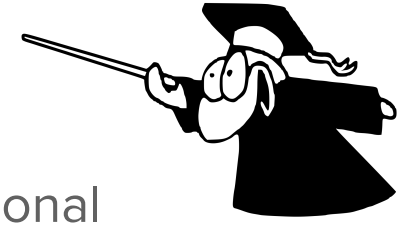
Gerência de Processos

EMB5632 - Sistemas Operacionais

Prof. Dr. Ricardo José Pfitscher
ricardo.pfitscher@ufsc.br

Objetivos de aprendizagem



- Entender o que é um processo no sistema operacional
- Entender os conceitos principais relacionados à processos

UFSC

Cronograma

- Conceito
- Multiprogramação
- Criação de processos /Término de Processos
- Hierarquias de Processos
- Estados de Processos
- Implementação de Processos

UFSC

Conceito [1/2]

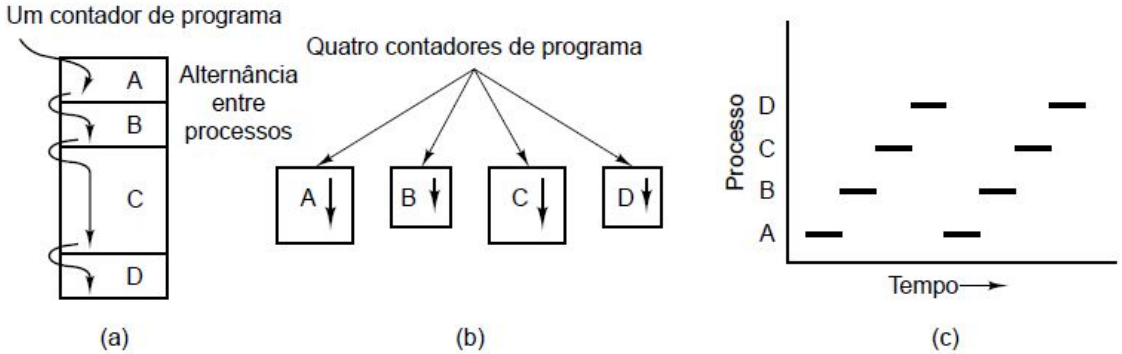
- Um processo é um programa em execução
  - Acompanhado do contador de programa, dos registradores e variáveis
  - Conceitualmente cada processo tem sua própria CPU
    - Eles não sabem que estão compartilhando com os outros

Conceito [2/2]

- Na prática: Utiliza-se o conceito de **Multiprogramação**:
  - Vários processos são carregados na memória ao mesmo tempo
  - A CPU alterna entre os processos em execução
    - Máquinas monoprocessadas: Somente um processo executa por cada vez
      - Pseudoparalelismo (parece que executam em paralelo, mas o chaveamento é muito rápido, o que dá esta impressão)
    - Máquinas multiprocessadas:
      - Paralelismo real, ou seja, cada processador executa um processo.

Multiprogramação

- Exemplo com quatro processos



a) Quatro programas na memória
b) Cada um com seu próprio fluxo de controle
   - Somente um contador de programa físico, cada um possuí um contador de programa lógico
c) A cada instante somente um processo está executando

- Processos não devem fazer hipóteses temporais ou sobre a ordem de execução (CPU, Escalonamento)

Uma analogia para processo [1/3]

- Um aluno de SO está preparando um bolo para seu filho.
  - Receita (Programa, um algoritmo)
  - Ingredientes (Dados de entrada)
  - Ler receita, buscar ingredientes, assar bolo (Processo) □ Executado pelo confeiteiro (CPU)

Uma analogia para processo [2/3]

- No meio da receita seu filho chega machucado de um tombo (o escalonador escolheu outro)
  - O homem registra onde ele estava na receita (salva o estado atual do processo)
  - Busca o livro de primeiro socorros, começa a seguir as instruções contidas nele. (alternância de processos)
  - Assim que tratar o ferimento, o homem volta ao bolo a partir do ponto em que parou

Uma analogia para processo [3/3]

- Processo:
  - Programa (algoritmo, receita)
  - Dados de Entrada (variáveis, ingredientes)
  - Saída (retorno, resultado, bolo)
  - Estado

Contextos

- Os processos possuem um estado interno bem definido
- Representam a situação atual do processo:
  - Instrução que está executando
  - Valores de suas variáveis
  - Arquivos que está utilizando, etc.

