Relatório da APS de Sistemas Operacionais

Simulador de Escalonador de Processos

github.com/wagjub/SimuladOS

Vitório Miguel Prieto Cilia
vitorio@alunos.utfpr.edu.br
Aluno de Ciência da Computação
Universidade Tecnológica Federal do Paraná
Campo Mourão – Brasil

Daniel Costa Valério
danielvalerio@alunos.utfpe.edu.br
Aluno de Ciência da Computação
Universidade Tecnológica Federal do Paraná
Campo Mourão – Brasil

***Abstract*—Este documento discorre sobre a teoria de escalonadores de processo e a implementação das principais politicas de forma modular as politicas implementeadas foram Shortest Job First, Random, First Come First Served, Round Robin e Fila de Prioridade, algoritmos comunmente estudados na matéria de sistemas operacionais que ajudam na intuição sobre esta área da Ciência da Computação.**

Keywords—Escalonadores de processo, politicas escalonadoras, sistemas operacionais, sjf, rr, random, fp,fcfs

I. Introducão

Um simulador é uma importante ferramenta quando se lida com o projeto de novas tecnologias em que a implementação é cara ou demorada, apesar do funcionamento simplificado os fatores decisivos da continuação de um projeto podem ser bem estudados em termos quantitativos. No presente relatório será discutido a implementação dos algoritmos, o funcionamento modular e os dados pesquisados. O foco do simulador é a coleta de dados sobre vários algoritmos escalonadores de processos, isto é, algoritmos que escolhem um elemento de um conjunto finito de processos aptos a executarem com base em diversos atributos, tanto do próprio conjunto quanto de cada processo individual. Os algoritmos aceitos pelo simulador são: SJF, Random, FCFS, Fila de Prioridade e Round Robin.

Primeiramente analisamos o código fonte do simulador disponibilizado pelo professor no moodle, está foi uma ótima iniciativa, já que assim nós compreendemos o que já havia sido implementado e também pudemos nos esclarecer a respeito do funcionamento e função de um simulador de modo geral.

Com o auxílio do professor nas aulas tivemos, antes da implementação, um entendimento amplo e suficiente do que realmente nos interessava modificar e acrescentar no código que já tínhamos em mãos para que as exigências mínimas do projeto fossem atendidas e concluíssemos com exito esta atividade.

II. Modularização

Quando um projeto apresenta uma dimensão maior, com mais integrantes e funcionalidades intercambiáveis, busca-se uma organização que permita que cada integrante possa focar em uma parte ou módulo do projeto sem que as outras funcionalidades ajam como empecilho ou gargalo no desenvolvimento.

A. Solução

Idealmente, antes da implementação do programa, todas as interfaces estavam definidas ou documentadas, isso propõe que o projeto seja feito de tal forma que para cada variação de uma parte do programa, seja necessário editar o mínimo possível de código.

Entre muitos, um exemplo de padrão de projeto que faz uso de tais conceitos é o *Model View Controller* (MVC) que reparte o software nestes três blocos (modelo, visualização e controle) que, em teoria, podem ser trocados sem que seja necessário a reescrita de toda a aplicação, assim, pode-se criar versões das interfaces de comunicação com o usuário que oferecem o mesmo acoplamento a nível de código mas que atendam a necessidades diferentes.

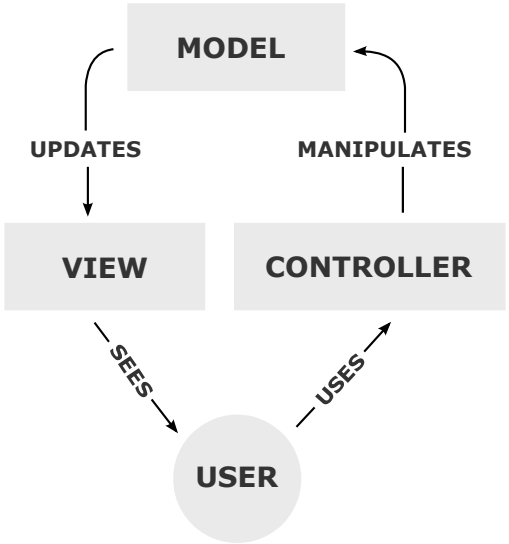

Fig. 1: [3] Representação MVC

.

