Bytes Negativo

Sincronização e Comunicação Entre Processos

Brasil

2022, v-1.0.0

Bytes Negativo

Sincronização e Comunicação Entre Processos

Esse artigo tem o intuito de sistematizar o conhecimento sobre o paradigma do paralelismo na computação e como este funciona em seus minuciosos tramites.

Universidade Federal De Rondônia – UNIR

Trabalho de Conclusão da Disciplina de Sistemas Operacionais

Orientador: Jonathan da Silva Ramos

Brasil

2022, v-1.0.0

Resumo

A Ciência da Computação permite-nos explorar cada vez mais a operação lógica das máquinas, juntamente em extrair sua capacidade máxima de funcionamento. No presente artigo, ele tem como objetivo mostrar as formas de comunicação que ocorre entre os processos, utilizando métodos de pesquisa, imagens e exemplos claros e concisos de forma que o leitor se sinta seguro no aprendizado adquirido, apresentando as funções dos Threads, Processos Concorrentes, Funções como Fork, Quit e Join, além da comunicação apresentar as forma de sincronização que os processos utilizam para manter suas informações atualizadas constantemente e para que evite problemas que possam surgir, a partir do funcionamento de algum programa ou do próprio Sistema Operacional, outro objetivo é a apresentar as características do que é um processo e a importância dela para o funcionamento do SO, no decorrer da leitura outro ponto característico será o paralelismo entre os processos no qual suas funções, serão explicadas para que ocorra um melhor entendimento e domínio do presente conteúdo.

Palavras-chaves: threads. processos. compartilhamento de memória. paralelismo. concorrência. sincronização de processos. comunicação de tarefas.

1 Introdução

Processos precisam se comunicar, eles necessitam de recursos externos e tendem a ser colaborativos. Com o surgimento dos primeiros sistemas operacionais multiprogramáveis, os programas de computadores começaram a ser pensados em sua execução de forma concorrente. A partir disso, surgiu o paradigma da programação concorrente, na qual possui como seu alicerce a execução simultânea de múltiplos processos ou threads que trabalham em envolto de uma mesma tarefa.

Uma aplicação concorrente pode ser vista como o oposto da programação sequencial, esta que executa cada sentença de comando de forma linear, esperando o resultado de determinada subrotina terminar para que a seguinte seja executada. Seguindo a programação concorrente, agora iremos imaginar uma aplicação que calcula intersecções entre 20 polígonos, comparando em pares, e gera um mapa no final de toda execução, somente com essas áreas de confluência. Nesse problema, definiremos 10 nanosegundos para cada chamada da subrotina *comparaPoligono*(Pn, Pm). Como a comparação é feita em pares, teremos 10 comparações. Em um programa sequencial, 10 comparações levariam cerca de 100 nanosegundos de processamento, ou seja, uma rotina deve terminar para outra começar. No processamento paralelo, teríamos que todas as 10 comparações seriam feitas de forma simultânea, no tempo de uma subrotina, levando a conclusão de 10 nanosegundos de processamento para 10 tarefas.

Também, em uma aplicação concorrente, é necessário que rotinas compartilhem informações entre si. Levando isso por base, consegue-se pensar a necessidade do estudo da comunicação entre esses processos, implementando diversos mecanismos para essa finalidade como, por exemplo, variáveis compartilhadas na memória central ou sistemas de mensageria entre essas tarefas. Isso tudo também implica a necessidade de que tais tarefas sejam sincronizadas pelo sistema operacional.

Desse modo, podemos notar que o paradigma paralelo é muito interessante para diversas áreas, o que torna seu estudo totalmente viável para aperfeiçoar e otimizar algoritmos. A programação concorrente é mais complexa que a programação sequencial, uma aplicação nesse paradigma envolve diversos outros conhecimentos de processamento, dentre eles o compartilhamento entre processos e a colaboratividade entre os processos que estão em funcionamento.

