

Processos: Implementação

Prof. Gustavo Girão
girao@imd.ufrn.br

Roteiro

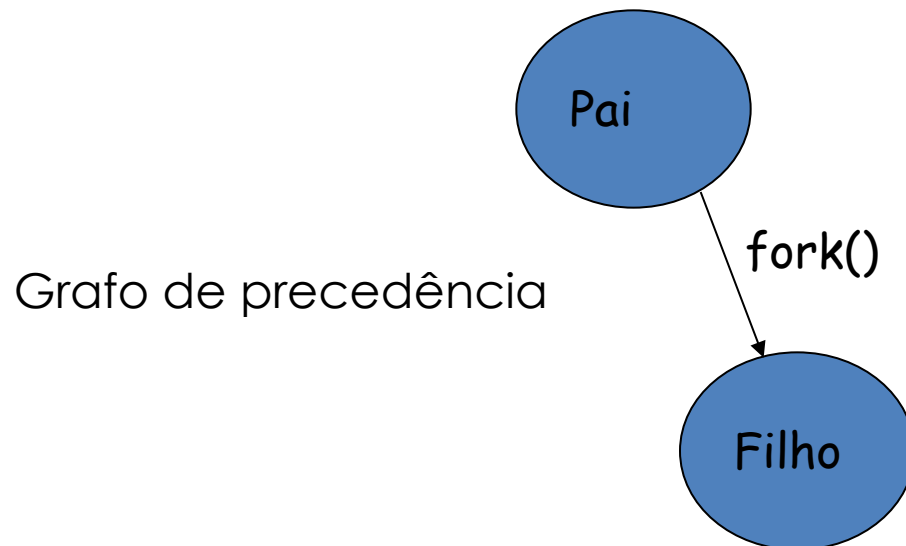
- Criação de processos
- Chamadas de sistema
- Exemplos
- Exercícios
- Comunicação entre processos

Criação de Processos

- Processo pai cria processo filho, o qual, por sua vez, pode criar outros processos, formando uma árvore de processos.
- Geralmente, processos são identificados e gerenciados via um **Identificador de Processos** (*Process Identifier* - **PID**)
- Compartilhamento de Recursos
 - Pai e filho compartilham todos os recursos.
 - Filho compartilha um subconjunto dos recursos do pai.
 - Pai e filho não compartilham recursos.
- Execução
 - Pai e filho executam concorrentemente.
 - Pai espera até filho terminar.

Hierarquia entre Processos

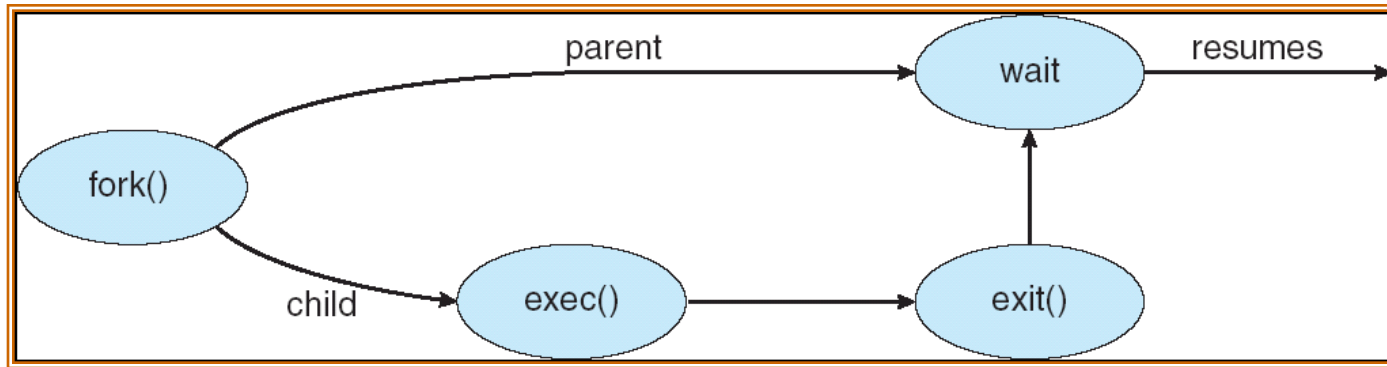
- Por exemplo, no Unix cria-se um processo via a primitiva `fork()`;
 - O criador é o Pai
 - O processo criado é o Filho



Criação de Processos (Cont.)

