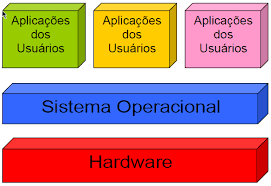
**Conceito de sistemas operacionais**

Os sistemas operacionais essencialmente são softwares que operam diretamente no hardware como uma camada de abstração entre o usuário e os recursos da máquina. Via de regra, todas as operações relacionadas aos componentes físicos de um computador como unidades de disco, gerenciamento de memórias, manipulação de pacotes de rede; exigem extensos e complexos comandos de baixo nível para realizarem uma determinada rotina. O que, certamente, tornaria o uso da máquina sem um Sistema operacional inviável.

Um exemplo que demonstra essa capacidade dos sistemas operacionais de tornar os computadores mais simples é o uso dos discos rígidos SATA, que apresenta uma descrição inicial da interface do disco contida em um livro de 450 páginas (ANDERSON, 2007), ou seja, o que o programador deveria saber para usar o disco a nível de hardware. Em vez disso, com um software chamado de driver de disco instalado no sistema operacional poderia fazê-lo de maneira muito simples e intuitiva.

Além dessa visão que caracteriza o sistema operacional como uma máquina estendida, ele também é responsável por gerenciar todos os recursos do hardware para a execução de todos os processos.

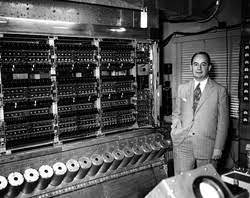


**Contexto histórico de sistemas operacionais**

Primeira geração de computadores (1945-1995):

O primeiro computador digital funcional foi criado na universidade de Iowa. Ele usava 300 válvulas. Nessa mesma época que, Konrad Zuse construiu em Berlim, o Z3, o primeiro computador a partir de relés eletromagnéticos.

De um ponto de vista mais pragmático, os computadores dessa geração ainda não usavam Sistemas Operacionais, o que obrigavam os programadores a usarem a linguagem máquina para realizar uma determinada operação que geralmente se resumia em resolver problemas matemáticos.



Segunda geração (1955-1965):

Essa fase foi marcada pela implementação dos primeiros transistores. O que tornava os computadores mais confiáveis e capazes de funcionar por tempo suficiente para realizar algum trabalho útil. Pela primeira vez, houve uma separação entre projetistas, construtores, operadores, programadores e pessoal da manutenção.

Nessa fase, os computadores eram grandes máquinas chamadas de MAINFRAMES que ficavam em grandes salas climatizadas. Para executar uma tarefa, o programador escrevia seu programa em linguagem de montagem (FORTRAM ou ASSEMBLY) e as convertia em forma de instruções em cartões perfurados. Depois levava esses cartões para a sala de entradas e passava para um operador. O operador executava esse código e entregava o resultado da execução na sala de saídas para que o programador pudesse busca-lo mais tarde.

Ainda nessa segunda fase, como forma de otimizar esse processo foi adotado o sistema em lote(batch). A ideia por trás disso era reunir um lote de tarefas passadas pelos programadores em fitas magnéticas usando um computador pequeno (IBM1401), para executa-las todas de uma vez. Após essa etapa, os resultados dos códigos compilados eram escritos em uma nova fita que era impressão por um outro IBM1401.

Essa estratégia de otimização proporcionada pela integração entre o IBM7094 e o IBM1401 que permitia a execução de vários programas sequencialmente teve uma grande influência nos modelos atuais de desktops que estendem esse conceito de multiprogramação.

A terceira geração (1965-1980):

Essa fase foi marcada pela implementação dos circuitos integrados. Até então, a maioria dos fabricantes tinham duas linhas de produtos (grandes computadores científicos ou pequenos computadores comerciais orientados a caracteres). Com o tempo, os usuários dos computadores ficavam limitados em suas tarefas por conta da capacidade computacional de seus equipamentos. Com base nessa demanda, a IBM tentou solucionar esse problema introduzindo o primeiro sistema operacional (Systyem306) compatível com os dois computadores de sua linha.