Contextos

Estado do processo é determinado pelas seguintes informações:

- Registradores do processador
  - Contador de programa (PC – *Program Counter*) – posição corrente de execução no código da tarefa
  - Ponteiro de pilha (SP – *Stack Pointer*) – aponta para o topo da pilha de execução
  - *Flags* indicando aspectos do processador naquele momento (nível usuário ou nível núcleo, *status* da última operação realizada, etc.)
  - Demais registradores (acumulador, de uso geral, de mapeamento da memória, etc.)
- Áreas de memória usadas pelo processo
- Recursos usados pelo processo
  - Arquivos abertos
  - Conexões de rede, etc.

Contextos

- Contexto do processo são as informações que permitem definir completamente o estado de um processo
- PCB (*Process Control Block*) – estrutura de dados onde são armazenadas informações de contexto e outros dados necessários à gerência do processo
- No Linux:
  - https://github.com/torvalds/linux/blob/master/include/linux/sched.h
  - Pesquise por `task_struct (linha 818)`

Trocas de contexto

- Trata-se do processo de salvar os valores do contexto atual em um PCB e restaurar o contexto de outro processo, salvo previamente em um PCB
- Operação delicada – manipulação de registradores e *flags* específicos de cada processador

Dispatcher

- Aspecto mecânico da troca de contexto
- Armazenamento do contexto do processo atual e atualização do PCB
- Recuperação do contexto do processo através do PCB
- Conjunto de rotinas chamada de **despachante** ou ***dispatcher***

Escalonador

- Aspecto político ou estratégico da troca de contexto
- Escolha do próximo processo a receber o processador
- Influência de diversos fatores:
  - Prioridades
  - Tempo de vida
  - Tempo de processamento restante, etc.
- Decisões ficam a cargo do **escalonador** ou ***scheduler***

Troca de contexto

- A troca de contexto completa é relativamente rápida:
  - Interrupção de um processo
  - Armazenamento do contexto
  - Escalonamento
  - Reativação do processo escolhido
- Quanto mais rápida for a troca de contexto, mais eficiente é a gerência de processos
- Fatores:
  - Carga do sistema: quanto mais processos ativos, mais tempo será gasto pelo escalonador
  - Perfil das aplicações: muita operação de E/S faz com que o processo saia do processador antes do final do *quantum*

Criação de Processos [1/5]

- Principais eventos que levam à criação de processos:
  - Início do sistema
  - Execução de uma chamada de sistema de criação de processos
    - fork (UNIX), CreateProcess (Windows), SYS$CREPRC (VAX/VMS)
  - Requisição do usuário para criar um novo processo (dois cliques em um ícone)
  - Início de um tarefa em lote (*batch*)
    - Computadores de grande porte
    - O sistema operacional criará um novo processo para executar uma nova tarefa quando necessário

Criação de Processos [2/5]

- Tipos de Processos:
  - Processos interativos: Interagem com usuários
    - Primeiro plano (*foreground*)
  - Processos de segundo plano (*background*): serviços do sistema
    - *Daemons:* Processos que ficam em *background* com a finalidade de lidar com alguma atividade
      - Ex.: Servidores *web:* fica inativo até que chegue uma nova requisição

Criação de Processos [3/5]

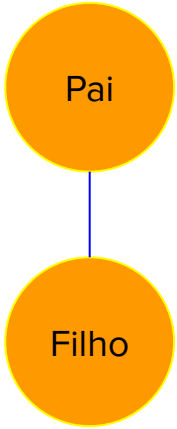
- Exemplo: Chamada *fork*
  - No UNIX, processos são criados através das chamadas de sistema *fork*
  - O processo filho é **idêntico** ao processo pai:
    - Código e dados são copiados
    - A diferença entre o pai e o filho, está no valor de retorno da função fork()
      - No processo pai, a função retorna o identificador (PID) do filho, ou seja, >0
      - No processo filho, a função retorna 0
    - A chamada *execve* pode ser usada para substituir o processo corrente Muda a imagem de memória e executa um novo programa

```
f = fork();
if (f == 0)  {                    /* processo filho */
   printf("processo filho\n");
   exit(4);                          /* retorna 4 */
} else {                           /* processo pai */
   printf("processo pai\n")
   w = waitpid(f, &rc, 0);      /* espera retorno   */
}                               /* do filho (rc==4) */
```

Criação de Processos [4/5]

- Árvore de Processos [1/2]

