Processos

Wedson Almeida Filho
Samuel Xavier de Souza

Processo

- Materialização de um programa em execução
  - Normalmente carregado de um ou mais arquivos
- Exemplos:
  - Navegador, editor de texto, compilador, servidor web, etc.
- É possível ter mais de uma instância do mesmo programa executando
- Cada processo é isolado dos demais

Exemplo: listando processos no Linux

```
$ ps
    PID TTY          TIME CMD
  32581 pts/2    00:00:00 bash
  45266 pts/2    00:00:00 ps
$ pstree
systemd─┬─ModemManager───3*[{ModemManager}]
        ├─NetworkManager───3*[{NetworkManager}]
        ├─accounts-daemon───3*[{accounts-daemon}]
        ├─agetty
        ├─avahi-daemon───avahi-daemon
        ├─colord───3*[{colord}]
        ├─cron
        ├─cups-browsed───3*[{cups-browsed}]
        ├─cupsd
        ├─dbus-daemon
        ├─fwupd───5*[{fwupd}]
        ├─gdm3─┬─gdm-session-wor─┬─gdm-wayland-ses─┬─gnome-session-b───3*[{gnome-session-b}]
        │      │                 │                 └─3*[{gdm-wayland-ses}]
        │      │                 └─3*[{gdm-session-wor}]
        │      └─3*[{gdm3}]
        ├─gnome-remote-de───3*[{gnome-remote-de}]
        ├─2*[kerneloops]
```

Exemplo: conteúdo de um arquivo executável no Linux

```
$ objdump -p /usr/bin/ls

/usr/bin/ls:     file format elf64-littleaarch64

Program Header:
    PHDR off    0x0000000000000040 vaddr 0x0000000000000040 paddr 0x0000000000000040 align 2**3
         filesz 0x00000000000001f8 memsz 0x00000000000001f8 flags r--
  INTERP off    0x0000000000000238 vaddr 0x0000000000000238 paddr 0x0000000000000238 align 2**0
         filesz 0x000000000000001b memsz 0x000000000000001b flags r--
    LOAD off    0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**16
         filesz 0x000000000001fe6c memsz 0x000000000001fe6c flags r-x
    LOAD off    0x000000000002ef30 vaddr 0x000000000002ef30 paddr 0x000000000002ef30 align 2**16
         filesz 0x0000000000001368 memsz 0x0000000000002638 flags rw-
 DYNAMIC off    0x000000000002f9e8 vaddr 0x000000000002f9e8 paddr 0x000000000002f9e8 align 2**3
         filesz 0x0000000000000210 memsz 0x0000000000000210 flags rw-
    NOTE off    0x0000000000000254 vaddr 0x0000000000000254 paddr 0x0000000000000254 align 2**2
         filesz 0x0000000000000044 memsz 0x0000000000000044 flags r--
EH_FRAME off    0x000000000001cecc vaddr 0x000000000001cecc paddr 0x000000000001cecc align 2**2
         filesz 0x00000000000005ec memsz 0x00000000000005ec flags r--
   STACK off    0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**4
         filesz 0x0000000000000000 memsz 0x0000000000000000 flags rw-
   RELRO off    0x000000000002ef30 vaddr 0x000000000002ef30 paddr 0x000000000002ef30 align 2**0
         filesz 0x00000000000010d0 memsz 0x00000000000010d0 flags r--
```

Exemplos de propriedades de processos

- Identidade do processo
  - Usado para diferenciar instâncias
- Identidade do usuário
  - Usado para decidir os recursos aos quais o processo tem acesso
- Espaço de endereçamento virtual
  - Cada processo tem sua própria memória virtual, independente dos demais
- Estado da CPU
  - Registradores usados na execução de código
- Estado do escalonador
  - Usado para determinar quando um processo está pronto para rodar, sua prioridade, etc.
- Tabela de descritores de arquivos
  - Cada processo tem o seu conjunto de arquivos abertos

Criação de processos

- A partir de um executável
  - Como no Windows com CreateProcess
- Fazendo a cópia de um processo existente
  - Como no Unix com a chamada de sistema fork
  - Normalmente seguida da chamada de sistema execve ou variantes
    - Carrega um executável no processo atual
  - Tinha um custo baixo no passado

Exemplo no Linux

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
   pid_t pid = fork();
   if (pid < 0) {
       return -1;
   }

   if (pid == 0) {
       printf("Novo processo: %d\n", getpid());
   } else {
       printf("Processo original, filho=%d\n", pid);
   }

   return 0;
}
```

Saindo de processos

- Normalmente uma chamada de sistema
  - exit no Unix
  - ExitProcess no Windows
- Acontece automaticamente nas linguagens de alto nível
  - Quando a função principal retorna, como `main` do C
- Normalmente podem comunicar o status da execução

Exemplo em C

