

Sistemas Operacionais Processos

Profª Drª Thaína Aparecida Azevedo Tosta
tosta.thaina@unifesp.br

Aula passada

- Chamadas de sistema para gerenciamento de diretórios
- Chamadas de sistema diversas
- Estrutura de sistemas operacionais

Sumário

- O modelo de processo
- Criação de processos
- Término de processos
- Estados de processos
- Implementação de processos

Objetivo: conceitos e códigos úteis para o Trabalho 1.

Processos

- O conceito mais central em qualquer sistema operacional é o processo: uma abstração de um programa em execução;
- Eles dão suporte à possibilidade de haver operações concorrentes mesmo em uma única CPU, transformando-a em múltiplas CPUs virtuais;
- Sem a abstração de processo a computação moderna não poderia existir.

Processos

- Todos os computadores modernos frequentemente realizam várias tarefas ao mesmo tempo;
- Exemplo: servidor web com solicitações de páginas chegando de toda parte;
 - Se a página requisita está em cache, ela é retornada;
 - Se não, uma solicitação de acesso ao disco é iniciada para buscá-la, o que é demorado.
- Como outras solicitações podem chegar, é necessário modelar e controlar essa concorrência: **processos**.

Processos

- Em qualquer sistema de multiprogramação, a CPU muda de um processo para outro rapidamente;
- Enquanto, estritamente falando, em qualquer dado instante a CPU está executando apenas um processo, no curso de 1s ela pode trabalhar em vários deles, dando a ilusão do paralelismo (pseudoparalelismo).

O modelo de processo

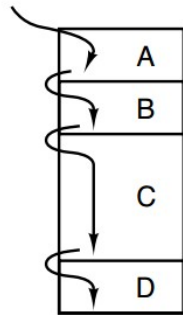
- Todos os softwares executáveis no computador, às vezes incluindo o sistema operacional, são organizados em uma série de processos sequenciais, ou, simplesmente, processos;
- Um processo é apenas uma instância de um programa em execução, incluindo os valores atuais do contador do programa, registradores e variáveis;
- CPU virtual vs CPU real: troca de processos (multiprogramação).

O modelo de processo

- Computador multiprogramando quatro programas na memória:

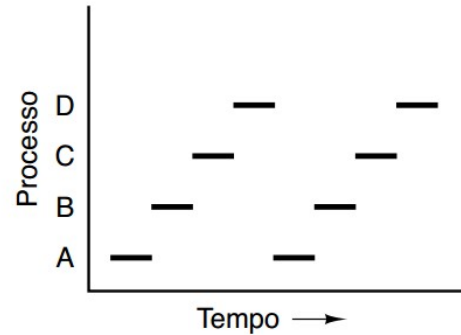
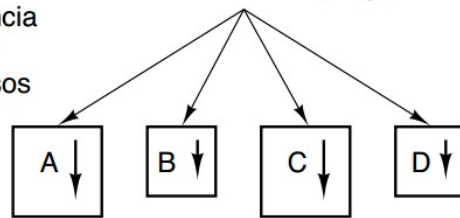
(a) Multiprogramação de quatro programas. (b) Modelo conceitual de quatro processos sequenciais independentes. (c) Apenas um programa está ativo de cada vez.

Um contador de programa



Alternância
entre
processos

Quatro contadores de programa



Se há dois núcleos (ou CPUs) cada um deles pode ser executado apenas com um processo de cada vez.

- Cada processo tem seu próprio fluxo de controle (seu próprio contador de programa lógico) com execução /independente dos outros processos.
 - Contador de programa físico vs contador de programa lógico do processo.

O modelo de processo

Qual a diferença entre programa e processo?

- Uma pessoa está preparando um bolo de aniversário:
 - Programa: receita;
 - Processador: pessoa;
 - Dados de entrada: ingredientes;
 - Processo: atividades de leitura da receita, busca de ingredientes e preparo do bolo.
- Criança aparece chorando por picada de abelha (pega livro de primeiros socorros e segue instruções):
 - Programa: livro de primeiros socorros;
 - Estado atual do processo em execução: registra onde estava na receita;
 - Novo processo: prestar cuidado médico;
 - Troca de processos: preparo do bolo torna-se prestar cuidado médico.