2 Sincronização e Processos

2.1 Processos

O entendimento de processos, no contexto do sistema operacional, é crucial para a compreensão do assunto. Antes de tudo, podemos conceituar processos no âmbito geral. A título de ilustração, conjecturarmos o cenário de um processo no sentido jurídico, a ideia de um é entrar com uma quantidade de informações como, por exemplo, argumentos, provas, documentos e um bom advogado - será o intermediador de uma pessoa leiga com a esfera judicial - e termos uma saída, esta dependerá das atividades que ocorrerá dentro desse tempo de atividade.

Figura 1 – Generalização de um processo

Ademais, a partir da imagem e da exemplificação com um cenário longe da abstração computacional, conseguimos encontrar uma definição formal para esse fenômeno. Observamos o processo como um conjunto lógico e sequencial de tarefas, as quais estão inter-relacionadas e interligadas e , também, recebem informações - entradas - de um fornecedor e transformam-nas em saídas, esta que possui um valor agregado para seu cliente, tendo em vista que já é um produto "entregável".

Dessa forma, com essas informações, é possível trazer essa ideia para o eixo computacional. Em computação, um processo é a abstração usada pelo Sistema Operacional para constituir a execução de um programa. Isso é caracterizado por uma thread - Em resumo, thread é a menor unidade de processamento que pode ser executada em um sistema operacional. Na maioria dos sistemas operacionais modernos, uma thread existe dentro de um processo e um único processo pode conter várias threads -. Assim, para manter essas tarefas, conseguimos observar um processo em funcionamento e para ele se manter, ele dispõe de recursos como, por exemplo, memória, arquivos, processos filhos, tratadores de sinais, entre diversas outras funcionalidades para auxiliá-lo em sua produção do produto "entregável".

2.2 Threads

A compreensão sobre threads é crucial para o desenvolvimento do raciocínio sobre a sincronização e comunicação entre os processos de um programa. A thread é caracterizada como a menor unidade de processamento de um programa, semelhante ao bit, que é a menor unidade lógica para as máquinas. Essa conjuntura pode ser entendida como um pequeno programa que trabalha como um subsistema, seria uma forma de um processo computacional ser subdividido em duas - tendo em vista que um processo possui uma thread principal - ou mais tarefas. Em outras palavras, podemos denotar thread como sendo um encadeamento de execução.

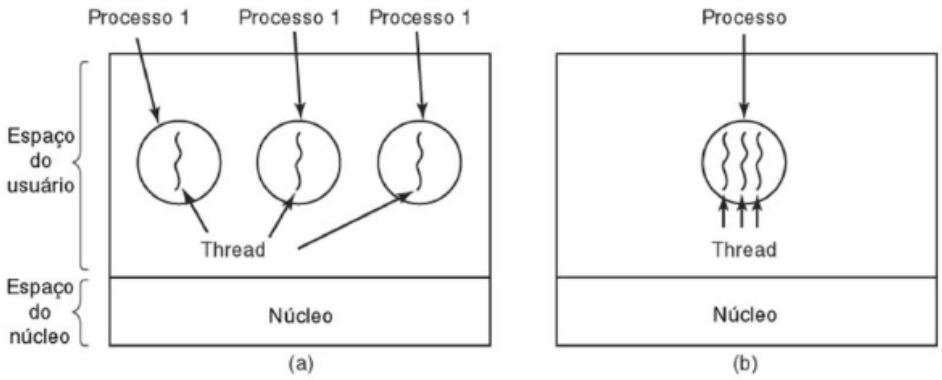

Figura 2 – Exemplos da existência de uma thread dentro de um programa

a) três processos cada um com uma thread.

b) um processo com três threads. 2

Em adição, as threads que existem em um programa podem trocar dados e informações entre elas e compartilhar os recursos postos a elas durante o funcionamento do programa, isso também inclui a mesma célula de memória. Desse modo, um usuário pode usar outras ferramentas do sistema enquanto as threads trabalham de forma oculta no sistema operacional. Podemos ilustrar esse cenário da seguinte maneira: Gabriel é usuário do Linux e ele permitiu que seu computador pudesse ser acessado de forma remota via ssh. Gabriel precisa criar 20 pastas em seu computador para organizar as disciplinas de sua faculdade como, por exemplo, pastas “Geometria Análitica”, “Sistemas Operacionais”, etc. William, um certo dia, decidiu se oferecer para facilitar essa atividade para seu amigo. De forma genial, Gabriel deu a ideia de passar seu usuário ssh para William e assim o convidado possa criar as pastas. A forma mais interessante de fazer essa atividade seria William e Gabriel trabalharem juntos na criação das pastas, com a condição de ambos não criarem uma pasta com o mesmo nome ao mesmo tempo.