- Espaço de endereçamento
 - Filho duplica espaço do pai.
 - Filho tem um programa carregado no seu espaço.
- Exemplos no UNIX
 - Chamada de sistemas **fork** cria um novo processo.
 - Chamada de sistemas **exec** é usada após o **fork** para sobrescrever o espaço de memória do processo com um novo programa.

Criação de Processos (Cont.)



Chamadas de sistema

- fork
 - `int fork();`
 - ✧ Retorna para o processo-pai o pid do processo criado
 - ✧ Retorna para o processo-filho o valor 0
 - ✧ Retorna um valor negativo em caso de erro
 - Exemplo:

```
int main()
{
    pid_t  pid;
    /* cria outro processo */
    pid = fork();
    if (pid < 0) { /* ocorrência de erro*/
        fprintf(stderr, "Criação Falhou");
        exit(-1);
    }
    else if (pid == 0) { /* processo filho*/
    }
    else { /* processo pai */
    }
}
```

Chamadas de sistema

- fork
 - Quando o processo é criado, é feita uma cópia do espaço de endereçamento do processo pai
 - ✧ Código + todas as variáveis
 - O processo filho herda do pai todos os atributos
 - Ambos executam a instrução seguinte ao fork

Informações de um processo

- Obter PID do processo
 - getpid()
 - ✧ Retorna o PID do processo
 - getppid()
 - ✧ Retorna o PID do processo-pai
 - getuid()
 - ✧ Retorna o ID do usuário
 - getgid()
 - ✧ Retorna o ID do grupo

Chamadas de sistema

- wait()
 - int wait();
 - ✧ Retorna o valor do processo-filho finalizado
 - ✧ Retorna -1 caso o processo não tenha filhos
 - Exemplo:

```
int main()
{
    pid_t  pid;
    /* cria outro processo */
    pid = fork();
    if (pid < 0) { /* ocorrência de erro*/
        fprintf(stderr, "Criação Falhou");
        exit(-1);
    }
    else if (pid == 0) { /* processo filho*/
    }
    else { /* processo pai */
        wait(NULL);
    }
}
```

Chamadas de sistema

- `execvp()`
 - `execvp(char * caminho, char * parametros);`
 - ✧ A lista de parâmetros é terminada com `NULL`
 - Exemplo:

```
int main()
{
    pid_t  pid;
    /* cria outro processo */
    pid = fork();
    if (pid < 0) { /* ocorrência de erro*/
        fprintf(stderr, "Criação Falhou");
        exit(-1);
    }
    else if (pid == 0) { /* processo filho*/
        execvp("/bin/ls", "ls", "-l", NULL);
    }
    else { /* processo pai */
    }
}
```

Chamadas de sistema

- `exit()`
 - `void exit(char n);`
 - ✧ Deve ser passado como parametro um valor entre 0 e 255 que representa uma informação sobre a finalização do processo

- Exemplo:

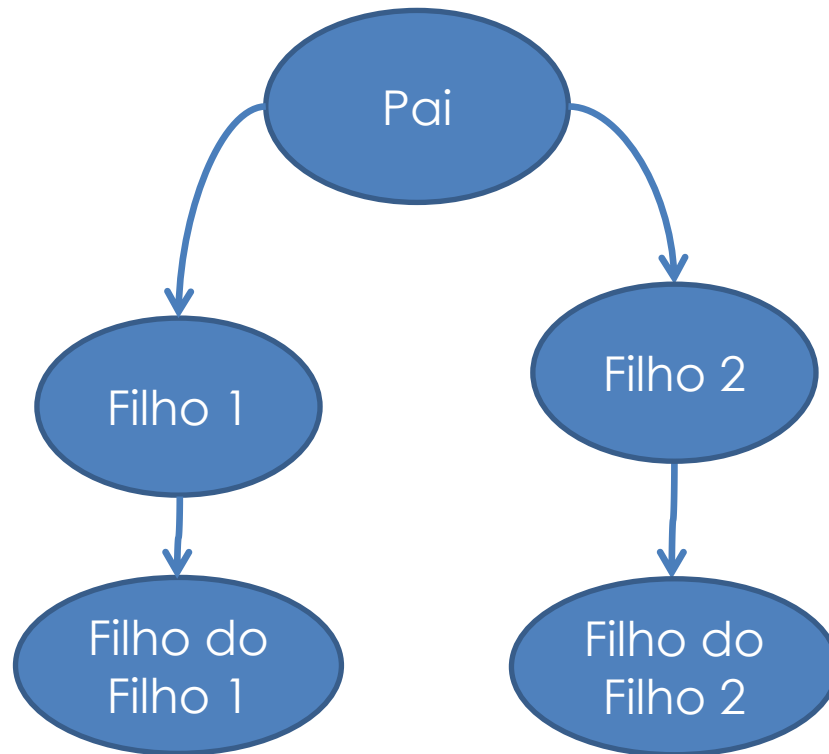
```
int main()
{
    pid_t  pid;
    /* cria outro processo */
    pid = fork();
    if (pid < 0) { /* ocorrência de erro*/
        fprintf(stderr, "Criação Falhou");
        exit(-1);
    }
    else if (pid == 0) { /* processo filho*/
    }
    else { /* processo pai */
    }
    exit(0);
}
```