Embora esse sistema não tenha obtido grande êxito em sua implementação por conta de seu complexo código, repleto de falhas, que não rodava como o esperado nas maquinas comerciais, ele foi um grande marco para o que conhecemos atualmente como sistemas operacionais por conta da implementação do Time-sharing, uma variante da multiprogramação, na qual cada usuário tem um terminal on-line, onde a CPU pode ser alocada para apenas 3 tarefas que demandam serviço.

Como evolução dessa estratégia, projetistas desenvolveram o MULTICS que envolvia o uso de um único computador fornecendo suporte a centenas de maquinas em Boston. Na realidade o MULTICS foi um sucesso relativo porque só conseguia proporcionar capacidade computacional pouco mais poderosa do que um 386Intel. (Essa ideia é algo próximo do que conhecemos como computação em nuvem).

Apesar da falta de sucesso, o MULTICS influenciou enormemente os sistemas operacionais atuais como o UNIX e seus derivados (LINUX, IOS, Android).

***Observação:*** *o POSIX é uma regulamentação que define uma interface minimalista de chamadas de sistema á qual os sistemas UNIX devem seguir.*



Quarta geração (1980-1990)

Essa geração foi marcada pelo desenvolvimento dos circuitos integrados em larga escala, que são chips contendo milhares de transistores. (3° fase = surgimento | 4° fase = desenvolvimento)

Os sistemas operacionais que marcaram essa fase foram: DOS, MAC OS X e Windows.

Além disso outro fator marcante foi o desenvolvimento da internet.

Quinta geração(1990-HOJE)

Geração marcada pela difusão da internet e aplicações no modelo cliente servidor. (Com suporte ao TCP/IP)

Além disso é nesse contexto que surgiram os sistemas de tempo real.

**Tipos de Sistemas Operacionais**

1. Sistemas operacionais de computadores de grande porte (MAINFRAMES): por serem computadores de processamento de dados de grandes corporações, possuem como principal diferencial a alta capacidade de I/o. Por conta disso, os sistemas operacionais desses computadores são intensamente orientados a processamento de muitas tarefas ao mesmo tempo, que pode ser realizada em lote(batch), processamento de transações e tempo compartilhado. Em geral esse tipo de máquina usa o OS/390(descendente do OS/360), mas pouco a pouco estão sendo substituídos por variantes do UNIX como o Linux.
2. Sistemas operacionais de servidores: em geral, os servidores são grandes máquinas que apresentam uma serie de redundância em seus componentes internos para garantir que consiga atender, initerruptamente, a todas as requisições feitas por outras máquinas conectadas a uma rede. Os sistemas operacionais mais comuns nesse tipo de computação são: Linux, Windows Server e Solaris.
3. Sistemas operacionais de multiprocessadores: uma das formas de se conseguir um potencial computacional é por meio da conexão entre múltiplas CPUs em um único sistema. Essas máquinas precisam de S.Os capazes de integrar os vários processadores. Entre os mais comuns nesse caso são: Windows e Linux.
4. Sistemas operacionais para computadores pessoais: são os sistemas mais comuns encontrados nos ambientes residenciais e comercias. Sua principal característica é o suporte a multiprogramação. Sistemas mais comuns: Linux, IOS, Windows.
5. Sistemas operacionais para computadores portáteis: dispositivos portáteis como celulares, tablets, smartwatch tem em comum a necessidade de dar suporte a multiprogramação e serem capazes de atender a diversos aplicativos instalados pelos usuários (muito semelhante aos computadores pessoais). Os sistemas mais comuns são: Android, IOS.
6. Sistemas operacionais embarcados: são sistemas responsáveis pelo gerenciamento de dispositivos que não costumam ser vistos como computadores. Em essência, os sistemas operacionais embarcados não aceitam softwares instalados pelos usuários, por conta disso, fazem parte do grupo dos sistemas monoprogramados (executam apenas um programa principal). Exemplos onde esse tipo de sistema é encontrado são: DVDs, Televisores, Automóveis. (QNX e VxWorks)
7. Sistemas operacionais de nós de sensores: existem aplicações em que é desejado coletar dados de uma determinada área. Para isso, dezenas de computadores são dispersados na região para enviar algum tipo especifico de dados. Esses computadores são chamados de nós de sensores. Em geral, são programados para rodar um sistema operacional simples e capaz de operar em rede. Estes enviam dados para uma central até que suas baterias sejam descarregadas. Exemplo de sistemas usados nesse tipo de operações: TinyOs.
8. Sistemas operacionais de cartões inteligentes: são sistemas projetados para serem executados em pequenos chips inseridos em cartões. Quando são expostos a algum sensor, eles são carregados por indução e executam um programa simples.
9. Sistemas operacionais em tempo real: há casos em que o processamento de dados deve estar sincronizado com o tempo real. Em geral, sistemas aviônicos e de linhas de montagem exigem esse grau de precisão de processamento. Por conta disso, o parâmetro tempo é um fator crucial nesse tipo de sistema.

