Operating System Workshop (EWiLi 2013)*, 11(1):8–19, February 2014.

[10] Giorgio C. Buttazzo. Rate monotonic vs. edf: Judgment day. *Springer Science + Business Media, Inc. Manufactured in The Netherlands*, 29(5–26), September 2005.

[11] John M. Calandrino, James H. Anderson, and Dan P. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. *Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on*, pages 247 – 258, July 2007.

[12] Qiang Wang, Hong-An Wang, Hong Jin, and Guo-Zhong Dai. Design and evaluation of priority table based real-time scheduling algorithmss. *Intemational Conference on Robotics Intelligent Systems and Signal Processing*, 2:1294 – 1299, October 2003.

[13] Biju K. Raveendran, Sundar Balasubramaniam, and S. Gurunarayanan. Evaluation of priority based real time scheduling algorithms: choices and tradeoffs. *Proceedings of the 2008 ACM symposium on Applied computing*, pages 302–307, March 2008.

[14] Lalatendu Behera and Durga Prasad Mohapatra. Schedulability analysis of task scheduling in multiprocessor real-time systems using edf algorithm. *International Conference on Computer Communication and Informatics (ICCCI -2012)*, pages 1–9, Jan. 2012.

[15] Operating System Engineering MIT. Xv6, a simple unix-like teaching operating system, 2014.

[16] Russ Cox, M Frans Kaashoek, and Robert T Morris. Operating System Engineeringxv6, a simple unix-like teaching operating system, 2011.