O modelo de processo

- A ideia fundamental aqui é que um processo é uma atividade de algum tipo;
 - Essa atividade tem um programa, uma entrada, uma saída e um estado.
- Um único processador pode ser compartilhado entre vários processos, com algum algoritmo de escalonamento para a troca entre eles;
- Em comparação, um programa é algo que pode ser armazenado em disco sem fazer nada;
- Vale a pena observar que se um programa está sendo executado duas vezes, é contado como dois processos.

Criação de processos

- Alguma maneira é necessária para criar e terminar processos, na medida do necessário, durante a operação;
 - Com exceção de sistemas simples ou de única aplicação (todos os processos na memória).
- Os quatro eventos para criação de processos são:
 1. Inicialização do sistema;
 2. Execução de uma chamada de sistema de criação de processo por um processo em execução;
 3. Solicitação de um usuário para criar um novo processo;
 4. Início de uma tarefa em lote.

Criação de processos

Inicialização do sistema

Nesse momento, uma série de processos é criada, divididos em:

- Primeiro plano: interagem com usuários e realizam trabalho para eles;
- Segundo plano: sem associação com usuários com funções específicas.
 - Processo para aceitar emails: inativo na maior parte do tempo, mas ativo quando chega um email (daemons).

Criação de processos

Execução de uma chamada de sistema de criação de processo por um processo em execução

- Muitas vezes, um processo em execução criará um ou mais processos novos para ajudá-lo em seu trabalho;
- Útil quando o trabalho pode ser formulado com vários processos relacionados, mas interagindo de maneira independente (↑rápido):
 - Dados buscados em uma rede para processamento
 - Processo 1: busca os dados e os coloca em memória compartilhada;
 - Processo 2: remove itens de dados e os processa.

Criação de processos

Solicitação de um usuário para criar um novo processo

- Em sistemas interativos, os usuários podem começar um programa digitando um comando ou clicando duas vezes sobre um ícone;
- Cada uma dessas ações inicia um novo processo e executa nele o programa selecionado.

Criação de processos

Início de uma tarefa em lote

- A última situação na qual processos são criados aplica-se somente aos sistemas em lote encontrados em grandes computadores;
- Gerenciamento de estoque:
 - Submissão de tarefas em lote ao sistema, com criação de novos processos quando ele tem todos os recursos para a sua tarefa.

Criação de processos

- No **UNIX**, após a `fork`, os processos pai e filho têm a mesma imagem de memória, as mesmas variáveis de ambiente e os mesmos arquivos abertos;
- Normalmente, o processo filho então executa uma chamada de sistema para mudar sua imagem de memória e executar um novo programa;
- Nenhuma memória para escrita é compartilhada;
- Alternativamente, o filho pode compartilhar toda a memória do pai, mas nesse caso, a memória é compartilhada no sistema ***copy-on-write*** (cópia-na-escrita).

Criação de processos

- No **Windows**, a chamada de *CreateProcess* (com 10 parâmetros) lida tanto com a criação do processo, quanto com o carga do programa correto no novo processo;
- Tanto no sistema UNIX quanto no Windows, após um processo ser criado, o pai e o filho têm os seus próprios espaços de endereços distintos;
- Se um dos dois processos muda uma palavra no seu espaço de endereço, a mudança não é visível para o outro processo.

Criação de processos

1. Inicialização do sistema;
- 2. Execução de uma chamada de sistema de criação de processo por um processo em execução;**
- 3. Solicitação de um usuário para criar um novo processo;**
4. Início de uma tarefa em lote.

Criação de processos

<https://man7.org/linux/man-pages/man2/fork.2.html>

SYNOPSIS [top](#)

```
#include <unistd.h>

pid_t fork(void);
```

DESCRIPTION [top](#)

fork() creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

The child process and the parent process run in separate memory spaces. At the time of **fork()** both memory spaces have the same content. Memory writes, file mappings (**mmap(2)**), and unmappings (**munmap(2)**) performed by one of the processes do not affect the other.

RETURN VALUE [top](#)

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and *errno* is set to indicate the error.

