

Aula 2.2: Processos

Criação e manipulação de processos

Prof. Rodrigo Campiolo
Prof. Rogério A. Gonçalves¹

¹Universidade Tecnológica Federal do Paraná (UTFPR)
Departamento de Computação (DACOM)
Campo Mourão, Paraná, Brasil

Ciência de Computação

BCC34G - Sistemas Operacionais

- Aprender comandos básicos para manipulação de processos no SO GNU/Linux.
- Compreender a estrutura de processos no SO GNU/Linux.
- Criar processos no SO GNU/Linux.

Listando processos I

- Os comandos **ps**, **top** e **htop** possibilitam visualizar os processos em execução.
- Algumas das informações providas são:
 - pid: identificação do processo.
 - user: usuário que iniciou o processo.
 - pr: prioridade do processo (escalonamento).
 - ni: “nice” do processo (escalonamento).
 - virt (VSZ), res (RSS), shr, %MEM: uso de memória pelo processo (total de memória usada, memória usada em RAM, memória compartilhada, memória RAM disponível).
 - S ou STAT: estado do processo.
 - TIME+: total de tempo do processo desde sua inicialização.
 - COMMAND: nome do programa.
 - TTY: terminal associado ao processo.
 - %CPU: tempo de CPU.

- Principais estados (S):
 - R (RUNNING): em execução.
 - S (INTERRUPTIBLE SLEEP): esperando por um evento.
 - I (IDLE TASK): usado por threads ociosas em nível núcleo.
 - D (UNINTERRUPTIBLE SLEEP): esperando por E/S mas não pode ser interrompido.
 - T (STOPPED): processo suspenso (sinal de controle - CTRL + Z ou depuração).
 - Z (ZUMBI): processo finalizado, mas não “limpo” pelo processo-pai.

Listando processos III

Exemplos ps

```
# ps aux
```

```
# ps -eo pid,ppid,user ,cmd
```

```
# ps -eo pid,cmd,%mem,%cpu --sort=-%mem
```

Exemplo: jobs, &, bg, fg

```
# jobs
# sleep 20 &
# jobs
# fg
# pico teste.txt
CTRL + Z
# fg %1
```

Processos: CPU bound x IO bound

```
/* Processo CPU bound */
int main() {
    int a = 0;

    while (1)
    {
        a = a + 1;
        if (a == 32000) a = 0;
    }

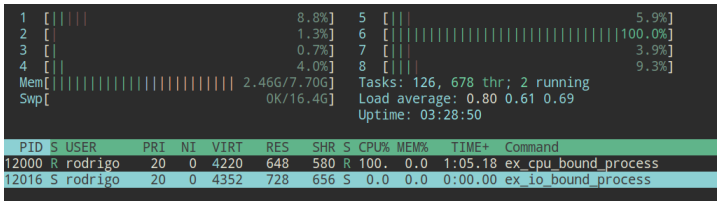
    return 0;
}
```

```
/* Processo IO bound */
int main() {
    char pnome[30];
    char unome[30];

    printf("Digite primeiro nome: ");
    scanf("%s", pnome);

    printf("Digite ltimo nome: ");
    scanf("%s", unome);

    printf("\t%s, %s\n", unome, pnome);
    return 0;
}
```



Criação de processos: fork

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>      // fork()
#include <sys/types.h>   // pid_t

int main(){
    pid_t  pid;
    int    valor = 0;
    pid = fork();        /* cria um processo e devolve pid do filho
                           para o pai e 0 para o filho */

    if (pid){            /* trecho executado pelo pai */
        printf("Eu Sou o Processo Pai - Filho %d \n", pid);
        valor = 5;
        printf("Valor: %d \n", valor);
    } else {              /* trecho executado pelo filho */
        printf("Eu Sou o Processo Filho - Filho %d \n", pid);
        valor = 10;
        printf("Valor: %d \n", valor);
    }

    exit(0);
}
```



Criação de processos: exec

```
#include <unistd.h>          // execl()
#include <stdio.h>

int main() {
    printf("Antes do exec\n");
    execl("/bin/ls", "/bin/ls", "-r", "-t", "-l", (char *) 0);
    printf("Depois do exec\n");

    return 0;
}
```

```
#include <unistd.h>          // execv()
#include <stdio.h>

int main() {
    printf("Antes do exec\n");
    char *cmd[] = { "ls", "-l", (char *)0 };
    execv("/bin/ls", cmd);
    printf("Depois do exec\n");

    return 0;
}
```

Processos: *Zombies*

```
#include <unistd.h>    // _exit()

int main(){
    for(int i=0; i<5; i++){
        if (fork() == 0) { // processo filho
            _exit(0); // and exit
        }
    }

    while(1);    // processo pai continua executando
}
```



Processos: Eliminando *Zombies*