Programa em C Criando Processos Separados

```
int main()
{
    Pid_t  pid;
    /* cria outro processo */
    pid = fork();
    if (pid < 0) { /* ocorrência de erro*/
        fprintf(stderr, "Criação Falhou");
        exit(-1);
    }
    else if (pid == 0) { /* processo filho*/
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* processo pai */
        /* pai irá esperar o filho completar
        execução */
        wait (NULL);
        printf ("Filho Completou Execução");
        exit(0);
    }
}
```

Exercícios

- Tente recriar o grafo de precedência abaixo



Como funciona a execução de dois processos?

```
int main()
{
    Pid_t  pid;
    /* cria outro processo */
    pid = fork();
    if (pid < 0) { /* ocorrência de
erro*/
        fprintf(stderr, "Criação
Falhou");
        exit(-1);
    }
    else if (pid == 0) { /* processo
filho*/
        execlp("/bin/ls", "ls",
NULL);
    }
    else { /* processo pai */
        /* pai irá esperar o filho
completar execução */
        wait (NULL);
        printf ("Filho Completou
```

Como funciona a execução de dois processos?

```
int main()
{
    Pid_t  pid;
    /* cria outro processo */
    pid = fork();
    if (pid < 0) { /* ocorrência de
erro*/
        fprintf(stderr, "Criação
Falhou");
        exit(-1);
    }
    else if (pid == 0) { /* processo
filho*/
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* processo pai */
        /* pai irá esperar o filho
completar execução */
        wait (NULL);
        printf ("Filho Completou
Execução");
    }
}
```




Como funciona a execução de dois processos?

```
int main()
{
    Pid_t pid;
    /* cria outro processo */
    pid = fork();
    if (pid < 0) { /* ocorrência de erro*/
        fprintf(stderr, "Criação Falhou");
        exit(-1);
    }
    else if (pid == 0) { /* processo filho*/
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* processo pai */
        /* pai irá esperar o filho completar execução
        */
        wait (NULL);
        printf ("Filho Completou Execução");
        exit(0);
    }
}
```




Como funciona a execução de dois processos?

- Cada processo tem seu próprio contador de programa
- Quais dos dois irá executar depende da escolha do escalonador




```
int main()
{
    Pid_t pid;
    /* cria outro processo */
    pid = fork();
    if (pid < 0) { /* ocorrência de erro*/
        fprintf(stderr, "Criação Falhou");
        exit(-1);
    }
    else if (pid == 0) { /* processo filho*/
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* processo pai */
        /* pai irá esperar o filho completar execução */
        wait (NULL);
        printf ("Filho Completou Execução");
        exit(0);
    }
}
```




```
int main()
{
    Pid_t pid;
    /* cria outro processo */
    pid = fork();
    if (pid < 0) { /* ocorrência de erro*/
        fprintf(stderr, "Criação Falhou");
        exit(-1);
    }
    else if (pid == 0) { /* processo filho*/
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* processo pai */
        /* pai irá esperar o filho completar execução */
        wait (NULL);
        printf ("Filho Completou Execução");
        exit(0);
    }
}
```

Como funciona a execução de dois processos?

- Cada processo tem seu próprio contador de programa
- Quais dos dois irá executar depende da escolha do escalonador



```
int main()
{
    Pid_t pid;
    /* cria outro processo */
    pid = fork();
    if (pid < 0) { /* ocorrência de erro*/
        fprintf(stderr, "Criação Falhou");
        exit(-1);
    }
    else if (pid == 0) { /* processo filho*/
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* processo pai */
        /* pai irá esperar o filho completar execução */
        wait (NULL);
        printf ("Filho Completou Execução");
        exit(0);
    }
}
```




```
int main()
{
    Pid_t pid;
    /* cria outro processo */
    pid = fork();
    if (pid < 0) { /* ocorrência de erro*/
        fprintf(stderr, "Criação Falhou");
        exit(-1);
    }
    else if (pid == 0) { /* processo filho*/
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* processo pai */
        /* pai irá esperar o filho completar execução */
        wait (NULL);
        printf ("Filho Completou Execução");
        exit(0);
    }
}
```

Processo **PAI** foi escalonado: executa a(s) instrução(ões) seguinte(s)

Como funciona a execução de dois processos?