Criação de processos

```
1#include <stdio.h>
2#include <unistd.h>
3#include <errno.h>
4
5int main(int argc, char **argv) {
6    pid_t pid;
7
8    pid = fork();
9    if (pid == 0) { // processo filho
10        while(1) {
11            printf("Filho pid=%ld, PID=%ld\n", (long)pid, (long)getpid());
12            sleep(1);
13        }
14    } else if (pid > 0) { // processo pai
15        while(1) {
16            printf("Pai pid=%ld, PID=%ld\n", (long)pid, (long)getpid());
17            sleep(1);
18        }
19    } else { // erro
20        perror("fork()");
21        return -1;
22    }
23    return 0;
24 }
```

Término de processos

Um processo terminará, normalmente devido a uma das condições:

- 1.Saída normal (voluntária);
- 2.Erro fatal (involuntário);
- 3.Saída por erro (voluntária);
- 4.Morto por outro processo (involuntário).

Término de processos

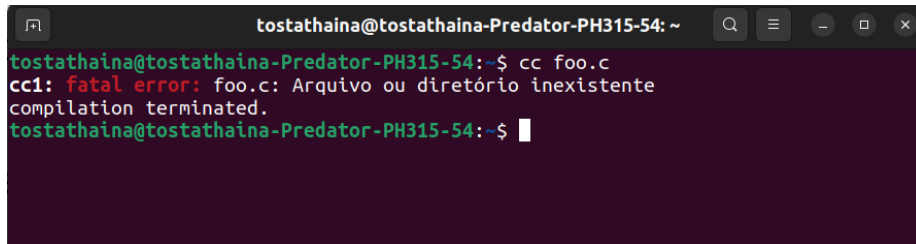
Saída normal (voluntária)

- A maioria dos processos termina por terem realizado o seu trabalho;
 - Exemplo: quando um compilador termina de traduzir o programa dado a ele, é executada uma chamada para dizer ao sistema operacional que ele terminou.
- Essa chamada é `exit` em UNIX e `ExitProcess` no Windows.

Término de processos

Erro fatal (involuntário)

- Quando o processo descobre um erro fatal;
 - Exemplo: compilar um arquivo inexistente.

A screenshot of a terminal window with a dark purple background. The window title is 'tostathaina@tostathaina-Predator-PH315-54: ~'. The terminal shows the command 'cc foo.c' being executed. The output is 'cc1: fatal error: foo.c: Arquivo ou diretório inexistente' followed by 'compilation terminated.' on the next line. The prompt 'tostathaina@tostathaina-Predator-PH315-54: \$' is visible at the bottom with a cursor.

```
tostathaina@tostathaina-Predator-PH315-54: ~  
tostathaina@tostathaina-Predator-PH315-54:~$ cc foo.c  
cc1: fatal error: foo.c: Arquivo ou diretório inexistente  
compilation terminated.  
tostathaina@tostathaina-Predator-PH315-54:~$
```

- Processos interativos com base em tela geralmente abrem uma caixa de diálogo e pedem ao usuário para tentar de novo.

Término de processos

Saída por erro (voluntária)

- Decorrente de um erro de programa;
 - Exemplos: executar uma instrução ilegal, referenciar uma memória não existente, ou dividir por zero.
- Em alguns sistemas, como o UNIX, um processo pode dizer ao sistema operacional que ele gostaria de lidar sozinho com determinados erros: interrupção e não término.

Término de processos

Morto por outro processo (involuntário)

- Um processo executa uma chamada de sistema dizendo ao sistema operacional para matar outro processo;
- Em UNIX, essa chamada é `kill` e a função Win32 correspondente é `TerminateProcess`;
- Em ambos os casos, o processo que mata o outro processo precisa da autorização necessária para fazê-lo.