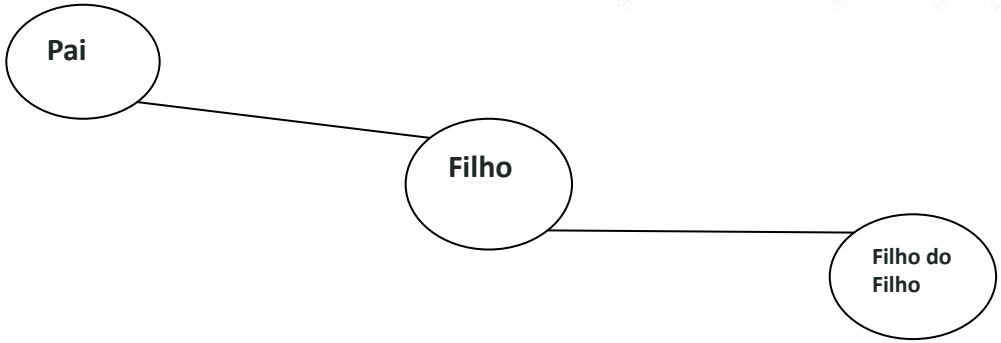
```
f = fork();
if (f == 0)  {                    /* processo filho */
   printf("processo filho\n");
   exit(4);                          /* retorna 4 */
} else {                           /* processo pai */
   printf("processo pai\n")
   w = waitpid(f, &rc, 0);      /* espera retorno   */
}                                /* do filho (rc==4) */
```



Criação de Processos [5/5]

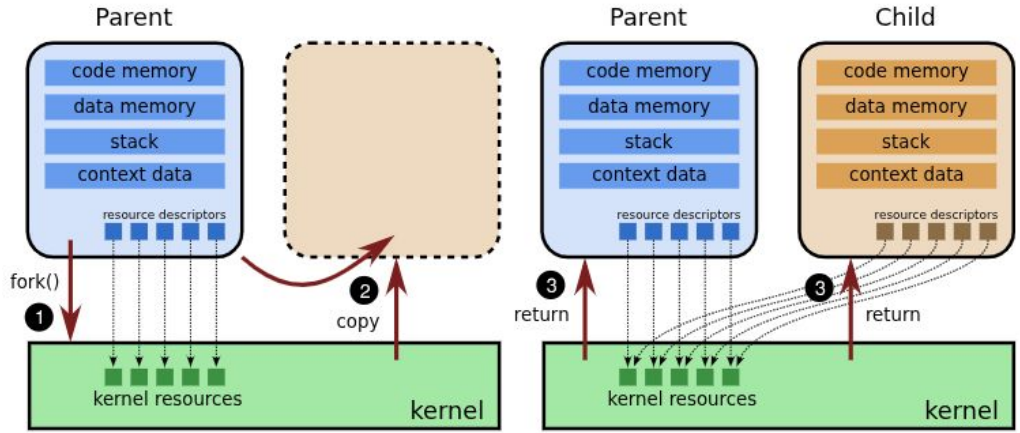
- Árvore de Processos [2/2]

```
f = fork();
if (f == 0) {                              /* processo filho */
        printf("processo filho\n");
        g = fork();
        if (g == 0)
                printf("processo filho do filho\n");
        exit(4);                           /* retorna 4 */
} if(f > 0) {                              /* processo pai */
        printf("processo pai\n");
        w = waitpid(f, &rc, 0);            /* espera retorno */
}                                          /* do filho (rc==4) */
```



Criação de Processos [3/5]

- Exemplo: Chamada *fork*



Fonte: (Maziero, 2019)

1 - o processo invocou fork()
2 - o sistema operacional faz uma cópia do processo pai
3 - Tanto o filho quanto o pai recebem o retorno de fork(), filho recebe 0 e pai o PID do filho
4 - os processos seguem fluxos independentes

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

int main (int argc, char *argv[], char *envp[])
{
   int pid ;                          // identificador de processo