Entre os sistemas operacionais de tempo real existem: sistemas operacionais de tempo real crítico (Aviônica e Linha de montagem) e sistemas operacionais de tempo real não crítico (Áudio digital e multimídia);

**Estruturas de sistemas operacionais**

Sistemas operacionais monolíticos: são as organizações mais comuns nos sistemas operacionais. Todo o sistema operacional é executado como um único programa em modo núcleo. Basicamente, ele é divido em dois módulos (núcleo e usuário). Sendo assim, toda a programação do sistema está contida no modo núcleo o que o torna mais eficiente é que qualquer estrutura pode chamar qualquer rotina. Além disso, o tempo de resposta é muito menor.

(UNIX, MAC OS, Windows e Linux)

Resumidamente: sistemas monolíticos podem ser compreendidos como um grande bloco.

Problemas:

* Difícil gerenciar estruturas complexas com grande número de processo
* Documentação e difícil compreensão
* Fácil de torna um sistema instável.

Sistemas operacionais microkernel: nesse caso busca tornar o núcleo do sistema menor o possível. A principal função do núcleo é gerenciar a comunicação entre processos. O grande diferencial desse tipo de sistema é que todos os processos são executados de forma que em caso de erros ou falhar não prejuízos para o modo núcleo do sistema. Isso o torna muito mais estável. Exemplo: MINIX

Estrutura de camadas: nesse caso, a ideia é criar um programa operacional dividido em módulos de menor complexidade. Estabelecendo entre os mesmos relações de hierarquias.

Um exemplo desse tipo de sistema é o MULTICS

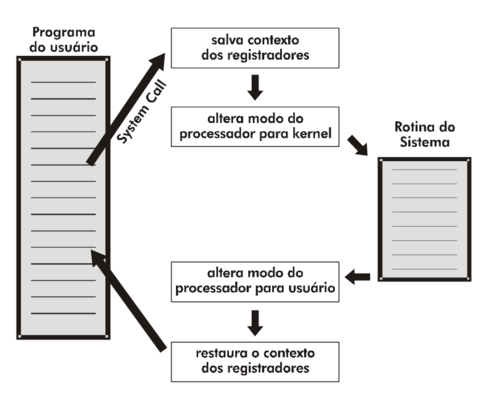
Sistemas micronúcleo tendem a melhorar a segurança, mas piorar o desempenho quando comparados a sistemas monolíticos. Por que?

**Chamadas de Sistemas e interrupção**

Quando uma determinada aplicação em modo usuário precisa executar uma tarefa privilegiada pelo sistema, como acessar o disco, por exemplo. Ela deve fazer uma chamada ao sistema, o que chamamos de SystemCall. Como essas chamadas ocorrem?

