UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO – UFERSA

PRÓ-REITORIA DE ENSINO E GRADUAÇÃO - PROGRAD

JEFFERSON CARLOS DA COSTA LINS

THIAGO DA SILVERIA BATISTA

PRIMEIRO RELATÓRIO DE SISTEMAS OPERACIONAIS

MOSSORÓ

2016

JEFFERSON CARLOS DA COSTA LINS

THIAGO DA SILVERIA BATISTA

PRIMEIRO RELATÓRIO DE SISTEMAS OPERACIONAIS

Relatório elaborado pelos alunos Jefferson Lins e Thiago Batista, do curso de Ciências da Computação, como método avaliativo na disciplina de Sistemas Operacionais, ministrada pelo professor Silvio Roberto.

Mossoró

2016

**SUMÁRIO**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1. **INTRODUÇÃO E OBJETIVOS** | | **4** | | | | | |
| **2. NOVAS CLASSES AUXILIARES** |  | | | | | | **4** |
| **2.1. PCB** | | | | | |  | **4** |
| **2.2. TABELA DE PROCESSOS** | | | | | |  | **5** |
| **2.3. ESCALONADOR** | | | **5** | | | | |
| **2.4. GERENCIADOR DE PROCESSOS** | | | **6** | | | | |
| **3. NOVAS CLASSES DE CHAMADA DE SISTEMA** | | | **6** | | | | |
| **3.1. FORK** | | | **6** | | | | |
| **3.2. PROCESS CHANGE** | | | **7** | | | | |
| **4. NO MIPS** | | | | **8** | | | |
| **4.1. CÓDIGO ASSEMBLY** | | | **8** | | | | |
| **4.2. MACROS** | | | | | **9** | | |
|  | | |  | | | | |
|  | | |  | | | | |
|  | | |  | | | | |
|  | | |  | | | | |

1. **INTRODUÇÃO E OBJETIVOS**

O MIPS é uma arquitetura de [microprocessadores](https://pt.wikipedia.org/wiki/Microprocessador) [RISC](https://pt.wikipedia.org/wiki/RISC) desenvolvida pela [MIPS Computer Systems](https://pt.wikipedia.org/wiki/MIPS_Computer_Systems). Essa arquitetura é baseada em registrador, ou seja, a CPU usa apenas registradores para realizar as suas operações aritméticas e lógicas. Um dos ambientes de desenvolvimento para a programação em linguagem assembly MIPS é o MARS, que é uma IDE muito usada a nível educacional para quem deseja a aprender a desenvolver nessa linguagem.

O MARS é oferecido gratuitamente para *download* em seu site e o seu código é completamente aberto, dando a liberdade para os usuários de criarem novas funcionalidades para a IDE.

O que nos interessa no MARS, nesse momento, é a capacidade que ele oferece de podermos criar chamada de sistema, que é um mecanismo usado pelo [programa](https://pt.wikipedia.org/wiki/Programa) para requisitar um serviço do [sistema operacional](https://pt.wikipedia.org/wiki/Sistema_operativo), ou mais especificamente, do [núcleo](https://pt.wikipedia.org/wiki/N%C3%BAcleo_(inform%C3%A1tica)) do sistema operacional.

O objetivo desse trabalho é desenvolvermos duas chamadas de sistemas, sendo a primeira chamada de *Fork,* na qual servirá para a criação de um novo processo, e a segunda chamada de *ProcessChange,* que permitirá trocar de processo em um determinado ponto do algoritmo.

1. **NOVAS CLASSES AUXILIARES**

Para o desenvolvimento das nossas novas chamadas de sistemas, tivemos que criar novas classes no pacote *mars.so,* que servirão de auxílio para o que desejamos construir. Nos próximos tópicos veremos cada uma delas.

* 1. **PCB**

A primeira delas é a classe *PCB* (*Process Control Block*) que serve para armazenar todas as informações de contexto de um processo, como: informações do hardware: o conteúdo de todos os registradores, e informações lógicas: endereço do início do programa, PID (Identificador do processo), estado do processo (pronto, executando ou bloqueado).

Essa classe também contém todos os métodos de *getters* e *setters* de todos os atributos, para podermos manipular as informações desses atributos em outras classes.

* 1. **TABELA DE PROCESSOS**

Também criamos a classe *Tabela de Processos* que instancia um objeto da classe PCB para cada novo processo, ou seja, essa classe possui tabela do tipo *Collection,* que irá armazenas os processos criados. Essa classe ainda nos permite realizar as seguintes operações: adicionar processo, remover processo, modificar o estado do processo, ver os processos da tabela, buscar um processo na tabela, atualizar um processo da tabela e buscar um processo que esteja executando.

* 1. **ESCALONADOR**

A classe *Escalonador* possui o algoritmo para escolha dos processos que estão no estado de “pronto” na Tabela de Processos.

Essa classe tem como atributos duas filas, uma para processos prontos e outra para processos bloqueados. Temos também um objeto da classe *Tabela de Processos,* e por fim um atributo que indicará qual o processo que está executando no momento (*pid\_executando*). Lembrando que cada um desses atributos possuem seus respectivos métodos de *get* e *set*.

A classe *Escalonador* possuem métodos que dão todo o controle de escolha e amazenameto dos processos. Nessa classe, temos os métodos: adicionar na tabela, adicionar na fila de prontos, ver processos da fila de prontos, adicionar processo na fila de bloqueados, ver processos da fila de bloqueados, escolher um processo pronto e buscar um processo que esteja executando.

* 1. **GERENCIADOR DE PROCESSOS**

Para finalizar as classes auxiliares, criamos a classe *Gerenciador de Processos*, que instancia estaticamente um objeto da classe *Escalonador*, nos permitindo ter acesso à escolha de processos e também à tabela de processos criados.

