

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/281745188>

"Crianças Brincando": Uma ferramenta para o auxílio na aprendizagem de programação concorrente

Conference Paper · September 2015

CITATION

1

READS

247

3 authors:



Stéphanie A Braga

Instituto Federal de Educação, Ciência e Tecnologia do Ceará

7 PUBLICATIONS 32 CITATIONS

[SEE PROFILE](#)



Hugaleno C Bezerra

2 PUBLICATIONS 1 CITATION

[SEE PROFILE](#)



Fernando Parente Garcia

Instituto Federal de Educação, Ciência e Tecnologia do Ceará

10 PUBLICATIONS 35 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



Home Energy Management Systems - Load profiling for Demand Side Management [View project](#)



“CRIANÇAS BRINCANDO”: UMA FERRAMENTA PARA O AUXÍLIO NA APRENDIZAGEM DE PROGRAMAÇÃO CONCORRENTE.

Stéphanie A. Braga – stephanie.abraga@gmail.com

Hugaleño C. Bezerra – hugaleño@gmail.com

Fernando P. Garcia – fernandoparente@ifce.edu.br

Instituto Federal de Educação, Ciência e Tecnologia do Ceará.

Avenida 13 de maio, 2081, Benfica.

60040531 - Fortaleza, CE – Brasil

Resumo: Neste trabalho é apresentada uma ferramenta para o auxílio na aprendizagem de programação concorrente e paralela, utilizando o conceito de semáforos. A aplicação desenvolvida simula o caso “Crianças Brincando”: um problema que envolve sincronização e exclusão mútua entre processos concorrentes. Essa ferramenta foi desenvolvida com o objetivo de facilitar o entendimento do problema de comunicação entre processos concorrentes, um desafio presente em muitas áreas da programação.

Palavras-chave: programação concorrente, semáforos, threads.

1. INTRODUÇÃO

Os primeiros sistemas de computação só permitiam que um programa fosse executado de cada vez. Esse programa tinha controle completo do sistema e acesso a todos os recursos. Já os sistemas de computação modernos permitem que múltiplos programas sejam carregados na memória e executados de forma concorrente (SÍLBERSCHATZ *et al*, 2001).

Segundo SARMANHO (2009), o projeto e desenvolvimento de programas concorrentes é, por natureza, mais complexo que o de programas sequenciais tradicionais. O projetista de aplicações concorrentes deve considerar tudo o que se aplica a programas sequenciais, além de precisar considerar os aspectos específicos da programação concorrente. Algumas primitivas são citadas pelo autor: definição dos processos que irão executar concorrentemente; inicialização e finalização dos processos concorrentes e; coordenação da interação entre esses processos, enquanto estiverem executando.

As disciplinas de sistemas operacionais e programação, presentes em alguns cursos de engenharia e áreas de Tecnologias de Informação e Comunicação (TICs), têm o objetivo de fornecer ao estudante conhecimentos relativos aos fundamentos da arquitetura de um sistema operacional, tais como gerenciamento do processador, da memória, dos dispositivos de entrada e saída e do sistema de arquivos. Entender esses pontos, no entanto, não é uma tarefa fácil. De acordo com FONSECA & NASCIMENTO (2014), a forma tradicional que a disciplina é ensinada, utilizando apenas teoria, dificulta o entendimento do aluno e, em alguns casos, pode até desmotivá-lo, por conta da falta de ferramentas capazes de mostrar na prática os conceitos vistos na teoria.

Com base nesses argumentos, neste artigo é apresentada uma aplicação que mostra, na prática, um problema de comunicação entre processos concorrentes utilizando o conceito de semáforos, um recurso amplamente utilizado na programação concorrente e paralela que garante a sincronização e exclusão mútua dos processos. A aplicação tem o objetivo de facilitar o entendimento do problema de comunicação entre processos concorrentes, um desafio presente em muitas áreas da programação.



2. FUNDAMENTAÇÃO TEÓRICA

Nessa seção serão abordados alguns tópicos que foram de fundamental importância para o embasamento da aplicação proposta.

2.1. A Multiprogramação

Quando se deseja realizar duas ou mais tarefas simultaneamente, a solução trivial a disposição do programador é dividir as tarefas a serem realizadas em dois ou mais processos. Assim como os processos comuns, os *threads* executam em paralelo, porém compartilham a mesma área de memória. Quando um processo é dividido em *threads*, todo o recurso alocado para o processo é compartilhado entre seus *threads*, tornando o compartilhamento de informações bem mais simples do que quando são usados processos independentes (JANDL, 2004).