As chamadas de sistemas são realizadas por meio das instruções TRAPS, elas são interrupções de software. Após o término da chamada a execução continua do ponto em que parou.

O que é um sistema preemptivo e não-preemptivo? No escalonamento **preemptivo**, o **sistema** operacional pode interromper um processo em execução e passá-lo para o estado de pronto, com o objetivo de alocar outro processo na UCP. No escalonamento **não**-**preemptivo**, quando um processo está em execução, nenhum evento externo pode ocasionar a perda do uso do processador



**Processos**

Em linhas gerais, um processo é definido como uma instancia de um programa com seu próprio contador de programas, estado interno e variáveis (fatores que terminam seu contexto de execução). Nos sistemas multiprogramados, cada processo recebe uma determinada fatia de tempo(time slice) para ser executado na CPU, quando esse tempo termina, seu contexto é armazenado e outro processo da fila entra em seu lugar para o processamento. Dessa forma, quando esse processo voltar para o processador ele pode ser executado do ponto em que foi interrompido.



Criação de processos em diferentes sistemas:

No Unix, há apenas uma chamada de sistema para criar um novo processo, o “fork”. Esse comando cria um novo processo idêntico ao processo pai (com a mesma imagem de memória), em seguida o processo filho executa um “execve” que é responsável por mudar sua imagem de memória e executar um novo programa.

No Windows, um novo processo é criado por meio do comando create process que recebe 10 parâmetros. Esse novo processo, recebe um espaço de memória distinto do processo pai.

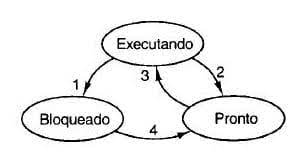
Hierarquia entre processos:

No Unix, todos os processos são gerados a partir de um único processo pai, o “init”. A partir do init é gerada uma árvore de processos filhos organizados hierarquicamente. (Não é possível um processo pai se desligar de um processo filho)

No Windows, não há esse tipo de hierarquia. Quando novo processo é criado, o processo pai pode controlar o processo filho por meio de um parâmetro handle. Este parâmetro pode ser passado para outros processos.

Estados de um processo:

Um dos pontos mais importantes e que certamente serão abordados na avaliação são os estados de um processo.



Embora cada processo seja uma entidade independente, com seu próprio contador de programa e estado interno, processos muitas vezes precisam interagir entre si. Algumas vezes, a saída de um processo pode servir como entrada para outro. Em situações como essa, onde algum dado de entrada esta condicionado a algum evento que ainda deve ocorrer é necessário estabelecer um bloqueio para o processo.

Além da dependência de dados, como sabemos, em sistemas multiprogramados os processos recebem uma parcela de tempo finita na CPU para executar parte de suas tarefas. Quando esse tempo termina, ocorre uma interrupção e todos os dados do contexto do processo atual são guardados para que outro processo possa entrar na CPU.

1. O estado de um processo é alterado para bloqueado quando o mesmo exige que algum evento deva ocorrer para que ele possa continuar sua execução. Exemplo: quando um processo necessita de acessar a leitura de algum arquivo na memória secundaria, ele deve permanecer bloqueado até que a informação seja lida e trazida para a memória principal.
2. O estado de um processo é alterado para pronto quando ocorre uma interrupção na CPU para que outro processo entre em seu lugar. Assim, seu contexto de execução é armazenado e, em seguida, ele é alocado para a fila de prontos até que possa receber uma nova fatia de tempo no processador.
3. O evento em questão ocorre quando um processo é escalonado da fila de processo para entrar na CPU.
4. Quando o evento esperado pelo processo bloqueado ocorre, seu estado é alterado para pronto e é alocado na fila de prontos para que possa continuar sua execução.

OBS: as transições 2 e 3 são causadas pelo escalonador de processo, uma parte do sistema operacional, sem o processo nem saber a respeito delas.

