

ANEXOS

Anexo 1 – Processos UNIX

Anexo 2 – Pipes UNIX

Anexo 3 – IPC UNIX

ANEXO 1

PROCESSOS NO UNIX

Processos são o coração do sistema UNIX. A grande maioria é gerada a partir da chamada de sistema **fork**. Quando executada com sucesso a chamada **fork** produz um processo filho que continua sua execução no ponto de sua invocação pelo processo pai.

Criação de Processo

- `fork (2)` [→]

```
#include <sys/types.h>
```

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

```
pid_t fork ( void ) ;
```

Retorno: 0 para o filho, PID do filho para o pai

```
/*
```

```
* geração de um processo filho
```

```
*/
```

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

```
#include <sys/types.h>
```

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

```
void
```

```
main ( void ) {
```

```
    if ( fork() == 0 )
```

```
        printf(“Processo FILHO\n”);
```

```
    else
```

```
        printf(“Processo PAI\n”);
```

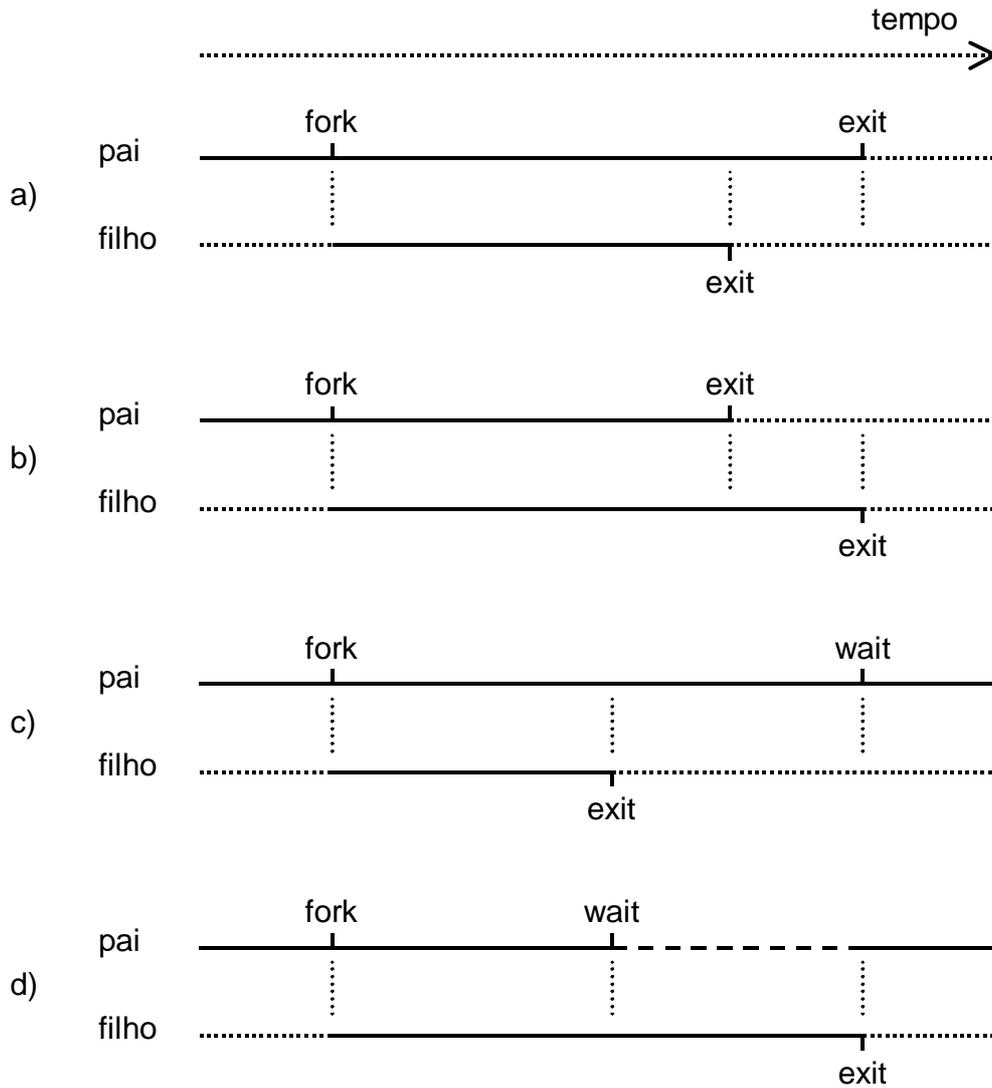
```
}
```

```
shell> a.out
```

- Diferenças entre pai e filho
 - Identificação (pid)
 - Identificação do pai (ppid)

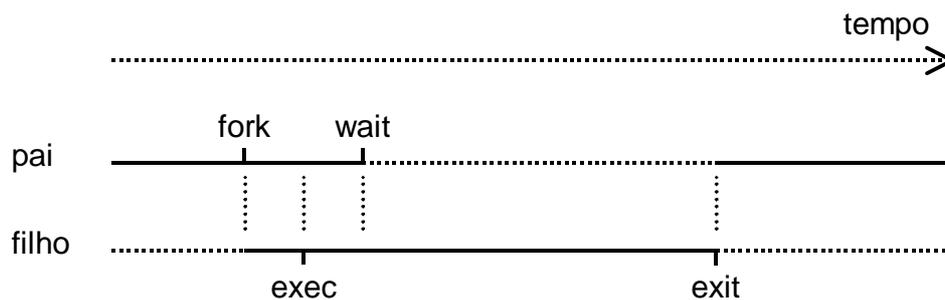
Sincronização entre processo pai e processo filho

- Término de processo: `exit (3) [->]` ou `_exit (2) [->]`
- Espera pelo término de processo filho: `wait (2) [->]`



Execução de programas por um Processo

Processos geram filhos por várias razões. No UNIX, existem vários processos com vida longa que executam continuamente em *background* e fornecem serviços sob demanda (daemon – lpd, inetd, routed). O mais comum é gerar um filho para executar algum outro código diferente do pai. A chamada *exec* permite transformar o processo filho através da mudança do código a ser executado. Ex: shell> cat file.txt > file2.txt.



- Chamadas de sistema exec

Existem 6 chamadas diferentes para o exec. A diferença básica está na forma como os argumentos são colocados (l:lista, v:vetor de pointers) e se o programador constrói o ambiente (envp) do processo ou se será usado o PATH corrente para buscar o arquivo executável.

```
#include <sys/types.h>
```

```
int execl ( const char *pathname, const char *arg0, ..., NULL ) ;
```

```
int execv ( const char *pathname, char *const argv[]);
```

```
int execl ( const char *pathname, const char *arg0, ..., NULL,  
           char *const envp[] );
```

```
int exeve ( const char *pathname, char *const argv[], char *const envp[] )
```

```
int execlp ( const char *filename, const char *arg0, ..., NULL ) ;
```

```
int execvp ( const char *filename, char *const argv[]);
```

Retorno: 0 para OK, -1 erro

- `execlp (2) [→]`

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main (int argc, char *argv[ ]) {

    if (argc > 1 ) {
        execlp("/bin/cat", "cat", argv[1], (char *) NULL);
        perror("falha no exec ");
        exit(1);
    }
    fprintf (stderr, "Uso: %s arquivo texto\n",*argv);
    exit (1);
}
```

```
shell> a.out test.txt
```

```
...
```

- `execvp (2) [→]`

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
main (int argc, char *argv[]) {
    execvp (argv[1], &argv[1]);
    perror ("falha no exec");
    exit (1);
}
```

```
shell> a.out cat test.txt
```

Término de Processo

Um processo pode ter seu término de forma normal ou ser sinalizado por outro processo e terminar de forma anormal.