B. *Análise prévia à Implementação*

Com esta abordagem semelhante a *Orientação a Objetos*, o simulador base foi criado em C de forma que cada *Algoritmo escalonador* seja tratado da mesma forma, sobre o nome de *política*. Ou seja, o trabalho do simulador é simular a execução do processo enquanto que a *politica* faz o trabalho de escolher o próximo processo, e com isso, na implementação de cada parte, não foi necessário alterar todo o código do simulador nem as bibliotecas que o auxiliam. Assim pode-se dizer que utilizando a orientação a objetos e o conceito do MVC, foram implementados novos *controllers* e uma nova *view,* sendo que este é o arquivo de saída (interface de saída) em contra partida o *debug;* e aquele as novas politicas.

Mais especificamente, todas políticas apresentam basicamente a mesma estrutura, porém, não só isso, cada uma apresenta um conjunto de funções que são conhecidas ao simulador que tem o mesmo objetivo, a nomear, *escalonar, tick, novoProcesso, fimProcesso* e *desbloqueio.* E estas funções tem não só o objetivo de atenderem as necessidades do simulador, mas oferecem *hooks* ou pontos de execução do programa principal (*simulador)* que são usados para a coleta de dados sobre a própria politica. Mais adiante neste trabalho isso é fica bem evidente, com políticas que precisam alterar informações que serão usadas na escolha de um processo.

III. A Implementação Geral

A. *A Divisão do Trabalho*

Um arquivo chamado *TODO.txt* foi criado e nele foram inseridos as tarefas necessárias para o termino deste projeto identificado com o nome de cada integrante para a realização da tarefa. Inicialmente, foram arbitrariamente definidos para cada integrante uma politica escalonadora simples (*fcfs e random)* e outra complexa (*sjf e fp).* As outras tarefas, como cálculo do *Tempo Médio de Retorno*(*TMR), Tempo Médio de Espera (TME)* e da *Vazão, Consertar Tick, elaborar este Relatório* e *Implementar o Arquivo de Saída* foram escolhidos arbitrariamente e executados por ambos, como é o caso do relatório ou atomicamente como aconteceu com o *TMR.*

Foi usado o *git* como software de versionamento e colaboração, hosteado no *https://github.com* sob o domínio *https://github.com/wagjub/simuladOS,* onde é possível o acesso ao código fonte.

B. *Tempo Médio de Espera (TME)*

Para a implementação do tempo médio de espera, a sua definição foi previamente analisada. Obtivemos com resultado da definição que o TME pode calculado como o quociente da divisão da soma do tempo de cada processo que estava na fila de prontos pela quantidade de processos existentes. Ou seja:

TME = *(somaTempoProntos)/ qtdProcessos*

Sabendo disto, necessitamos calcular o divisor e o dividendo para que possamos calcular o TME. Assim, buscamos no código fonte do simulador o momento ideal para calcular a soma do tempo de prontos sem que deixássemos passar nenhum processo sem a soma. A maneira mais simples de calcular o dividendo foi: somar em uma variavel a quantidade de elementos existentes na lista de prontos a cada unidade de tempo. Nós deparamos com um "jump" no relógio quando o simulador está ocioso, e neste momento precisamos calcular quantos elementos haviam na lista de pronto multiplicado pela quantidade de "jumps" que foram dados, assim impedimos que durante o tempo ocioso não esqueçamos de somar o quociente corretamente.

Após realizar esta soma do divisor, podemos pegar no arquivo de processos quantos processos ali existiam para tirar uma média aritmética simples com este valor que é identificado como o nosso divisor.

C. *Tempo Médio de Retorno (TMR)*

Também chamado de *TurnArount Time (TAT)* é a diferença entre o tempo do término do processo e do tempo de submissão (quando ficou apto a executar) de um processo. Se houver mais de um processo a ser analisado, o *TMR* final é a média to *TMR* de cada processo. Esse cálculo pode ser melhor entendido no contexto de hospitais, no qual é fornecido um tempo médio que um exame leva para ser concluído ou entregue ao paciente. Já em computação, o *Tempo Médio de Espera* é interpretado como a média de tempo que um processo leva para terminar, mesmo dependendo do tempo de cada processo, essa métrica é um indicador válido da eficiência do escalonador.

