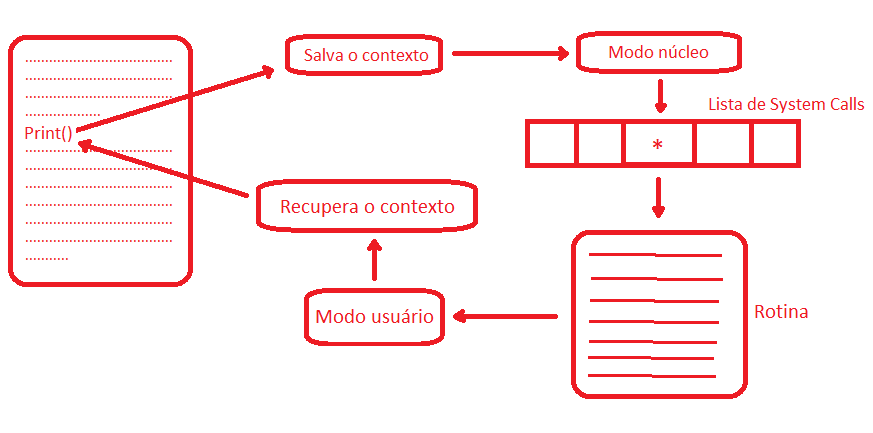
**Abordagem das Funções**

Bottom up

* Focado em eficiência
* Escalonamento da CPU
* Gerência de Memória
* Gerência de Entrada / Saída

Top down

* Focado em comodidade



Programas != Processos

Programa

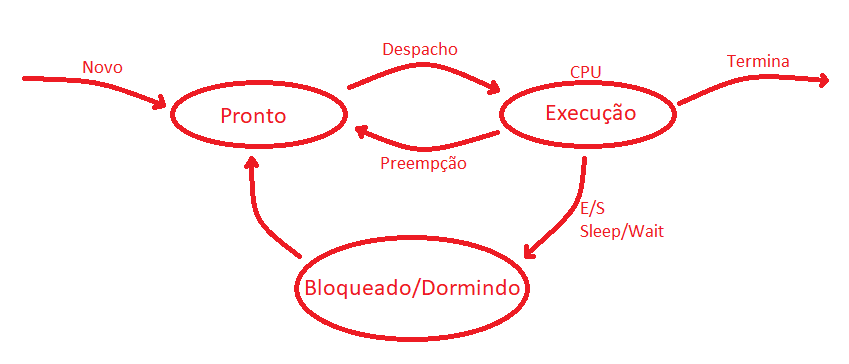
* Lista de instruções/comandos

Processos

* Instância de execução do programa

**Aula 03**

**Diagrama de estados de processos suspendidos:**



* Preempção: O processo para de ser executado para uma tarefa mais importante tomar o seu lugar.
* Suspensão: Quando há memória insuficiente, os processos menos importantes são mandados para o disco.

**Threads:**

* Thread é a tarefa que um determinado programa realiza. Também conhecida como fio/linha/encadeamento de execução. É uma forma de um processo se dividir em duas ou mais tarefas para serem executadas simultaneamente.
* Possuem um ciclo de vida similar aos processos:

Pronto ⇆ Executando

↖ Bloqueado/Dormindo ↲

**Aula 04**

**Projeto de programação com Threads:**

* Threads executam códigos específicos.
* Forks executam a partir do ponto que foram gerados.
* A ideia das threads é que elas sejam executadas em paralelo e em qualquer ordem.
* É mais fácil criar e destruir threads do que destruir processos.
* Comum as threads trabalharem na mesma base de dados.

**Modelos de gerenciamento de Threads:**

* 1:n - Escalonamento apenas dos processos.
* n:n - Escalonamento dos processos e das threads.
* n:m - Escalonamento dos processos e de parte das threads.

Aula: Alguma aí

* **Condição da corrida:**
  + Ocorre quando mais de 1 Thread acessam uma mesma variável ao mesmo tempo, causando inconsistências nos dados.
* **Região Crítica:**
  + Local do código onde é feito um acesso à memória compartilhada.
* **Exclusão Mútua:**
  + Garantir que nunca terá mais de uma thread na região crítica ao mesmo tempo.
* 4 Regras para evitar corridas e concorrências:

1 - **Exclusão mútua:** Garantir 1 thread por vez na região crítica. A Thread que estiver dentro da RC deve bloquear uma thread que quiser entrar.