Estados de processos

- Embora cada processo seja uma entidade independente (com seu próprio contador de programa e estado interno), eles podem interagir entre si, como em:

```
cat chapter1 chapter2 chapter3 | grep tree
```

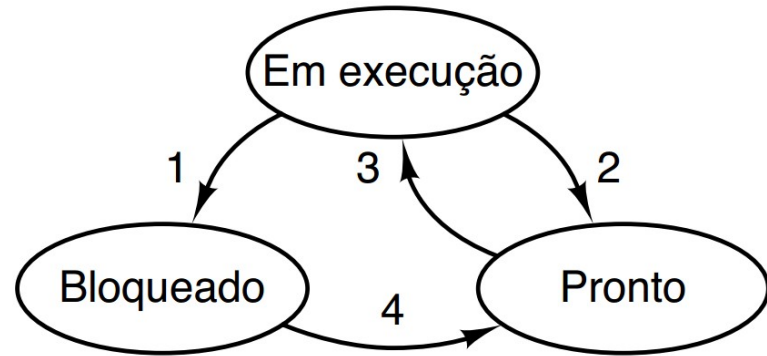
- Dependendo das velocidades relativas dos dois processos (pela complexidade relativa dos programas ou pelo tempo de CPU que cada um teve), grep pode estar pronto para ser executado, mas sem entrada disponível: bloqueado.

Estados de processos

- Os processos podem se encontrar em um de três estados:
 - 1.Em execução** (realmente usando a CPU naquele instante);
 - 2.Pronto** (executável, temporariamente parado para deixar outro processo ser executado);
 - 3.Bloqueado** (incapaz de ser executado até que algum evento externo aconteça).
- O terceiro estado não permite a execução do processo mesmo que a CPU esteja ociosa e não tenha nada mais a fazer.

Estados de processos

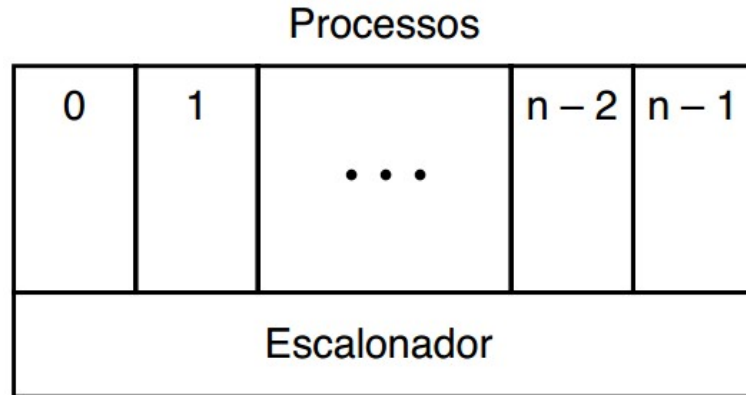
- O escalonamento, isto é, decidir qual processo deve ser executado, quando e por quanto tempo, é um assunto importante;
- Muitos algoritmos foram desenvolvidos para tentar equilibrar as demandas concorrentes de eficiência para o sistema como um todo e justiça para os processos individuais.



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

Estados de processos

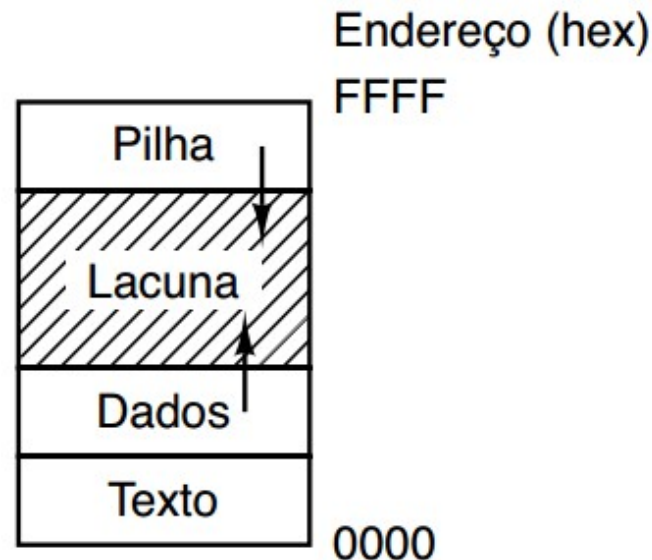
- O nível mais baixo do sistema operacional é o escalonador, com uma variedade de processos acima dele;
- Todo o tratamento de interrupções e detalhes sobre o início e parada de processos estão ocultos naquilo que é chamado aqui de escalonador.



Implementação de processos

Processos em UNIX têm sua memória dividida em:

- Segmento de texto: código de programa;
- Segmento de dados: variáveis;
- Segmento de pilha: dados temporários (como parâmetros de funções, endereços de retorno e variáveis locais).