- Normal
 - Execução completa do procedimento principal (main)
 - Execução de return no procedimento principal (main)
 - Chamada de exit (3) [->] ou de _exit (2) [->]
- Anormal
 - Sinal não tratado

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

```
int exit (int status);
```

Retorno: não retorna

```
#include <stdio.h>
#include <fcntl.h>
extern int errno;
main () {
    if (open ("/etc/passwd", O_RDWR) == -1) {
        perror ("main");
        exit (1);
    }
    printf ("Tenho superpoderes - Posso alterar /etc/passwd.\n");
    exit (0);
}
```

```
shell> a.out
main: Permission denied
```

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

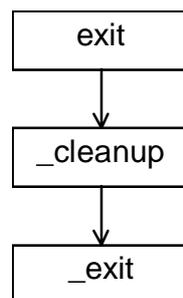
```
int _exit (int status);
```

Retorno: não retorna

```
#include <stdio.h>
#include <fcntl.h>
extern int errno;
main () {
    if (open ("/etc/passwd", O_RDWR) == -1) {
        perror ("main");
        _exit (1);
    }
    printf ("Tenho superpoderes - Posso alterar /etc/passwd.\n");
    _exit (0);
}
```

```
shell> a.out
main: Permission denied
```

- exit versus _exit



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

```
int atexit ( void (*func) (void) );
```

Retorno: não retorna

```
#include <stdio.h>
#include <stdlib.h>
int main( void ) {
    void f1(void), f2(void), f3(void);
    atexit(f1);
    atexit(f2);
    atexit(f3);
    printf( "pronto para terminar\n");
    exit(0);
}
void f1(void){ printf("executando f1\n"); }
void f2(void) { printf("executando f2\n"); }
void f3(void) { printf("executando f3\n"); }
```

```
shell> a.out
pronto para terminar
executando f1
executando f2
executando f3
```

- Funções executadas pelo sistema quando um processo chama `_exit`
 - O pai é notificado através do sinal SIGCHLD
 - O código de término é retornado ao pai (se ele quiser recebê-lo)
 - Todos os filhos recebem `ppid = 1` (init)
 - ...

Espera pelo término de processo filho

Freqüentemente o processo pai precisa sincronizar suas ações esperando até que o filho termine ou pare. A chamada *wait* permite a suspensão do processo pai até que isto aconteça.

- `wait (2) [→]`

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

pid_t wait ( int *stat_loc );
```

Retorno: identificação do processo filho

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
main () {
    int
    ipf, /* Identificacao do Processo Filho */
    estado = 0;
    if (fork () != 0) {
        sleep (1);
        ipf = wait (&estado);
        printf ("ipf: identificacao do processo filho\n");
        printf ("ct: codigo de termino\n\n");
        printf ("ipf: %6d, ct: %08x\n", ipf, estado);
    }
    else {
        sleep (2);
        exit (1);
    }
}
shell> a.out
```

```
ipf: identificacao do processo filho
ct: codigo de termino
```

```
ipf: 156, ct: 00000100
```

Processo chamador tem pelo menos um filho?

Não: Retorna `errno == 10 (ECHILD)`

Sim: Pelo menos um dos filhos é zumbi?

Sim: Retorna `stat_loc == código de término (figura 1)`

Não: Bloqueia processo chamador até término de um filho

Término normal: `stat_loc == código de término (figura 1)`

Término anormal: `stat_loc == sinal não tratado (figura 2)`



figura 1



figura 2

Algumas variações da chamada `wait` são fornecidas : `waitpid`, `wait3` e `wait4`

Comunicando processos através de Sinais

Uma forma de comunicar a ocorrência de eventos para os processos são os sinais. Os sinais são gerados quando o evento ocorre pela primeira vez e são considerados entregues quando o processo reage ao sinal. Os sinais são considerados como comunicação assíncrona.

- Origens
 - Hardware
 - SIGSEGV (segmentation fault)
 - SIGFPE (arithmetic exception)
 - ...
 - Núcleo

- SIGIO (
 - ...
- Processos
 - alarm (2) [->]
 - kill (2) [->]
- Usuário
 - kill (1)
 - control-c (SIGINT) (interrupt)
 - control-\ (SIGQUIT) (quit)
 - ...
- Espera
 - pause (2) [->]
- Respostas
 - Padrão (conforme tabela)
 - *exit* - O processo é encerrado
 - *core & exit* - Uma "imagem do processo" é gerada e ele é encerrado
 - *stop* - O processo é suspenso
 - *ignore* - O sinal é ignorado
 - Ignora (signal (2) [->])
 - Trata (signal (2) [->])
- Lista de sinais

Obs: man 5 signal e <sys/signal.h>

Obs: Alguns sinais não podem ser tratados nem ignorados (*)

Nome	Nº	Resposta padrão	Evento
SIGALARM	14	exit	relógio
SIGFPE	8	core & exit	exceção aritmética
SIGINT	2	Exit	interrupção
SIGKILL (*)	9	Exit	término
SIGPIPE	13	Exit	rompimento de <i>pipe</i>
SIGQUIT	3	core & exit	término
SIGSEGV	11	core & exit	endereçamento inválido

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

```
unsigned int alarm ( unsigned int sec );
```

Retorno: intervalo de tempo residual

```
#include <unistd.h>
main () {
    int
        estado = 0;
    if (fork () != 0) { /* Pai */
        wait (&estado);
        printf ("[Pai] sinal == %08x\n", estado);
    }
    else { /* Filho */
        alarm (2);
        pause ();
    }
}
```

```
shell$ a.out
[Pai] sinal == 0000000e
shell$
```

```
#include <sys/types.h>
```

```
#include <signal.h>
```

```
int kill ( pid_t pid, int sig );
```

Retorno: 0 para OK e -1 para erro

pid	processos destinatários
> 0	processo identificado por pid
0	processos pertencentes ao mesmo grupo do emissor
< -1	processos pertencentes ao grupo abs (-pid)

```
#include <sys/types.h>
```

```
#include <signal.h>
```

```
main () {
```

```
    int
```

```
    ipf, /* identificacao do processo filho */
```

```
    estado = 0;
```

```
    if ((ipf = fork ()) != 0) { /* Pai */
```

```
        kill (ipf, SIGKILL);
```

```
        wait (&estado);
```

```
        printf ("[Pai] sinal == %08x\n", estado);
```

```
    }
```

```
    else /* Filho */
```

```
        sleep (2);
```

```
}
```

```
shell$ a.out
```

```
[Pai] sinal == 00000009
```

```
shell$
```

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

```
int pause ( void );
```