Três benefícios da programação com múltiplos *threads* são mostrados a seguir:

- Capacidade de resposta: Os *threads* de uma aplicação interativa podem permitir que um programa continue executando mesmo se parte dele estiver bloqueada ou executando uma operação demorada, aumentando, assim, a capacidade de resposta para o usuário (SÍLBERSCHATZ *et al*, 2001).
- Compartilhamento de recursos: Por padrão, os *threads* compartilham a memória e os recursos do processo aos quais pertencem. O benefício do compartilhamento do código permite que uma aplicação tenha vários *threads* diferentes, em atividade, todos dentro do mesmo espaço de endereçamento (SÍLBERSCHATZ *et al*, 2001).
- Utilização de arquiteturas multiprocessador: Os *threads* são úteis em sistemas com múltiplas CPUs, para os quais o paralelismo real é possível (TANENBAUM, 2009).

2.2. Programação Concorrente e Paralela

O surgimento da programação concorrente ocorreu quando os sistemas operacionais evoluíram, passando a implementar a multiprogramação. Apesar dos benefícios do cenário da multiprogramação, citados na seção 2.1., os processos não trabalham de forma colaborativa, interagindo entre si com o objetivo de produzir um resultado único. De acordo com SARMANHO (2009), esses processos trabalham de forma concorrente, tentando obter para si os recursos necessários à sua execução em particular.

A programação paralela consiste em dividir uma aplicação em partes menores para serem executadas por diversos elementos de processamento, com a condição de interagir entre si para obter o resultado correto. Um programa paralelo é composto por processos ou *threads*. Já a programação concorrente é uma generalização da programação paralela, não havendo a necessidade de diversos elementos de processamento. Um programa concorrente pode, inclusive, executar em uma plataforma sequencial, desde que a plataforma suporte multiprogramação (SARMANHO, 2009).

Ainda segundo SARMANHO (2009), a programação concorrente e paralela é uma alternativa para atender a demanda das aplicações computacionais: maior poder de processamento aliado a menor tempo de resposta, por conta da crescente quantidade de informação a ser tratada e da necessidade dos negócios das empresas, respectivamente.

2.3. Região Crítica

Todos os *threads* de um processo compartilham a mesma área de memória. Quando um ou mais *threads* precisam acessar o valor de uma mesma área de memória compartilhada, são necessários certos cuidados para evitar inconsistência dos dados. É preciso encontrar algum modo de impedir que mais de um *thread* leia e escreva ao mesmo tempo na memória compartilhada. Em outras palavras, precisa-se de exclusão mútua, ou seja, alguma maneira de



garantir que outros *threads* sejam impedidos de usar algum recurso compartilhado, quando este já tiver sendo usado (TANENBAUM, 2009).

O *thread* durante algum tempo pode estar realizando tarefas internas que não interferem em outros *threads*. A região crítica de um programa é determinada pela parte do código na qual o mesmo precisa acessar alguma área de memória compartilhada ou qualquer outro recurso compartilhado que possa ocasionar disputas (TANENBAUM, 2009).

Existem diversos recursos para se obter exclusão mútua, sendo que enquanto um processo está ocupado atualizando a memória compartilhada em sua região crítica, nenhum outro processo terá acesso a essa região. Segundo TANENBAUM (2009), alguns desses mecanismos são: desabilitar as interrupções, utilização de variáveis de bloqueio, alternância escrita e semáforo, entre outros.

2.4. Semáforos

A maioria dos mecanismos citados na seção 2.3. possui a desvantagem de apresentar o problema de espera ociosa. Na espera ociosa ocorre a execução de um laço que testa possibilidade de acesso à região crítica. Esse tipo de espera é indesejável, pois acarreta o desperdício de ciclos de CPU (SARMANHO, 2009).