No simulador, esse valor foi calculado criando uma variável *tempoRetorno* e atribuindo a ela o somatório de todas as diferenças de *tUltimaExec e tPrimeiraExec* (atributos do *bcp_t*) e finalmente, para a variável *TMR* foi atribuído o valor de *tempoRetorno* dividido pela quantidade de processos. Ou seja,

$$TME = tempoRetorno / processos \rightarrow nProcessos$$

Em que *tUltimaExec[i]* é o tempo da ultima execução do processo *i* e *tPrimeiraExec[i]* é o tempo da primeira execução do processo *i*.

$$tempoRetorno = \sum_{i=0}^{processos \rightarrow nProcessos} tUltimaExec[i] - tPrimeiraExec[i]$$

D. *Vazão*

A *Vazão* é uma grandeza na ordem de $\frac{Volume}{tempo}$ e é principalmente utilizada na área de Engenharia civil em Hidráulica, mas uma outra boa métrica quando se analisa o escalonamento de processos é a quantidade de processos que terminam por uma determinada quantidade de tempo, semelhante a vazão previamente citada, neste contexto, volume é a quantidade de processos que terminaram e o tempo foi especificado de 1000 ticks.

Para o cálculo do simulador , a vazão foi definida da seguinte maneira:

$$Vazão = \frac{processos \rightarrow nProcessos}{\frac{relogio}{1000}}$$

E. Tick

Tick é o termo dado a cada unidade de tempo que se passa no simulador, na implementação base um artifício foi usada para reduzir o tempo de execução. Quando não há processos bloqueados, não há processos prontos nem processos novos, e não há nenhum processo executando, o relógio é avançado até o momento que o próximo processo entra em execução. A falha constada está no fato que quando o avanço acontece, os callbacks das políticas não são chamados de acordo com o tempo avançado e assim o simulador foi modificado para que quando um avanço ocorresse, os callbacks seriam chamados de acordo.

Por fim, verificou-se que não havia uma implementação cem por cento fiel à um caso real da implementação do tick, pois assim como no TME o tempo ocioso era "jumpeado" para algum evento e por tanto a função de tick não era realmente chamada a cada iteração do relógio, por isso a função foi feita para que chamássemos a função de tick a cada iteração do relógio porem com uma peculiaridade. Colocamos as mudanças dentro de defines e "ifs" assim podemos ativar e dasativar essa simulação mais "real", porém o tempo de execução é notavelmente muito maior, e por isso deixamos está possibilidade de preferir executar ou não está mudança visto que a diferença de resultado pode ocorrer apenas na politica Round-Robin e FP caso está use o RR como política em uma ou varias listas que à compõem.

F. Arquivo de Saída

O arquivo de saida faz parte da especificação do projeto, e deve ter o seguinte formato:

CHAVEAMENTOS: ZZZ

TME: ZZZ
TMR: ZZZ
VAZAO: ZZZ
TERMINO: PID PID ...
DIAGRAMA_DE_EXEC

No qual o DIAGRAMA_DE_EXEC é definido por um diagrama de eventos, que por sua vez é composto no presente formato:

TTT PID EVENTO

TTT PID EVENTO
TTT PID EVENTO
...
TTT PID EVENTO

Em que *TTT* é o tempo do relógio que o evento aconteceu, *PID* é o id do processo e *EVENTO* é qual o acontecimneto que ocorreu. O evento pode apresentar os seguintes valores

BLOQUEIO
DESBLOQUEIO
TERMINO

A implementação foi bem diferente do que estamos acostumados, pois verificamos que havia uma ferramenta na biblioteca do stdio.h que nos permite criar um arquivo temporário chamado tmpfile(), ela foi muito útil e essencial para a criação da saída. Criamos dois arquivos temporários e um arquivo final, nos arquivos temporários inserimos a ordem que os processos foram finalizados e no outro todos os os dados do diagrama_exec. Com o resultado final pudemos criar o cabeçalho inicial que colocamos diretamente no arquivo final junto com as informaçoes do arquivo temporário que nos diz a orde que os processos foram finalizados, e logo em seguida colocamos o resto dos dados que estavam no outro arquivo temporário. Para colocar os dados de um arquivo no outro, utilizamos um vetor de caracteres de 255 elementos e através do fgets armazenamoss em blocos pedaços do arquivo que eram inseridos no outro arquivo.

IV. Algoritmos Escalonadores

A. Introdução

Na computação, escalonadores são métodos de seleção de tarefas para uso de um certo recurso. Ou seja, um escalonador de processos seleciona seleciona processos de forma que todos possam executar e terminar suas respectivas tarefas. Essa escalonação pode ser feita de forma a refletir as necessidades de cada sistema, como aumentar a vazão (quantidade de processos por unidade de tempo), manter todos os recursos do computador em uso, oferecer sensação de execução paralela ou mesmo terminar as tarefas de maneira mais rápida. Neste contexto está incluído o conceito de escalonador preemptivo, que é aquele que troca a tarefa atual sem que esta faça uma requisição ou permissão de troca, ou seja, em sistemas em lotes não-preemptivos, a comutação da tarefa ocorre quando ela é bloqueada para esperar um acesso ao disco ou usar um periférico externo.