```c
#include <stdlib.h>

int main(int argc, char **argv) {
    if (argc > 1) {
        exit(atoi(argv[1]));
    } else {
        exit(0);
    }
}
```

Executando no Linux

```
$ ./02-exit
$ echo $?
0
$ ./02-exit 1
$ echo $?
1
$ ./02-exit 2
$ echo $?
2
$ ./02-exit -2
$ echo $?
254
```

Esperando um filho terminar

- Normalmente uma chamada de sistema
  - WaitForSingleObject no Windows
  - wait no Unix

- É possível receber o status
  - GetProcessExitCode no Windows
  - `WEXITSTATUS` no status do wait no Unix

Exemplo: esperando o filho terminar no Linux

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char **argv)
{
   pid_t completed, pid = fork();
   if (pid < 0) {
       return -1;
   }

   if (pid == 0) {
       printf("Novo processo: %d\n", getpid());
   } else {
       printf("Processo original, filho=%d\n", pid);
       completed = wait(NULL);
       printf("filho acabou, pid=%d\n", completed);
   }

   return 0;
}
```

Prática: ler o código de saída do filho

- Modificar código anterior
- Fazer o filho retornar um valor diferente de 0, 10 por exemplo
- Fazer o pai esperar pelo filho e ler o código de saída
  - Escrever o código de saída na saída padrão

Solução: lendo código de saída do filho no Linux

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char **argv)
{
   pid_t pid = fork();
   if (pid < 0) {
       return -1;
   }

   if (pid == 0) {
       printf("Novo processo: %d\n", getpid());
       return 10;
   }

   int status;
   printf("Processo original, filho=%d\n", pid);
   if (wait(&status) != -1) {
       printf("filho acabou, status=%d\n", WEXITSTATUS(status));
   }
   return 0;
}
```

Segurança: isolamento do espaço de endereçamento

- Cada processo tem o seu próprio espaço de endereçamento
- Permite que o mesmo endereço numérico tenha valores diferentes
- Impede que um processo altere o conteúdo da memória de outro

Exemplo: isolamento de memória

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char **argv)
{
   pid_t pid = fork();
   if (pid < 0) {
       return -1;
   }

   int v = 10;
   if (pid == 0) {
       printf("Novo processo: %d\n", getpid());
       v = 20;
   } else {
       printf("Processo original, filho=%d\n", pid);
       wait(NULL);
       printf("Valor de v: %d\n", v);
   }

   return 0;
}
```

Segurança: ataques no espaço de endereçamento

- Que tipo de ataques são impedidos por esse isolamento?

Segurança: ataques no espaço de endereçamento

- Que tipo de ataques são impedidos por esse isolamento?

- Modificação direta de instruções
- Modificação de ponteiros em v-tables
- Modificação de dados em variáveis
- Modificação do endereço de retorno na pilha de chamadas
- Etc.

Pilha de chamadas

- Para cada chamada de função:
  - Um "quadro" novo é adicionado à pilha de chamadas
- Quadro contém
  - Ponteiro para posição de retorno
  - Conteúdo de registradores não-voláteis
  - Variáveis locais
- Para cada retorno de função:
  - Um quadro é retirado do topo da pilha

Prática: alterar o fluxo de um programa

- Escrever um programa que chama uma função
- A função altera o ponteiro de retorno na pilha
  - Fazer a função "retornar" para outra função
- Notar que o fluxo de controle é alterado

Solução: modificando o ponteiro de retorno

```c
#include <stdio.h>
#include <stdlib.h>

void alterado(void)
{
   printf("Fluxo alterado!\n");
   exit(0);
}

void exemplo(void)
{
   void *local[1];
   printf("dentro de exemplo\n");
   local[3] = (void *)alterado;
}

int main(int argc, char **argv)
{
   exemplo();
   printf("Fluxo normal\n");
   return 1;
}
```

Descobrindo distância usando o gdb

```
$ gdb ./06-fluxo
[...]
(gdb) b exemplo
Breakpoint 1 at 0x11b6: file 06-fluxo.c, line 11.
(gdb) r
[...]
Breakpoint 1, exemplo () at 06-fluxo.c:11
11   {
(gdb) p/x &local
$1 = 0x7fffffffe350
(gdb) bt
#0  exemplo () at 06-fluxo.c:11
#1  0x000055555555520e in main (argc=1, argv=0x7fffffffe498)
    at 06-fluxo.c:19
(gdb) x/10gx 0x7fffffffe350
0x7fffffffe350:   0x0000000000000000 0x0000000000000000
0x7fffffffe360:   0x00007fffffffe380 0x000055555555520e
0x7fffffffe370:   0x00007fffffffe498 0x0000000100000000
0x7fffffffe380:   0x0000000000000001 0x00007ffff7c29d90
0x7fffffffe390:   0x0000000000000000 0x000055555555551f6
```