Implementação de processos

- Para implementar o modelo de processos, o sistema operacional mantém um arranjo de estruturas chamado **tabela de processos**:
 - Uma entrada para cada processo (blocos de controle de processo).
- Essas entradas contêm tudo o que deve ser salvo quando há a troca de processo (em execução → pronto ou bloqueado):
 - Contador de programa, ponteiro de pilha, alocação de memória, estado dos arquivos abertos, informação sobre sua contabilidade e escalonamento, etc.

Implementação de processos

Entrada típica na tabela de processos (altamente dependente do sistema)

Gerenciamento de processo	Gerenciamento de memória	Gerenciamento de arquivo
Registros	Ponteiro para informações sobre o segmento de texto	Diretório-raiz
Contador de programa		Diretório de trabalho
Palavra de estado do programa	Ponteiro para informações sobre o segmento de dados	Descritores de arquivo
Ponteiro da pilha		ID do usuário
Estado do processo	Ponteiro para informações sobre o segmento de pilha	ID do grupo
Prioridade		
Parâmetros de escalonamento		
ID do processo		
Processo pai		
Grupo de processo		
Sinais		
Momento em que um processo foi iniciado		
Tempo de CPU usado		
Tempo de CPU do processo filho		
Tempo do alarme seguinte		

Implementação de processos

- Associada com cada classe de E/S, há um local de memória chamado de vetor de interrupção, com o endereço da rotina de serviço de interrupção;
- Exemplo: processo sendo executado com interrupção de disco
 - 1.O hardware de interrupção coloca na pilha do processo atual seu contador de programa, palavra de estado de programa e às vezes um ou mais registradores;
 - 2.O computador desvia a execução para o endereço especificado no vetor de interrupção;
 - 3.Software realiza a rotina do serviço de interrupção.

Implementação de processos

Nas interrupções:

1. Os registradores são salvos, muitas vezes na entrada da tabela de processo para o processo atual *;
2. A informação empurrada para a pilha pela interrupção é removida;
3. O ponteiro de pilha é configurado para uma pilha temporária usada pelo tratador de processos *;
4. Rotina do tipo específico da interrupção é chamada **;
5. Com a conclusão da interrupção, o escalonador é chamado;
6. O controle é passado de volta ao código de linguagem de montagem para carregar os registradores e mapa de memória para o processo agora atual e iniciar a sua execução.

* pequena rotina de linguagem de montagem

** rotina em C (comum em sistemas operacionais reais)

Implementação de processos

O esqueleto do que o nível mais baixo do sistema operacional faz quando ocorre uma interrupção

1. O hardware empilha o contador de programa etc.
2. O hardware carrega o novo contador de programa a partir do arranjo de interrupções.
3. O vetor de interrupções em linguagem de montagem salva os registradores.
4. O procedimento em linguagem de montagem configura uma nova pilha.
5. O serviço de interrupção em C executa (em geral lê e armazena temporariamente a entrada).
6. O escalonador decide qual processo é o próximo a executar.
7. O procedimento em C retorna para o código em linguagem de montagem.
8. O procedimento em linguagem de montagem inicia o novo processo atual.



**Objetivo: conceitos e códigos úteis
para o Trabalho 1.**

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <errno.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
6
7 int main(int argc, char **argv) {
8     pid_t pid;
9     if (argc == 1) {
10         printf("Uso: %s <comando> <p_1> <p_2> ... <p_n>\n", argv[0]);
11         return 0;
12     }
13
14     pid = fork();
15     if (pid == 0) { // processo filho: executa comando
16         char **cmd;
17         cmd = &argv[1];
18         execvp(cmd[0], cmd);
19     } else if (pid > 0) { // processo pai: aguarda termino do filho
20         int status;
21         waitpid(-1, &status, 0);
22     } else { // erro
23         perror("fork()");
24         return -1;
25     }
26     return 0;
27 }

```