B. Shortest Job First (SJF)

O SJF, também conhecido como *Shortest Job Next* é uma politica de escalonamento preemptivo que escolhe a tarefa que precisa de menos tempo para terminar, favorecendo, talvez a vazão de processos. De início, ele não pode ser aplicado a problemas do reais pois normalmente não se sabe quanto tempo uma tarefa demorará para terminar, então, para implementa-la foi necessário adicionar um campo de tempo de execução total no *bcp_t* e soma-lo com o tempo do evento de desbloqueio durante a criação dos eventos. A razão disso é que no arquivo do processo, o número que precede o nome do evento é o tempo decorrido dês do ultimo evento, por exemplo:

```
1 3320  BLOQUEIO
2 763   DESBLOQUEIO
3 6872  BLOQUEIO
4 885   DESBLOQUEIO
5 10001 BLOQUEIO
6 929   DESBLOQUEIO
7 12854 BLOQUEIO
```

Na primeira linha, temos a seguinte interpretação "O processo executa por 3320 tempos e é bloqueado" e na segunda linha "O processo espera 763 tempos e é desbloqueado". Ou seja, pode-se inferir que a cada evento de bloqueio temos uma parte do tempo de execução e que o tempo total de execução de um dado processo é a soma de todos os tempos relativos encontrados nos eventos de bloqueio. Isso foi implementado com uma variável global externa *tempo_restante* a biblioteca eventos.h que é zerada pelo invocador da função *EVENTO_criar* e é somada por esta a cada instância de um evento de bloqueio e por fim, a função invocadora, quando terminar de criar todos os eventos do processo, atribui ao processo o valor de *tempo_restante*.

Na execução do SJF, a função *SJF_tick* decrementa o *tempoRestante* do processo em execução, tornando-o mais apto a ser escalonado novamente. Na função *SJF_escalonar* uma varredura é feita em todos os processos aptos a executarem e o de menor *tempoRestante* é retornado como próximo processo.

Nota-se na Ilustração 2 como um processo menor tende a continuar executando até o seu termino e como isso implica no *starvation* de outros processos, como é o caso do processo 12 (*P12)* e do processo 7(*P7)* que foram deixados para o final do escalonamento (*P12)* ou para a outra remessa de processos (*P7).* Interessante também como o *starvation* do processo 5 (*P5)* atrasou a execução de todos os próximos processos, executando-os de forma injusta em relação ao tempo.

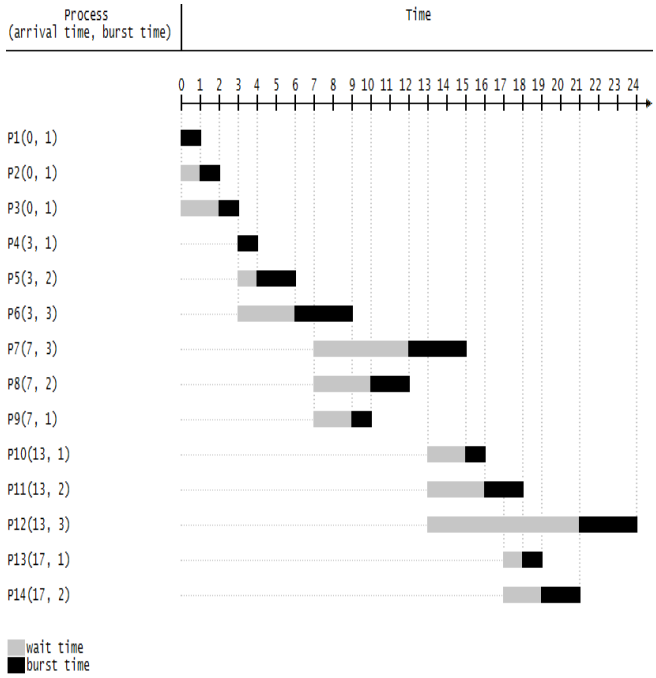

Fig. 3. [5] Politica de escalonamento SJF

C. Random

O Algoritmo Escalonador *Aleatório* pode ser classificado como um dos mais simples sobre implementação e sobre a tática de funcionamento, entretanto, o seu custo não é nem próximo do que se espera. Este algoritmo é simples porque ele não leva em consideração nenhuma informação sobre os processos em execução, simplesmente, seleciona um processo qualquer da lista de prontos ou, se não houver ninguém, não escolhe nenhum e o simulador fica ocioso.