Retorno: -1 se o sinal não termina o processo

```
#include <unistd.h>
#include <signal.h>
main () {
    int
        ipf, /* identificacao do processo filho */
        estado = 0;
    if ((ipf = fork ()) != 0) { /* Pai */
        sleep (2);
        kill (ipf, SIGKILL);
        wait (&estado);
        printf ("[Pai] sinal == %08x\n", estado);
    }
    else /* Filho */
        pause ();
}
```

```
shell$ a.out
[Pai] sinal == 00000009
shell$
```

```
#include <signal.h>
```

```
oldfunction = signal ( sig , function ) ;
```

Retorno: função anterior

```
#include <unistd.h>
#include <signal.h>
void sigint () { }
main () {
```

```

int
    ipf, /* identificacao do processo filho */
    estado = 0;
if ((ipf = fork ()) != 0) { /* Pai */
    sleep (2);
    kill (ipf, SIGINT);
    sleep (2);
    kill (ipf, SIGINT);
    wait (&estado);
    printf ("[Pai] sinal == %08x\n", estado);
}
else { /* Filho */
    signal (SIGINT, sigint);
    pause ();
    printf ("[Filho] primeiro SIGINT: recebido e tratado\n");
    pause (); /* A instrucao abaixo nao deve ser executada */
    printf ("[Filho] segundo SIGINT: recebido e nao tratado\n");
}
}

```

```

shell$ a.out
[Filho] primeiro SIGINT: recebido e tratado
[Pai] sinal == 00000002
shell$

```

ANEXO 2

COMUNICAÇÃO ENTRE PROCESSOS

Comunicando Processos via pipes

Outra forma de comunicação entre processos (com troca de informações) é através dos **pipes**. Um pipe pode ser visto como um arquivo especial que pode armazenar uma quantidade de dados limitada com política FIFO (First-in-First-out). Na maioria dos sistemas os pipes são limitados a 10 blocos lógicos de 512bytes. Como regra geral, um processo escreve em um pipe (como se fosse um arquivo) enquanto outro processo lê do pipe. O funcionamento de um pipe é análogo a uma esteira composta por 10 blocos com dados que são continuamente preenchidos (escritos) e esvaziados (lidos). Os dados são escritos numa ponta do pipe e lidos na outra. O sistema prove o sincronismo entre os processos leitores e escritores. Se um processo tenta escrever em um pipe cheio, o sistema bloqueia o processo até que o pipe possa receber dados. Da mesma forma, se um processo tenta ler em um pipe vazio o processo será bloqueado até que o dado esteja disponível. Quando um pipe foi aberto para leitura por um processo mas não foi aberto para escrita por outro processo o primeiro processo será bloqueado.

A escrita dos dados em um pipe é feita usando a chamada de sistema **write**.

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

```
ssize_t write ( int fildes, const void *buf, size_t nbyte);
```

Retorno: número de bytes escritos

Algumas diferenças em relação a escrita (write) em arquivos:

- cada write é sempre adicionado no final do pipe
- write do tamanho do pipe ou menor são não interrompidos
- um write pode bloquear o processo e retardar a escrita (flags em sys/fcntl.h)
- um write em um pipe que não esta aberto para leitura causa um SIGPIPE.

A leitura dos dados é feita usando a chamada de sistema **read**.

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

```
ssize_t read ( int fildes, const void *buf, size_t nbyte);
```

Retorno: número de bytes escritos

- se o valor retornado é 0 então é o fim do pipe
- todas as leituras são iniciadas da posição corrente (sem seek)

- a chamada read pode bloquear o processo

Os pipes podem ser divididos em duas categorias pipes sem nome (unnamed) e pipes com nome (named). Pipes sem nome só podem ser usados por processos relacionados (pai/filho) e existem enquanto os processos o utilizam. Os pipes com nome existem como entradas de diretório (com restrições de acesso) e portanto podem ser usados por processos não relacionados.

Pipes sem nome:

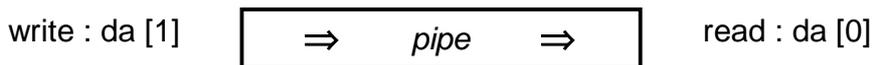
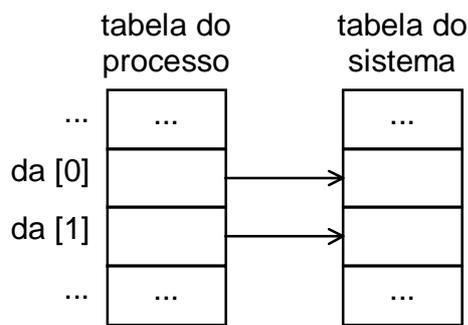
```
#include <unistd.h>

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

Retorno: 0 para OK e -1 para erro

- pipe (2) [→]
- Descritores:

```
...
pipe (da);
...
```

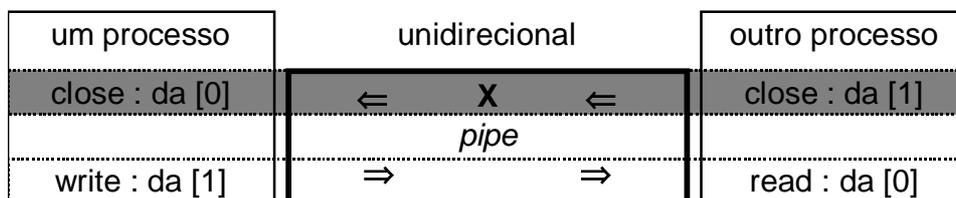
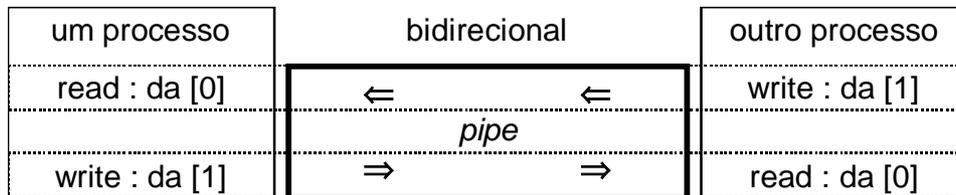


- Sentido:

```
...
pipe (da);
```

```
fork ();
```

```
...
```



Exemplo:

```
int main(void) {
    int    n,fd[2];
    pid_t  pid;
    char   line[MAXLINE];

