MAC5753 - Sistemas Operacionais

Daniel Macêdo Batista

IME - USP, 15 de Setembro de 2020

Roteiro

Um pouco mais sobre chamadas de sistema

Mais um pouco sobre processos

Threads

Um pouco mais sobre chamadas de sistema

Para manipular a execução de outros processos


```
while (1) {
  type_prompt();
  read_command(command, parameters);

  if (fork() != 0) {
     /* Codigo do pai */
     ...
  } else {
     /* Codigo do filho */
     execve(command,parameters,0);
  }
}
```

Para manipular a execução de outros processos


```
while (1) {
  type_prompt();
  read_command(command, parameters);

  if (fork() != 0) {
     /* Codigo do pai */
     waitpid(-1,&status,0);
  } else {
     /* Codigo do filho */
     execve(command,parameters,0);
  }
}
```

Variáveis de ambiente

- Costumam ser úteis para facilitar a execução de alguns processos
- Podem ser vistas como mais uma forma de permitir comunicação entre processos
- O `execve` recebe a lista de variáveis de ambiente como terceiro parâmetro
- O comando `export` no linux permite atribuir valores a variáveis de ambientes e o comando `echo` permite imprimir na tela o valor de uma variável de ambiente

Mais um pouco sobre processos

Resumindo o que já sabemos

- Processos são programas em execução
- Processos competem pelos recursos do computador
- Os processadores chaveiam rapidamente de processo em processo dando a impressão de que tudo está executando ao mesmo tempo (se tiver mais de um processador, de fato isso acontece)

Algumas observações sobre o modelo de processo

- Uma linha do tempo que mostre o processo que está usando a CPU dificilmente vai ser igual mesmo se você executar os mesmos programas duas vezes seguidas
- As decisões sobre qual o próximo processo que vai executar são tomadas por uma parte do SO chamada de **escalonador de processos**
- Exemplo: antes de ler arquivos de backup de uma fita magnética, importante esperar a unidade de fita alcançar a velocidade de rotação. Rodar um laço 1000 vezes para esperar isso acontecer nem sempre vai dar certo (A CPU não vai ficar só rodando esses 1000 laços)

Algumas observações sobre a criação de processos

- Nem sempre os processos precisam de um terminal ou uma janela para interagirem com os usuários. Servidores de serviços da Internet são um exemplo. Eles costumam rodar em segundo plano e geralmente são iniciados na inicialização do SO. Os processos desses servidores são chamados de daemons
- Em clones do Unix, a criação de um processo é feita apenas com a chamada de sistema `fork`. O que o `execve` faz, embora na prática seja a criação de um processo para rodar aquele programa, é mudar a "imagem" do processo que o executou na memória e executar essa "imagem". Por isso que fazer o `execve` sem um `fork` antes finalizaria um shell (Testem isso).
- Da manpage do execve: *"execve() does not return on success, and the text, data, bss, and stack of the calling process are overwritten by that of the program loaded."*

Algumas observações sobre o término de processos

- Processos podem terminar de 4 formas:
  - Saída normal (voluntária)
  - Saída por causa de erro (voluntária) – relacionado com o que o programa faz. Por exemplo, arquivo não existe
  - Erro fatal (involuntária) – acesso a endereço de memória inexistente, divisão por zero
  - Morto por outro processo (involuntária)