Na implementação desta politica, os *callbacks* de *Tick* e desbloqueio funcionam apenas para gerar um numero pseudo randômico, que neste caso, é referente ao conjunto de eventos referente a cada experimento. Isso acontece porque a cada *callback* para a política, independente do processo, altera a variável *numeroAleatorio*, normalmente, somando-a com o pid do processo que está em execução no momento e no momento em que o *RANDOM_escalonar* é chamado, o processo escolhido é dado por

$$numeroAleatorio \,\%\, prontos{\rightarrow}tam$$

Em que $prontos{\rightarrow}tam$ é a quantidade de processos prontos. Fazendo dessa forma, o experimento é reprodutível mas ainda sim consegue oferecer certa imprevisibilidade da saída, semelhante a geração de números aleatórios utilizando a mesma semente para a geração.

D. First Come First Served (FCFS)

A politica FCFS também conhecida como FIFO tem um funcinamento bem simples, até mais simples do que a Round Robin por dois motivos, primeiro que este é um algoritmo não preemptivo e isto torna-o mais simples pelo fato de que não há necessidade de se implementar um código no tick para o controle de quantum dos processos. O primeiro processo a ser escalonado deve ser executado até o fim, sem preempção. Como documentação na especificação do projeto, quando um processo é bloqueado ele cede sua vez na execução ao próximo processo que no seu bloqueio ou término cede a sua vez ao anterior novamente ou se este estiver bloqueado então a sua vez será cedida ao próximo não bloqueado e assim por diante. A implementação foi muito simples, basicamente tivemos que analisar a fundo as estruturas do simulador e pelo fato de este ser o primeiro algoritmo que implementamos no simulador, então verificamos como a programação em OO foi importante para está implementação do algoritmo. Pois bastou observarmos bem a implementação do RR, como havia sido feita e fazermos de maneira muito semelhante porém suprimindo as características do RR e incrementando as características do FCFS.