Seguindo o problema anterior, o William e Gabriel seriam duas threads em funcionamento, criando diversas pastas - tarefa a ser processada - e o ssh seria a porta que

permitiria essa ocorrência. O espaço de memória compartilhado seria a pasta raiz, na qual as outras subpastas seriam criadas. Logo, conseguimos ver de forma macro como as threads funcionam em prol de uma atividade.

Embora isso pareça encantador, as threads possuem desvantagens também, uma delas é que o trabalho fica mais complexo com a quantidade de threads e isso surge justamente por causa da interação que ocorre entre elas. Vimos isso quando impomos uma condicional para todo orquestramento da criação das pastas, mesmo que 2 pessoas estejam criando pastas de locais diferentes sem se comunicar, ambas podem colidirem, fazendo uma mesma sub-atividade, que seria criar pastas com os mesmo nomes.

Também, é importante distinguirmos a diferenciação de processos e threads, ambos parecem iguais, entretanto um é a unidade do outro. O que melhor distingue uma thread de um processo é o espaço de endereçamento. Todas as threads de um processo trabalham no mesmo espaço de endereçamento, que é a memória lógica do "processo hospedeiro". Isto é, quando se tem um conjunto de threads dentro de um processo, todas as threads executam o código do processo e compartilham as suas variáveis. Por outro lado, quando se tem um conjunto de processos, cada processo trabalha num espaço de endereçamento próprio, com um conjunto separado de variáveis.

2.3 Definindo Paralelismo: Fork, Quit e Join

Para construir um programa concorrente, antes de mais nada, é necessário ter a capacidade de **especificar o paralelismo** dentro do programa. Essa especificação pode ser feita de diversas maneiras. Uma delas utiliza os comandos **fork, quit e join** (CONWAY, 1963).

O comando (ou função) fork pode ser implementado de duas maneiras distintas: ele pode criar um novo processo ou criar apenas um novo fluxo de execução dentro de um processo já existente. A função fork retorna um número inteiro que é a identificação do novo processo ou do novo fluxo criado. Um fluxo de execução também é denominado linha de execução ou thread.

Os comandos quit e join são auxiliares ao fork. Quando o comando quit é executado, o processo (ou thread) que o executa termina imediatamente. O comando join(id) bloqueia quem o executa até que termine o processo (ou thread) identificado por id.

Dessa forma, operações primitivas como essas permitem o manejamento de processos e threads, facilitando o trabalho do sistema operacional. Visto que elas partem desde a criação, até o gerenciamento das sub-atividades.

2.4 Paralelismo em Processos

Paralelismo é sobre a execução paralela de tarefas, ou seja, mais de uma por vez (de forma simultânea), a depender da quantidade de núcleos (cores) do processador. Quanto mais núcleos, mais tarefas paralelas podem ser executadas. É uma forma de distribuir processamento em mais de um núcleo.

Um programa é considerado paralelo quando este é visto como um conjunto de partes que podem ser resolvidas concorrentemente. Cada parte é igualmente constituída por uma série de instruções sequenciais, mas que no seu conjunto podem ser executadas simultaneamente em vários processadores.

Como já mencionado, os processos paralelos são a aversão dos processos seriais. Em vista da multiplixedidade que é posta nesse eixo. O processamento serial é totalmente dependente de uma sequência, todos os componentes de uma determinada tarefa são computados por dependências, um deve terminar para que outro possa iniciar.