   pid = fork () ;                    // replicação do processo

   if ( pid < 0 )                     // fork funcionou?
   {
      perror ("Erro: ") ;             // não, encerra este processo
      exit (-1) ;
   }
   else                               // sim, fork funcionou
      if ( pid > 0 )                  // sou o processo pai?
         wait (0) ;                   // sim, vou esperar meu filho concluir
      else                            // não, sou o processo filho
      {
         // carrega outro código binário para executar
         execve ("/bin/date", argv, envp) ;
         perror ("Erro: ") ;          // execve não funcionou
      }
   printf ("Tchau !\n") ;
   exit(0) ;                          // encerra este processo
}
```

1 - o processo invocou fork()
2 - o sistema operacional faz uma cópia do processo pai
3 - Tanto o filho quanto o pai recebem o retorno de fork(), filho recebe 0 e pai o PID do filho
4 - os processos seguem fluxos independentes
    4.1 - o processo filho pode usar a chamada de sistema execve() para carregar um novo código binário para a memória

Fonte: (Maziero, 2019)

Exemplo de aplicação- link - moodle

fork-aula.c > main()

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


int main() {
  pid_t pid;
  int vals[200];
  int i, j, n=90, status;

  //preenchendo o vetor
  srand(time(NULL));
  for(i=0; i<200; i++) {
    vals[i]=rand()%300;
  }

  //criando processos filhos e procurando no vetor
  for(i=0; i<4; i++) {
    pid=fork();
    if(pid==0) {
      for(j=50*i; j<50*(i+1); j++) {
        if(vals[j]==n) {
          exit(j);
        }
      }

      exit(255);
    }
  }

  //aguardando os retornos
  int cont=0;
  for(i=0; i<4; i++) {
    wait(&status);

    if(WEXITSTATUS(status)!=255) {
      cont++;
      printf("O valor está na posição: %d\n", WEXITSTATUS(status));
    }
  }
  printf("Encontrado %d vezes!\n",cont);

  return 0;
}
```

```
rjp@turing:~/Documentos/UFSC/2025.2/EMB5632-SO/Exemplos$ ./fork-aula
O valor está na posição: 146
Encontrado 1 vezes!
rjp@turing:~/Documentos/UFSC/2025.2/EMB5632-SO/Exemplos$ ./fork-aula
O valor está na posição: 22
O valor está na posição: 59
Encontrado 2 vezes!
rjp@turing:~/Documentos/UFSC/2025.2/EMB5632-SO/Exemplos$
```

UFSC

Término de Processos

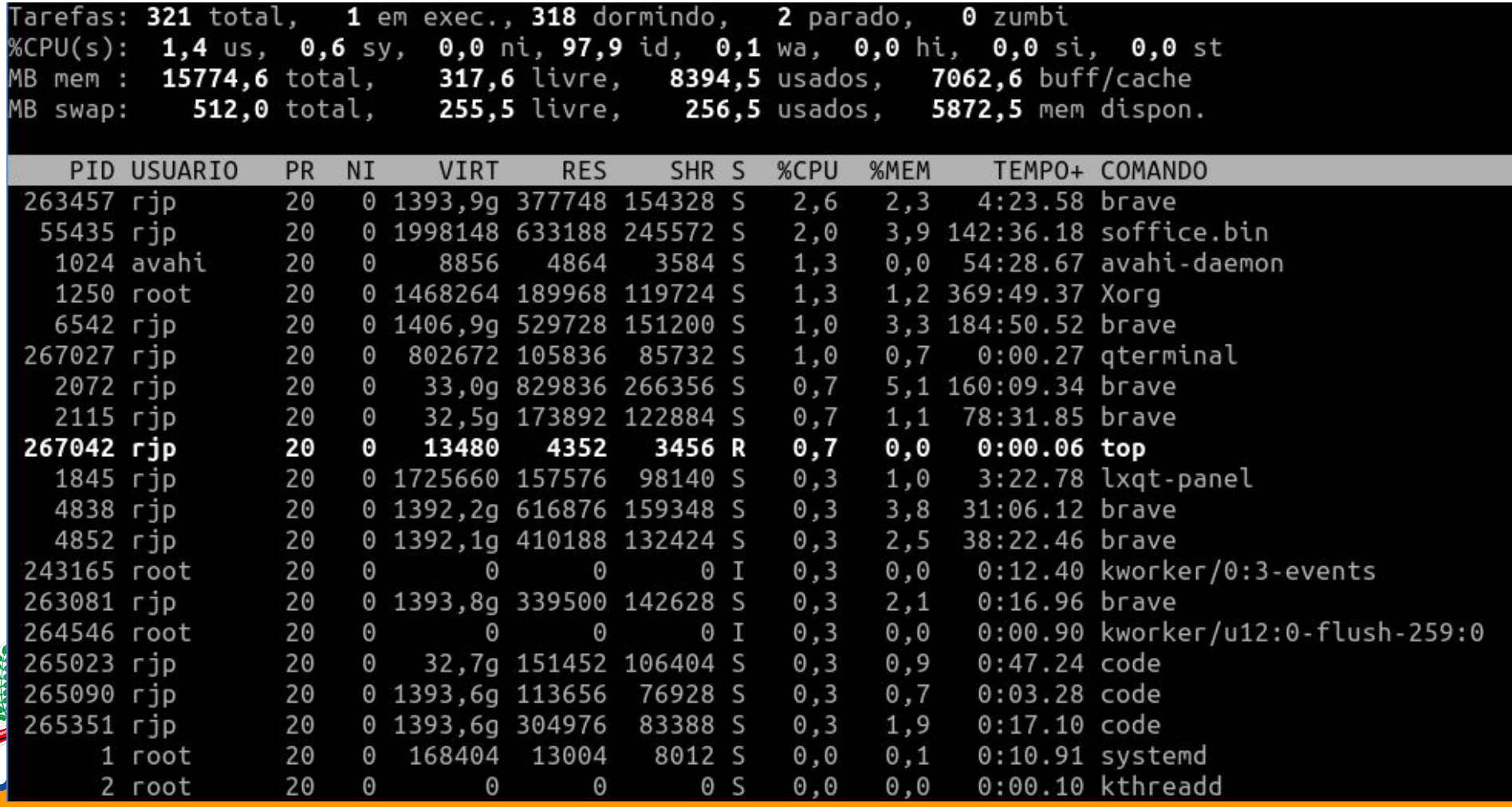
- Condições para o término de um processo:
  - Saída normal (voluntária)
    - O processo termina
    - Ex.: O usuário seleciona para fechar um programa
  - Saída por erro (voluntária)
    - Programa detecta um erro
    - Ex.: tentar compilar um arquivo inexistente
  - Erro fatal (involuntário)
    - Programa faz algo ilegal
    - Ex.: Referência a memória inexistente, divisão por zero
  - Cancelamento por outro processo (involuntário)
    - Um processo executa uma chamada de sistema para encerrar outro
    - Ex.: `kill` (UNIX), `TerminateProcess` (Windows)
- O término de um processo pode causar o término dos processos que ele criou
  - Isto não ocorre nem em UNIX nem em Windows