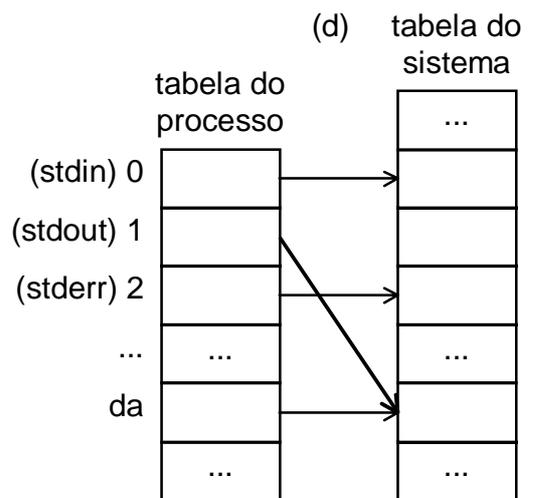
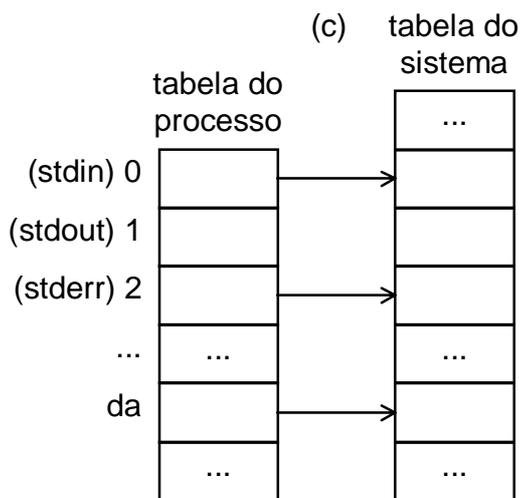
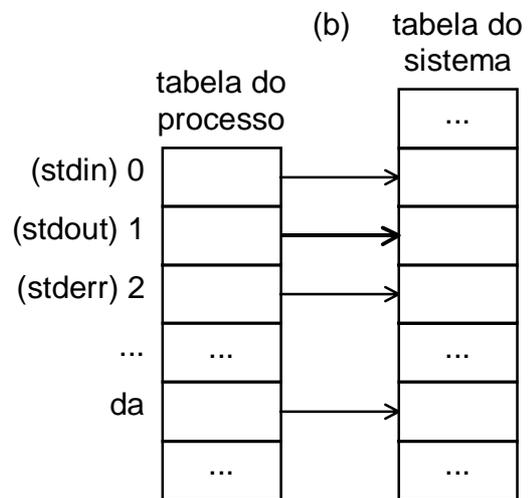
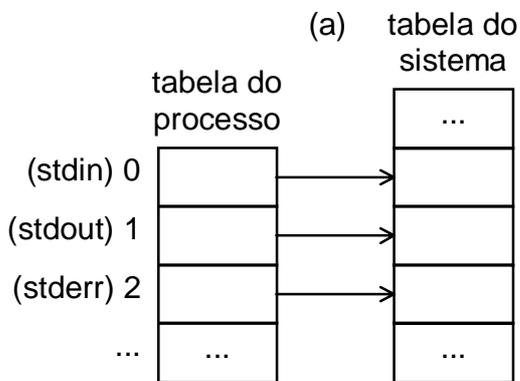
    if (pipe(fd) < 0)
        perror("pipe");
    if ( (pid=fork()) < 0)
        perror("fork");
    else if (pid > 0) {
        close(fd[0]);
        write(fd[1],"hello world\n",12);
    } else {
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

Pipes são bastante utilizados na linha de comando para conectar programas de forma que a saída padrão de um programa torna-se a entrada padrão do outro.

```
Shell> last | sort
```

Neste caso é preciso associar entrada e saída padrão com o pipe. As chamadas dup e dup 2 permitem isto.

```
... (a)
da = open ( ... ); (b)
close (1); (c)
dup (da); (d)
```



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

```
int dup ( int fd );
```

Retorno: o menor descritor de arquivo disponível

- Exemplo:

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

```
#include <fcntl.h>
```

```
main () {
```

```
    int
```

```
    da; /* descritor de arquivo */
```

```
    write (1, "Uma linha com 30 caracteres.\n", 30);
```

```
    da = open ("dup_teste", O_CREAT | O_WRONLY);
```

```
    close (1);
```

```
    dup (da);
```

```
    close (da);
```

```
    write (1, "Uma linha com 30 caracteres.\n", 30);
```

```
}
```

```
shell$ a.out
```

```
Uma linha com 30 caracteres.
```

```
shell$ cat dup_teste
```

```
Uma linha com 30 caracteres.
```

```
shell$
```

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

```
int dup2 ( int fd, int fd2 );
```

Retorno: o menor descritor de arquivo disponível

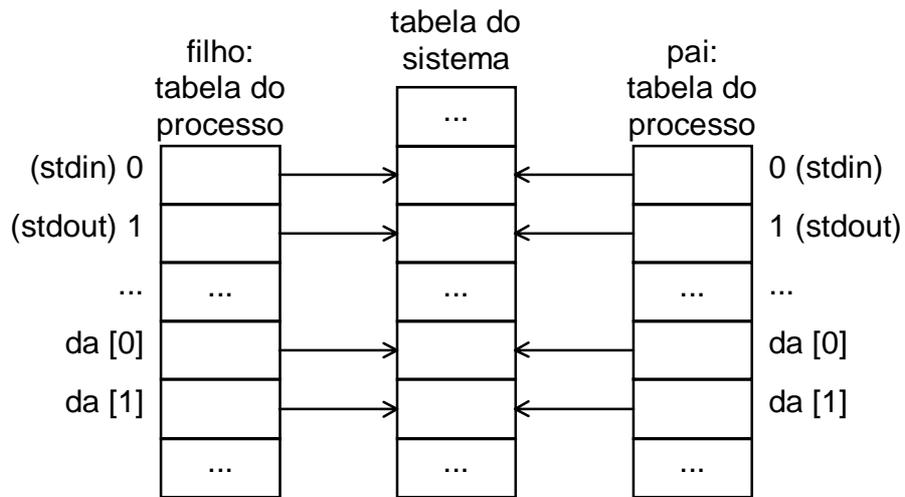
Exemplo : associação de pipe com entrada e saída padrão

```
...
```

```
    pipe (da);
```

```
    fork ();
```

```
...
```



/ filho */*

```

...
close (1);
dup (da [1]);
close (da [0]);
close (da [1]);
...

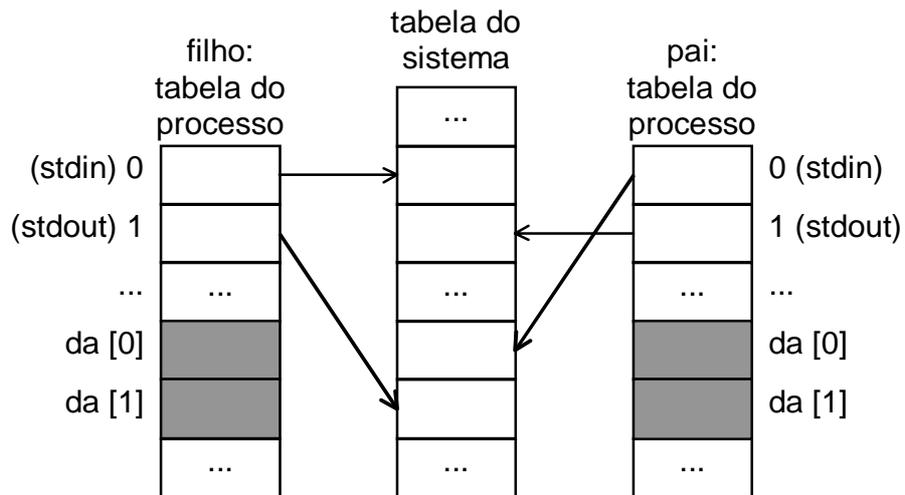
```

/ pai */*

```

...
close (0);
dup (da [0]);
close (da [0]);
close (da [1]);
...