Implementação de processos

Para implementar o modelo de processo, o sistema operacional mantém uma tabela de processo, com uma entrada para cada um deles. Essas entradas contêm informações importantes sobre o estado do processo, incluindo o seu contador de programa, ponteiro de pilham alocação de memória, estado dos arquivos abertos, informação sobre sua contabilidade e escalonamento e tudo mais que leva a ser salvo quando o processo é trocado do estado em execução para pronto ou bloqueado.

* PCB (PROCESS CONTROL BLOCK): é uma estrutura de dados criado pelo sistema operacional para guardar informações sobre cada processo; algumas informações armazenadas:

**Threads**

Conceito de threads: são linhas de execução de um processo que compartilham o mesmo espaço em memória. A principal razão para a utilização delas é o fato delas permitirem que determinadas tarefas de um processo possam ser decompostas e executadas paralelamente. Além disso, podem ser criadas e excluídas com maior facilidade se comparadas com os processos.

* Lembrar do exemplo do editor de texto que usa 3 threads (Editar, alterar formatação e salvar).

Threads também possuem um contador de programas, registradores que armazenam variáveis de trabalho e pilha de execução.

**Diferença principal entre threads e processos:**

*Processos agrupam recursos, threads são escalonadas para executar tarefas na CPU.*

Além disso, é válido lembrar que threads também apresentam 3 estados de execução: PRONTO, BLOQUEADO, EXECUTANDO.

Embora as threads sejam essenciais para a melhora no desempenho das máquinas atuais por proporcionarem o paralelismo entre tarefas de um mesmo processo, elas também podem causar alguns problemas indesejados por conta do acesso compartilhado a determinados recursos críticos. Para compreender esses problemas, considere o seguinte exemplo:

Quando um processo quer imprimir um arquivo, ele entra com o nome do arquivo em

um **diretório de spool** especial. Outro processo, o **daemon de impressão**, confere periodicamente para ver sehá quaisquer arquivos a serem impressos, e se houver,ele os imprime e então remove seus nomes do diretório.Imagine que nosso diretório de spool tem um númeromuito grande de vagas, numeradas 0, 1, 2, ..., cadauma capaz de conter um nome de arquivo. Tambémimagine que há duas variáveis compartilhadas, *out*,

que apontam para o próximo arquivo a ser impresso, e *in*, que aponta para a próxima vaga livre no diretório. Essas duas variáveis podem muito bem ser mantidas em um arquivo de duas palavras disponível para todos os processos. Em determinado instante, as vagas 0 a 3 estarão vazias (os arquivos já foram impressos) e as vagas 4 a 6 estarão cheias (com os nomes dos arquivos na fila para impressão). De maneira mais ou menos simultânea, os processos *A* e *B* decidem que querem colocar um arquivo na fila para impressão. Nas jurisdições onde a Lei de Murphy2 for aplicável, pode ocorrer o seguinte: o processo *A* lê *in* e armazena o valor, 7, em uma variável local chamada *next\_free\_slot*. Logo em seguida uma interrupção de relógio ocorre e a CPU decide que o processo *A* executou por tempo suficiente, então, ele troca para o processo *B*. O processo *B* também lê *in* e recebe um 7. Ele, também, o armazena em*sua* variável local *next\_free\_slot*. Nesse instante, ambos os processos acreditam que a próxima vaga disponível é 7. O processo *B* agora continua a executar. Ele armazena o nome do seu arquivo na vaga 7 e atualiza *in* para ser um 8. Então ele segue em frente para fazer outras coisas.

Por fim, o processo A executa novamente, começando do ponto onde ele parou. Ele olha para o next\_free\_slot, encontrar um 7 ali e escreve seu nome de arquivo na vaga 7, sobreescrevendo o que o processo B recém-colocou ali. Então calcula next\_free\_slot +1, que é 8 e configura in para 8. Assim, o arquivo escrito por B nunca será impresso.

