

Fundamentos de Sistemas Operacionais

Professores: Cristiano Bonato Both



JESUÍTAS BRASIL

 UNISINOS

Somos infinitas possibilidades

Sumário

- Primitivas de sistemas operacionais UNIX
- *Inter-Process Communication (IPC)*
- Comunicação em Sistemas Distribuídos
- Referências



JESUÍTAS BRASIL

 UNISINOS

Somos infinitas possibilidades

Relembrando a aula passada

- Programação concorrente
 - Composta por um conjunto de processos sequenciais que são executados concurrentemente
 - Processos disputam recursos comuns
 - Um processo pode cooperar com uma tarefa, mas é capaz de afetar, ou ser afetado, pela execução de outro processo

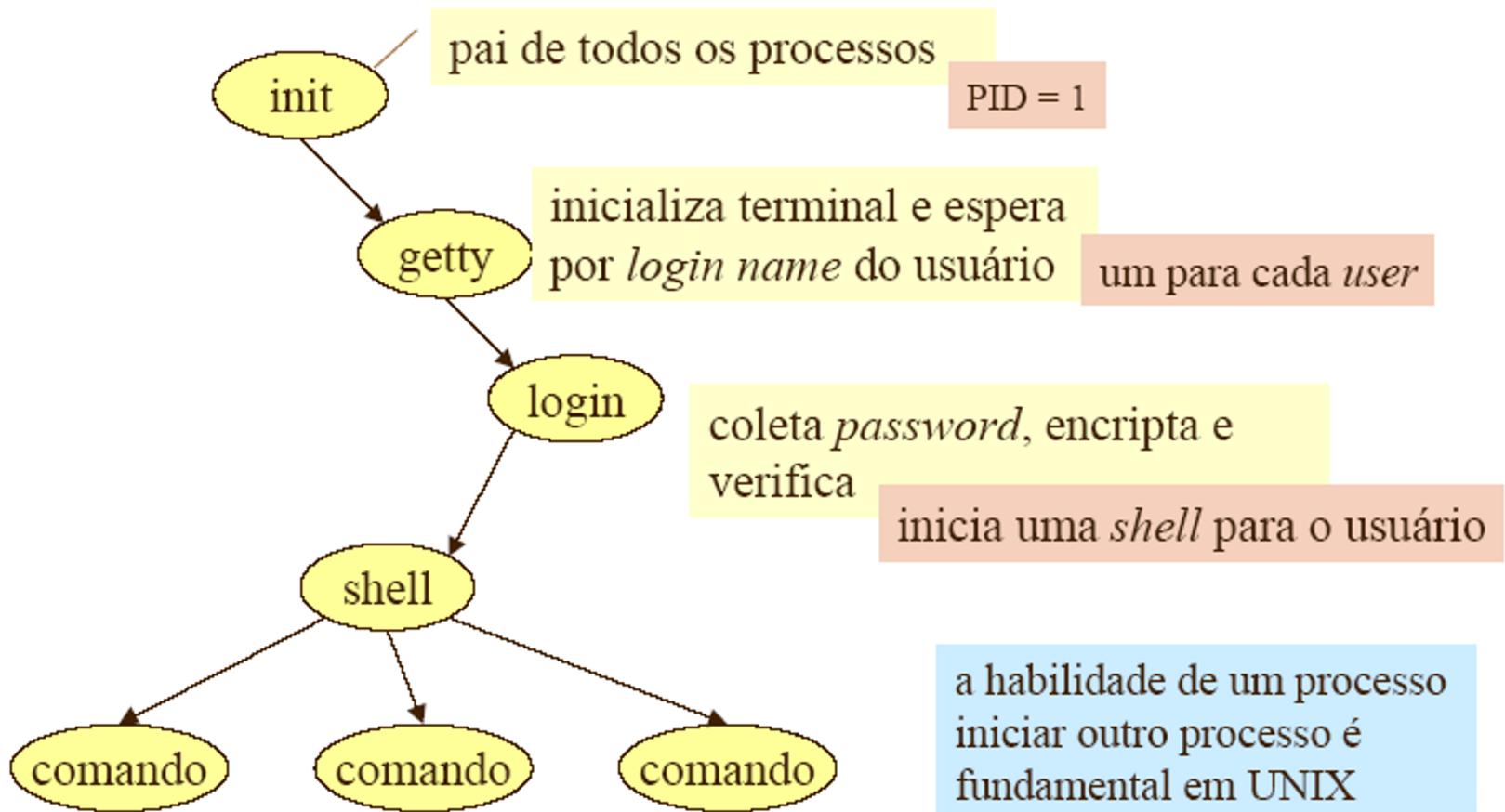


JESUÍTAS BRASIL

 UNISINOS

Somos infinitas possibilidades

Processos iniciais



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Exec

- Substituindo a imagem do processo
 - “exec”: substitui o processo atual por um novo código, cujo arquivo ou *path* é passado como parâmetro



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Execlp

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

int main( ) {
    printf("Running ps with execlp\n");
    execlp("ps", "ps", "-ax", NULL);
    printf("Done.\n");
    exit(0);
}
```

- O que aconteceu com o “Done.”?
- Qual o PID do exemplo?
- Qual o PID do comando ps?



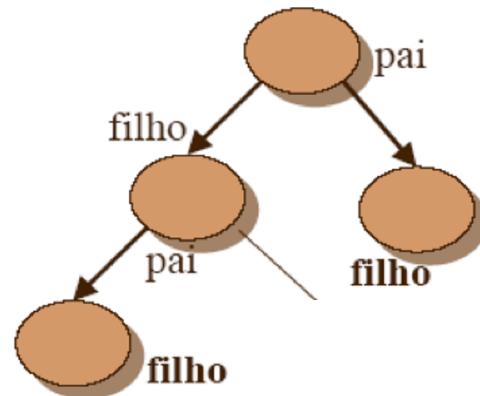
JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Criação de processos

- Pai:
 - Um processo pode criar outros processos
- Filho:
 - Pode receber dados inicializados pelo pai
 - Pode obter recursos diretamente do Sistema Operacional
 - Um filho pode ser pai de outros filhos



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Criação de processos

- Execução:
 - Pai executa concorrentemente com os filhos
 - Pai espera que alguns ou todos os filhos concluam a execução
- Conteúdo de memória:
 - O filho é uma duplicata do pai