Prática: executando outro programa em Linux

- Modificar exemplo do fork/wait
- Adicionar chamada de sistema execve
  - Variantes como execlp também são aceitáveis
- Executar o comando "ls -l"

Solução: executando "ls -l"

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char **argv)
{
   pid_t pid = fork();
   if (pid < 0) {
       return -1;
   }

   if (pid == 0) {
       execlp("/usr/bin/ls", "ls", "-l", NULL);
   } else {
       printf("Processo original, filho=%d\n", pid);
       wait(NULL);
   }

   return 0;
}
```

Tabela de descritores de arquivos

- Cada processo tem uma tabela de "arquivos abertos"
- É uma camada de indireção para usuários se referirem a arquivos
  - Em vez de usar o ponteiro para a memória em modo kernel, o que seria inseguro
- Usuários podem se referir a arquivos abertos usando um número
- Alguns número têm significado no Unix:
  - 0: Entrada padrão
  - 1: Saída padrão
  - 2: Saída de erro padrão

Exemplo: redirecionando a saída padrão no Linux

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
   int fd = open("saida.txt", O_WRONLY | O_CREAT, 0600);
   dup2(fd, 1);
   close(fd);
   printf("Hello world!\n");
   return 0;
}
```

Prática: redirecionar os arquivos de saída do filho

- Modificar o exemplo do fork/exec
- Redirecionar saída padrão (1) e saída de erro padrão (2) do filho
- Executar "ls -l"

Solução

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
   int fd = open("saida.txt", O_WRONLY | O_CREAT, 0600);
   pid_t pid = fork();
   if (pid < 0) {
       return -1;
   }

   if (pid == 0) {
       dup2(fd, 1);
       dup2(fd, 2);
       close(fd);
       execlp("/usr/bin/ls", "ls", "-l", NULL);
   } else {
       printf("Processo original, filho=%d\n", pid);
       wait(NULL);
   }
   return 0;
}
```

Pipe: comunicação entre processos

- Canal unidirecional de dados
  - Um descritor de arquivo permite leitura
  - Outro descritor permite escrita
- Tem um *buffer* para guardar dados escritos mas não lidos ainda
  - De tamanho limitado
- Bloqueia
  - Leitores quando o buffer está vazio
  - Escritores quando o buffer está cheio
- Leitura completa com 0 quando escritor estiver fechado
  - E não houver mais nenhum dado pendente
- Em Unix, criada com a chamada de sistema pipe ou variantes

Prática: usar uma pipe para comunicação entre pai e filho

- Modificar exemplo do fork
- Pai cria uma pipe antes do fork
- Filho escreve mensagem na pipe
- Pai lê mensagem da pipe
  - Escreve para a saída padrão

Solução

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
   int fd[2];
   pipe(fd);

   pid_t pid = fork();
   if (pid < 0) {
       return -1;
   }

   if (pid == 0) {
       close(fd[0]);
       write(fd[1], "do filho", 9);
   } else {
       char buf[100] = {0};
       close(fd[1]);
       wait(NULL);
       read(fd[0], buf, sizeof buf - 1);
       printf("Lido: %s\n", buf);
   }

   return 0;
}
```

Estados de escalonamento

- Processos podem estar em estados diferentes
  - Pronto para executar
  - Executando
  - Esperando
  - Finalizado
- Exemplos
  - Após fork: ambos os processos estão prontos para executar, escalonador vai escolher um
  - Quando o pai executar, faz a chamada `wait`
    - Vai para o estado *Esperando*, ou seja, não está mais pronto para execução
  - Quando filho executar, vai até o fim
    - Ao terminar, acorda o pai, que passa ao estado *Pronto para executar*
    - Filho vai para o estado *Finalizado*
  - Quando o pai executar, lê o estado do filho, o que faz o que ele seja apagado
  - Pai termina e vai para o estado *Finalizado* até o seu pai ler o seu resultado

Segurança: como evitar DoS (denial of service)?

- Como um sistema operacional impede um processo de rodar para sempre?

Segurança: como evitar DoS (denial of service)?

- Como um sistema operacional impede um processo de rodar para sempre?

- Com uma interrupção
- Quando começa a rodar um processo, o S.O. liga um temporizador (*timer*)
- Quando ele expira, a CPU é interrompida
  - Traz de volta o controle para o S.O.
  - O escalonador escolhe outro processo para executar

Comunicação entre processos

- Pipe
  - Anônima ou no sistema de arquivos
- Memória compartilhada
  - shmem, mmap
- Sockets
  - Modelo cliente/servidor
- Primitivas de sincronização
  - Eventos, mutexes, etc.
- Remote procedure calls (RPCs)
  - Mais alto nível, usando as primitivas anteriores