Esse tipo de situação é conhecida como CONDIÇÃO DE CORRIDA.

(Inserir aqui a ilustração sobre regiõe criticas do livro Tanembaun)

**Soluções para resolver problemas relacionados a condição de corrida**

Propostas de exclusão mutua com espera ocupada:

1) Variáveis do tipo trava:

Nesse tipo de proposta é estabelecido uma variavel binaria(Lock) para definir se um processo esta na região critica ou não. Dessa forma, caso o valor de de Lock seja 0 é permitido entrar na região critica, assim o processo que entra na região critica altera o valor dessa variavel para 1. Assim quando o processo sai da região critica ele retorna o valor 0 para essa variavel permitindo que o outro processo possa acessar a região critica.

Veja sua implementação em Linguagem C:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <unistd.h>  #define THREADS 2  #define NUMREP 1000000  long compartilhada =0;  void \*threadFuncIncrement(void \*arg){  int a, b, condicao=1;  for(long j=0; j<NUMREP; j++){  b=1;  while(trava != 0);  trava = 1;  a = compartilhada;  compartilhada = a+b;  trava = 0;  }  }  void \*threadFuncDecrement(void \*arg){  int a,b;  for(long j=NUMREP; j>0; j--){  trava = 1;  a = compartilhada;  compartilhada = a – b;  trava = 0;  }  }  int main(int argc, char \*arg[]){  pthread\_t t[THREADS];  int s;  s = pthread\_create(&t[0], NULL, threadFuncIncrement, NULL);  s = pthread\_create(&t[1], NULL, threadFuncIncrement, NULL);  s = pthread\_join(t[0], NULL);  s = pthread\_join(t[1], NULL);  printf(“Valor da variavel compartilhada: %1ld\n”, compartilhada);  }  (Para compilar use gcc -nomeDoArquivo.c -o nomeExecutavel -lpthread) |

Nessa proposta não foi possível eliminar o erro da condição de corrida porque caso ocorra uma interrupção logo após a saída da região critica, um segundo processo pode ler a veriavel Lock com Zero e entrar na região critica. Quando o processo que foi pausado entrar em execução novamente, ele terminará de mudar o valor da Variavel do tipo trava e também entrará na região crítica.

2) Algoritmo SpinLock

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <unistd.h>  #define THREADS 2 // Numero de threads desejadas  #define NUMREP 10000000  long compartilhada = 0; // memória compartilhada  int turn = 0;  //Função executada em paralelo  void \*processoA(void \*arg){  int a, b;  for(long j=0; j<NUMREP; j++){  b = 1;  while(turn != 0);//Loope até que a trava seja igual a 1  a = compartilhada;  compartilhada = a + b; //incrementando em 1(condicao de corrida)  turn = 1;  }  }  void \*processoB(void \*arg){  int a, b;  for(long j=NUMREP; j>0; j--){  b = 1;  while(turn != 1);//Loop até que trava seja igual a 1  a = compartilhada;  compartilhada = a - b; //incrementando em 1(condicao de corrida)  turn = 0;  }  }  int main(int argc, char \*argv[]){  pthread\_t t[THREADS];  int s;  s = pthread\_create(&t[0], NULL, processoA, NULL); //criado  s = pthread\_create(&t[1], NULL, processoB, NULL); //criado  s = pthread\_join(t[0], NULL); //esperando as threads terminarem  s = pthread\_join(t[1], NULL); //esperando as threads terminarem  printf("compartilhada = %1ld\n", compartilhada);  }  } |

Embora o algoritmo SpinLock consiga resolver o problema da condição de corrida, quando existem dois ou mais processos com rotinas assimétricas(que exigem tempos distintos para a execução) ocorrerá um desperdício de processamento, visto que o processo mais rápido deverá permanecer em espera ocupada até que o processo mais longo saia da região crítica.