 - É carregado um programa no espaço de endereçamento de memória do filho



JESUÍTAS BRASIL

 UNISINOS

Somos infinitas possibilidades

Fork

- Pai executa *fork*
 - Os dois processos continuam a execução, após *fork* (na instrução seguinte ao *fork* em cada processo)
 - Permite ao pai se comunicar com o filho
 - O filho é uma cópia do pai (mesmo conteúdo, mas em posições diferentes de memória)
 - O filho recebe 0
 - O pai recebe o PID do filho
 - Todos os arquivos abertos antes do *fork* continuam abertos



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Fork

- O *fork* dá ao filho um novo identificador de processo PID (que é informado ao pai)
- Memória principal é alocada para dados e pilha
 - Uma nova *Page Table* é construída
- Geralmente, não é necessário copiar o código, pois os processos compartilham o mesmo código
- São preservados arquivos abertos, identificadores de *user* e *group*, manipuladores de sinais



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Fork

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

int main( ) {
    pid_t pid;
    char *message;
    int n;

    printf("Iniciando FORK\n");
    pid = fork( );
```

- O que aconteceu?

```
switch(pid) {
    case -1:
        perror("errouu!\n");
        exit(1);
    case 0:
        message = "\nFILHO\n";
        n = 5; break;
    default:
        message = "\nPAI\n";
        n = 3; break;
}
for(; n > 0; n--) {
    printf("PID=%d\n", pid);
    puts(message);
    sleep(1);
}
exit(0);}
```



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Fork e exec

- Pai
 - Se não tem nada para fazer, espera
- Filho
 - Até chamar exec, o filho executa o mesmo programa do pai
- Tipicamente, um dos dois processos chama exec
- execve:
 - Carrega um arquivo binário na memória do processo (destruindo a imagem da memória do processo que chamou execve)
 - É um dos mais usados, mas qualquer exec serve!