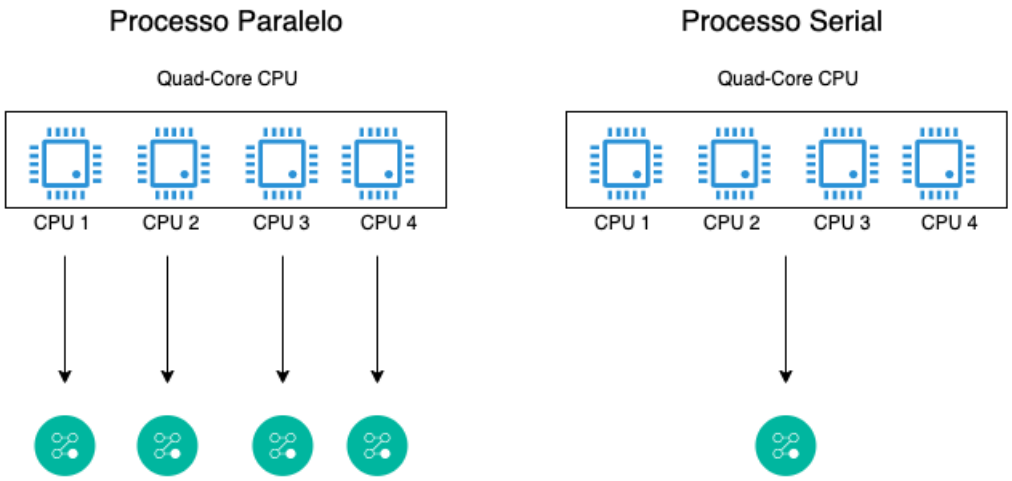

Figura 3 – Exemplificação do processamento paralelo e do processamento serial.

2.5 Concorrência em Processos

O sistema de concorrência é o **princípio básico para o projeto e a implementação dos sistemas multiprogramados** (CONWAY, 1963). A possibilidade de o processador executar várias tarefas ao mesmo tempo permite que vários programas sejam executados concorrentemente pelo sistema operacional.

A utilização concorrente da UCP deve ser implementada de maneira que, quando um programa perde o uso do processador e depois retorna para continuar o processamento seu estado deve ser idêntico ao do momento em que foi interrompido. O programa deverá

continuar sua execução exatamente na instrução seguinte àquela em que havia sido interrompido. Para o usuário este processo é transparente, ficando ao usuário a percepção de que o computador está totalmente dedicado ao usuário e a mais nenhum outro processo. Com isso, é exatamente este mecanismo de concorrência que permite o compartilhamento de recursos pelos vários programas sendo executados pelo processador.

Esse poderoso mecanismo não só permite a fluidez e usabilidade do sistema operacional, mas também gera estruturas complexas para serem analisadas, como a comunicação entre todos os processos concorrentes participantes de um conjunto simbólico.

2.6 Sincronização de Processos

Um processo paralelo cooperativo pode afetar ou ser afetado pelos outros processos que estão executando no sistema. Os **processos cooperativos podem compartilhar diretamente um espaço de endereçamento lógico** (SILBERSCHATZ ABRAHAM GALVIN, 2001) (ou seja, código e dados) ou ter permissão para compartilhar dados apenas através de arquivos. O acesso concorrente a dados compartilhados pode resultar em inconsistência de dados. Isso enquadra-se diretamente no problema da sincronização de processos, o qual iremos situar nessa seção.

Para os processos concorrentes não é possível determinar a ordem em que os eventos irão ocorrer seja de leitura, seja de escrita em uma determinada célula da memória. Levando isso por base, diversos processos nessa situação são factos para possíveis problemas de compartilhamento de informação simultânea.

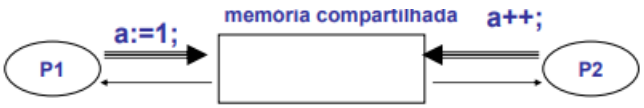

Figura 4 – Demonstração de 2 processos distintos tentando acessar o mesmo espaço da memória, considerae a = 10.