```



Os passos para comunicação com pipes sem nome:

- criação dos pipes
- geração dos processos filhos
- fecha/duplica descritores para fechar as pontas do pipe
- fecha pontas não necessárias
- realiza comunicação
- fecha descritores

PIPES com nome

Este tipo de pipe é suportado pelo sistema e fornecido ao usuário através da chamada *mknod*. Esta chamada cria um arquivo (referenciado por *path*) cujo tipo (FIFO, character ou block especial, directory ou comum) e as permissões de acesso são determinados por *mode*. Neste caso, processos não relacionados podem se comunicar entre si. Existe uma função específica *mkfifo* que cria um pipe com nome do tipo FIFO.

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
int mknod ( const char *path, mode_t mode, dev_t dev );
```

```
int mkfifo ( const char *path, mode_t mode);
```

Retorno: 0 para OK e -1 para erro

ANEXO 3

IPC (*interprocess communication*)

Considerando que PIPES e SIGNALS são formas de comunicação restritas, aumenta-se a flexibilidade para comunicação entre processos com outros mecanismos.

- Recursos para *IPC* (Unix System V)
 - Filas de mensagens: As informações a serem comunicadas são colocadas em uma estrutura de mensagens predefinida. O processo gerador da mensagem especifica seu tipo e coloca a mensagem em uma fila de mensagens mantida pelo sistema. Os processos que acessam a fila podem selecionar as mensagens para leitura pelo tipo numa política FIFO.
 - Semáforos: Estruturas de dados implementadas pelo sistema para comunicar pequenas quantidades de dados entre processos. Frequentemente utilizado para sincronização entre processos.
 - Memória compartilhada: As informações são comunicadas pelo acesso espaço de dados compartilhado por processos. É o método mais rápido de comunicação entre processos. Os processos acessam diretamente segmento de memória compartilhada. Para sincronizar o acesso são usados semáforos.

Da mesma forma que os arquivos todos estes mecanismos devem ser gerados antes de utilizados, sendo que cada um tem um criador, dono e permissões de acesso. Estes mecanismos existem e mantém seus conteúdos mesmo depois do processo criador terminar. A remoção pode ser feita pelo seu dono através de chamada de sistema ou por comando de usuário (**iperm**).

- Chamadas de sistema

	Filas de mensagens	Semáforos	Memória compartilhada
criação/acesso	msgget	semget	shmget
controle	msgctl	semctl	shmctl
operação	msgrcv msgsnd	semop	shmat shmdt

As chamadas do tipo **get** são usadas para alocação de um mecanismo (gerando estrutura associada) ou para ganhar acesso a um mecanismo existente. Quando um novo mecanismo é criado, o usuário deve especificar as permissões de acesso do mesmo. Assim como uma chamada **open** a chamada *get* retorna um valor inteiro que é o identificador do mecanismo (IPC identifier). Existem dois argumentos comuns para cada das 3 chamadas *get*. Cada chamada *get* tem um argumento conhecido como valor *chave*, usado pela chamada para gerar o identificador do mecanismo IPC (relacionamento 1-para-1). A padronização da produção de *chaves* é feita pela função de biblioteca **ftok**.

- Autorização de acesso (<sys/ipc.h>)

```
struct ipc_perm
{
    key_t key;
    ushort uid;           /* owner euid and egid */
    ushort gid;
    ushort cuid;         /* creator euid and egid */
    ushort cgid;
    ushort mode;         /* access modes */
    ushort seq;          /* sequence number */
};
```

- Chave para as chamadas de sistema ...get
 - [ftok \(3\) \[→\]](#)

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok ( char *path , int id ) ;
```

Retorno: chave para get

- Exemplo:

```
#include <sys/ipc.h>
main (void) {
    key_t chf; /* chave de acesso a recurso ipc */
    chf = ftok (".", 'a');
    printf ("Chave == %08x\n", chf);
    exit (0);
}
```

shell\$ a.out

Chave == 6102d02f
shell\$

- IPC_PRIVATE

O valor *chave* para a chamada *get* também pode ser uma constante definida IPC_PRIVATE, dizendo para a chamada *get* para criar um mecanismo IPC com um identificador único que será compartilhado por processos relacionados (pai/filho ou filho/filho) ou em relações cliente-servidor.

O segundo argumento comum é o flag da mensagem, um valor inteiro, que é usado para ajustar as permissões de acesso no momento da criação do mecanismo IPC. As permissões são de escrita (msgsnd, msgctl, semop, semctl, shmat, shmctl) e leitura (msgrcv, msgctl, semop, semctl, shmat, shmctl). Junto com as permissões de acesso (OR) especifica-se ações na criação. Por exemplo, IPC_CREAT diz a chamada *get* para criar um mecanismo se ele não existe, se ele já existe e foi criado sem a utilização de IPC_PRIVATE, retorna o identificador. Outra ação, IPC_EXCL, assegura a um processo se ele é o criador do mecanismo ou se ele esta apenas ganhando acesso.

Mensagens

Criação de fila de mensagens

A criação de uma fila de mensagens é feita através do uso da chamada **msgget** que retorna um numero inteiro positivo, identificador da fila de mensagens, se completada com sucesso. Este identificador é usado nas demais chamadas para o mecanismo IPC fila de mensagens.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget ( key_t key, int msgflg ) ;
Retorno: identificador da fila associada a chave
```

- Exemplo:

```
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
main (void) {
    int ifm; /* identificador da fila de mensagens */
    key_t chave;
```

```

chave = ftok (".", 'F');
if ((ifm = msgget (chave, IPC_CREAT | 0660)) == -1) {
    perror ("msgget");
    exit (1);
}
printf ("Identificador da fila == %d\n", ifm);
msgctl (ifm, IPC_RMID, (struct msqid_ds *) 0); /* remocao da fila */
exit (0);
}

```

```

shell$ a.out
Identificador da fila == 128
shell$

```

- Estruturas de dados das Filas de mensagens (<sys/msg.h>)

- Descritor

/* one msqid structure for each queue on the system */

```

struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first; /* first message on queue */
    struct msg *msg_last; /* last message in queue */
    time_t msg_stime; /* last msgsnd time */
    time_t msg_rtime; /* last msgrcv time */
    time_t msg_ctime; /* last change time */
    struct wait_queue *wwait;
    struct wait_queue *rwait;
    ushort msg_cbytes; /* current number of bytes on queue */
    ushort msg_qnum; /* number of messages in queue */
    ushort msg_qbytes; /* max number of bytes on queue */
    ushort msg_lspid; /* pid of last msgsnd */
    ushort msg_lrpid; /* last receive pid */
};

```

- Fila de mensagens

/* one msg structure for each message */

```

struct msg {
    struct msg *msg_next; /* next message on queue */
    long msg_type;
};

```


Operações de controle

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl ( int msqid, int cmd, struct msqid_ds *buf );
Retorno: 0 para OK e -1 para erro
```