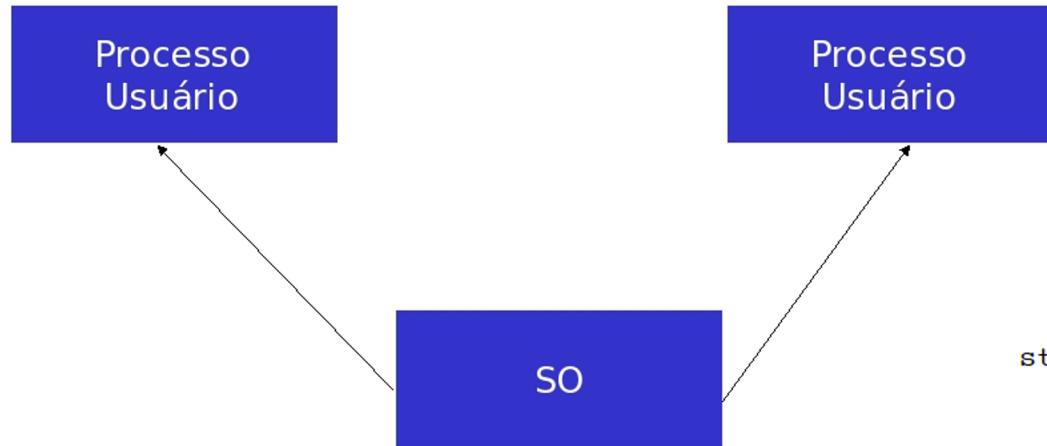


JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Inter-Process Communication



IPC em um mesmo sistema

```
struct desc_proc{  
    char     estado_atual;  
    int      prioridade;  
    unsigned inicio_memoria;  
    unsigned tamanho_mem;  
    struct   arquivos arquivos_abertos[20];  
    unsigned tempo_cpu;  
    unsigned proc_pc;  
    unsigned proc_sp;  
    unsigned proc_acc;  
    unsigned proc_rx;  
    struct   desc_proc *proximo;  
}  
  
struct desc_proc tab_desc[MAX_PROCESS];
```



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Inter-Process Communication

- *Pipes* (disciplina de Sistemas Operacionais)
 - Mecanismo IPC mais característico em UNIX
 - A mais antiga forma de IPC
 - Usado também na linguagem *shell*
- *Sockets* (disciplina de Redes)
 - Mecanismo genérico de IPC que suporta comunicação entre processos pela rede de computadores
 - Pode ser utilizado também para comunicação de processos no mesmo sistema



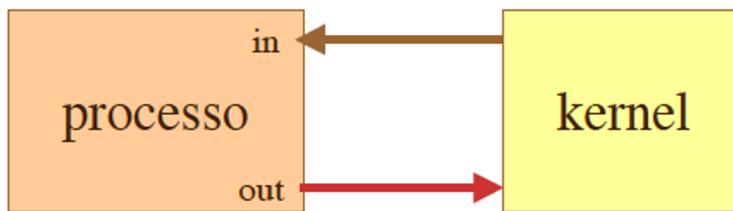
JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Pipe

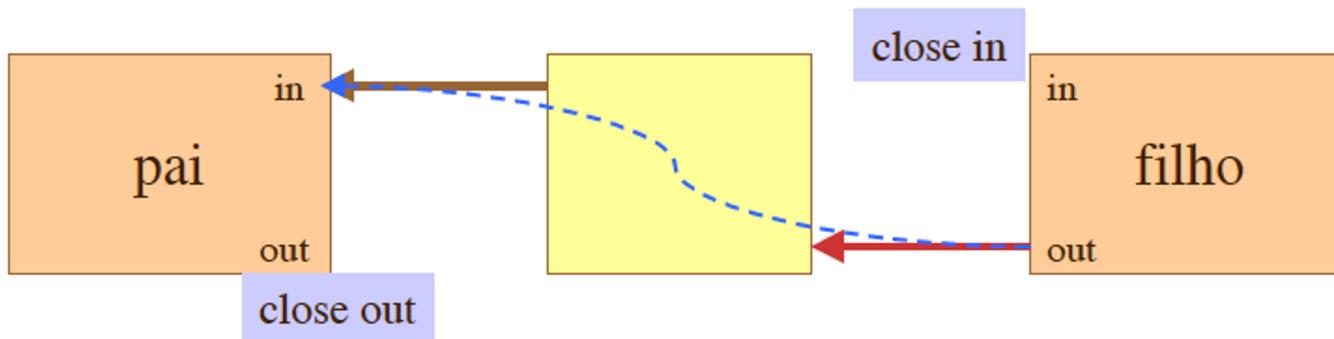
- Conecta uma saída de um processo a entrada de outro processo