2 - **Progressão:** Nenhuma thread fora da região crítica pode impedir outra thread de entrar na região crítica.

3 - **Espera limitada:** Uma Thread não pode esperar infinitamente para entrar na região crítica.

4 - **Hardware:** Não deve ser feita presunção de hardware. É necessário funcionar em qualquer sistema.

**21/09/22**:

Semáforos: Utilizados para controlar recursos.

(...)

—----------------------

Problema do produtor consumidor (buffer limitado):

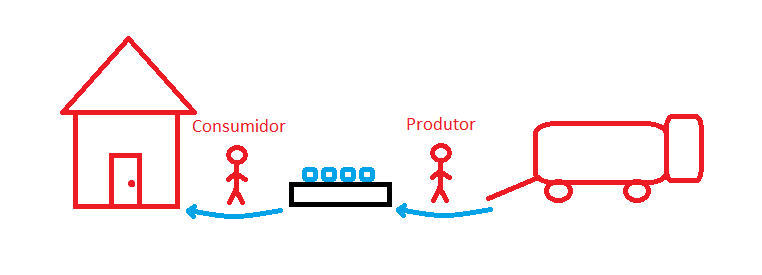
0 1 2 3 4

Buffer | || || || || |

O produtor deseja posições vazias no buffer.

O consumidor deseja itens no vetor.

Exemplo em sala: Estoque.



**Deadlock / Impasse:** O código fica parado em determinado ponto, ocasionado por um recurso indisponível.

**4 condições para ocorrer um deadlock (Juntas):**

**1 - Exclusão Mútua**. Se não tivermos exclusão mútua, um deadlock não acontecerá.

**2 - Uso e espera**. Se não tem uso e espera, não tem deadlock.

**3 - Não há preempção de recursos**.

**4 - Espera Circular**.

Starvation ou Inanição: Condição em que um processo nunca executa (nunca recebe processamento) por causa de uma competição com eventuais processos.

**Prova: Tudo até aqui.**

Recomendação do professor: Moodle -> UNIVESP Aulas 25 e 16 - Tratamentos de Deadlocks.

Conceitos iniciais de SO

**Aula que vem:**

**Anotações do Davizeira:**

- tipos de SOs (2):

- monoprogramáveis/monotarefa;

- multiprogramados/multitarefas.

- Programa em execução: processo/tarefa (job).

- Pode ter múltiplos processos para um mesmo programa.

- SOs monoprogramados executam apenas um único processo por vez (dois processos nunca estarão ativos ao mesmo tempo).

- SOs podem múltiplos processos ao mesmo tempo em memória e há alternância entre processos.

- A CPU não precisa ficar ociosa quando um processo não está sendo executado, ela pode executar outro processo. O conjunto de processos são executados com mais eficiência.

- O SO vai escalonando os processos - cada processo tem uma fatia de tempo para ser executado.

- Multiprogramados = múltiplos processos em um único processador.

- Multiprocessados = múltiplos processos em múltiplos processadores.

- SOs multiprogramados em Lote: execução de processos em ordem pela CPU. SOs sem interação com o usuário.

SOs multiprogramáveis iterativos: dividem os recursos de forma justa para ter múltiplos processos em paralelo (em teoria).

- Preempção: SO multiprogramados podem tirar um processo da CPU porque já acabou seu tempo de execução (mesmo que não tenha sido encerrado) para executarem outro processo.

- Processos não sabem que estão sendo escalonados. Um processo acha que está sendo executado ininterruptamente.

- Um processo pode estar em estado de PRONTO, EXECUTANDO ou DORMINDO/ESPERANDO.

- Quando um processo é finalizado, deixa de existir.

- Quando está dormindo, outro processo precisa acordar esse processo para ficar em estado de pronto.

- Estruturas mais comuns de SOs:

- monolítico: dentro do SO tem muitos programas (todos funcionam como um único programa - como o SO). Possuem hardware, SO e aplicativos.

- microkernel: Programas que fazem acesso ao hardware ficam no espaço de usuário. Possuem o hardware, microkernel e o espaço de usuário (onde ficam os programas que gerenciam o hardware).