- Exemplo:

```
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
main (void) {
    int ifm; /* identificador da fila de mensagens */
    key_t chave;
    struct msqid_ds dfm; /* descritor da fila de mensagens */
    chave = ftok (".", 'M');
    if ((ifm = msgget (chave, IPC_CREAT | 0660)) == -1) {
        perror ("msgget");
        exit (1);
    }
    msgctl (ifm, IPC_STAT, &dfm);
    printf ("Sobre a fila de mensagens %08x:\n", chave);
    printf (" Identificacao do criador == %d\n", dfm.msg_perm.cuid);
    printf (" Identificacao do proprietario == %d\n", dfm.msg_perm.uid);
    printf (" Autorizacoes originais (octal) == %04o\n",
dfm.msg_perm.mode);
    dfm.msg_perm.mode = 0600;
    msgctl (ifm, IPC_SET, &dfm);
    printf (" Autorizacoes alteradas (octal) == %04o\n",
dfm.msg_perm.mode);
    msgctl (ifm, IPC_RMID, (struct msqid_ds *) 0);
    exit (0);
}
```

```

shell$ a.out
Sobre a fila de mensagens 4d02d02f:
  Identificacao do criador == 501
  Identificacao do proprietario == 501
  Autorizacoes originais (octal) == 0660
  Autorizacoes alteradas (octal) == 0600
shell$

```

IPC_STAT: retorna informações da fila de mensagens (dono, modos de acesso, etc)

IPC_SET: modifica características da fila de mensagens (modos de acesso)

IPC_RMID: remove todas as estruturas associadas a fila de mensagens

Envio / recepção de mensagens

A chamada **msgsnd** tem 4 argumentos : msqid(descriptor da fila de mensagens), msgp(endereço da mensagem), msgsz(tamanho da mensagem), msgflg (*flags*). As flags servem para indicar ao sistema se o processo deve ser bloqueado ou não (IPC_NOWAIT) em situações onde os limites do sistema foram alcançados (espaço na fila).

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd ( int msqid, void *msgp, size_t msgsz, int msgflg ) ;
                Retorno: 0 para OK e -1 para erro

```

```

/* Emissor (mssor) */
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
main (int argc, char *argv []) {
  int
    ifm, /* identificador da fila de mensagens */
    i;
  struct {
    long int tipo;
    char linha [76];
  } mensagem;
  ifm = atoi (argv [1]);

```

```

mensagem.tipo = 1;
for (i = 1; i <= 3; i++) {
    gets (mensagem.linha);
    msgsnd (ifm, &mensagem, sizeof (mensagem), 0);
}
exit (0);
}

```

A chamada de sistema **msgrcv** tem 5 argumentos: *msqid* (descritor da fila de mensagens), *msgp* (endereço da mensagem), *msgsz* (tamanho da mensagem), *msgtyp* (tipo da mensagem), *msgflg* (*flags*). O valor do tipo da mensagem pode indicar algumas ações para a execução da chamada: 0 – recupera a primeira mensagem de qualquer tipo; >0 – recupera a primeira mensagem do tipo especificado; <0 – recupera a primeira mensagem do tipo <= ao valor de *msgtyp*.

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

```

```

int msgrcv ( int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg ) ;

```

Retorno: 0 para OK e -1 para erro

```

/* Receptor (rcptor) */
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
main (int argc, char *argv []) {
    int
        ifm, /* identificador da fila de mensagens */
        i;
    struct {
        long int tipo;
        char linha [76];
    } mensagem;
    ifm = atoi (argv [1]);
    for (i = 1; i <= 3; i++) {
        msgrcv (ifm, &mensagem, sizeof (mensagem), 1L, 0);
        puts (mensagem.linha);
    }
    exit (0);
}

```

- Exemplo:

```
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
main (void) {
    int
        ifm, /* identificador da fila de mensagens */
        estado = 0,
        i;
    char ifms [10]; /* ifm em ascii */
    if ((ifm = msgget (IPC_PRIVATE, 0600)) == -1) {
        perror ("msgget");
        exit (1);
    }
    sprintf (ifms, "%d", ifm);
    if (fork () != 0) { /* pai */
        if (fork () != 0) { /* ainda o pai */
            wait (&estado);
            wait (&estado);
            msgctl (ifm, IPC_RMID, (struct msqid_ds *) 0);
        }
        else /* segundo filho */
            execlp ("rcptor", "rcptor", ifms, (char *) 0);
    }
    else /* primeiro filho */
        execlp ("mssor", "mssor", ifms, (char *) 0);
}
```

Semáforos

Antes de da utilização os semáforos devem ser criados. A criação de um conjunto de semáforos gera uma única estrutura de dados usada pelo sistema para identificar e manipular os semáforos. A estrutura de dados é encontrada no arquivo <sys/sem.h>.

- Descritor

```
/* One semid structure for each set of semaphores on the system. */
struct semid_ds {
    struct ipc_perm sem_perm; /* permissions see ipc.h */
    time_t sem_otime;        /* last semop time */
    time_t sem_ctime;        /* last change time */
    struct sem *sem_base;    /* ptr to first semaphore in array */
    struct sem_queue *sem_pending; /* pending operations */
    struct sem_queue **sem_pending_last; /* last pending operation */
    struct sem_undo *undo;   /* undo requests on this array */
    ushort sem_nsems;        /* no. of semaphores in array */
};
```

Cada semáforo no conjunto tem uma estrutura associada que contém além do valor do semáforo e do PID da última operação, o número de processos esperando o semáforo ser incrementado e chegar a zero.

- Conjunto de semáforos

```
/* one semaphore structure for each semaphore on the system. */
struct sem {
    short semval;           /* current value */
    short sempid;          /* pid of last operation */
};
```

- Criação de /Acesso a um conjunto de semáforos

Para criar um semáforo, ou garantir acesso a um existente, a chamada **semget** é usada. Esta chamada requer 3 argumentos: *key* (chave para criação/acesso a um conjunto de semáforos), *nsems* (número de semáforos no conjunto) e *semflg* (autorização de acesso ao conjunto de semáforos). O argumento *nsems* é usado para alocar o array das estruturas *sem*. Os bits de mais baixa ordem do argumento *semflg* são usados para especificar as permissões de acesso, sendo que flags como `IPC_CREAT` e `IPC_EXCL` podem ser incluídas nesta especificação através de uma operação OU.

- semget (2) [→]

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

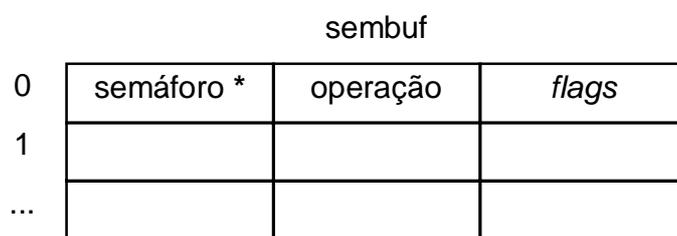
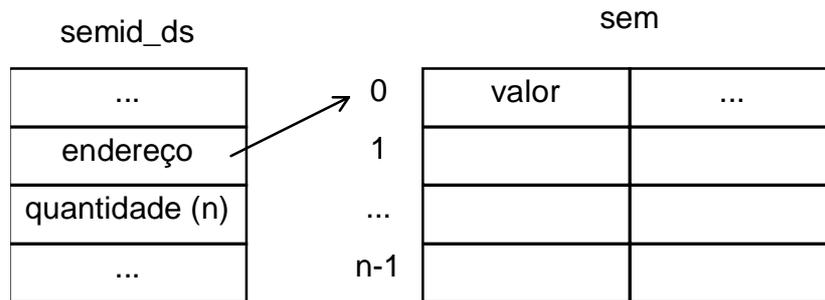
```
int semget ( key_t key, int nsems, int semflg ) ;
```