Após a execução
da chamada pipe



Com fork()



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Pipe - chamada

- Argumento: array de inteiros de 2 posições que representa *File Descriptions* (fd):
 - fd[0] - é usado para **ler**
 - fd[1] - é usado para **escrever**
- Resultado: 0 (sucesso); -1 (em caso de erro)
- Se a chamada for bem sucedida:
 - Um descritor é usado para escrever dados no Pipe
 - O outro é usado para ler dados no pipe



JESUÍTAS BRASIL

 UNISINOS

Somos infinitas possibilidades

Pipe - função

`#include <unistd.h>`

`int pipe(int file_descriptor[2]);`

- Descritores de arquivo e não stream
 - *Read* e *Write* de baixo nível
 - Em caso de falha, retornar -1 e seta erro
- Escrita e leitura obedecem uma ordem FIFO
 - Após a criação do filho, **pai e filho precisam estabelecer o sentido da comunicação**
 - Pai para filho ou filho para pai

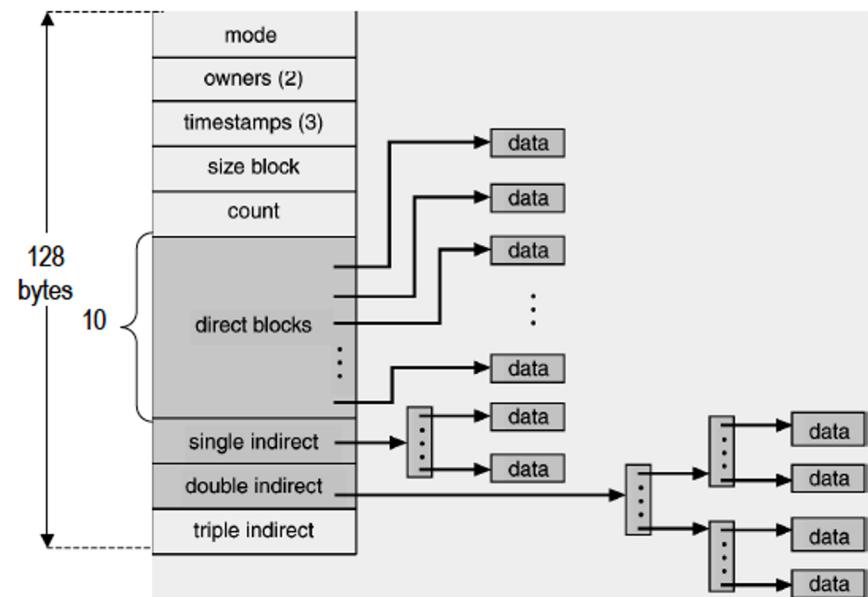
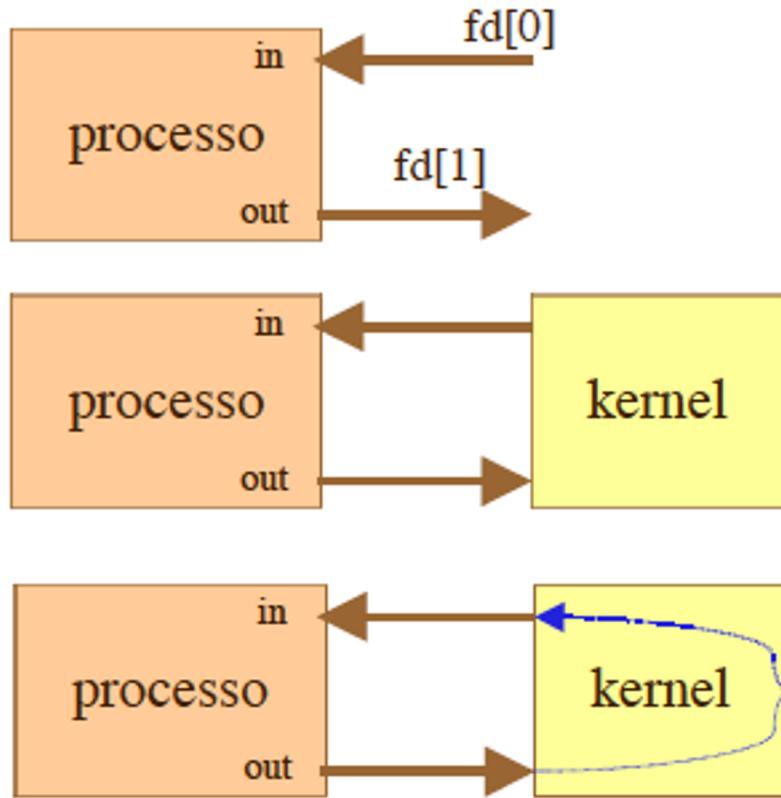


JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Pipe - criação



*Pipes são representados internamente por um *inode**

O *Kernel* cria 2 *file descriptors*:

- `fd[0]` para obter os dados (*read*)
- `fd[1]` para saída (*write*)

Até esse ponto, o processo só pode se comunicar com ele mesmo



JESUÍTAS BRASIL

UNISINOS

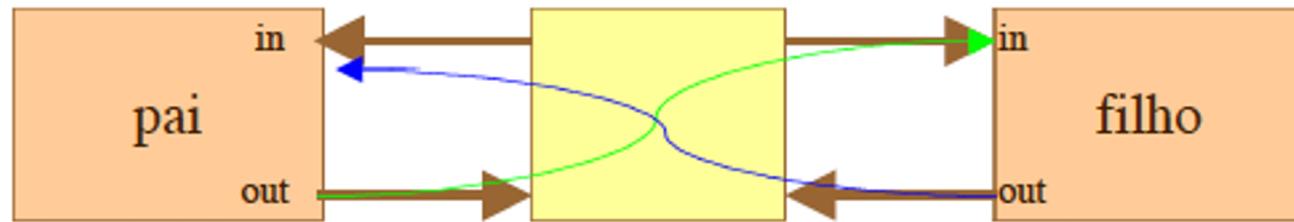
Somos infinitas possibilidades

Inter-Process Communication

- **Unidirecional**



- Os 2 processos entram em acordo e fecham a ponta do Pipe que cada um não vai usar



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Pipe - função

- Acesso a um Pipe
 - Mesmas funções usadas para *low-level file I/O*
 - *write()* para enviar dados pelo Pipe
 - *read()* para obter dados do Pipe
- Certas funções, como *Iseek()*, não operam com descritores para Pipes



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Pipe – exemplo 1 - criação

```
1 #include <unistd.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <string.h>
5
6 int main(){
7     int data_processed;
8     int file_pipes[2];
9     const char some_data[] = "123";
10    char buffer[BUFSIZ + 1]; // BUFSIZ = 1024 --> default value
11
12    memset(buffer, '\0', sizeof(buffer));
13
14    if (pipe(file_pipes) == 0) {
15        data_processed = write(file_pipes[1], some_data, strlen(some_data));
16        printf("Wrote %d bytes\n", data_processed);
17        data_processed = read(file_pipes[0], buffer, BUFSIZ);
18        printf("Read %d bytes: %s\n", data_processed, buffer);
19        exit(EXIT_SUCCESS);
20    }
21    exit(EXIT_FAILURE);
22 }
```

```
[aCusco Aula_88 $ gcc pipel.c -o pipel
[aCusco Aula_88 $ ./pipel
Wrote 3 bytes
Read 3 bytes: 123
```

Matthew & Stones, cap 12



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Pipe – exemplo 2 – pipe & fork

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(){
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1]; // BUFSIZ = 1024 --> default value
    pid_t fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == -1) {
        fork_result = fork();
        if(fork_result == -1){
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }
        if (fork_result == 0) {
            data_processed = read(file_pipes[0], buffer, BUFSIZ);
            printf("Read %d bytes: %s\n", data_processed, buffer);
            exit(EXIT_SUCCESS);
        }
        else {
            data_processed = write(file_pipes[1], some_data, strlen(some_data));
            printf("Wrote %d bytes\n", data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}
```

- Lembrete: pai e filho executam o mesmo programa