Hierarquias de Processos

- Processos “procriam” por várias gerações
  - Um processo pai cria processos filhos, que por sua vez também criam seus filhos, e assim por diante
- Leva a formação de **hierarquias** de processos
- No UNIX as hierarquias são chamadas de “Grupos de Processos”
  - Sinalizações de eventos se propagam através do grupo, e cada processo decide o que fazer com o sinal (ignorar, tratar, ou “ser morto”)
  - Todos os processos UNIX descendem de *init*
- Windows não possui hierarquias de processos
  - Todos os processos são criados iguais

Estados de um Processo [1/2]

- Um processo pode assumir diversos estados no sistema
  - **Em execução:** processo que está usando a CPU
  - **Pronto:** processo temporariamente parado enquanto outro processo executa
    - Fila de prontos (aptos a executar)
  - **Bloqueado:** esperando por um evento externo


Tarefas: 321 total, 1 em exec., 318 dormindo, 2 parado, 0 zumbi
%CPU(s): 1,4 us, 0,6 sy, 0,0 ni, 97,9 id, 0,1 wa, 0,0 hi, 0,0 si, 0,0 st
MB mem : 15774,6 total, 317,6 livre, 8394,5 usados, 7062,6 buff/cache
MB swap: 512,0 total, 255,5 livre, 256,5 usados, 5872,5 mem dispon.
PID USUARIO PR NI VIRT RES SHR S %CPU %MEM TEMPO+ COMANDO
263457 rjp 20 0 1393,9g 377748 154328 S 2,6 2,3 4:23.58 brave
55435 rjp 20 0 1998148 633188 245572 S 2,0 3,9 142:36.18 soffice.bin
1024 avahi 20 0 8856 4864 3584 S 1,3 0,0 54:28.67 avahi-daemon
1250 root 20 0 1468264 189968 119724 S 1,3 1,2 369:49.37 Xorg
6542 rjp 20 0 1406,9g 529728 151200 S 1,0 3,3 184:50.52 brave
267027 rjp 20 0 802672 105836 85732 S 1,0 0,7 0:00.27 qterminal
2072 rjp 20 0 33,0g 829836 266356 S 0,7 5,1 160:09.34 brave
2115 rjp 20 0 32,5g 173892 122884 S 0,7 1,1 78:31.85 brave
267042 rjp 20 0 13480 4352 3456 R 0,7 0,0 0:00.06 top
1845 rjp 20 0 1725660 157576 98140 S 0,3 1,0 3:22.78 lxqt-panel
4838 rjp 20 0 1392,2g 616876 159348 S 0,3 3,8 31:06.12 brave
4852 rjp 20 0 1392,1g 410188 132424 S 0,3 2,5 38:22.46 brave
243165 root 20 0 0 0 0 I 0,3 0,0 0:12.40 kworker/0:3-events
263081 rjp 20 0 1393,8g 339500 142628 S 0,3 2,1 0:16.96 brave
264546 root 20 0 0 0 0 I 0,3 0,0 0:00.90 kworker/u12:0-flush-259:0
265023 rjp 20 0 32,7g 151452 106404 S 0,3 0,9 0:47.24 code
265090 rjp 20 0 1393,6g 113656 76928 S 0,3 0,7 0:03.28 code
265351 rjp 20 0 1393,6g 304976 83388 S 0,3 1,9 0:17.10 code
1 root 20 0 168404 13004 8012 S 0,0 0,1 0:10.91 systemd
2 root 20 0 0 0 0 S 0,0 0,0 0:00.10 kthreadd