- Cada processo tem seu próprio contador de programa
- Quais dos dois irá executar depende da escolha do escalonador



```
int main()
{
    Pid_t pid;
    /* cria outro processo */
    pid = fork();
    if (pid < 0) { /* ocorrência de erro*/
        fprintf(stderr, "Criação Falhou");
        exit(-1);
    }
    else if (pid == 0) { /* processo filho*/
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* processo pai */
        /* pai irá esperar o filho completar execução */
        wait (NULL);
        printf ("Filho Completou Execução");
        exit(0);
    }
}
```



```
int main()
{
    Pid_t pid;
    /* cria outro processo */
    pid = fork();
    if (pid < 0) { /* ocorrência de erro*/
        fprintf(stderr, "Criação Falhou");
        exit(-1);
    }
    else if (pid == 0) { /* processo filho*/
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* processo pai */
        /* pai irá esperar o filho completar execução */
        wait (NULL);
        printf ("Filho Completou Execução");
        exit(0);
    }
}
```

Processo **FILHO** foi escalonado: executa a(s) instrução(ões) seguinte(s)

Exercícios

- O que o programa ao lado exibe na tela?

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

int main(){
    pid_t pid, pid1;
    pid = fork();

    if (pid < 0){
        printf("Erro ao criar processo\n");
        return 1;
    }
    else if(pid == 0){
        pid1 = getpid();
        printf("Filho: pid = %d\n", pid);
        printf("Filho: pid1 = %d\n", pid1);
    }
    else{
        pid1 = getpid();
        printf("Pai: pid = %d\n", pid);
        printf("Pai: pid1 = %d\n", pid1);
        wait(NULL);
    }
    return 0;
}
```

Exercícios

- Qual o valor escrito na tela para “value”?

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

```
int value = 5;
```

```
int main(){
    pid_t pid;