```
[aCuscoII:Aula_08 cbboth$ ./pipe2
Wrote 3 bytes
Read 3 bytes: 123
```



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Pipe – exemplo 3 – pipe & exec

- Problema:
 - Processo que chama `exec()` substitui imagem do processo atual
- Solução:
 - Passar o *fd* como parâmetro ao novo programa
 - *fd* é apenas um número
 - Pode ser facilmente passado como parâmetro



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Pipe – exemplo 3 – pipe & exec

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(){
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1]; // BUFSIZ = 1024 --> default value
    pid_t fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == -1) {
        fork_result = fork();
        if(fork_result == (pid_t)-1){
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }
        if (fork_result == 0) {
            sprintf(buffer, "%d", file_pipes[0]); // coloca o descriptor no buffer
            (void)execl("pipe4", "pipe4", buffer, (char *)0);
            exit(EXIT_FAILURE);
        }
        else {
            data_processed = write(file_pipes[1], some_data, strlen(some_data));
            printf("Wrote %d bytes\n", data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}
```

- O que é o pipe4?



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Pipe – exemplo 3 – pipe & exec

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int data_processed;
    char buffer[BUFSIZ + 1];
    int file_descriptor;

    memset(buffer, '\0', sizeof(buffer));
    sscanf(argv[1], "%d", &file_descriptor);
    data_processed = read(file_descriptor, buffer, BUFSIZ);

    printf("%d - read %d bytes: %s\n", getpid(), data_processed, buffer);
    exit(EXIT_SUCCESS);
}
```

pipe4.c