Retorno: identificador do semáforo

- Exemplo:

```
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/sem.h>
main (void) {
    int ics; /* identificador do conjunto de semaforos */
    key_t chave;
    chave = ftok (".", 'S');
    if ((ics = semget (chave, 2, IPC_CREAT | 0660)) == -1) {
        perror ("semget");
        exit (1);
    }
    printf ("Descritor do conjunto de semaforos == %d\n", ics);
    semctl (ics, 0, IPC_RMID, 0); /* remocao */
    exit (0);
}
```

```
shell$ a.out
Descritor do conjunto de semaforos == 128
shell$
```



(*) índice de sem

Operações de controle

A chamada de sistema **semctl** é usada pelo usuário para realizar operações de controle da estrutura de semáforos do sistema, no conjunto de semáforos ou individualmente. A chamada apresenta 4 argumentos. O primeiro, *semid*, é o identificador de um semáforo válido. O segundo, *semnum*, é o numero de semáforos no conjunto. O terceiro, *cmd*, valor inteiro que representa o comando de controle em <sys/ipc.h> ou <sys/sem.h>. O quarto, *arg*, representa os argumentos do comando e é definido por uma estrutura de dados do tipo *union*.

```

/* arg for semctl system calls. */
union semun {
    int val; /* value for SETVAL */
    struct semid_ds *buf; /* buffer for IPC_STAT & IPC_SET */
    ushort *array; /* array for GETALL & SETALL */
    struct seminfo * __buf; /* buffer for IPC_INFO */
    void * __pad;
};

```

- `semctl (2)` [→]

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl ( int semid, int semnum, int cmd, /*union semnum arg*/ );
           Retorno: 0 para OK e -1 para erro
```

```
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/sem.h>
main (void) {
    int ics; /* identificador do conjunto de semaforos */
    union semun arg;
    ushort ics_ini [2] = {1, 2};
    key_t chave;
    chave = ftok (".", 'S');
    if ((ics = semget (chave, 2, IPC_CREAT | 0660)) == -1) {
        perror ("semget");
        exit (1);
    }
    arg.array = ics_ini;
    semctl (ics, 0, SETALL, arg);
    ics_ini [0] = 0; ics_ini [1] = 0;
    semctl (ics, 0, GETALL, arg);
    printf ("Semaforo 0 == %d\n", ics_ini [0]);
    printf ("Semaforo 1 == %d\n", ics_ini [1]);
    semctl (ics, 0, IPC_RMID, 0);
    exit (0);
}
```

Operação

A chamada **semop** tem 3 argumentos e retorna um valor inteiro. O primeiro argumento, *semid*, é o identificador do semáforo. O segundo argumento, *sops*, é uma referência estrutura de dados *sembuf* que define as operações a serem realizadas no semáforo. O terceiro argumento, *nsops*, indica o numero de elementos no array de operações do semáforo.

```

/* semop system calls takes an array of these. */
struct sembuf {
    ushort sem_num;           /* semaphore index in array */
    short  sem_op;           /* semaphore operation */
    short  sem_flg;         /* operation flags */
};

```

- semop (2) [→]

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop ( int semid, struct sembuf *sops, size_t nsops ) ;
Retorno: 0 para OK e -1 para erro

```

```

#include <fcntl.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/sem.h>
main (void) {
    int
        ia, /* identificador de arquivo */
        ics, /* identificador do conjunto de semaforos */
        chave,
        estado = 0,
        n,
        i;
    union semun arg;
    ushort ics_ini [2] = {1, 0};
    struct sembuf
        P0 = {0, -1, SEM_UNDO},
        V0 = {0, +1, SEM_UNDO},
        P1 = {1, -1, SEM_UNDO},
        V1 = {1, +1, SEM_UNDO};
    chave = ftok (".", 'S');
    if ((ics = semget (chave, 2, IPC_CREAT | 0600)) == -1) {
        perror ("semget");
    }
}

```

```

    exit (1);
}
ia = open ("temp", O_CREAT | O_RDWR);
arg.array = ics_ini;
semctl (ics, 0, SETALL, arg);
if (fork () != 0) {                                     /* pai == produtor */
    for (i = 0; i < 5; i++) {
        semop (ics, &P0, 1);
        lseek (ia, 0, SEEK_SET);
        write (ia, &i, sizeof (i));
        /* sleep (1); */
        semop (ics, &V1, 1);
    };
    wait (&estado);
    semctl (ics, 0, IPC_RMID, 0);
    exit (0);
}
else {                                                 /* filho == consumidor */
    for (i = 0; i < 5; i++) {
        semop (ics, &P1, 1);
        lseek (ia, 0, SEEK_SET);
        read (ia, &n, sizeof (i));
        printf ("n == %d\n", n);
        semop (ics, &V0, 1);
    };
    exit (0);
}
}
}

```

Memória Compartilhada

Memória compartilhada permite que múltiplos processos compartilhem um espaço de memória. Este mecanismo pode ser considerado a forma mais rápida de comunicação entre processos. Entretanto, não é a forma mais fácil (problemas de coordenação). Em geral, um processo cria/aloca o segmento de memória compartilhada, sendo que o tamanho e as permissões de acesso são estabelecidas neste momento. Em seguida o processo anexa o segmento compartilhado fazendo com que o mesmo seja mapeado em seu atual espaço de dados. Quando necessário, o processo criador inicializa o segmento. Depois, outros processos podem ganhar acesso ao segmento e mapea-lo em seus espaços de dados. Após usar o segmento compartilhado o processo libera (detach) o mesmo. O processo criador é responsável pela remoção do segmento.

Criação de /Acesso a um segmento de memória

A chamada de sistema **shmget** é usada para criar um segmento de memória compartilhada e gera uma estrutura de dados associada (<sys/shm.h>). Os argumentos utilizados são 3: *key*(chave para criação/acesso a um segmento(memória)), *size*(Tamanho do segmento (em bytes)) e *shmflg* (autorização de acesso ao segmento). Um novo segmento compartilhado será criado se: o argumento *key* é a constante `IPC_PRIVATE`, o valor de *key* não esta associado a um identificador existente e a flag `IPC_CREAT` esta ativa ou o valor *key* não esta associado a um identificador existente e o conjunto `IPC_CREAT` e `IPC_EXCL` estão ativos. A chamada não permite a utilização da memória alocada apenas reserva a memória requisitada. Para utiliza-la o processo deve anexa-la (`attach`). Alguns limites da memória compartilhada são: tamanho máximo – 1.048.576 bytes, tamanho mínimo – 1 byte, número máximo de segmentos – 100 e número máximo de segmentos por processo – 6.

- `shmget (2) [→]`

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
int shmget ( key_t key, int size, int shmflg ) ;
```

Retorno: identificador do segmento

```
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
main (void) {
    int is; /* identificador do segmento de memoria */
```

```

key_t chave;
chave = ftok(".", 'M');
if ((is = shmget (chave, 1024, IPC_CREAT | 0600)) == -1) {
    perror ("shmget");
    exit (1);
}
printf ("Identificador do segmento == %d\n", is);
shmctl (is, IPC_RMID, 0); /* remocao */
exit (0);
}

```

```

shell$ a.out
Identificador di segmento == 128
shell$

```

A estrutura a seguir é gerada quando da criação de um segmento de memória compartilhada.