```

tostathaina@tostathaina-Predator-PH315-54: ~/Documentos
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$ gcc -o exemplo-shell exem
plo-shell.c
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$ ./exemplo-shell
Uso: ./exemplo-shell <comando> <p_1> <p_2> ... <p_n>
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$ ./exemplo-shell ls -l -a
total 68
drwxr-xr-x  2 tostathaina tostathaina  4096 mar 11 09:27 .
drwxr-x--- 22 tostathaina tostathaina  4096 mar  6 13:41 ..
-rwxrwxr-x  1 tostathaina tostathaina 16136 mar 11 09:04 exemplo-fork
-rw-rw-r--  1 tostathaina tostathaina   453 mar 11 09:04 exemplo-fork.c
-rwxrwxr-x  1 tostathaina tostathaina 16192 mar 11 09:27 exemplo-shell
-rw-rw-r--  1 tostathaina tostathaina   544 mar 11 09:25 exemplo-shell.c
-rwxrwxr-x  1 tostathaina tostathaina 16024 mar  8 11:10 saida
-rw-rw-r--  1 tostathaina tostathaina   163 mar  8 11:10 teste.c
-rw-rw-r--  1 tostathaina tostathaina    0 mar  8 10:00 testeS0
-rwxrwxrwx  1 tostathaina tostathaina    0 mar  8 10:00 testeS02
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$

```

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <errno.h>
4 #include <string.h>
5 #include <sys/wait.h>
6 #define STR_LEN 100
7 int main(void) {
8     int fd[2];
9     pid_t child;
10    if (pipe(fd) < 0) {
11        perror("pipe()");
12        return (-1);
13    }
14    child = fork();
15    if (child == 0) { // processo filho le no 0
16        char msg[STR_LEN];
17        close(fd[1]);
18        bzero(msg, STR_LEN);
19        if (read(fd[0], msg, STR_LEN) > 0) {
20            printf("Filho (PID=%ld) leu: %s\n", (long)getpid(), msg);
21        }
22        printf("Filho terminando e dormindo por 5s. \n");
23        sleep(5);
24        return 0;
25    } else { // pai escreve no 1
26        char pa_msg[STR_LEN];
27        int status;
28        close(fd[0]);
29        bzero(pa_msg, STR_LEN);
30        printf("Pai (PID=%ld) escrevendo para o filho... \n", (long)getpid());
31        sprintf(pa_msg, "Filho, preste atencao! \n");
32        write(fd[1], pa_msg, strlen(pa_msg));
33        waitpid(-1, &status, 0);
34        printf("Pai: filho terminou... \n");
35    }
36    return 0;
37 }

```

```

tostathaina@tostathaina-Predator-PH315-54: ~/Documentos
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$ gcc -o exemplo-pipe1
exemplo-pipe1.c
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$ ./exemplo-pipe1
Pai (PID=13137) escrevendo para o filho...
Filho (PID=13138) leu: Filho, preste atencao!

Filho terminando e dormindo por 5s.
Pai: filho terminou...
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$

```

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <errno.h>
4 #include <string.h>
5 #include <sys/wait.h>
6 #define STR_LEN 100
7 int main(void) {
8     int fd[2], ch_fd[2];
9     pid_t child;
10    if (pipe(fd) < 0) {
11        perror("pipe()");
12        return(-1);
13    }
14    if (pipe(ch_fd) < 0) {
15        perror("pipe(ch_fd)");
16        return(-1);
17    }
18    child = fork();
19    if (child == 0) { // processo filho le no 0
20        char msg[STR_LEN];
21        close(fd[1]);
22        close(ch_fd[0]);
23        bzero(msg, STR_LEN);
24        if (read(fd[0], msg, STR_LEN) > 0) {
25            printf("Filho (PID=%ld) leu: %s\n", (long)getpid(), msg);
26        }
27        bzero(msg, STR_LEN);
28        sprintf(msg, "OK, papai!");
29        write(ch_fd[1], msg, strlen(msg));
30        printf("Filho terminando e dormindo por 5s. \n");
31        sleep(5);
32        return 0;
33    } else { // pai escreve no 1
34        char pa_msg[STR_LEN];
35        int status;
36        close(fd[0]);
37        close(ch_fd[1]);
38        bzero(pa_msg, STR_LEN);
39        printf("Pai (PID=%ld) escrevendo para o filho... \n", (long)getpid());
40        sprintf(pa_msg, "Filho, preste atencao! \n");
41        write(fd[1], pa_msg, strlen(pa_msg));
42        bzero(pa_msg, STR_LEN);
43        read(ch_fd[0], pa_msg, STR_LEN);
44        printf("Pai leu: %s\n", pa_msg);
45        waitpid(-1, &status, 0);
46        printf("Pai: filho terminou... \n");
47    }
48    return 0;
49 }

```

```

tostathaina@tostathaina-Predator-PH315-54: ~/Documentos
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$ gcc -o exemplo-pipe2
exemplo-pipe2.c
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$ ./exemplo-pipe2
Pai (PID=13543) escrevendo para o filho...
Filho (PID=13544) leu: Filho, preste atencao!