Solução de Peterson: esse algoritmo funciona, mas desperdiça processamento com espera ocupada.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <unistd.h>  #define THREADS 2 // Numero de threads desejadas  #define NUMREP 10000000  #define TRUE 1  #define FALSE 0  #define N 2  long compartilhada = 0; // memória compartilhada  int turn = 0;  int interested[N];  void enter\_region(int process){  int other;  other = 1 - process; //Identifica qual é o outro processo  interested[process] = TRUE; //Indica que o processo atual 1 ou zero quer entrar na RC  turn = process;// Turn recebe o processo que chamou essa função  while(turn == process && interested[other] == TRUE);  //Enquanto turn == processo e o outro processo for true permance no looping  //Se o interested[other] trocar para false ele entra na região critica  }  void leave\_region(int process){  interested[process] = FALSE;  }  //Função executada em paralelo  void \*threadFuncIncrement(void \*arg){  int a, b;  for(long j=0; j<NUMREP; j++){  enter\_region(0);  a = compartilhada;  compartilhada = a++; //incrementando em 1(condicao de corrida)  leave\_region(0);  }  }  void \*threadFuncDecrement(void \*arg){  int a, b;  for(long j=NUMREP; j>0; j--){  enter\_region(1);  a = compartilhada;  compartilhada = a--; //incrementando em 1(condicao de corrida)  leave\_region(1);  }  }  int main(int argc, char \*argv[]){  //interested[0] = 0;  //interested[1] = 0;  pthread\_t t[THREADS];  int s;  s = pthread\_create(&t[0], NULL, threadFuncIncrement, NULL); //criado  s = pthread\_create(&t[1], NULL, threadFuncDecrement, NULL); //criado  s = pthread\_join(t[0], NULL); //esperando as threads terminarem  s = pthread\_join(t[1], NULL); //esperando as threads terminarem  printf("compartilhada = %1ld\n", compartilhada);  } |

Conceito de semáforos:

Semáforos são operações atômicas (não sofrem interrupções) implementadas nos sistemas operacionais para contar quantos recursos compartilhados estão disponíveis. Na prática, quando um determinado processo deseja alterar o valor de uma variável compartilhada de forma que não ocorra inconsistência de dados, ele deve chamar dois tipos de funções down(variável) ou up(variável). A função down verifica se o valor da variável passada é maior ou igual que 1, se a condição for verdadeira essa variável é decrementada em uma unidade, caso seja falsa deve permanecer em uma fila de espera de processos. A função up, incrementa o valor da variável passada e em seguida, verifica se há algum processo na fila de processos, se sim um processo é liberado dessa fila para a execução.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <unistd.h>  #include <semaphore.h> //biblioteca para semaforos  #define THREADS 40 //Número de threads desejadas  long compartilhada = 0; // memória compartilhada  sem\_t sem;  //Função executada em paralelo  void \*threadFunc(void \*arg){  int a, b;  for(long j=0; j<100000; j++){  sem\_wait(&sem);  a = compartilhada;  b = 1;  compartilhada = a + b; //incrementando em 1(condicao de corrida)  sem\_post(&sem);  }  }  int main(int argc, char \*argv[]){  pthread\_t t[THREADS];  int s;  sem\_init(&sem, 0, 1);//Inicialização do semaforo  for(int i = 0; i<THREADS; ++i){  s = pthread\_create(&t[i], NULL, threadFunc, NULL); //criado  }  for(int i = 0; i<THREADS; ++i){ s = pthread\_join(t[i], NULL); //esperando as threads terminarem  }  printf("compartilhada = %1ld\n", compartilhada);  // O resultado esperado aqui era que cada uma das 40 threads acrescentassem 100000. No entanto  // o que resultaria em 4000000 mil na variavel compartilhada. O que não ocorrre.  // Use gcc semaforo.c -o executavel -lpthread para compilar esse código  } |

.