Cap. 2 - Processos no UNIX

- **objetivo:** com base nos conceitos de processo apresentados nas seções anteriores, detalharemos a estruturação de processos no UNIX (como são controlados, escalonados e se comunicam).

2.1 - Gerenciamento de Processos no UNIX

- Transições de Estado e Descritores de Processos.

2.2 - Escalonamento de Processos no UNIX

- Algoritmos de Escalonamento.

2.3 - Controle de Processos no UNIX

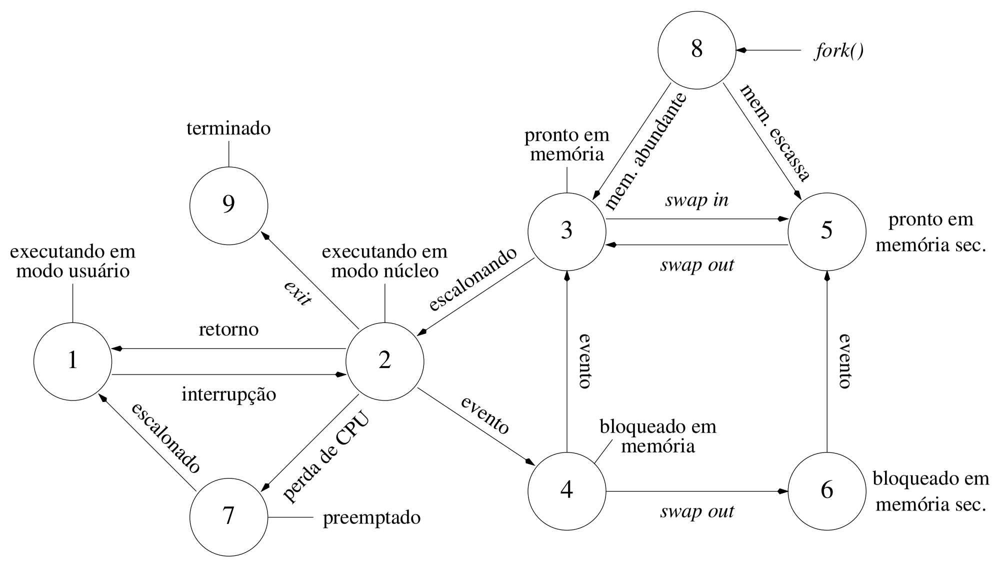
- Instanciação de Processos e Interrupção de Processos.

2.4 - Comunicação e Sincronização entre Processos no UNIX

- Pipes; Mensagens; Memória Compartilhada e Semáforos.

2.1 - Gerenciamento de Processos no UNIX

- No UNIX, um processo pode estar em um dos 09 estados possíveis:



- ... vejamos o que tipicamente ocorre a partir da criação de um processo:

... 2.1 - Gerenciamento de Processos no UNIX

- após executado o *fork*, são criados para o novo processo um campo na tabela de processos, sua área U e apontadores para sua *pregion*, ou seja, estado 8;
- ... dependendo da disponibilidade ou não de memória primária, o processo é movido, respectivamente, para os estados 3 ou 5;
- ... assumindo que o processo tenha ido para o estado 3, o escalonador seleciona então o processo para executar, movendo-o para o estado 2 (modo núcleo) onde a chamada *fork* será completada para este processo;
- ... a partir daí o processo é executado no modo usuário, processando suas instruções uma a uma e uma vez que seu tempo (*quantum*) tenha sido expirado, uma interrupção faz com que o processo retorne ao modo núcleo;
- ... terminado o tratamento da interrupção, o escalonador pode decidir alocar o processador a um outro a um outro processo, movendo-o para o estado 7;

... 2.1 - Gerenciamento de Processos no UNIX

- ... o estado 7 é equivalente ao 3, mas esta distinção é feita para enfatizar que um processo tem a CPU somente quando está apto a retornar para o modo usuário, diferente do estado 3 onde deve voltar ao modo núcleo para completar uma chamada;
- ... a eventual execução de uma chamada de sistema faz com que o processo abandone o modo usuário (estado 1) e continue no modo núcleo (estado 2);
- ... suponha que o processo requisitou uma operação de entrada/saída, assim o núcleo coloca o processo no estado 4 até que o mesmo seja notificado que a operação foi completada — envia sinal para o processo para acordá-lo;
- ... se ao ser desbloqueado o processo ainda residir em memória primária, o mesmo é posto no estado 3, aguardando a CPU para ser escalonado;
- ... entretanto, durante sua permanência no estado 4 o núcleo pode necessitar de espaço na memória primária o que acarreta a sua remoção da memória primária para a memória secundária (tipicamente disco) — estado 6;