- Top Down: Como os apps enxergam o SO.

- Bottom Up: Como o Hardware enxerga o SO.

- A perspectiva Top Down se refere muito à comodidade que os SO's fornecem aos usuários (programadores e processos).

Utilizar os mesmos comandos em diferentes arquiteturas e SO's é uma comodidade que a perspectiva Top Down proporciona aos usuários.

- A perspectiva Bottom Up pensa na eficiência da CPU, para que a mesma não fique ociosa.

- Ao executar uma system call, o programa é pausado e o seu contexto é salvo (o conteúdo dos registradores é salvo em memória) e entramos em modo núcleo para chegarmos em uma lista de chamadas de sistema que, dependendo de qual for executada, um programa/código específico é executado referente a system call. Ao terminar de executar a system call, voltamos ao modo user e recuperamos o contexto (conteúdos dos registradores salvos são recuperados na CPU).

- O software solicita a troca de user mode para modo núcleo. As interrupções ocorrem em nível de hardware (um sinal que vem de um dispositivo de hardware). Também é preciso salvar o contexto e entrar no modo núcleo. Após isso, ao observar a controladora de interrupções, é necessário encontrar o código que vai manipular a interrupção. Terminando, recupera o contexto e volta ao modo user.

- Interrupções podem ser causadas por system calls no momento em que são finalizadas. Por exemplo, enquanto uma system call executa uma operação no disco (bloqueando o processo corrente), um outro processo pode ser executado na CPU até que a operação seja finalizada. Ao finalizar, é gerada uma interrupção por parte do disco alertando que a operação terminou e o escalonador de processos pode decidir se vai continuar executando o processo anteriormente bloqueado ou processo que passou a ser executado na CPU.

- Um processo é uma instância de execução de um programa. Um processo tem algo chamado de "bloco de controle de processo". Esse bloco tem um espaço de contexto com informações sobre o processo, por exemplo, seu identificador (PID).

- Um novo processo pode ser criado ao início do sistema (em seu boot).

- Interação com usuário também cria um novo processo (por exemplo, quando o usuário executa um programa).

- Tirando o início do sistema, a criação de processos se resume a processos criando outros processos.

- Em ambientes Linux, processos podem ser criados a partir de forks: chamadas de sistemas que clonam processos, por exemplo, um processo pai gera um processo filho.

- Um processo pode ser encerrado de 4 maneiras diferentes:

- Quando finaliza (voluntário);

- Erro não fatal. Por exemplo, quando o tratamento de exceções não dá conta da exceção, o programa é encerrado (voluntário);

- Um processo mata outro processo (término involuntário);

- Erro fatal/crash (término involuntário).

- Quando um processo é criado, é colocado em estado de PRONTO (pronto para ser executado na CPU).

- Quando um processo é executado por muito tempo, ele sofre uma preempção (acabou o tempo do processo e outro em estado de PRONTO é executado). O processo interrompido volta a ficar em estado de pronto.

- Um processo pode se bloquear (dormir) quando estiver impedido de continuar sua execução e alguma outra operação que não use a CPU esteja sendo executada no momento (por exemplo, uma leitura no disco).

- Um processo suspenso é mandado para o disco (processos PRONTOS ou BLOQUEADOS podem ser suspensos - permanecem no mesmo estado, porém no disco) quando não há memória suficiente e há muitos processos.

- O SO opta por suspender processos menos importantes.

- Quando tiver memória disponível, o processo deixa de estar suspenso.

- A tabela de controle de processos é montada em memória.

- Quando um fork é feito, um processo é clonado (filho) de um processo original (pai). Com exceção do ID, todo o processo é exatamente igual.

- Threads permitem paralelismo dentro de processos.

- Threads vem de "Threads of Execution" (linhas de execução). Um único processo pode ter várias linhas de execução em paralelo.

- As threads precisam ser independentes umas das outras.

- Todo processo tem pelo menos uma thread. Threads compartilham o mesmo bloco de controle de processo.

- Uma vantagem de usar threads em relação a processos é que threads se comunicam mais facilmente.

- Cada thread vai ter uma stack diferente e um contexto individual de registradores. De resto, a área de dados estáticos, de dados dinâmicos e de código são as mesmas.