Para esse problema, existem alguns mecanismos que possibilitam essa sincronização e trabalho em equipe entre esses processos. Dessa forma, isso garante a sequência adequada para a execução de eventos. Um dos mecanismos seria o WAIT, no qual um processo consegue ter a capacidade de adquirir o recurso compartilhado, evitando possíveis colisões de acessos. Quando esse processo terminar de usar determinado recurso, ele libera um SGINAL, para avisar que terminou de usar determinado componente da organização.

Desse modo, de forma genérica, é possível observar como o sistema operacional comporta-se diante de situações de acessos mútos ao seus recursos. Alguns serviços aguardam e, depois que terminarem de usar determinados recursos,eles emitem sinais para disponibilizar aquela área.

3 Comunicação Entre Processos

3.1 Definindo Comunicação

Os processos relacionados que estão cooperando para fazer algum trabalho **frequentemente necessitam se comunicar uns com os outros e sincronizar suas atividades** (TANENBAUM, 2008). Nos referimos a isso como comunicação entre processos (Inter Process Communication – IPC). É crucial que as informações entre os processos sejam compartilhadas, entretanto, isso para isso, é necessário que haja um orquestramento dessas relações dos processos com os recursos existentes, para que não haja falhas.

Os processos que estão trabalhando juntos podem compartilhar algum armazenamento comum onde cada um deles pode ler e escrever. O armazenamento compartilhado pode estar na memória principal (possivelmente em uma estrutura de dados do núcleo) ou pode ser um arquivo; a localização exata do compartilhamento não muda , fazendo com que haja padrões no compartilhamento de memória, até mesmo nos erros e nas interrupções, as quais são evitadas.

Uma forma de vermos esse compartilhamento seria em uma aplicação que ler um arquivo e carrega essas informações para um buffer na memória. A partir disso, podemos criar pequenos mecanismos que acessam esse buffer. Nessa situação, é notório que nas ações de escrita ou leitura, o buffer ficaria ocioso, ou dois processos diferentes tentarão acessar a mesma informação, seja como ela esteja estruturada nesse buffer.

3.2 Forma de Comunicação

Dentre diversas formas dos processos se comunicarem, buscaremos a generalização dessa atividade. Os processos, quando precisam se comunicar, é quando há necessidade deles compartilharem recursos ou deles se sincronizarem diante o uso de um mesmo endereço de memória. Uma forma disso ocorrer é fornecermos meios para os processos cooperativos se comunicarem entre si através de um recurso de comunicação entre processos (IPC).

O IPC é um meio que possibilita um mecanismo para permitir que os **processos comuniquem e sincronizem suas ações sem compartilhar o mesmo espaço de endereçamento** (SILBERSCHATZ ABRAHAM GALVIN, 2001). O IPC é particularmente útil em um ambiente distribuído no qual os processos em comunicação podem residir em diferentes computadores conectados via rede.

As mensagens enviadas por um processo podem ser de tamanho fixo ou variável.Se os processos T1, T2 e T3 quiserem se comunicar, deverão enviar e receber mensagens um do outro; um canal de comunicação deve existir entre eles. Esse canal pode ser implementado de

várias formas. Estamos preocupados não com a implementação física do canal , mas sim com sua implementação lógica. Existem vários métodos para implementar logicamente um canal e as operações de send/receive. Um deles é a comunicação direta, nesse método cada processo que deseja se comunicar precisa identificar explicitamente o destinatário ou remetente da comunicação.