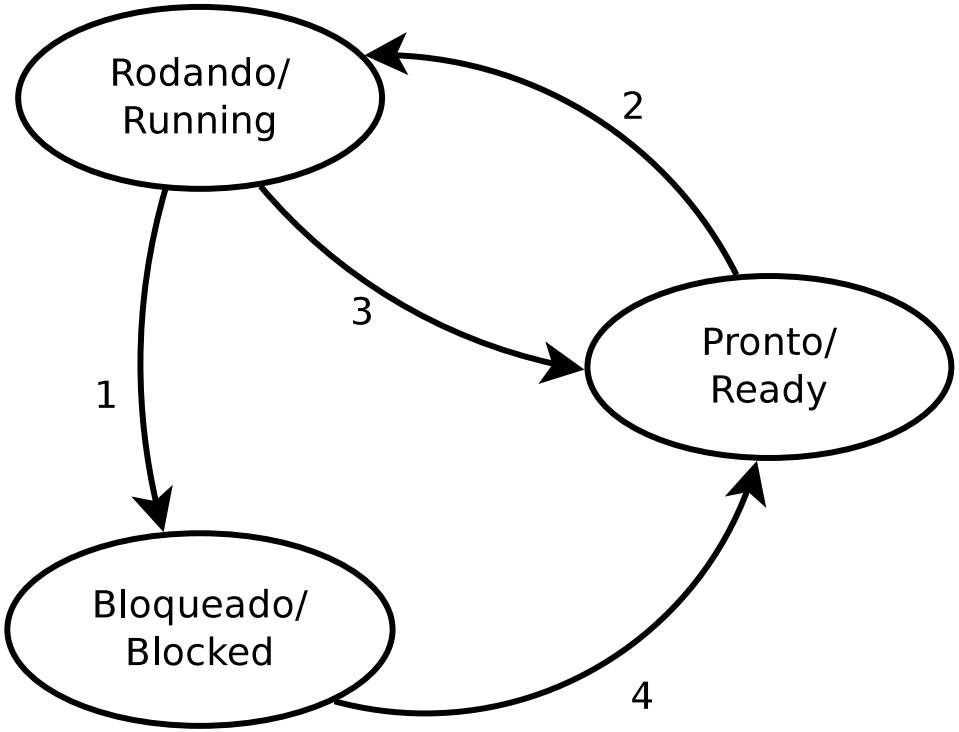
Estados de um processo

- É comum processos bloquearem durante as suas execuções (por exemplo no caso de pipe conectando dois processos, o segundo processo pode estar pronto para processar a entrada mas a entrada pode não estar pronta)

```
tail -f /var/log/syslog | grep daniel
```

- Um processo também pode ser bloqueado porque o SO escalonou outro processo para usar a CPU
- O comando `ps` informa o estado de cada processo em execução

Estados de um processo



1: Processo bloqueia esperando entrada
2: Escalonador escolhe este processo
3: Escalonador escolhe outro processo
4: Entrada disponível

O escalonador de processos no fim das contas é a base sobre a qual os processos são executados

Threads

Motivação

- Muitas vezes o modelo de processos é muito rígido para resolver alguns problemas
- Por exemplo, um software que tem partes bem definidas que possam ser realizadas em paralelo mas que precisam manter comunicação entre si por meio de variáveis na memória seria difícil de ser escrito como múltiplos processos (cada processo tem seu próprio espaço **privado** de endereço!)
- Threads ajudam a resolver esse problema! Com elas, variáveis podem ser vistas e podem ser usadas para comunicação entre as threads

Threads

- As threads podem ser vistas como miniprocessos mas, diferente de processos, tem a vantagem de compartilhar o espaço de endereço
- O fato de não precisar de uma nova entrada na tabela de processos, nem criação de área de memória específica, faz a gerência das threads ser mais leve do que a gerência de processos (criar uma thread em certos SOs pode ser até 100 vezes mais rápido do que criar um processo)
- As threads podem rodar em núcleos separados, tirando proveito do paralelismo não apenas entre processos mas em vários processos (mas transformar códigos monothread em multithread nem sempre é trivial)

Threads

- Um jogo digital é um bom exemplo que mostra a vantagem de multithread. Uma thread pode cuidar do áudio do jogo, outra da renderização das imagens na tela e outra pode cuidar da comunicação via rede. Todas compartilhando a mesma área de memória. Assim, se vem uma ação de um jogador via rede, isso já pode ser exibido na tela de forma que o usuário vai acreditar que foi instantâneo.

Em C, no Linux

- `pthread.h`
- Funções `pthread_create` e `pthread_join`
- Outras funções: `pthread_exit`, `pthread_tryjoin_np`, `pthread_timedjoin_np`, ...
- Importante bloquear as threads pela lógica do código. Também há funções e técnicas para isso

Outros argumentos sobre quando usar

- A eficiência em fazer a comunicação por meio de variáveis na memória é maior do que pipes e variáveis de ambiente
- Útil quando um software precisa atender diversos clientes/usuários mantendo o status de cada um deles (o código para atender cada um é igual) – Servidores de rede por exemplo com 1 **dispatcher** e vários **workers** – Desempenho melhora significativamente

Processos vs Threads

- Processos podem ser vistos como unidades que agrupam recursos juntos
- Threads podem ser vistas como unidades que são escalonadas para execução nas CPUs
- As threads permitem que execuções múltiplas aconteçam dentro de um único processo.
- Múltiplas threads de um processo rodando em paralelo é análogo a múltiplos processos monothread de um computador rodando em paralelo (Threads compartilham espaço de endereço e outros recursos. Processos compartilham a memória física, discos, etc...)
- Mesmo que a CPU não tenha múltiplas unidades de processamento pode tirar proveito das threads (alguma hora vai ter uma atividade de E/S)
- Hierarquia entre threads pode existir mas tem que ser feita manualmente pelo programador

Usando threads

- Na prática todo programa começa monothread. A thread principal tem poder de criar outras threads.
- Cada thread independente do SO normalmente vai ser criada com uma função como parâmetro. Essa função vai ser executada para aquela thread
- Algumas funções básicas que podem ser realizadas com threads: criar, terminar, esperar uma thread específica terminar, liberar a CPU para outra thread (útil para escalonamento. Não faria sentido para processos)

Consequências da área de memória compartilhada

- As variáveis globais são compartilhadas (bom e ruim ao mesmo tempo)
- Não há controle algum no acesso a essas variáveis (desnecessário pois as threads são de um mesmo processo e por sua vez serão de um mesmo usuário)
- Mas nem tudo é compartilhado. É preciso manter **por thread**: Program counter, registradores, pilha, estado (aqui no estado é a mesma ideia dos estados de processos)
- Se são implementadas a nível de usuário, o escalonador pode ser personalizado para cada processo