```

/* One shmid structure for each memory segment on the system. */
struct shmid_ds {
    struct ipc_perm shm_perm; /* operation perms */
    int shm_segsz; /* size of segment (bytes) */
    time_t shm_atime; /* last attach time */
    time_t shm_dtime; /* last detach time */
    time_t shm_ctime; /* last change time */
    unsigned short shm_cpid; /* pid of creator */
    unsigned short shm_lpid; /* pid of last operator */
    short shm_nattch; /* no. of current attaches */
    /* the following are private */
    unsigned short shm_npages; /* size of segment (pages) */
    unsigned long *shm_pages; /* array of ptrs to frames */
    struct vm_area_struct *attaches; /* descriptors for attaches */
};

```

Operações de controle

Através da chamada **shmctl** o usuário pode realizar operações de controle (IPC_STAT, IPC_SET, IPC_RMID, SHM_LOCK, SHM_UNLOCK) em um segmento existente. Os argumentos são: *shmid* (identificador do segmento), *cmd* (comando (IPC_STAT, IPC_SET, IPC_RMID, SHM_LOCK, SHM_UNLOCK) e *buf* (cópia local do

descriptor

do

segmento).

- shmctl (2) [→]

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
int shmctl ( int shmid, int cmd, struct shmid_ds *buf ) ;
```

Retorno: 0 para OK e -1 para erro

```
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
main (void) {
    int is; /* identificador do segmento */
    key_t chave;
    struct shmid_ds ds; /* descritor do segmento */
    chave = ftok (".", 'M');
    if ((is = shmget (chave, 1024, IPC_CREAT | 0660)) == -1) {
        perror ("shmget");
        exit (1);
    }
    shmctl (is, IPC_STAT, &ds);
    printf ("Autorizacoes originais (octal) == %04o\n",
ds.shm_perm.mode);
    ds.shm_perm.mode = 0600;
    shmctl (is, IPC_SET, &ds);
    printf ("Autorizacoes alteradas (octal) == %04o\n",
ds.shm_perm.mode);
    shmctl (is, IPC_RMID, (struct shmid_ds *) 0);
    exit (0);
}
```

Anexação / Desanexação

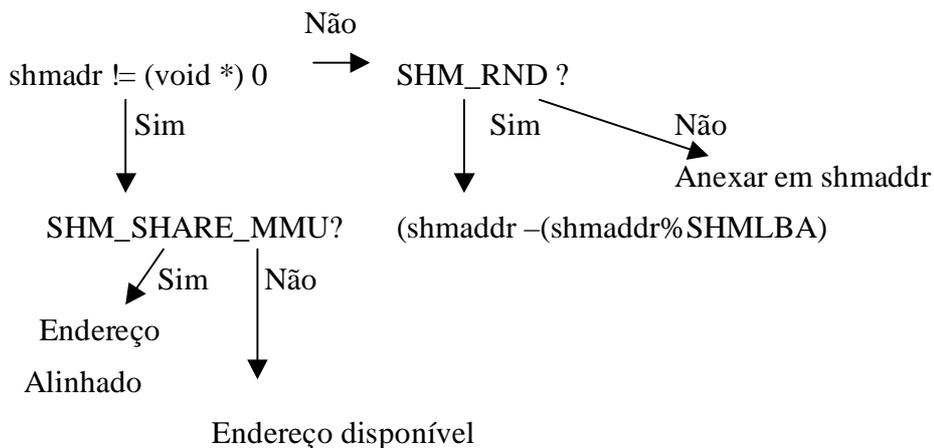
Existem duas chamadas de sistema para memória compartilhada. A primeira **shmat** é usada para anexar (attach) o segmento de memória compartilhada no segmento de dados do processo.

- **shmat (2) [→]**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat ( int shmids, void *shmaddr, int shmflg ) ;
                Retorno: endereço do segmento
```

Os argumentos desta chamada são 3. O primeiro *shmids* é um descritor de segmento compartilhado. O segundo *shmaddr* permite alguma flexibilidade ao usuário para atribuir o endereço do segmento. Um valor diferente de zero significa o endereço a ser usado para o segmento, enquanto que um valor igual a zero indica que o sistema determina o endereço. O terceiro argumento, *shmflg*, é usado para especificar as permissões de acesso para segmento. O valor do endereço junto com as permissões de acesso permitem o sistema determinar o endereço de anexação. Os segmentos são acessados para leitura e escrita, por default. Entretanto, é possível através da flag SHM_RDONLY determinar acesso apenas para leitura. Na determinação dos endereços, o sistema pode pegar o primeiro endereço disponível *alinhado* ou não(flag SHM_SHARE_MMU). Se o endereço usado é o informado na chamada o sistema pode escolher o próprio ou o mais próximo alinhado a página(flag SHM_RND). Veja o algoritmo a seguir.



A Segunda chamada, **shmdt**, é usada para desanexar o segmento do processo. Esta chamada possui apenas um argumento, *shmaddr*, que é o endereço do segmento.

- `shmdt (2) [→]`

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
int shmdt ( void *shmaddr );
```

Retorno: 0 para OK e -1 para erro

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
main (void) {
    int
        ics, /* identificador do conjunto de semaforos */
        is, /* identificador do segmento */
        chave_cs,
        chave_s,
        estado = 0,
        n,
        i;
    union semun arg;
    ushort ics_ini [2] = {1, 0};
    struct sembuf
        P0 = {0, -1, SEM_UNDO},
        V0 = {0, +1, SEM_UNDO},
        P1 = {1, -1, SEM_UNDO},
        V1 = {1, +1, SEM_UNDO};
    int *es; /* endereco do segmento */
    chave_cs = ftok (".", 'S');
    if ((ics = semget (chave_cs, 2, IPC_CREAT | 0600)) == -1) {
        perror ("semget");
    }
}
```

```

    exit (1);
}
chave_s = ftok (".", 'M');
arg.array = ics_ini;
semctl (ics, 0, SETALL, arg);
if ((is = shmget (chave_s, sizeof (int), IPC_CREAT | 0600)) == -1) {
    perror ("shmget");
    exit (1);
}

if (fork () != 0) { /* pai == produtor */
    es = (int *) shmat (is, 0, 0);
    for (i = 0; i < 5; i++) {
        semop (ics, &P0, 1);
        *es = i;
        printf ("produtor\n");
        /* sleep (1); */
        semop (ics, &V1, 1);
    };
    shmdt ((void *) es);
    wait (&estado);
    semctl (ics, 0, IPC_RMID, 0);
    shmctl (is, IPC_RMID, 0);
    exit (0);
}
else { /* filho == consumidor */
    es = (int *) shmat (is, 0, 0);
    for (i = 0; i < 5; i++) {
        semop (ics, &P1, 1);
        n = *es;
        printf ("consumidor == %d\n", n);
        semop (ics, &V0, 1);
    };
    shmdt ((void *) es);
    exit (0);
}
}
}

```