O matemático holandês E. Dijkstra propôs um mecanismo de coordenação eficiente e flexível para o controle da exclusão mútua entre N tarefas: o semáforo. Apesar de antigo, o semáforo continua sendo o mecanismo de sincronização mais utilizado na construção de aplicações concorrentes. Um semáforo possui duas operações básicas: P (conhecida como *acquire* no Java) e V (conhecida como *release* no Java). Se uma operação *acquire* for realizada sobre um semáforo, ocorre a verificação se o seu valor é maior que zero. Caso verdade, seu valor é decrementado e o processo que realizou essa operação poderá continuar. Caso contrário, o processo que requisitou a operação irá para o estado suspenso. Já a operação *release*, incrementa o valor de um dado semáforo. Se um ou mais processos estiverem suspensos nesse semáforo, o sistema escolhe um dos processos para terminar a operação *acquire*. Portanto, depois de um *release* em um semáforo, com processos suspensos sobre ele, o semáforo permanecerá 0, mas haverá um processo a menos suspenso sobre ele. Todas essas operações são realizadas como uma única e indivisível ação atômica (MAZIERO, 2014; TANENBAUM 2009; JANDL, 2004).

Um caso especial de semáforo é o semáforo binário, onde assume-se apenas dois valores: 0 ou 1. Esse tipo de semáforo também é conhecido por *mutex* (*Mutual Exclusion*), e é usado para garantir a exclusão mútua no acesso a regiões críticas (TANENBAUM, 2009).

2.4.1. O Problema Produtor - Consumidor

A ferramenta proposta nesse artigo baseia-se no problema Produtor – Consumidor, um dos problemas clássicos da computação de comunicação entre processos. Os problemas clássicos são usados como base para resolução de outros desafios parecidos.

O problema consiste no seguinte: dois processos compartilham um *buffer* comum e de tamanho fixo. O produtor adiciona informação dentro do *buffer* e o consumidor, retira informação. O problema se origina quando o produtor quer colocar um novo item no *buffer*, quando o mesmo já está cheio. Outro problema ocorre quando o consumidor tenta remover um item do *buffer*, estando ele vazio. (TANENBAUM, 2009).

3. CARACTERIZAÇÃO DO PROBLEMA “CRIANÇAS BRINCANDO”

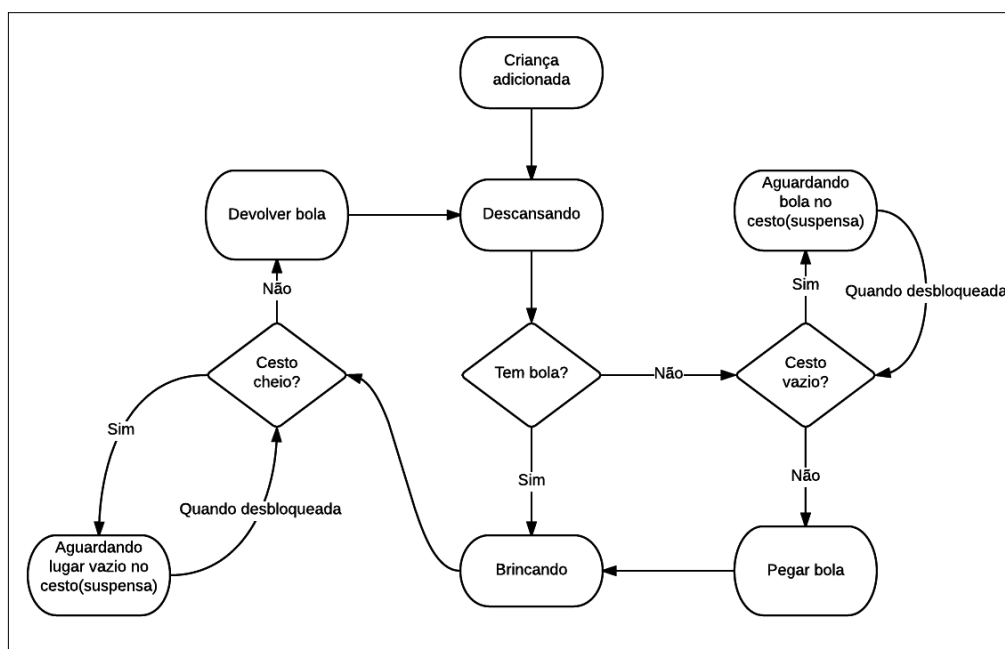
A ferramenta proposta tem o objetivo de simular e mostrar o caso de sincronização e exclusão mútua entre processos, utilizando o problema “Crianças Brincando”. O problema é o seguinte: Há várias crianças brincando com bolas e um cesto (com capacidade X) para guardar



as bolas. As crianças podem começar a brincar, já estando na posse de uma bola, ou podem pegar a bola no cesto. Elas passam um tempo B brincando e, quando terminam de brincar, guardam a bola no cesto, passando a descansar por um tempo D . Depois retornam a brincar e, assim, sucessivamente. Essas crianças podem ser caracterizadas em duas classes: “espertas” (crianças que passam mais tempo brincando do que descansando) e “preguiçosas” (crianças que passam mais tempo descansando do que brincando).