    pid = fork();
    if( pid == 0){
        value += 15;
        return 0;
    }
    else if(pid > 0){
        wait(NULL);
        printf("PAI: value = %d\n",
value);
        return 0;
    }
}
```

Exercícios

- Faça um programa em que três processos executam paralelamente as seguintes ações:
 - Pai – cria dois filhos e em seguida imprime os números de 1 a 50, com um intervalo de 2 segundos entre cada número.
 - ✧ Após imprimir todos os números, imprime a frase “Processo pai vai morrer”.
 - Filho1 - Imprime os números de 100 a 199, com um intervalo de 1 segundo entre cada número.
 - ✧ Antes de imprimir os números, imprime a frase “Filho 1 foi criado”.
 - ✧ Após imprimir todos os números, imprime a frase “Filho 1 vai morrer”.
 - Filho2 - Imprime os números de 200 a 299, com um intervalo de 1 segundo entre cada número.
 - ✧ Antes de imprimir os números, imprime a frase “Filho 2 foi criado”.
 - ✧ Após imprimir todos os números, imprime a frase “Filho 2 vai morrer”.
- Importante:
 - Em cada printf os processos devem imprimir o seu pid e o pid do seu pai.
 - A função **sleep(n)** faz com que o processo durma por **n** segundos

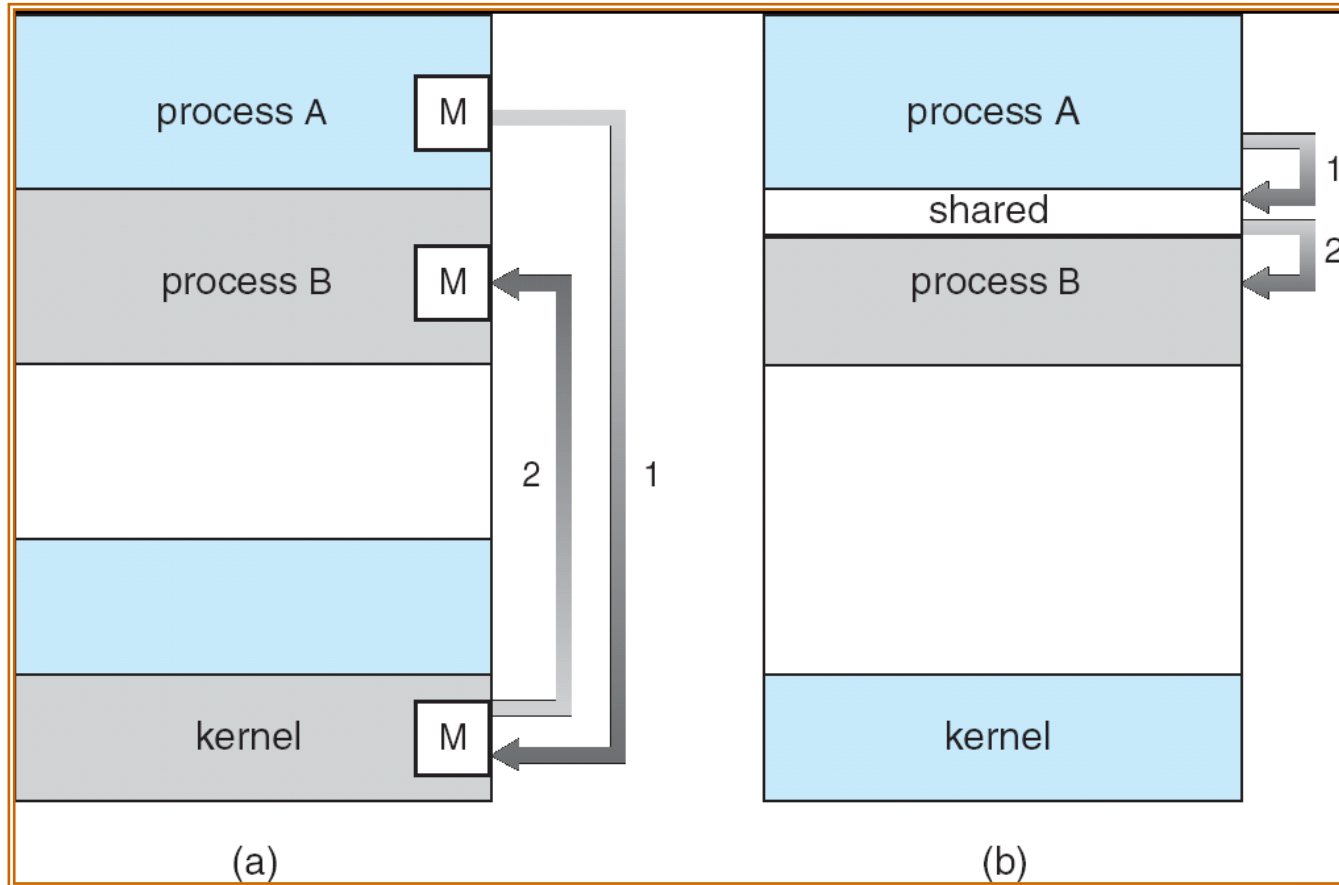
Terminação de Processos

- Processo executa última declaração e pede ao sistema operacional para decidir (**exit**).
 - Dados de saída passam do filho para o pai (via **wait**).
 - Recursos do processo são desalocados pelo sistema operacional.
- Pai pode terminar a execução do processo filho (**abort**).
 - Filho se excedeu alocando recursos.
 - Tarefa delegada ao filho não é mais necessária.
 - Pai está terminando.
 - ✧ Sistema operacional não permite que um filho continue sua execução se seu pai terminou.
 - ✧ Todos os filhos terminam - Terminação em cascata.
 - No Linux utilizamos **kill(pid, SIGNAL)**

Comunicação entre Processos (IPC)

- Processos em um sistema podem ser **Independentes** ou **Cooperantes**
- Processos **Independentes** não podem afetar ou ser afetados pela execução de outro processo.
- Processos **Cooperantes** podem afetar ou ser afetados pela execução de outro processo
- Razões para cooperação entre processos:
 - Compartilhamento de Informações
 - Aumento na velocidade da computação
 - Modularidade
 - Conveniência
- Processos cooperantes precisam de **Comunicação entre Processos (IPC – *interprocess communication*)**
- Dois modelos de IPC: memória compartilhada e troca de mensagens

Modelos de Comunicações



Comunicação por memória compartilhada

- Memória Compartilhada no POSIX
 - Processo cria primeiro um segmento de memória compartilhado
 - `int shmget(key_t key, int size, int shmflg);`
 - ✧ Key é a chave de acesso e identifica se o segmento de memória é compartilhado ou privado.
 - ✧ Size é o tamanho do segmento a ser criado em bytes
 - ✧ Shmflg é um argumento que identifica direitos de acesso de leitura, escrita e execução (similar ao uso do CHMOD)
 - ✧ O valor de retorno de um identificador do segmento de memória (shmid)