... 2.1 - Gerenciamento de Processos no UNIX

- ... uma vez completada a operação de entrada/saída o processo no estado 6, passa para o estado 5, que por sua vez pode retornar ao estado 3 quando o *swapper* escolhê-lo para voltar à memória principal;

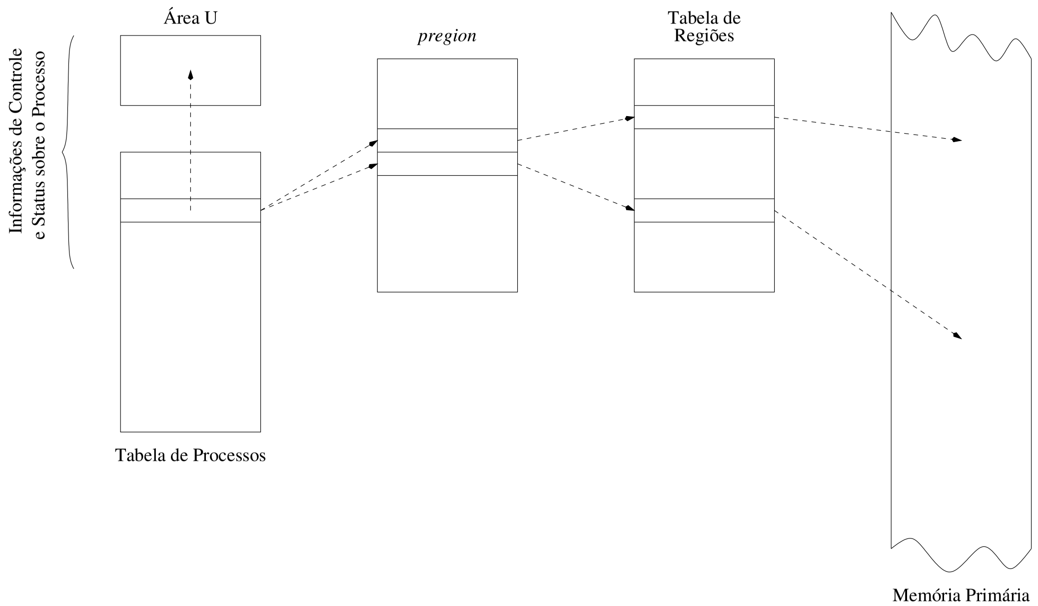
- ... no estado 3, quando o escalonador volta a atribuir a CPU ao processo, o mesmo atinge o estado 2 onde completa a chamada de sistema e volta ao estado 1, executando no modo usuário;

- ... quando um *exit* é executado pelo processo, o mesmo transita, via estado 2, para seu estado terminal (estado 9), permanecendo neste estado até que o processo pai seja notificado que pode estar sincronizado com o término do processo filho.

observaçao: uma vez criado, as transições de estado de um processo dependem exclusivamente do sistema operacional.

... 2.1 - Gerenciamento de Processos no UNIX

- vejamos agora que estruturas ou descritores estão disponíveis no sistema operacional UNIX para gerenciar e controlar os processos.



... 2.1 - Gerenciamento de Processos no UNIX

✰ **tabela de processos** - estrutura mantida pelo núcleo com os seguintes campos:

- estado do processo;
- localização da área U e do próprio processo;
- *process identification* ou pid;
- eventos aguardados pelo processo quando no estado bloqueado;
- parâmetros de escalonamento, utilizados para decidir quais processos transitarão entre estados de execução nos modos usuário e núcleo;
- sinais enviados para o processo, mas ainda não tratados;
- marcas de tempo como, por exemplo, tempo total de CPU, "despertadores" armados pelo processo, etc., além de recursos consumidos do núcleo que serão parâmetros considerados no cômputo da prioridade de escalonamento do processo;

... 2.1 - Gerenciamento de Processos no UNIX

✰ **área U** - de um processo armazena os seguintes campos:

- um ponteiro de volta ao respectivo índice da tabela de processos;
- privilégios que o processo dispõe, de acordo com o seu UID;
- tempos de execução nos modos usuários e núcleo;
- ponteiros para os gerenciadores de sinais definidos pelo processo;
- o terminal de *login* associado ao processo, caso exista;
- parâmetros de entrada/saída, tais como endereço de dados a serem transferidos, tamanho destes dados, destino, etc.
- o diretório corrente e o diretório raiz;
- descritores de arquivos abertos pelo processo;
- limites (tamanho máximo da pilha, arquivos, etc.);
- permissões (modos de acesso atribuídos durante a criação de arquivos).