O problema de sincronização e exclusão mútua deve-se aos fatos: quando começam a brincar sem bola, as crianças devem pegar uma bola no cesto. Se o cesto estiver vazio, devem esperar até que ele possua, pelo menos, uma bola para começarem a brincar; depois que terminam de brincar, as crianças devem guardar a bola no cesto. Se o cesto estiver cheio, ou seja, com X bolas, elas devem esperar até que o cesto possua espaço novamente para a bola ser guardada e, então começarem a descansar. Um diagrama de estados descrevendo esse problema é ilustrado na Figura 1.

Figura 1 – Diagrama de Estados do problema.



4. METODOLOGIA

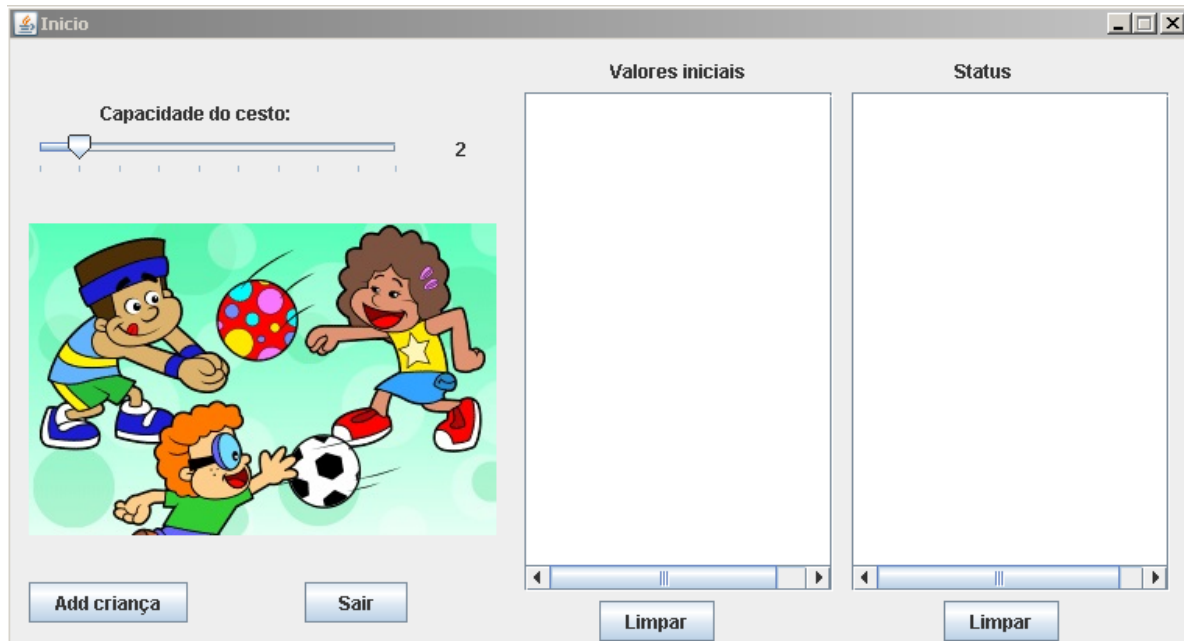
A aplicação foi desenvolvida por alunos do curso de Engenharia de Telecomunicações do IFCE (Instituto Federal de Educação, Ciência e Tecnologia do Ceará), na disciplina de Sistemas Operacionais e, proposta pelo professor que ministrava a disciplina. Nas próximas subseções serão descritas os recursos e metodologia usados no desenvolvimento da aplicação.

4.1. A Interface Gráfica e Inicialização da Aplicação

A aplicação foi desenvolvida na plataforma Java, utilizando o ambiente de programação NetBeans. Foi desenvolvida uma interface gráfica, de forma a ser mais dinâmica possível, facilitando a visualização do problema.

