**Processos e compartilhamento de memória**

**Processo** = instância de um programa em execução.

Podemos ter processos em dois tipos de ambientes em servidores:

* **Transacional** = onde os processos são acionados por requisições que chegam ao servidor de forma aleatória oriundas de usuários (pessoas ou outros softwares) não sincronizados entre si. Pode-se dizer que esses usuários não têm notícia do que os outros estejam fazendo. Eles simplesmente enviam sua requisição assim que necessitem, esteja o servidor congestionado ou não. Ou seja, não é uma preocupação delas.
* **Processamento em lote** (ou *batch*) = onde os processos são instanciados segundo uma programação (*schedule*) previamente estabelecida. Neste ambiente não há demanda proveniente de agentes externos. Toda a execução segue o cronograma.

**Estrutura de um processo:**

|  |  |
| --- | --- |
| Figura 1 | * **Text** = instruções do programa * **Data** = variáveis estáticas * **Stack** = para armazenamento de variáveis locais * **Heap** = para alocação dinâmica (new, malloc). |

**Estados de um processo**

|  |  |
| --- | --- |
| Figura 2 | * **New** - processo está em estágio de instanciamento. * **Ready** - está pronto (instanciado, variáveis inicializadas etc) mas a CPU ainda não iniciou a sua execução. * **Running** - Em execução. * **Terminated** - processo encerrado (todos os recursos de máquina são liberados). |

**Process Control Block (PCB)**

É uma espécie de planilha onde estão as informações referentes a um dado processo.

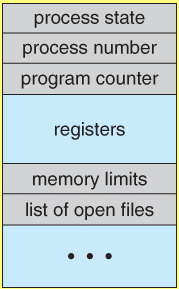


Figura 3

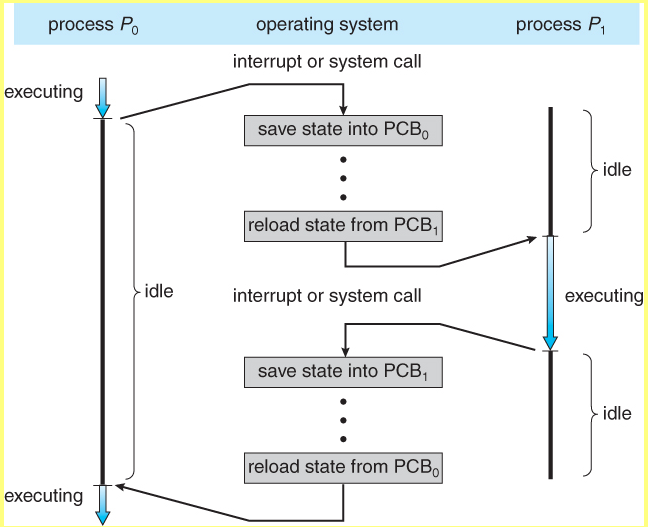


Figura 4

**Processamento paralelo**

A ideia básica do processamento paralelo é que diversos processos trabalhem simultaneamente na resolução de um mesmo problema.

Obviamente que para que esse palalelismo possa se efetivar, é necessário que se disponha de um processador para cada processo. Isto pode se dar de duas formas:

* Um computador multi-core, onde se tem diversos processadores em uma mesma CPU.
* Vários computadores trabalhando de forma colaborativa, chamado processamento distribuído.

|  |  |
| --- | --- |
| Figura 5 | Figura 6 |

Este trabalho conjunto entre os processos, para que o mesmo problema seja resolvido, é classificado em 2 tipos:

1. Divisão em tarefas de naturezas diferentes.
2. Divisão da massa de dados em tarefas da mesma natureza.

O caso 1 ocorre quando o algoritmo precisa realizar diversas tarefas diferentes como, por exemplo, compor um relatório financeiro anual consolidado com diversos indicadores. Sendo cada indicador obtido por cálculos diferentes, pode-se estudar a possibilidade de lançar um subprocesso para cada um desses cálculos.

O caso 2 ocorre quando existe uma massa de dados muito grande onde se deseja, por exemplo, classificar o seu conteúdo em: dentro da faixa média, acima da média e abaixo da média. Neste caso divide-se a massa de dados e atribui-se cada parte a um processo. Depois, neste exemplo simples, é só somar os resultados de cada um.

No caso 1, os processos têm codificação específica para cada tarefa e, no caso 2, os processos têm codificação idêntica, uma vez que estão fazendo a mesma coisa.

É importante observar que só é possível palalelizar atividades que são independentes entre si, ou seja, uma não depende dos resultados de outra para poder iniciar.

**Processamento multi-core:**

Neste ambiente, os processos utilizam a mesma memória física controlada pela mesma CPU que, por sua vez, possui vários processadores, o que possibilita executar mais de um processo ao mesmo tempo.

Há, também, a possibilidade de se simular um ambiente distribuído, fazendo-se uma divisão lógica da memória e atribuindo-se cada divisão e um processador a um "computador", mas há recursos mais eficientes e mais simples de serem controlados.

Esses recursos, chamados genericamente de *threads*, são processos disparados por outros processos. Nas linguagens de programação que dão suporte à criação de *threads*, um processo pode instanciar outro processo. Este último é um subprocesso do seu processo criador, mas para a CPU, trata-se de um processo como outro qualquer.