```
aCuscoII:Aula_08 cbboth$ ./pipe3
Wrote 3 bytes
13984 - read 3 bytes: 123
aCuscoII:Aula_08 cbboth$ ./pipe3
Wrote 3 bytes
aCuscoII:Aula_08 cbboth$ 13986 - read 3 bytes: 123
```

- Programa consumidor que lê os dados

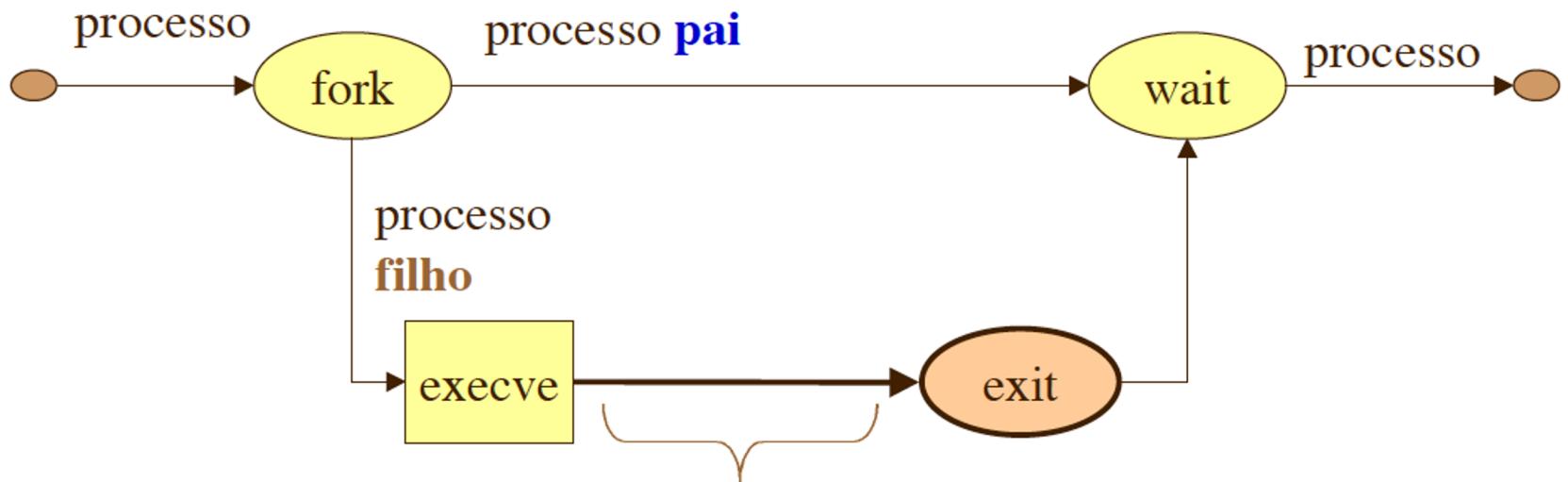


JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Wait - esperando por um filho



- Processo normal:
 - A associação de um filho com seu pai permanece, mesmo após o seu término, até que o pai termine ou chame `wait()`
 - Seus dados de saída estão armazenados e podem ser usado pelo pai em uma chamada `wait()` posterior



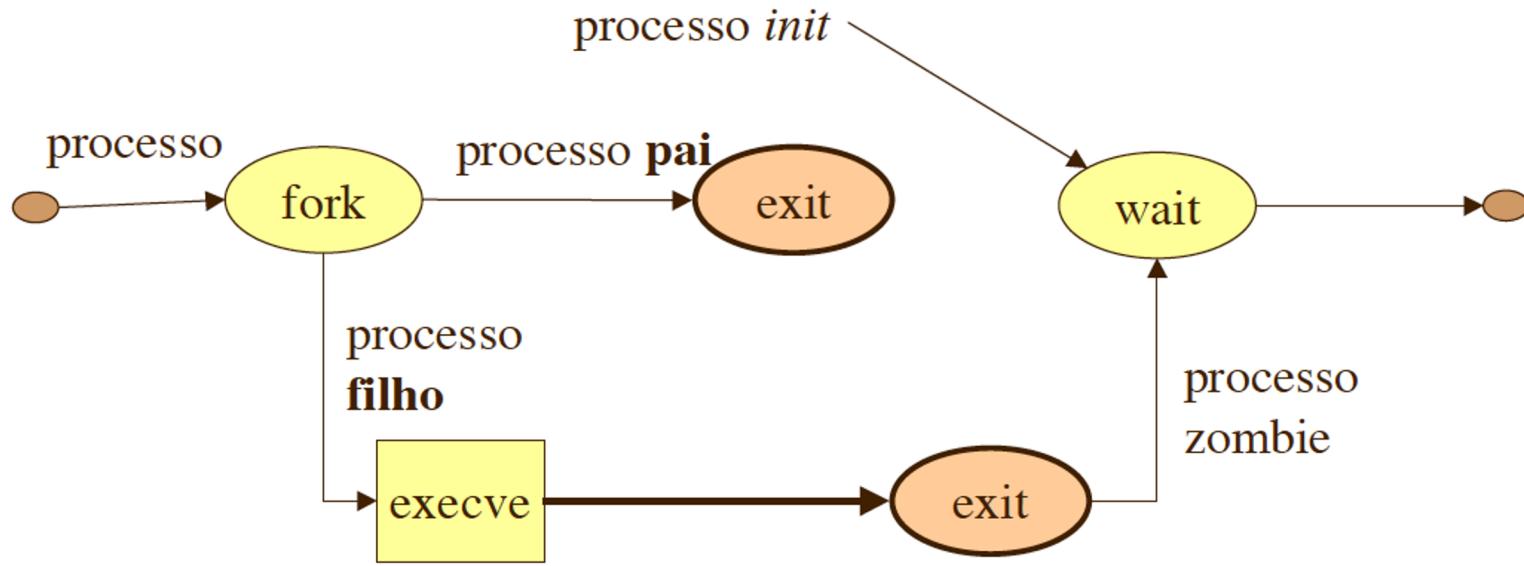
JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Zombie

- Quando o pai termina antes do filho, processo *zombie*
- O processo *zombie* é adotado pelo processo *init*
 - Processo *init* é o pai de todos os processos de usuários



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Exemplo: wait

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

int main( ) {
    pid_t pid;
    char *message;
    int n;
    int exit_code;

    printf("Fork program starting: PID=%d\n", getpid());
    pid = fork( );
    switch(pid) {
        case -1:
            perror("fork failed\n");
            exit(1);
        case 0:
            message = "child: %d\n";
            n = 5;
            exit_code = 37;
            break;
        default:
            message = "parent: %d\n";
            n = 3;
            exit_code = 0;
            break;
    }
    for(; n > 0; n--) {
        printf(message, getpid());
        sleep(1);
    }
}
```