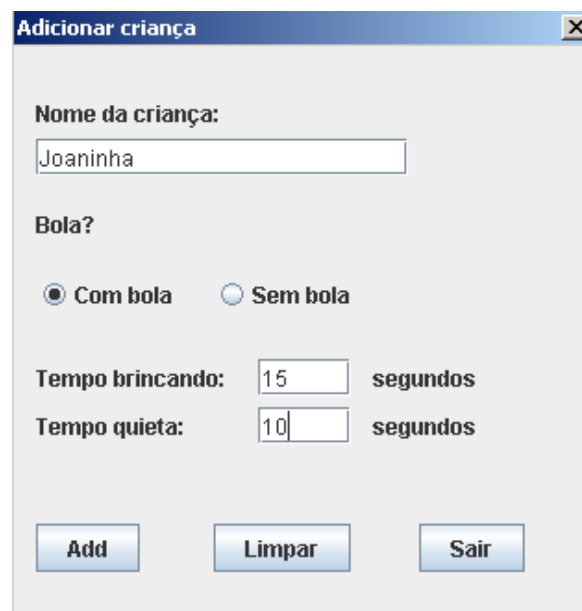
Na tela inicial da aplicação, há a opção para o usuário especificar a capacidade do cesto de bolas. Há também dois campos, onde serão mostradas informações das crianças que forem adicionadas, bem como seus parâmetros, e o estado em que cada criança se encontra no decorrer da execução. A tela Inicial da aplicação é ilustrada na Figura 2.

Figura 2 – Tela Inicial da Aplicação.



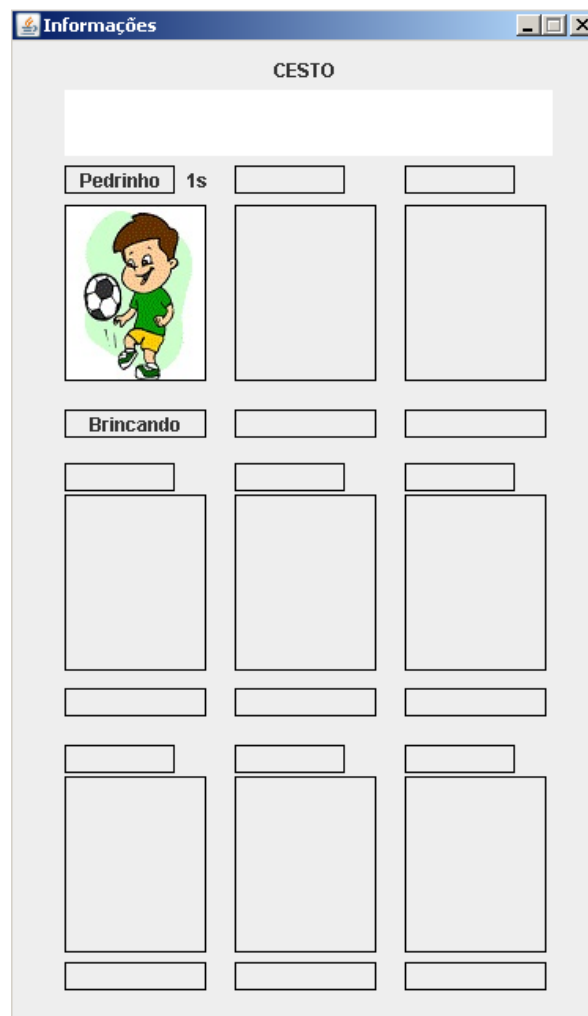
O botão “Add Criança” carrega uma nova tela, onde o usuário pode inicializar uma nova criança. Nessa tela, o usuário informa o nome da criança; marca a opção de inicializá-la com bola ou sem bola e; define os tempos (em segundos) que a criança passa brincando e descansando. A tela de adicionar as crianças é mostrada na Figura 3.

Figura 3 – Tela de Adicionar Crianças.



Quando o usuário adiciona a primeira criança, outra tela é carregada, onde é possível visualizar, dinamicamente, o estado de cada criança e do cesto de bolas. Nessa tela, o tempo de cada ação da criança é atualizado e mostrado, permitindo ao usuário visualizar quanto tempo falta para encerrar uma ação e inicializar outra. Na Figura 4, é ilustrada essa tela de informações.

Figura 4 – Tela de Informações.