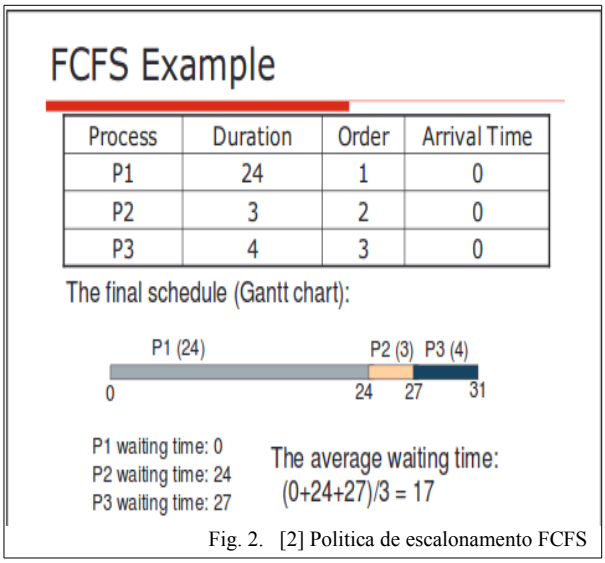

Fig. 2. [2] Politica de escalonamento FCFS

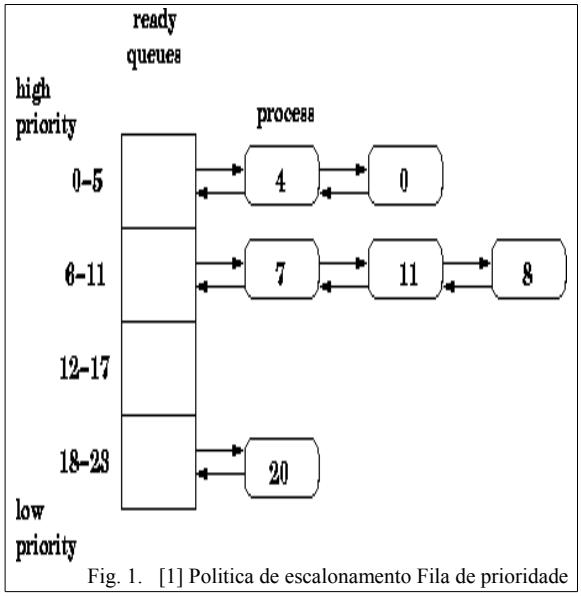

Fig. 1. [1] Politica de escalonamento Fila de prioridade

E. Fila de Prioridade (FP)

A fila de prioridades funciona com diversas filas (uma para cada prioridade) que tem uma politica de escalonamento distinta para cada fila de acordo com a prioridade. Processos com prioridade x são inseridos na fila com prioridade x,. Foi muito simples a implementação desta politica, pois basicamente precisamos apenas criar as filas e chamar os callbacks das politicas de cada fila, com isto foi muito simples de implementar graças novamente à programação em OO. Pois assim nós não precisamos nos preocupar com o funcionamento do escalonamento dentro de cada política, apenas repassamos os callbacks para a politica corretamente.

O principal desafio nesta politica foi ler o arquivo de processos corretamente, já que o parâmetro de quantum do RR tem uma característica morfológica diferente nesta política. Portanto foi necessário realizar algumas alterações para facilitar a implementação na politica do RR e na função de criar politicas, para que o RR pudesse entender os dois tipos morfologicamente distintos de parâmetro, para isso apenas passamos o quantum como parâmetro para a função que realiza uma implementação distinta caso o quantum seja passado como parâmetro ou caso seja NULL assim criamos basicamente uma chave para cada tipo de execução

F. Round Robin (RR)

O *Round Robin* é um escalonador de processos semelhante ao *FCFS* mas difere na execução cíclica dos processos baseado num tempo máximo de execução chamado de quantum. Assim, quando um processo está executando, a cada unidade de tempo, o *quantum* do processo é decrementado, o momento de escalonar o processo é quando o quantum chega a zero, ou seja, o processo já executou tudo o tempo que ele tinha disponível. O simulador então chama a rotina de escalonação que escolhe o primeiro processo de uma lista de processos aptos a executar e coloca o antigo processo em execução no final da lista de aptos com o seu quantum no valor original.

O *RR* é uma politica interessante porque ela é livre de *starvation* ou seja, nunca um processo deixará de executar, ou seja, a divisão de tempo do processador sempre abrange todos os processos vigentes. Por isso, ele não só serve como um bom exemplo de politica escalonadora mas também é de fato aplicado a problemas reais, como a escalonação de pacotes em redes de computadores.

Este algoritmo escalonador já foi entregue implementado no simulador base e serviu como exemplo para a implementação das outras políticas.

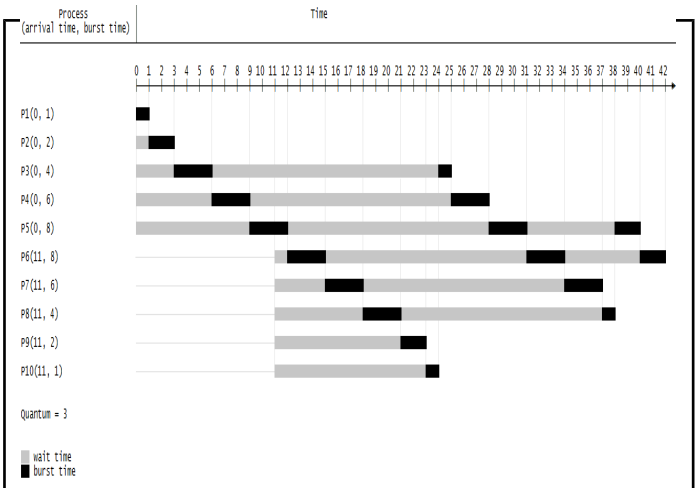

Fig. 4. [4] Politica de escalonamento do Round Robin com quantum 3

REFERENCIAS

[1] Imagem fila de prioridades – Computer Science Departament of The Ohio University
http://web.cse.ohio-state.edu/~mamrak/CIS662/bsd_ready_queues.gif

[2] Imagem FCFS – Programers Paradise website
http://program-aaag.rhcloud.com/wp-content/uploads/2013/11/Screenshot-from-2013-11-01-120933.png

[3] Imagem MVC – Wikipedia page of Model View Controler
https://pt.wikipedia.org/wiki/MVC#/media/File:ModelViewControllerDiagram2.svg

[4] Imagem Round Robin – Wikipedia page of Round Robin
https://pt.wikipedia.org/wiki/Round-robin#/media/File:Round-robin_schedule_quantum_3.png

[5] Imagem SJF – Wikipedia page of SJF
https://pt.wikipedia.org/wiki/Shortest_job_first#/media/File:Shortest_job_first.png