2.2 - Escalonamento de Processos no UNIX

- **escalonamento de processos** - com o objetivo de compartilhar a CPU de forma equânime entre múltiplos usuários o escalonamento seque um algoritmo de meio termo entre o *round robin* e o de prioridades.

- ... para aplicações onde restrições temporais se fazem necessárias, este algoritmo como apresentaremos a seguir não é adequado.

✰ **algoritmo** - o núcleo divide os processos segundo 02 classes de prioridades:

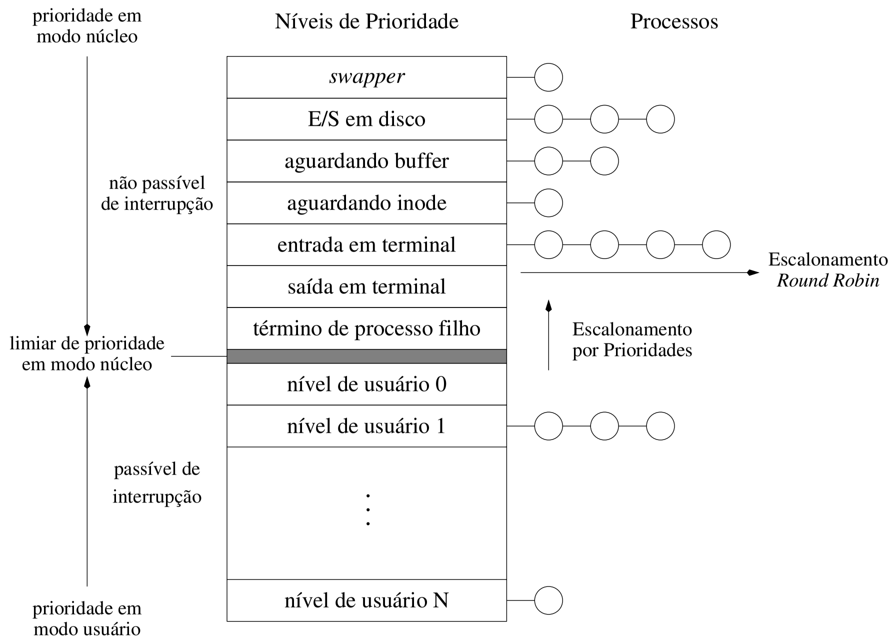
  - prioridades em do modo núcleo (altas), referentes a processos nos estados 4 ou 5;
  - prioridades em modo usuário (baixas), referentes a processos no estado 7.

✰ ... prioridades em modo núcleo são por sua vez subdivididas em 02 grupos:

  - constituído de processos bloqueados a espera de *swapping*, entrada/saída em disco, *buffers* de *cache* e *inodes*;
  - constituídos de processos bloqueados a espera de entrada de terminal, saída de terminal e terminação de processo filho.

... 2.2 - Escalonamento de Processos no UNIX

- ... a figura abaixo mostra as classes de prioridades adotadas:



... 2.2 - Escalonamento de Processos no UNIX

✫ o algoritmo de escalonamento é processado de modo que quando da ocorrência de uma interrupção de *hardware*, duas ações podem ser tomadas:

- caso a interrupção acorde um ou mais processos com prioridade de modo núcleo, aloque a CPU àquele de mais alta prioridade;
- caso contrário, se uma mudança de contexto se fizer necessária, aloque a CPU ao processo de mais alta prioridade dentre as de modo usuário.

- ... em existindo mais de um processo apto a executar no mesmo nível, a seleção se dá segundo a política do *round robin*.