```
segment id = shmget(IPC_PRIVATE, size,  
IPC_CREAT | 0666);
```

Comunicação por memória compartilhada

- Memória Compartilhada no POSIX
 - Processo que deseja acesso a essa memória compartilhada deve se **ANEXAR** (ou **acopla**) a ela:
 - `Void *shmat(int shmid, const void *shmaddr, int shmflg);`
 - ✧ Shmid é o identificador do segmento de memória que se deseja acoplar.
 - ✧ É o endereço do acoplamento. Se o valor for NULL ou 0, a memória compartilhada é anexada ao primeiro endereço possível determinado pelo sistema (caso mais comum)
 - ✧ Shmflg é um argumento que identifica parâmetros diferentes para o endereçamento.
 - ✧ O valor retornado é o endereço do segmento de memória compartilhado. Em caso de erro, o valor é -1

```
shared_memory = (char *) shmat(id, NULL, 0);
```

Comunicação por memória compartilhada

- Memória Compartilhada no POSIX
 - Agora o processo pode escrever na memória compartilhada

```
sprintf(shared_memory, "teste de mem  
compartilhada");
```

- Quando terminar, um processo pode desanexar a memória compartilhada do seu espaço de armazenamento

```
shmdt(shared_memory);
```

Exercício!

- Faça um programa que cria dois processos filhos.