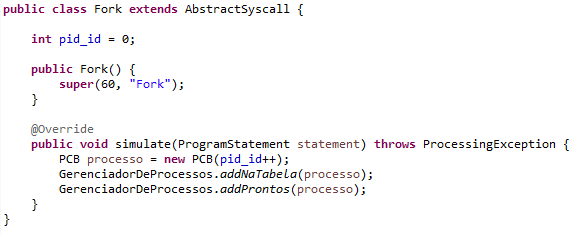
1. **NOVAS CLASSES DE CHAMADA DE SISTEMA**

Antes de criarmos nossas novas classes de chamadas de sistema, adicionamos duas propriedades no arquivo *Syscall.properties:* Fork (60), e ProcessChange (61). Essas propriedades estão relacionadas com as nova classes *Fork* e *ProcessChange*, que involuntariamente possuem os mesmos nomes.

Nos próximos tópicos analisaremos o desenvolvimento das classes que invocam as chamadas de sistemas que criamos.

* 1. **FORK**

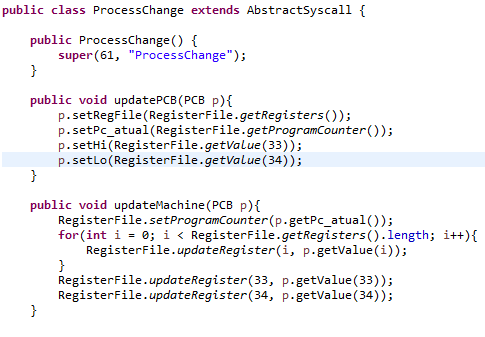
O objetivo da classe *Fork* é adicionar um processo na Tabela de Processos e também na fila de processos pronto. Vale lembrar que sempre que um processo é criado, o seu estado inicial é definido como pronto.

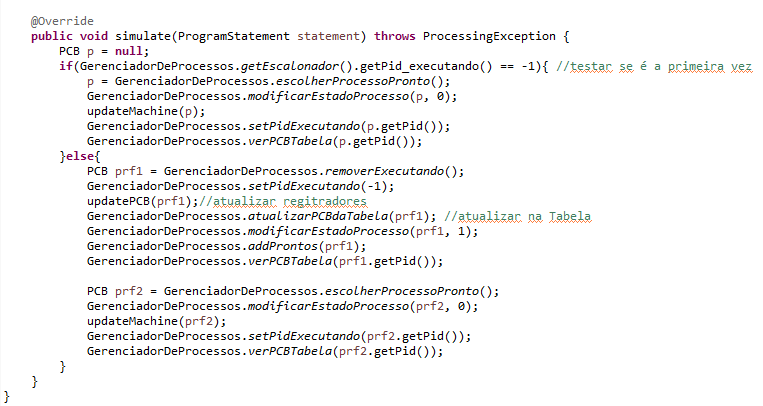


Depois de criar um objeto PCB, chamamos o nosso objeto *Gerenciador de Processos*. Vale salientar que esse objeto possui um objeto *Escalonador* estático, e esse último possui um objeto *Tabela de Processos* e uma fila para armazenamento dos processos prontos. O objeto gerenciador invoca o método de adicionar o processo na tabela, e em seguida, o também adiciona o mesmo processo na fila de prontos.

* 1. **PROCESS CHANGE**

A classe *ProcessChange* tem como objetivo trocar de processo sempre que ela é invocada.





Inicialmente, verificamos se existe algum processo em execução naquele momento, como será a primeira chamada do *ProcessChange*, não teremos nenhum processo em execução. Então, removemos da fila de prontos o primeiro processo inserido e em seguida modificamos seu estado para *executando*. Depois, invocamos o método *updateMachine,*  que irá fazer uma cópia de todos os registradores do PCB atual para os registradores da máquina. E por fim, atualizamos o atributo *pid\_executando* com o identificador do processo atual.

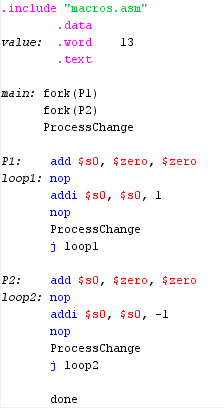
Agora, caso não seja exista algum processo em execução, pegamos esse processo, atualizamos seu *pid\_executando* com “-1” e o atualizamos todos os seus registradores com os valores dos registradores atuais da máquina, depois atualizamos esse mesmo processo na Tabela de Processos e em seguida, modificamos seu estado para *pronto.* Por fim, escolhemos o próximo processo que irá ser executado.

1. **NO MIPS**

Veremos agora o nosso código assembly no MIPS.

* 1. **CÓDIGO ASSEMBLY**

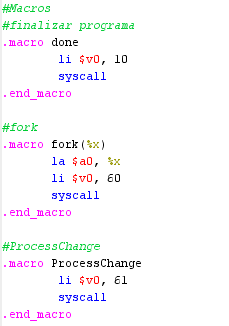
O programa assemby abaixo utiliza a chama de sistema *Fork* duas vezes para criar dois processos, sendo que a primeira recebe o endereço inicial de *P1* e a segunda recebe o endereço inicial de *P2.*



Quando *P1* chama o *ProcessChange,* todo o conteúdo desse processo deve ser salvo, de forma que, quando ocorrer uma nova chamada do *ProcessChange,* programa deverá voltar de onde parou em *P1*, contendo o mesmo conteúdo que foi armazenado anteriormente.

* 1. **MACROS**

Elaboramos três macros para facilitar a programação no código assembly.



Os macros criados foram: *done*, que apenas finaliza o programa; *fork,* que invoca a chamada de sistema *Fork;* e por último temos o macro *ProcessChange,* que invoca a chamada de sistema *ProcessChange*.