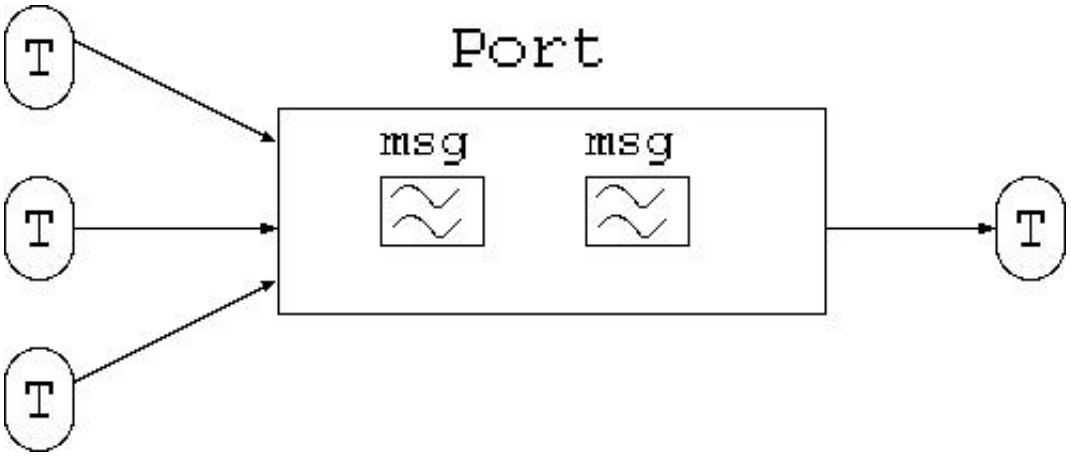

Figura 5 – Exemplo de comunicação de muitos para um , vários processos tentam se comunicar com um, de forma direta, podemos imaginar PORT como sendo um mecanismo IPC

.

3.3 Escalonamento de Processsos

O **escalonamento de processos é o ato de escolher qual processo executar na CPU em um determinado momento** (CARVALHO,). Em outras palavras , é o ato de decidir qual processo será executado na CPU em um determinado momento. O escalonamento de processos ajuda a melhorar o desempenho e a capacidade de resposta de um sistema computacional. É principalmente executado por sistemas operacionais. Os processos que ele gerencia são chamados de thread. Isso ocorre porque eles representam um trabalho útil feito por um computador. O escalonamento de processos é comumente referido como o escalonamento de processos ou agendador de tarefas. Este último significa agenda de tarefas, e o primeiro significa agir de forma organizacional. Em outras palavras, é a atividade do gerenciador que escolhe qual processo irá ter prioridade na CPU. O agendador de tarefas é uma ferramenta fundamental na rotina do gestor de agenda e muitas vezes é complexo de manusear. O escalonamento de processos ou agendador de tarefas é uma atividade organizacional feita pelo escalonador. Ele controla o que cada usuário vai executar no sistema ao definir as tarefas a serem executadas pelos usuários e os seus respectivos processos. Assim como as planilhas do excel, os softwares executados pelo computador são processados de várias maneiras.

Eles podem aprender mais em bases para Linux, Windows ou em outra plataforma específica. Existem softwares básicos que são executados por esses softwares e que facilitam

esses processos decisórios, como edições imediatas das bases eletrônicas - adaptabilidades imediatas - difíceis - arranjos - programação - desenvolvimento web - desenvolvimento de software embaralhado - análise financeira - gestão de marketing - entre outras. Esses agendamentos são feitos para nos ajudar em nossa rotina diária, e sem eles nossa vida seria muito mais chata e lenta do que ela é. A programação do processo pode ser bastante complicada e envolve muitos fatores, como carga da CPU e utilização da memória. Se alocar muitos ciclos de CPU para uma tarefa específica causar utilização excessiva de memória, o sistema ficará lento devido a recursos insuficientes disponíveis para outras tarefas.

Os algoritmos mais usados para escalonar processos são (AKITA,):

1. Algoritmo Round Robin
2. Algoritmo Shortest Job First
3. Algoritmo Priority Scheduling

O Algoritmo Round Robin é um algoritmo de escalonamento que prioriza processos de menor duração. O algoritmo funciona criando um ciclo de processos que são alternados, sendo executados por um tempo pré-definido, quando um processo excede esse tempo, ele é colocado no final da fila para ser executado novamente.

O Algoritmo Shortest Job First (SJF) é um algoritmo de escalonamento que prioriza os processos com menor tempo de execução, ou seja, o processo que possui o menor tempo de execução será o primeiro a ser executado.