- O programa deve criar um segmento de memória compartilhado que contém um inteiro. Este inteiro deve ser incrementado pelo filho 1 e em seguida multiplicado por 2 pelo filho 2.

Pipes

- **Pipes** permitem a comunicação no estilo produtor-consumidor
- Produtor escreve em um extremo (o extremo de escrita do pipe)
- Consumidor lê do outro extremo (o extremo de leitura do pipe)
- Pipes comuns são unidirecionais
- Necessitam de relação pai-filho entre os processos comunicantes

Pipes

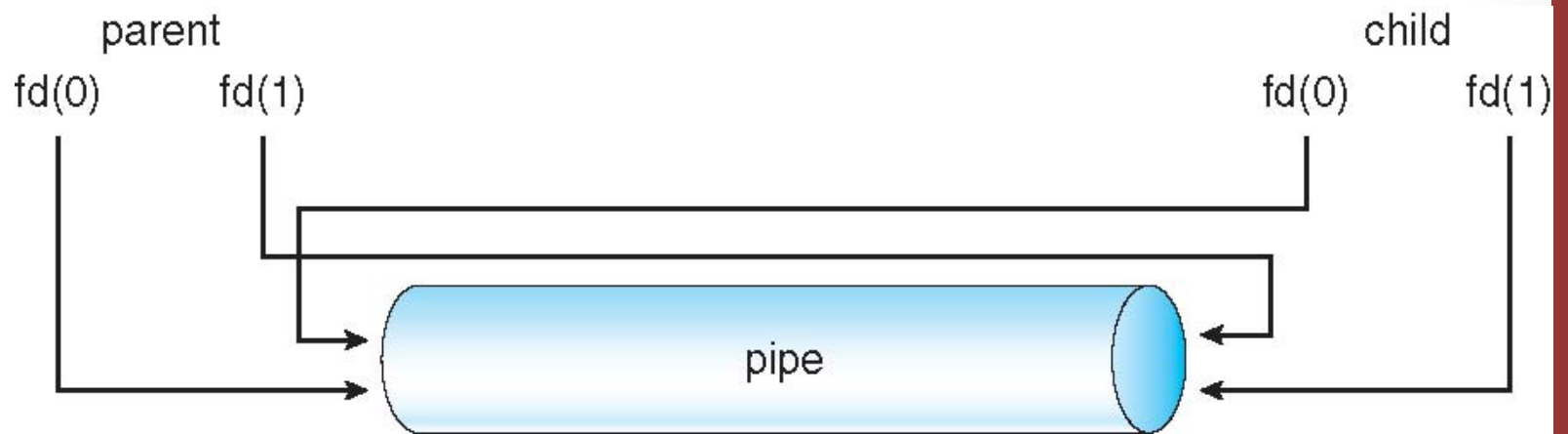
- **Formato:**

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

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

- É passado como parametro um array de dois descritores de arquivo. Cada arquivo representa um lado do pipe.
- Um estará aberto para leitura (receptor) e o outro para escrita (emissor).
- Para leituras, utiliza-se a primitiv:
 - **read(int filedes, void *buf, size_t nbyte)**
- Para escritas, utiliza-se a primitiva:
 - **write(int filedes, const void *buf, size_t nbyte)**

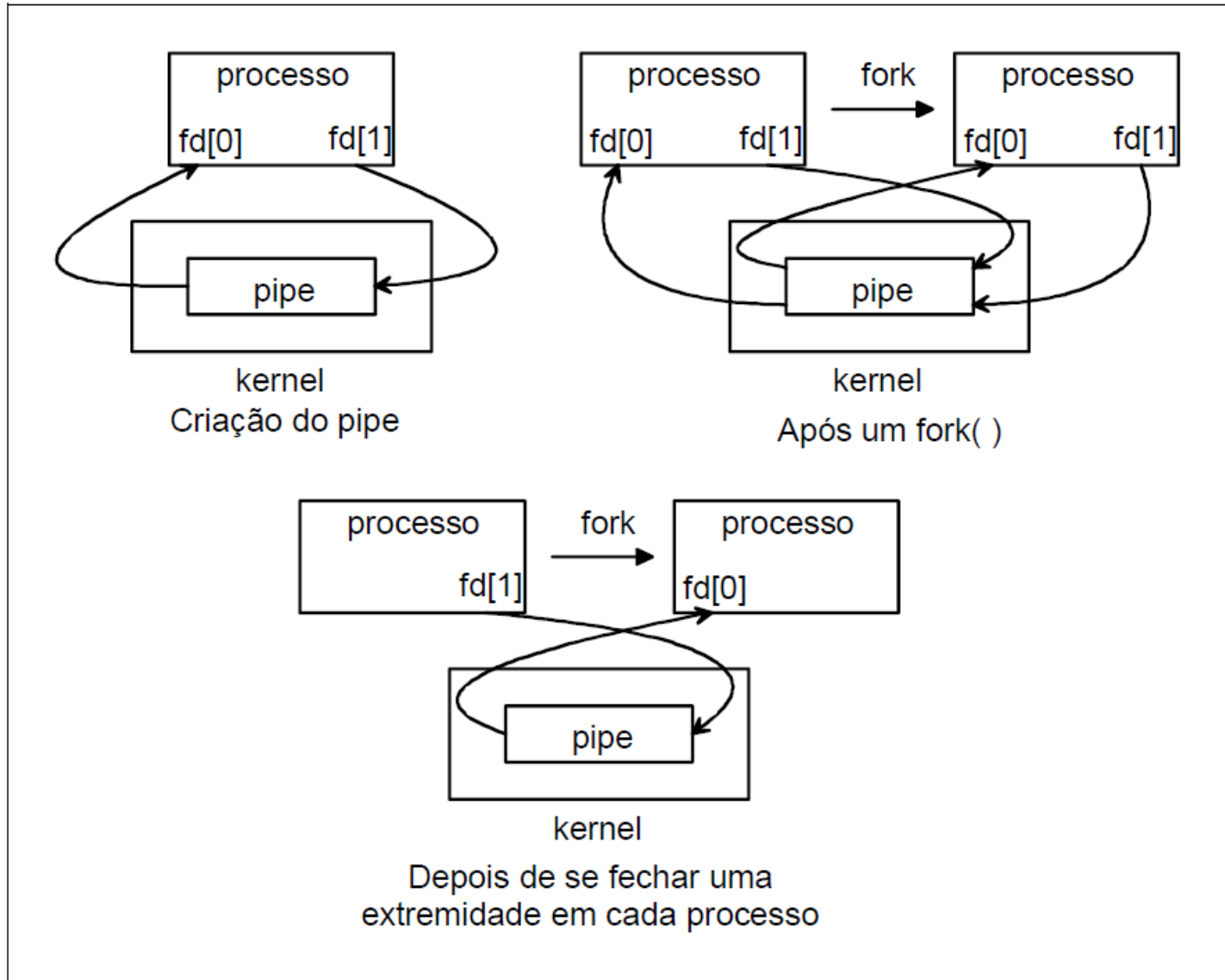
Pipes



Pipes

- Quando um dos lados está fechado e se tenta ler ou escrever:
 - Se for uma tentativa de leitura cujo lado emissor esteja fechado, o *read()* retorna 0
 - Se for uma tentativa de escrita num pipe cujo lado receptor esteja fechado, será gerado um sinal SIGPIPE e o *write()* retorna um erro.