✫ ... outras caracterísiticas importantes do algoritmo de escalonamento:

- prioridades no modo núcleo dependem apenas do evento aguardado pelo processo;
- prioridades de processos que transitam do modo núcleo para o modo usuário têm seus valores decrementados em relação ao valor inicial;
- em intervalos de tempos pré-definidos, as prioridades em modo usuário são recomputadas, de modo que processos que não usaram a CPU tenham a sua vez.

2.3 - Controle de Processos no UNIX

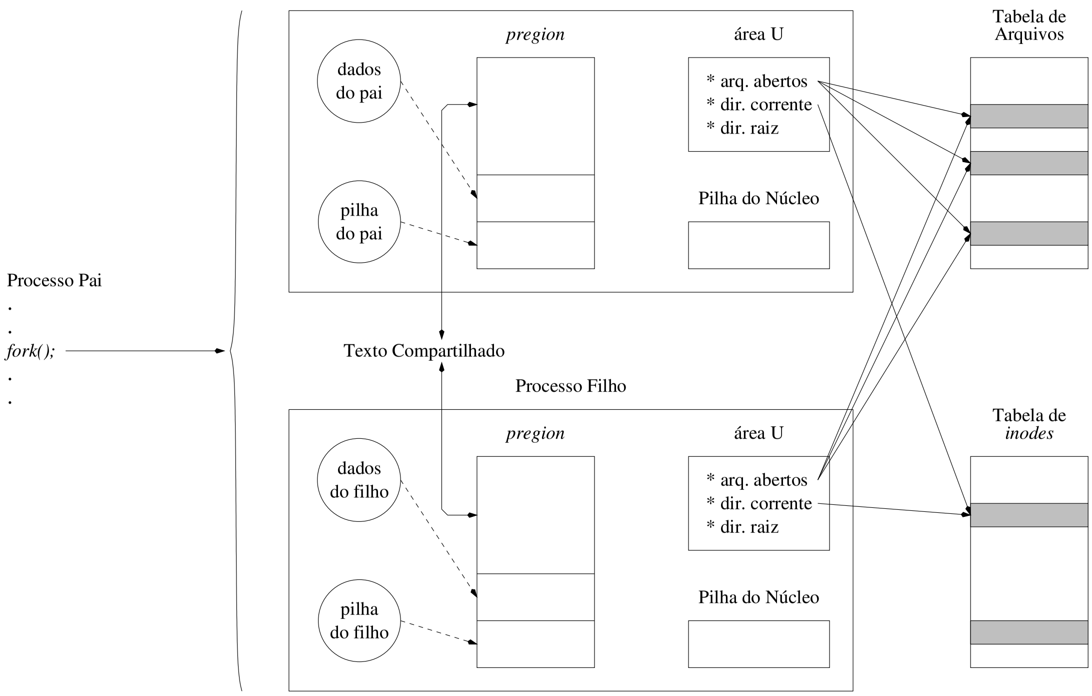
- **controle de processos** - divide-se em 02 categorias: instanciação e interrupção.

✫ **instanciação de processos** - o mecanismo básico de criação de processos no UNIX é a chamada de sistema *fork*, que pode ser descrita da seguinte forma:

- um novo elemento na tabela de processos mantida pelo núcleo é criado;
- a área U é criada à imagem da área U do processo pai;
- uma tabela de região é criada com áreas de pilha e dados copiados do processo pai, e área de texto compartilhada com o processo pai;
- a pilha do núcleo é por fim copiada.

- ... a cópia de regiões de dados e pilhas faz com que toda a história de execução do processo pai seja herdada pelo processo filho.

... 2.3 - Controle de Processos no UNIX



... 2.3 - Controle de Processos no UNIX

- ... uma alternativa de instanciação de processos é a família *exec* — que utiliza os recursos do processo executor para instalar um novo processo.

- ... *exec* tem como parâmetros o arquivo executável e dados a serem passados ao novo processo, sendo que estes são obtidos na função *main (int argc, char** argv)*;

- ... basicamente, uma chamada *exec* libera as regiões de memória relativas a texto, dados e pilha alocadas pelo processo executor da chamada, instalando novas regiões para o novo processo, ou seja, a história do processo antigo é perdida, mas os descritores de arquivo por ele abertos permanecem válidos.

✰ diferença entre a chamada de sistema *fork* e *exec* - criar um novo processo sem que o processo que o criou tenha sua execução terminada.