O Algoritmo Priority Scheduling é um algoritmo de escalonamento que prioriza os processos com maior prioridade

4 Conclusões

Chegamos ao entendimento da importância qua a comunicaçao, sincronização e a concorrência incrementam um bom funcionamento do Sistema Operacional, como os Threads realizam e organizam a demanda de processos concorrentes, auxiliando na velocidade de comunicação entre esses processos, entregando mais velocidade no tempo de resposta desses processos, e como uma peça tão pequena no processamento tem uma grande importância, influenciando e realizando uma mudança única e importante, dessa forma prosseguindo aprendemos também sobre a troca de dados e informações que ocorrem entre esses processos e como sua comunicação é essencial para que isso ocorra de forma objetiva , além de explorarmos como funciona o paralelismo no qual apresenta características de ser um paralelismo serial ou paralelo, isso facilita em como vários processos podem ser executados ao mesmo tempo sem ter que depender de uma sequência que acaba limitando o sistema operacinal, ocasionando numa perda de velocidade e como consequência perda de desempenho, além disso é extremamente necessário a possibilidade de implementar concorrência nos processos pois isso facilita a criação de sistemas multiprogramavéis que são mais capacitados, velozes e que possibilitam a realização de várias atividades simultaneamente, agregando em um sistema operacional mais fluido e melhor para o úsuario, ou seja é importante que metódos novos e configurações atualizadas sejam sempre implementadas em computadores novos com sistemas operacionais melhores e mais desenvolvidos para que a todo momento máquinas sejam mais capacitadas a realizarem atividades de alta complexidade e a velocidade e tempo de resposta diminuam a ponto de adquirirmos resultados precisos em uma quantidade de tempo pequena e de fato precisa, além de facilitar a vida do úsuario e melhorar tanto a segurança como o conforto ao utilizar determinado programa.

5 Participacoes

Aqui está as devidas referências para os membros da equipe de pesquisa que dedicaram-se e realmente propuseram-se a editar ou contribuir com o mínimo de conhecimento no artigo, todo citados nesta página pelo menos

1. WILLIAM CARDOSO BARBOSA (Estrutura do Artigo, Introdução, Revisão do Resumo e Conclusão, Criação de Imagens, Dissertado: Processos, Threads, Definindo Paralelismo, Paralelismo em Processos , Sincronização de Processos, Definindo Comunicação e Forma de Comunicação.)
2. BRUNO MCPHERSON SIMÔA CHIAMENTI (Resumo, Conclusão e Revisão)
3. JOÃO PEDRO DIAS MAGALHÃES (Revisão)
4. GABRIEL SERGIO SALDANHA DA SILVA (Escalonamento de Processos e Revisão)
5. KAROLYNE IMACULADA ANDRADE MUNIZ (Revisão)
6. JOAO GUILHERME TAVARES REIS (Revisão)

REFERÊNCIAS (TODAS)

AKITA, F. *Concorrência e Paralelismo (Parte 1) | Entendendo Back-End para Iniciantes (Parte 3)*. Disponível em: <https://www.youtube.com/watch?v=cx1ULv4wYxM&feature=youtu.be>. 13

CARVALHO, C. E. M. de. *Sistemas Distribuídos - Análise de Desempenho (Escalonamento de Processos)*. Disponível em: <https://www.linkedin.com/pulse/sistemas-distribu%C3%ADdos-an%C3%A1lise-de-desempenho-carlos-eduardo?trk=articles_directory&originalSubdomain=pt>. 12

CONWAY, M. E. A multiprocessor system design. *Proc. AFIPS Fall Joint Computer Conference*, p. 139–146, 1963. 7, 8

SILBERSCHATZ ABRAHAM GALVIN, P. G. G. *Sistemas Operacionais - Conceitos E Aplicacoes*. [S.l.]: Campus; 8ª edição (1 janeiro 2001), 2001. ISBN 978-8535207194. 9, 11

TANENBAUM, A. S. *Sistemas Operacionais: Projeto e Implementação*. [S.l.]: Bookman; 3ª edição, 2008. ISBN 978-8577800575. 11