```
#include <unistd.h>    // _exit(), sleep()
#include <sys/types.h>  // pid_t
#include <sys/wait.h>   // wait()
#include <stdio.h>

int main(){
    pid_t pid;
    int status;

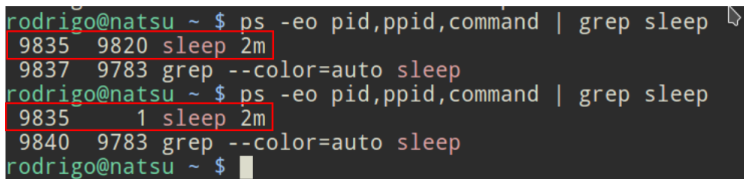
    for(int i=0; i<5; i++){
        if (fork() == 0) { // processo filho
            _exit(0); // and exit
        }
    }

    while (1) {
        pid = wait(&status);
        printf("-- Pai detecta processo %d foi finalizado com estado %d. \n", pid,
            status);
        sleep(1);
    }
}
```



Processos: Órfãos

- Um **processo órfão** é um processo cujo pai foi finalizado ou terminado.
- Processos órfãos são terminados ou adotados pelo processo **init**.
- **Órfão não intencional**: processo pai finaliza ou termina inesperadamente e os filhos são finalizados por um mecanismo de proteção contra órfãos acidentais da sessão/aplicação.
- **Órfão intencional**: processo desassociado da sessão/aplicação e executa em segundo plano; pai finaliza e os filhos são *re-parenting*.

A terminal window showing the execution of the 'ps' command. The first command is 'ps -eo pid,ppid,command | grep sleep', which shows two processes: '9835 9820 sleep 2m' and '9837 9783 grep --color=auto sleep'. The second command is 'ps -eo pid,ppid,command | grep sleep', which shows '9835 1 sleep 2m' and '9840 9783 grep --color=auto sleep'. Red boxes highlight the change in the parent PID (ppid) for the 'sleep' process from 9820 to 1.

```
rodrigo@natsu ~ $ ps -eo pid,ppid,command | grep sleep
9835 9820 sleep 2m
9837 9783 grep --color=auto sleep
rodrigo@natsu ~ $ ps -eo pid,ppid,command | grep sleep
9835 1 sleep 2m
9840 9783 grep --color=auto sleep
rodrigo@natsu ~ $
```

Figura 1: Processo sleep ppid: 9820 \Rightarrow 1

Processos: Mais comandos

- **kill**: envia um sinal para o processo.
`$ kill -SIGKILL 26004`
- **renice**: altera a prioridade de um processo.
`$ renice -n -10 -p 28990`
- **pidof**: devolve o(s) pid(s) de um processo a partir do nome.
`$ pidof kate`
- **chrt**: manipula atributos de tempo real de um processo.
`$ chrt -p 28990`
- **ps**: mostra a árvore de processos.
`$ ps`
- **watch**: executa um programa periodicamente.
`$ watch -n 1 'ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem'`
- **lsof**: mostra os arquivos abertos por processo.
`$ lsof -p 28990`

- ❶ Faça um programa que crie uma hierarquia de processos com 4 níveis ($1 + 2 + 4 + 8$) processos. Visualize a hierarquia usando um comando do sistema.
- ❷ Faça um programa que receba um comando Linux como parâmetro e execute como um filho do processo. O processo pai deve aguardar o término da execução do comando.
- ❸ Faça um programa que receba um vetor e divida para N filhos partes iguais de processamento para localizar um item. Exibir o PID dos filhos que encontrarem o valor procurado.

- POSIX Programmer's Manual, unistd.h(0P). Disponível em:
<http://man7.org/linux/man-pages/man0/unistd.h.0p.html>.