4.2. Programação *Multithread* em Java

A aplicação foi desenvolvida utilizando o conceito de Programação Orientada a Objetos, sendo dividida em algumas classes. A classe principal é a classe “Crianca”, responsável por simular o comportamento de cada criança, realizando instruções como brincar, descansar, pegar bola e devolver a bola para o cesto. Para aproveitar o suporte a *threads*, já presente na linguagem Java, a classe “Crianca” herda todos os recursos da classe *Thread*, nativa da linguagem. Dessa forma, podemos instanciar vários *threads*, cada uma representando uma criança.

Analisando o problema, pode-se perceber que a região crítica é a parte da aplicação onde são realizadas as ações que serão feitas no cesto. Como cada *thread* tem acesso ao mesmo cesto, é necessário controlar essas ações: é preciso impedir que bolas sejam adicionadas além da capacidade do cesto, bem como tirar bolas quando o cesto já se encontrar vazio. Em comparação com o problema clássico produtor-consumidor, apresentado acima, pode-se afirmar que o cesto representa o *buffer*, com a diferença de que as crianças tanto “produzem”, ao colocar a bola no cesto depois de brincar, como “consomem”, ao retirar uma bola do cesto para poderem brincar. Dessa forma, cada criança realiza o papel tanto do consumidor quanto do produtor.



4.3. Resolvendo o problema de Exclusão Mútua

Para que a aplicação funcionasse como esperado, foi necessário solucionar dois problemas: o primeiro surge quando uma criança quer brincar e tenta pegar uma bola no cesto, mas o mesmo se encontra vazio; o outro surge quando uma criança quer devolver a bola, com a qual estava brincando, porém o cesto se encontra cheio. Na solução de ambos os problemas foi utilizada a classe *Semaphore* do Java, a qual permitiu proteger a região crítica da aplicação. Foram criados dois semáforos, *full* e *empty*, um para representar quantas bolas há no cesto e o outro para representar quantos espaços vazios existem, respectivamente.

4.3.1. O caso do Cesto vazio

Quando a “criança” vai ao cesto pegar a bola, é preciso, antes, verificar se o mesmo não está vazio. Isso é feito pelo próprio método *acquire* do semáforo *full*. Se esse valor for igual a zero, significa que o cesto está vazio, ou seja, a quantidade de bolas no cesto é zero. Neste caso, a criança terá que aguardar, em estado suspenso, até que alguma criança termine de brincar e coloque a bola no cesto.

Se o cesto não estiver vazio, a criança deve pegar uma bola e começar a brincar. No entanto, como existem vários *threads* “Crianças” executando “paralelamente”, é necessário garantir que somente um *thread*, de cada vez, tente retirar uma bola do cesto. Isto faz-se necessário porque, computacionalmente, é possível que dois *threads* retirem a mesma bola do cesto, porém, isso acarretaria em um erro na aplicação. Para solucionar o problema, foi preciso implementar a exclusão mútua, utilizando o recurso de semáforo, suportado tanto pela linguagem quanto pelo Sistema Operacional, garante o acesso exclusivo de um único processo a um determinado recurso. No caso da aplicação, garantindo que apenas um *thread* esteja na região crítica. O trecho de código da Figura 5 ilustra como isto foi feito em Java.

Figura 5 – Exclusão mútua no caso do cesto vazio.

```
if(full.availablePermits() == 0){
    inicio.addMensagemStatus(getName() + " aguardando bola.");
    this.setImagem("Imagens/aguardando.jpg", "Aguard. bola");
    this.setTempo("");
}
full.acquire();
setImCapacidade();
pegarBola();
empty.release();
```

4.3.2. O caso do Cesto cheio

Outro caso onde precisa-se garantir a exclusão mútua é quando o cesto está cheio, pois quando a “Criança” termina seu tempo de brincar, ela deve devolver a bola para o cesto, porém, antes de devolver, é necessário verificar se o cesto já se encontra cheio. Esse problema foi solucionado de forma semelhante ao problema anterior, mas dessa vez foi preciso verificar se o valor do semáforo *empty* é zero. Caso verdade, isso significa que não há mais espaços vazios no cesto e a criança deverá esperar até que haja algum lugar vazio.

Caso haja espaço no cesto, a criança deve devolver a bola e descansar, durante o tempo determinado. No entanto, como no caso anterior, é necessário garantir que somente um *thread* “Criança” esteja devolvendo a bola. Se essa medida não fosse tomada, mais de uma “Criança” colocaria a bola de volta no cesto, podendo acarretar um funcionamento indesejado



à aplicação, já que o cesto poderia conter bolas além de sua capacidade. A proteção dessa região crítica da aplicação, também foi realizada utilizando os recursos da variável semáforo. A Figura 6 mostra o trecho de código no qual foi implementada a exclusão mútua no caso do cesto cheio.