Filho terminando e dormindo por 5s.
Pai leu: OK, papai!
Pai: filho terminou...
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$

```



```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <errno.h>
4 #include <string.h>
5 #include <sys/wait.h>
6 #define STR_LEN 100
7 int main(void) {
8     int fd[2], ch_fd[2];
9     pid_t child;
10    if (pipe(fd) < 0) {
11        perror("pipe()");
12        return(-1);
13    }
14    if (pipe(ch_fd) < 0) {
15        perror("pipe(ch_fd)");
16        return(-1);
17    }
18    child = fork();
19    if (child == 0) { // processo filho le no 0
20        char msg[STR_LEN];
21        close(fd[1]);
22        close(ch_fd[0]);
23        bzero(msg, STR_LEN);
24        if (read(fd[0], msg, STR_LEN) > 0) {
25            printf("Filho (PID=%ld) leu: %s\n", (long)getpid(), msg);
26        }
27        bzero(msg, STR_LEN);
28        sprintf(msg, "OK, papai!");
29        write(ch_fd[1], msg, strlen(msg));
30        printf("Filho terminando e dormindo por 5s. \n");
31        sleep(5);
32        return 0;
33    } else { // pai escreve no 1
34        char pa_msg[STR_LEN];
35        int status;
36        close(fd[0]);
37        close(ch_fd[1]);
38        bzero(pa_msg, STR_LEN);
39        printf("Pai (PID=%ld) escrevendo para o filho e dormindo por 3s... \n", (long)getpid());
40        sprintf(pa_msg, "Filho, preste atencao! \n");
41        sleep(3);
42        write(fd[1], pa_msg, strlen(pa_msg));
43        bzero(pa_msg, STR_LEN);
44        read(ch_fd[0], pa_msg, STR_LEN);
45        printf("Pai leu: %s\n", pa_msg);
46        waitpid(-1, &status, 0);
47        printf("Pai: filho terminou... \n");
48    }
49    return 0;
50 }

```

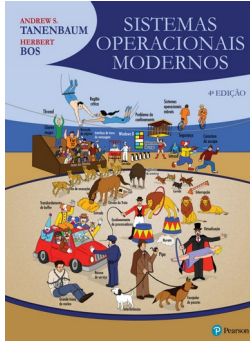
```

tostathaina@tostathaina-Predator-PH315-54: ~/Documentos
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$ gcc -o exemplo-pipe3
exemplo-pipe3.c
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$ ./exemplo-pipe3
Pai (PID=14013) escrevendo para o filho e dormindo por 3s...
Filho (PID=14014) leu: Filho, preste atencao!

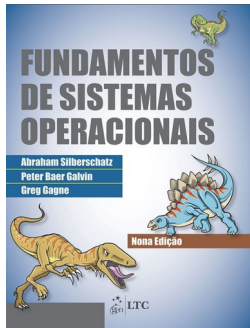
Filho terminando e dormindo por 5s.
Pai leu: OK, papai!
Pai: filho terminou...
tostathaina@tostathaina-Predator-PH315-54:~/Documentos$

```


Referências



TANENBAUM, Andrew S.; BOS, Herbert. Sistemas operacionais modernos. 4. edição. São Paulo: Pearson, 2016. xviii, 758 p. ISBN 9788543005676.



SILBERSCHATZ, Abraham.; GALVIN, Peter Baer.; GAGNE, Greg. Fundamentos de sistemas operacionais. 9. edição. Rio de Janeiro: LTC, 2015.

O modelo desta apresentação foi criado pelo Slidesgo.

Agradeço ao Prof. Bruno Kimura da Universidade Federal de São Paulo pelo material disponibilizado.