Pipes

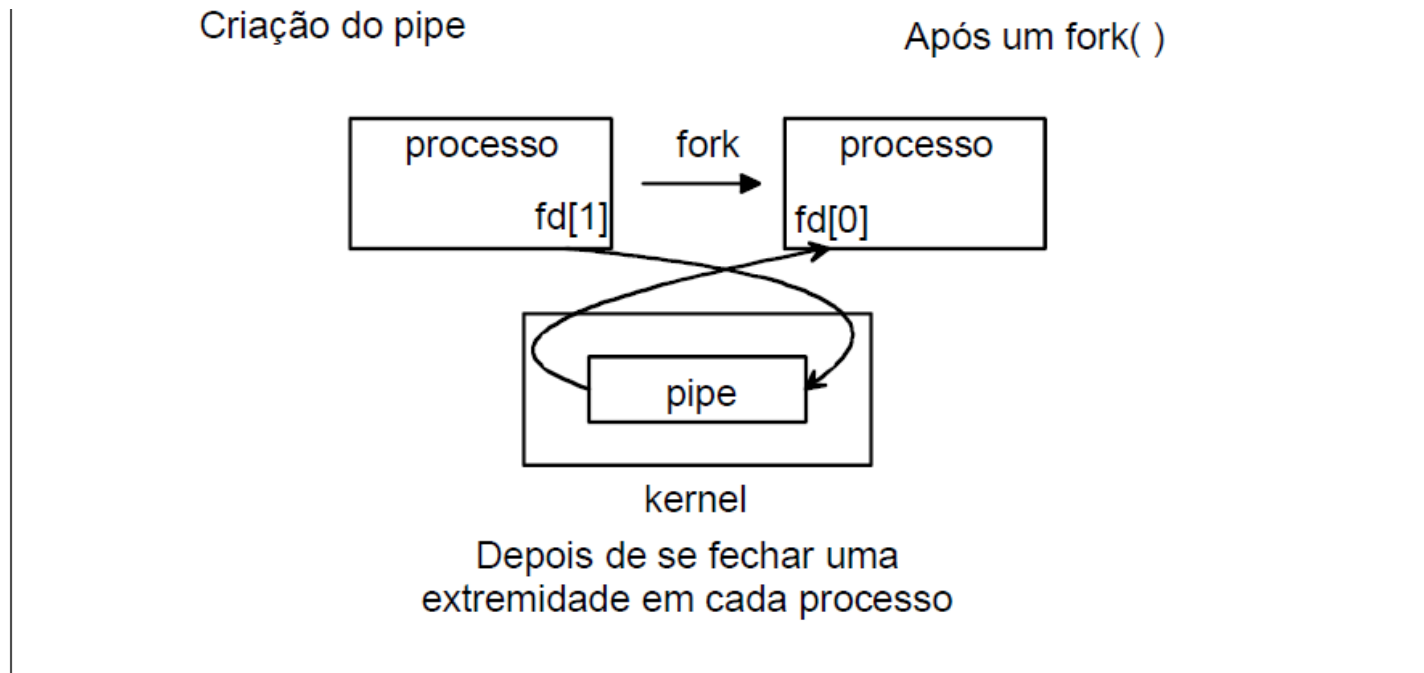


Pipes

Está é a criação de uma comunicação unidirecional.

Como criamos uma comunicação Bidirecional?

Criando outro pipe!



Exercício para fazer em casa!

- Utilizando comunicação entre processos, escreva um programa que realiza a soma dos N primeiros números naturais. O valor de N é passado como argumento do programa (`argc`, `argv`)
 - Cada soma deve ser realizada por um processo filho. Ou seja, se o valor de N for 3, deverão ser criados 3 filhos.
 - A primeira, soma será feita pelo filho, a próxima soma pelo filho do filho e assim por diante...
 - O filho **N** realiza a ultima soma e o pai escreve o resultado final.

Referências

- OLIVEIRA, Rômulo Silva de; CARISSIMI, Alexandre da Silva; TOSCANI, Simão Sirineo. **Sistemas operacionais**. 4. ed. Porto Alegre: Bookman, 2010. ISBN: 9788577805211.
 - **Capítulo 2**
- TANENBAUM, Andrew S.. **Sistemas operacionais modernos**. 3. ed. São Paulo: Prentice Hall, 2009. 653 p. ISBN: 9788576052371.
 - **Capítulo 2**
- SILBERCHATZ, A.; Galvin, P.; Gagne, G.; **Fundamentos de Sistemas Operacionais**, LTC, 2015. ISBN: 9788521629399
 - **Capítulo 3**

Próxima aula

- Exercícios em sala!
 - Valendo 1,5 pontos para a Unidade 1