Ajuda - WIFEXITED(status):
Retorna *true* se o processo-filho terminou normalmente

```
/* This section of the program waits for the child process to finish */

if(pid != 0) {
    int stat_val;
    pid_t child_pid;

    child_pid = wait(&stat_val);

    printf("Parent terminated\n");
    if(WIFEXITED(stat_val))
        printf("Msg from parent: child proc %d exited with code %d\n", child_pid, WEXITSTATUS(stat_val));
    else
        printf("Msg from Parent: child proc %d terminated abnormally\n", child_pid);
}
else {
    printf("Child has terminated\n");
}
exit(exit_code);
```



JESUÍTAS BRASIL

Somos infinitas possibilidades

Signal

- **Gerado pelo Sistema Operacional** para algumas condições de erro
 - Ex: instrução ilegal, violação de segmento de memória, etc.
- **Gerado por um outro processo**
 - Pode ser considerado uma mensagem especial
 - Processo reage a recepção do sinal executando alguma ação ou ignorando
 - Processo normalmente é interrompido
 - Interrupção de software



JESUÍTAS BRASIL

 UNISINOS

Somos infinitas possibilidades

Signal

- Exemplos:
 - SIGILL: produzido por instrução ilegal
 - SIGINT: gerado pelo terminal (Ctrl-C), envio ao processo em background
 - SIGHUP: comando *Kill* pode enviar um *signal* qualquer a um dado processo conhecendo seu PID

```
[aCusco Aula_06 $ ps -af
  UID  PID  PPID   C STIME   TTY          TIME CMD
    0  5020    280   0  9:51AM ttys000      0:00.06 login -pf cbboth
    501  5021  5020   0  9:51AM ttys000      0:00.04 -bash
    0  5028  5021   0  9:52AM ttys000      0:00.01 ps -af
aCusco Aula_06 $ kill -HUP 5021
```

- Veja todos os sinais no man page
 - \$ man signal



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Signal

```
1 #include <signal.h>
2 #include <stdio.h>
3 #include <unistd.h>
4
5 /* Função que será chamada */
6 void ouch(int sig) {
7     printf("OUCH! - I got signal %d\n", sig);
8     (void) signal(SIGINT, SIG_DFL); /* SIG_DFL : Replace the current signal handler with the default handler */
9 }
10
11 int main(){
12     (void) signal(SIGINT, ouch); /* SIGINT gerado pelo terminal (Ctrl-C)*/
13     while(1) {
14         printf("Hello Word!\n");
15         sleep(1);
16     }
17 }
```

```
[aCusco Aula_86 $ vim ctrl.c
[aCusco Aula_86 $ gcc ctrl.c -o ctrl
[aCusco Aula_86 $ ./ctrl
Hello Word!
Hello Word!
Hello Word!
^COUCH! - I got signal 2
Hello Word!
Hello Word!
Hello Word!
^C
```



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Signal

- Sinais podem ser perdidos
 - Se um segundo sinal do mesmo tipo ocorrer antes que o processo capturar o primeiro, o primeiro será sobescrito
- Não existe prioridade entre os sinais
 - Se dois sinais forem enviados a um processo ao mesmo tempo, é indeterminada a ordem de recebimento dos sinais
- Sinais podem conduzir a condições de corrida (*race conditions*)
 - Ex: o 4.3 BSD introduziu um padrão mais confiável para manipulação de sinais
 - Função *sigaction* é mais robusta que a função *signal*



JESUÍTAS BRASIL

UNISINOS

Somos infinitas possibilidades

Referências Bibliográficas

- SILBERSCHATZ, A.; GALVIN, Peter; GAGNE Greg, Operating System Concepts Essentials. John Wiley & Sons, Inc. 2th edition, 2013.
- TANENBAUM, Andrew S. Sistemas operacionais modernos. 3a. ed. São Paulo: Pearson, 2009-2013. p. 653.
- OLIVEIRA, Rômulo; CARÍSSIMI, Alexandre; TOSCANI, Simão. Sistemas Operacionais. Porto Alegre: Bookman, 4a. ed. 2010.



JESUÍTAS BRASIL

 UNISINOS

Somos infinitas possibilidades