Estados de um Processo [2/2]

- Transições de estado

1. O processo bloqueia aguardando uma entrada
2. O escalonador seleciona outro processo
3. O escalonador seleciona esse processo
4. A entrada torna-se disponível

- Processos entram no sistema na fila de prontos
- Transições dependem de interrupções para sinalizar condições
  - Término de operações de E/S, passagem do tempo, ...

Implementação de Processos [1/3]

- As informações sobre os processos do sistema são armazenados na **tabela de processos**
  - Uma entrada para cada processo
  - Cada uma delas é chamada de **descritor de processo** ou **bloco de controle de processo**
  - Os campos desta tabela devem ser salvos quando o processo for bloqueado, assim ele pode voltar do ponto que parou

Gerenciamento de processos	Gerenciamento de memória	Gerenciamento de arquivos
Registradores	Ponteiro para o segmento de código	Diretório-raiz
Contador de programa	Ponteiro para o segmento de dados	Diretório de trabalho
Palavra de estado do programa	Ponteiro para o segmento de pilha	Descritores de arquivos
Ponteiro de pilha		Identificador (ID) do usuário
Estado do processo		Identificador (ID) do grupo
Prioridade		
Parâmetros de escalonamento		
Identificador (ID) do processo		
Processo pai		
Grupo do processo		
Sinais		
Momento em que o processo iniciou		
Tempo usado da CPU		
Tempo de CPU do filho		
Momento do próximo alarme		

Implementação de Processos [2/3]

- O papel das interrupções
  - Interrupções são fundamentais para multiprogramação
    - Sinalizam eventos no sistema
    - Dão oportunidade para que o SO assuma o controle e decida o que fazer
  - **Processos não executam sobre o controle direto do SO**
    - O SO só assume quando ocorrem interrupções ou chamadas de sistema (implementadas com *traps*)

Implementação de Processos [3/3]

- Tratamento de Interrupções
  - O *hardware* empilha o contador de programa, PSW, etc
  - O *hardware* carrega o novo contador de programa a partir do vetor de interrupções
  - Rotina em ASSEMBLY salva os registradores
    - Na entrada da tabela de processos referente ao processo corrente
  - Rotina em ASSEMBLY configura uma nova pilha
    - Usada pelo manipulador de processos
  - Tratador de interrupções C executa
    - Possivelmente coloca algum processo no estado de ‘pronto’
  - O escalonador decide qual processo é o próximo a executar
  - Tratador de interrupções retorna para o código em ASSEMBLY
  - Rotina em ASSEMBLY inicia o novo processo corrente

Exercício [1/1]

- Elabore a árvore de processos para o segmento de código ao lado, e indique o valor de **n** em cada processo.
- Implemente o código em um ambiente Linux e verifique a árvore de processos gerada através do comando `pstree` do Linux

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

int main(void)
{
  int n = 5;

  if (fork() == 0)
     n++;
  n++;
  fork();
  n += 10;
  if (fork() > 0)
     n++;
  printf("n = %d\n", n);
  return 0;
}
```

UFSC

Referências

- *MAZIERO, C. Sistemas Operacionais: Conceitos e Mecanismos. Editora da UFPR, 2019. 456 p. ISBN 978-85-7335-340-2*