- Threads têm um ciclo de vida similar ao de um processo: estado de PRONTO, EXECUTANDO, BLOQUEADO. A thread não tem estado de SUSPENSO - o processo do qual a thread faz parte é suspenso.

- É muito comum ter threads bloqueadas esperando que outras threads acordem. Isso não é tão comum com processos.

- SO's antigos não conheciam as threads dos processos, apenas os próprios processos. O escalonamento das threads aconteciam por parte dos processos. n threads eram vistas como um único processos pelos SOs. Os SOs possuíam uma tabela para escalonar os processos. Não era possível threads de um único processo sendo executadas em vários processadores. Modelo n:1.

- Em SO's modernos, o escalonamento de threads acontece por parte do próprio SO (ele conhece todas as threads), o que permite paralelismo real. Threads de um mesmo processo podem ser executadas em múltiplos processadores. O SO tem tabelas de processos e tabelas de threads. Modelo n:n.

- Ainda existe o modelo n:m, da qual o SO pode não ver todas as threads. Por exemplo, enxerga duas threads como apenas uma entrada de thread - uma junção de threads.

IPC:

- Condição da corrida: ocorre quando dois ou mais threads acessam uma mesma variável ao mesmo tempo, causando inconsistência nos dados.

- Um dos problemas da programação com threads é a condição da corrida.

- A região crítica é o pedaço do código onde pode ocorrer a condição da corrida (o local do código onde é feito um acesso a memória compartilhada).

- A maneira de evitar a corrida é garantir a exclusão mútua (não permitir que várias threads acessem a região crítica ao mesmo tempo).

- Há quatro regras para evitar corrida:

- 1: Exclusão mútua: apenas uma thread na região crítica.

- 2: Progressão: nenhuma thread fora da região crítica pode impedir outra thread de entrar na região crítica (uma thread na região crítica deve proibir outra thread de entrar na região crítica).

- 3: Espera limitada: uma thread não deve esperar infinitamente para entrar na região crítica.

- 4: Hardware: não deve se considerar as características do hardware: quantidade de núcleos, memória, etc. (precisa funcionar em qualquer hardware).

Uma thread que não pode entrar na região crítica pode dormir (bloquear) ou entrar em estado de "espera ocupada" (continua em estado de execução, verificando em um loop quando pode entrar na região crítica). Ex:

Desabilitar interrupções quando uma thread entra na região crítica é uma técnica que não tem garantia de evitar a condição de corrida (em máquinas com multiprocessadores isso não funciona).

Alternância: uma variável seta qual thread está na vez de entrar na região crítica (alterna qual thread irá entrar na região crítica). Isso desrespeita a progressão, pois uma thread fora da região crítica pode impedir outra de entrar na região crítica.

Solução de Peterson: Originalmente proposto para duas threads. A ideia é ter um vetor compartilhado de "interesse" (flag) com duas posições (uma para cada processo).

- Semáforo é um contador de recursos (basicamente uma variável inteira que apenas permite valores >= 0). Tem apenas duas operações: incremento e decremento (feitas de forma atômica em baixo nível, ex.: "tsl" e "exch"). Tem a garantia de não sofrer preempção, haja vista que as operações são atômicas (executadas em uma única unidade de tempo). A thread dorme quando o valor que é 0 tenta ser decrementado.

Aula \_ \_:

**Escalonamento de Processos:** Permite uma melhor distribuição do processador entre os processos. Não existe intervenção do Sistema Operacional na execução do processo.

**FCFS:** First Come, First Served // "Primeiro a chegar, primeiro a ser servido"

* É o algoritmo de escalonamento de CPU mais simples que executa o processo que vem primeiro.
* É um algoritmo de escalonamento para estruturas de dados do tipo fila.

**SJF ou SJN:** Shortest Job First ou Shortest Job Next.

* Vantajoso por sua simplicidade e também porque minimiza o tempo médio que cada processo leva desde quando ele é criado até o fim de sua execução.

**HRRN:** Highest Response Ratio Next

* Inviável pois o tempo executando não é previamente conhecido.

**SRTN:** Shortest Remaining Time Next

Algoritmo Relógio em fila circular (Prova)

Algoritmos não usados recentemente:

* NRU
* LRU (Boa eficiência em termos de acerto, mas custa muito processamento