... 2.3 - Controle de Processos no UNIX

- seja o exemplo do uso das chamadas de sistema *fork* e *exec*:

```
/* Programa que invoca a chamada fork() */
#include <sdtio.h>

int main( int argc, char** argv )
{
    int pid, fd1, fd2;
    char arq1[16], arq2[16];

    fd1 = open( "arq1", O_RDONLY, 0 );
    fd2 = open( "arq2", O_RDONLY, 0 );
    sprintf( arq1, "%d", fd1 );
    sprintf( arq2, "%d", fd2 );
    pid = fork();
    if( pid == 0 ) {
        execlv( "prog1", arq1, arq2, NULL );
        printf("Execlv FALHOU !");
        exit( 0 );
    }
    . . .   /* Processo Pai continua . . . */
}
```

```
/* Programa 1 */
int main( int argc, char** argv )
{
    int fd1, fd2;
    sscanf( argv[1], "%d", &fd1 );
    sscanf( argv[2], "%d", &fd2 );
    . . .
    . . .
}
```


... 2.3 - Controle de Processos no UNIX

- **interrupção de processos** - processos podem ter sua execução alterada assincronamente por ação de um outro processo (p. ex., do usuário através da *shell*).

sinal	significado
SIGHUP	hang-up
SIGINT	interrupção
SIGILL	instrução ilegal (*trap*)
SIGFPE	exceção aritmética (*trap*)
SIGKILL	término forçado
SIGSEGV	violação de segmentação (*trap*)
SIGSYS	argumento inválido em chamada de sistema
SIGALRM	alarme de relógio
SIGSTOP	suspensão da execução
SIGCONT	continuação da execução
SIGCHLD	mudança de *status* de processo filho

- sinais são enviados a processos através da chamada de sistema *kill* com os parâmetros *pid* do processo a que se destina e o tipo de sinal.

... 2.3 - Controle de Processos no UNIX

✰ processos podem apresentar as seguintes reações a sinais:

- o processo é compulsoriamente terminado;
- o processo é bloqueado até receber um sinal de desbloqueio;
- o processo ignora o sinal;
- o processo captura o sinal.

✰ sinais são armazenados na tabela de processos mantida pelo núcleo, sendo que a ação correspondente ao sinal tem efeito quando o processo passa a execução do modo núcleo para o modo usuário.

- ... para cada sinal é definido um comportamento *default* para o processo que o recebe, que em geral implica no término da execução do processo.

- ... um processo, no entanto, pode alterar o seu comportamento frente a ocorrência de um sinal que tenha sido recebido via chamada de sistema *signal*.

... 2.3 - Controle de Processos no UNIX

✰ *signal* possui dois parâmetros: número do sinal definido em *signal.h* e a ação a ser executada quando do recebimento deste sinal:

- 0 - o processo é terminado quando recebe o sinal;
- 1 - o sinal é ignorado e nenhum efeito é gerado;
- funçao - um função é chamada assincronamente quando da ocorrência do sinal, funcionando assim como um gerenciador do sinal para o processo.

```
int main( int argc, char** argv )
{
    signal( SIGINT, ger_sigint );
    . . .
}
```

```
/* Gerenciado do SIGING */
int ger_sigint( )
{
    int c;
    printf("Tem certeza que quer abortar [s/n] ?");
    c = getchar();
    if( c == 's' )
        exit( 0 );
}
```

... 2.3 - Controle de Processos no UNIX

- ... quando um processo passa do modo núcleo para o usuário, o núcleo verifica se existe sinais enviados ao processo e que ainda não foram tratados.

- ... caso exista, o núcleo executa os seguintes passos:

❶ salva o contador de programa e o apontador de pilha do processo;

❷ executa a ação *default* se o processo não definiu uma função de tratamento de sinal, caso contrário segue o passo ❸;

❸ associa temporiamente a ação *default* para o sinal, como forma de evitar que uma rajada de sinais ocasione estouro de pilha;

❹ cria um novo quadro na pilha como se o processo estivesse evocando o gerenciador de sinal neste momento;

❺ direciona o contador de programa para o endereço da rotina de tratamento do sinal e atualiza o apontador de pilha para considerar o aumento da pilha causado pela chamada do gerenciador de sinal.

2.4 - Comunicação e Sincronização entre Processos no UNIX

✫ **pipes** - é um mecanismo original de comunicação entre processos geralmente empregado para estabelecer comunicação entre processos pai e filho.