Um processo pode instanciar vários subprocessos e um subprocesso, por sua vez, pode instanciar outros subprocessos.



Figura 7

É importante observar que o tempo de vida da *thread* está condicionado à vida do seu processo criador. Caso este se encerre, todas as suas *threads* serão destruídas independentemente de estarem fazendo algo que possa ser encerrado no momento ou não. Portanto, um correto planejamento se faz necessário por parte do programador.

**Memória compartilhada**

Em muitos casos, há situações em que os processos precisam acessar variáveis comuns. Os processos, em sua forma básica, têm espaços de memória exclusivos e invioláveis. Outros processos não podem invadir esse espaço. No entanto, para que um processo possa compartilhar uma, ou mais variáveis, com outros processos, há que se ter algum tipo de recurso programático para possibilitar esta situação.

Há recursos que já criam um ambiente de memória compartilhada de forma praticamente automática, mas vamos começar com recursos de nível programático mais baixo para entender como isto funciona.

**A instrução fork()**

No ambiente Unix e derivados (Linux, FreeBSD, etc.) existe a instrução fork(). Esta instrução instancia um novo processo réplica do processo pai até o ponto da instrução fork. A partir da instrução fork, a codificação do processo filho pode se diferenciar.

Na fork(), diferentemente das threads, o processo filho não se encerra, necessariamente, com o encerramento do processo pai. Se o pai se encerrar antes, o filho se torna um processo órfão e o sistema operacional o promove a processo titular.

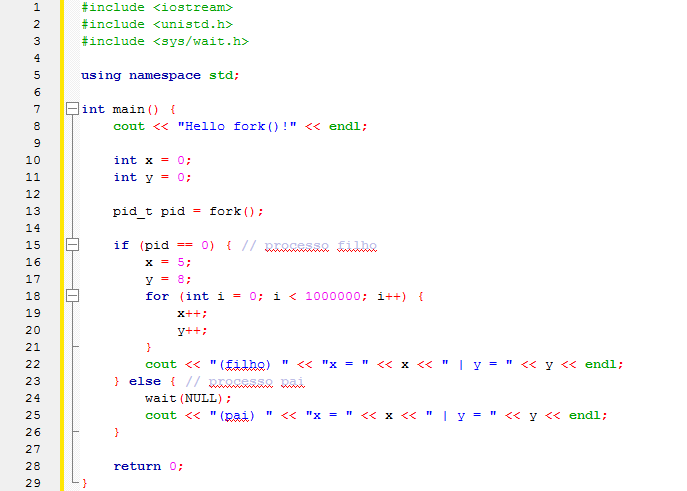


Figura 8

1. A linha 13 instancia um novo processo, réplica do pai deste ponto para trás. As variáveis são simplesmente replicadas. Toda e qualquer alteração
2. A linha 15 verifica qual o PID retornado:
   1. se for 0, significa que se está na instância do processo filho.
   2. maior que 0, significa que se está na instância do processo pai.
3. Nas linha 16 e 17, o processo filho alterou os valores de **x** e de **y**.
4. A linha 18, dentro do processo filho, está incrementando as variáveis e, ao mesmo tempo, fazendo um papel de temporizador para dar tempo de ser concluído depois que o processo pai terminou.
5. A linha 24 faz o processo pai aguardar o encerramento do filho. Caso esta linha seja comentada, o processo pai vai encerrar antes.

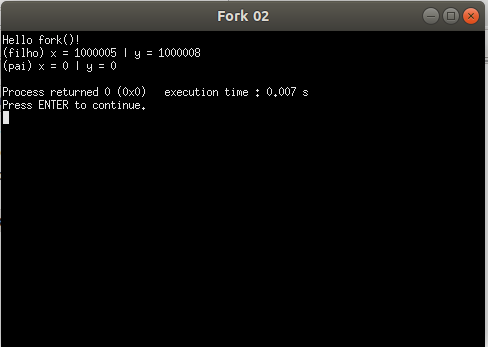


Figura 9 - Instrução wait(NULL) na linha 24 - processo pai aguarda o final do processo filho.

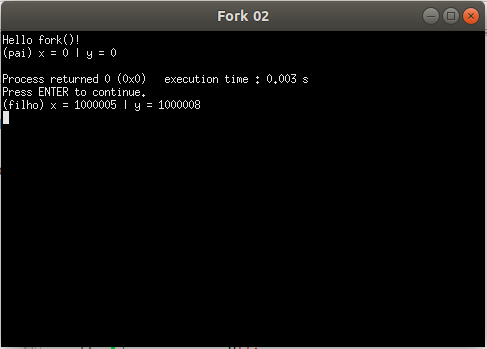


Figura 10 - Sem a instrução wait(NULL) na linha 24 - processo pai termina antes do processo filho.

Como se pode observar, as alterações que o processo filho fez nas variáveis x e y não se refletiram em **x** e **y** do processo pai. Isto porque no processo filho todo o ambiente é clonado, ou seja, não se trata das mesmas variáveis.

Para que **x** e **y** possam ser compartilhadas, é necessária a utilização da instrução shmget() (**sh**ared **m**emory **get**).