Figura 6 – Exclusão mútua no caso do cesto cheio.

```
if(empty.availablePermits() == 0){  
    inicio.addMensagemStatus(getName() + " aguardando lugar vazio no cesto.");  
    this.setImagem("Imagens/devolvendo.jpg", "Cesto cheio");  
    this.setTempo("");  
}  
empty.acquire();  
devolverBola();  
full.release();  
setImCapacidade();
```

Os trechos de código apresentados mostram como os recursos de semáforo foram utilizados para administrar de forma correta a região crítica do problema. Dessa forma, é possível garantir que a aplicação funcione corretamente, sem perda de informações e inconsistência dos dados.

5. RESULTADOS E DISCUSSÕES

Ao término do desenvolvimento da aplicação, foi possível garantir o funcionamento correto do problema “Crianças Brincando”, como esperado. Foram realizados testes em todos os cenários possíveis, induzindo situações que, se a sincronização e a exclusão mútua não tivessem sido implementadas de forma correta, acarretaria um erro na aplicação. Um exemplo disso é o cenário em que todas as crianças estão com a bola e o cesto já está cheio. A aplicação funcionou corretamente em todos os testes.

No decorrer do desenvolvimento foi possível compreender, na prática, como esses mecanismos utilizados em programação concorrente e paralela funcionam. Foi possível perceber que, baseando-se em problemas clássicos, já presentes na literatura, pode-se conseguir soluções mais facilmente de problemas que envolvem esses ramos tão importantes da programação. A ferramenta serve para que alunos que estão cursando disciplinas que envolvem a programação concorrente e paralela possam visualizar, de forma mais dinâmica, esses problemas de sincronismo e exclusão mútua. Uma ilustração da versão final aplicação funcionando com 9 crianças está ilustrada na Figura 7. Nesse caso, o cesto ainda possui uma bola disponível.

Figura 7 – Versão final da Aplicação funcionando.



6. CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Acredita-se que ferramentas desse tipo possam ajudar aos alunos a visualizar mais facilmente os conteúdos repassados em sala de aula. Para os desenvolvedores, a ferramenta foi grandiosamente útil, pois além de puderem solucionar o problema, o vivenciaram de forma prática.

Como trabalhos futuros, deseja-se implementar mais problemas, clássicos ou até mesmo baseados nos clássicos, como no caso do problema aqui proposto, e acrescentar à ferramenta. Pretende-se utilizar problemas que envolvam formas diferentes de implementar sincronismo e exclusão mútua, tornando a ferramenta mais diversificada, podendo ser utilizada por professores das disciplinas que envolvam o conteúdo.



REFERÊNCIAS BIBLIOGRÁFICAS

FONSECA, Fagner do Nascimento; NASCIMENTO, Flávia M. S. Uma Ferramenta de Simulação do Processo de Substituição de Páginas em Gerência de Memória Virtual. Anais: XXII Workshop sobre Educação em Computação. Brasília, 2014.

JANDL, Peter, Jr. Notas sobre Sistemas Operacionais. 2004

SARMANHO, Felipe Santos. UNIVERSIDADE DE SÃO PAULO, Instituto de Ciências Matemáticas e de Computação. Teste de programas concorrentes com memória compartilhada, 2009. 183p, il. Dissertação (Mestrado).

SÍLBERSCHATZ, Abraham *et al.* Sistemas Operacionais: Conceitos e Aplicações. Campus, 2004. 303p, il.

TANENBAUM, Andrew S. Sistemas Operacionais Modernos. 3. Ed. São Paulo: Pearson Prentice Hall, 2009. 638p, il.

"PLAYING CHILDREN": A TOOL FOR ASSISTANCE IN CONCURRENT PROGRAMMING LEARNING.

Abstract: *In this paper, is presented a tool that helps on the concurrent and parallel programming learning, using the semaphore concept. The developed application simulates the "Playing Children" case: a problem that involves synchronism and mutual exclusion among concurrent processes. This tool was developed aiming facilitate the understanding of concurrent processes communication, an impediment present at many programming areas.*

Key-words: *concurrent programming, semaphore, threads.*