- ... *pipes* são bloqueantes, assim a gravação em um *pipe* cheio ou a leitura em um *pipe* vazio causa o bloqueio do processo.

```
int main( int argc, char** argv )
{
    int fd[2];
    char buff[32];
    if( pipe(fd) == -1 ) {
        perror("pipe");
        exit( 0 );
    }
    . . .
```

```
if( fork()  != 0 ) {
    close( fd[0] );      /* close read descriptor */
    strcpy( buff, "Oi Filho!");
    write( fd[1], buff, strlen( buff ) + 1 );
    close( fd[1] );
    exit( 0 );
}
else {
    close( fd[1] );      /* close write descriptor */
    read( fd[0], buff, 32 );
    printf( "%s", buff );
    close( fd[0] );
    exit( 0 );
}
}       /* end of main program */
```

... 2.4 - Comunicação e Sincronização entre Processos no UNIX

✫ **troca de mensagens** - mais flexíveis que os *pipes*, mensagens são canais tanto síncronos quanto assíncronos e bidirecionais.

- ... para envio e recepção de mensagens cria-se um *port* de comunicação através da chamada *msgget*, que por sua vez retorna seu identificador local;

- ... a comunicação em um *port* é bidirecional, posto que, de posse de um identificador local de *port*, um processo pode tanto enviar quanto receber mensagens nesse *port*;

- ... mensagens são enviadas através da chamada *msgsnd* com 04 parâmetros:

❶ identificador do port para o qual a mensagem deve ser enviada;
❷ a estrutura que contém a mensagem;
❸ tamanho da mensagem em bytes;
❹ opção de envio assíncrono com retorno de código de erro caso não exista espaço e envio síncrono com bloqueio do processo ante a impossibilidade de armazenamento pelo núcleo da mensagem no *port*.

... 2.4 - Comunicação e Sincronização entre Processos no UNIX

- ... a recepção da mensagem se dá através da chamada *msgrcv*:

❶ identificador local do *port*;
❷ estrutura onde a mensagem será copiada;
❸ tamanho máximo em bytes alocados pela estrutura;
❹ tipo da mensagem desejada;
❺ opção de recebimento assíncrono (retornando código de erro caso o *port* não contenha mensagem do tipo especificado).

- ... o controle pode ser feito pela chamada *msgclt* afim de alterar o *status* de *ports*:

❶ identificador do *port*;
❷ tipo de operação a ser efetuada;
❸ endereço da estrutura com as informações de entrada ou retorno, de acordo com a ação explicitada no segundo parâmetro.

... 2.4 - Comunicação e Sincronização entre Processos no UNIX

✯ *memória compartilhada* - compartilhamento de um espaço virtual através da chamada de sistema *shmget* com 03 parâmetros:

❶ identificador global da região compartilhada;
❷ tamanho da região compartilhada em bytes;
❸ opções de controle, criação e acesso à região compartilhada.

- ... *msgget* retorna o identicador da região, podendo criá-la caso inexista;

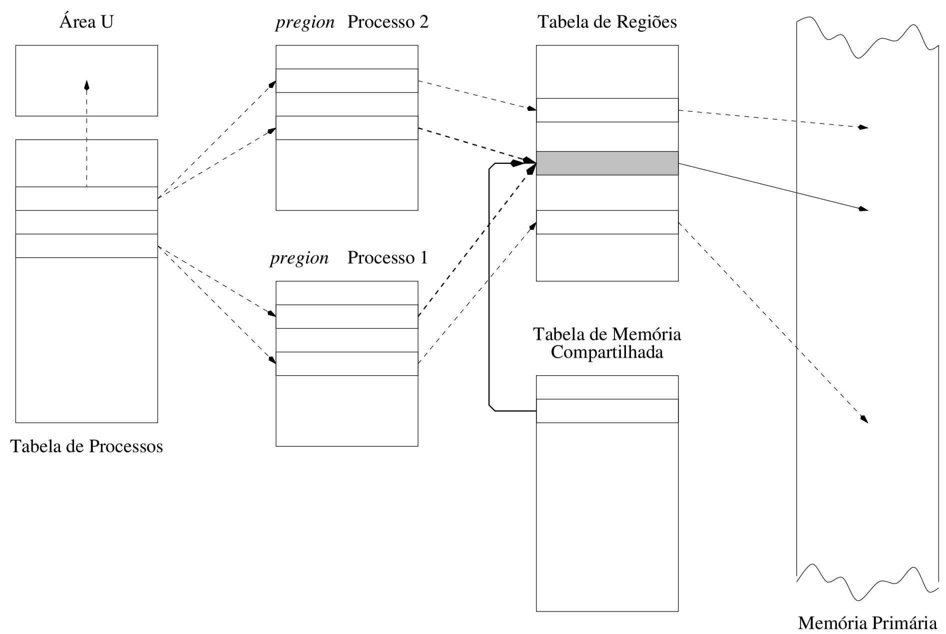
- ... uma vez acessada, um processo associa a região compartilhada em seu próprio espaço de endereçamento e para tal utiliza a chamada *shmat* com 03 parâmetros:

❶ identificador local da região compartilhada;
❷ endereço local que apontará para a região;
❸ parâmetros de controle se a região deve ser apenas de leitura, por exemplo.

- ... o procedimento inverso, ou seja, a desassociação de uma região compartilhada de um endereçamento local se dá com a chamada de sistema *shmdt*.

... 2.4 - Comunicação e Sincronização entre Processos no UNIX

- ... a figura abaixo ilustra o esquema da memória compartilhada:



... 2.4 - Comunicação e Sincronização entre Processos no UNIX

- ... a memória compatilhada é implementada pelo núcleo através de uma tabela de regiões alocadas por *shmget*, que por sua vez aponta para a tabela de regiões de memória mantida pelo núcleo;

- ... após a chamada *shmat*, a tabela de regiões do processo aponta para a respectiva região mantida pelo núcleo.

- ... finalmente, a chamada *shmctl* é empregada para obter e alterar o *status* de uma região compartilhada (permissões, desativação, etc.) de forma semelhante a *msgctl*.

- o mecanismo de memória compartilhada não necessita operações especiais para manipulação de dados, dado que a região é mapeada num endereço local;

- este mecanismo de comunicação é sempre assíncrono e pode suportar comunicação do tipo "de um para muitos", p. ex., um processo escreve na memória e vários lêem;

... 2.4 - Comunicação e Sincronização entre Processos no UNIX

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

#define KEY 67
extern char* shmat();
int main( int argc, char** argv )
{
    char *buff;
    char *poema[16];
    int i, sh;

    /* Cria área compartilhada */
    sh = shmget( KEY, 1024, 0777 | IPC_CREAT );
    /* Associa área compartilhada a um endereço local */
    buff = shmat( sh, 0, 0 );

    poema[0] = "Se não houver frutos";
    poema[1] = "Valeu a beleza das flores.";
    poema[2] = "Se não houver flores,";
    poema[3] = "Valeu a beleza das folhas.";
    poema[4] = "Se não houver folhas,";
    poema[5] = "Valeu a intenção da semente.";

    /* Armazena o Texto na área compartilhada */
    for( i=0; i < 6; i++)
        strcpy( (buff + i*100), poema[i] );
}
```

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

#define KEY 67
extern char* shmat();
int main( int argc, char** argv )
{
    char *buff;
    char *poema[16];
    int i, sh;

    /* Cria área compartilhada */
    sh = shmget( KEY, 1024, 0777 | IPC_CREAT );
    if( sh < 0 ) {
        printf("\nÁrea Compartilhada não criada.\n");
        exit( 0 );
    }
    /* Associa área compartilhada a um endereço local */
    buff = shmat( sh, 0, 0 );

    /* Acessa Texto no na área compartilhada */
    for( i=0; i < 6; i++)
        poema[i] = (buff + i*100);
    for( i=0; i < 6; i++)
        printf("\n%s", poema[i]);
    printf("\n\n");
}
```

... 2.4 - Comunicação e Sincronização entre Processos no UNIX

✰ **semáforo** - processos definem e/ou acessam semáforos com a chamada de sistema *semget* fornecendo 03 parâmetros:

❶ identificador global para um *array* de semáforos;
❷ o número de semáforos contido no *array*;
❸ *flag* estipulando ações relativas a permissões e criação.

- ... operações em semáforo se processam através da chamada *semop*, fornecendo-se 03 parâmetros:

❶ identificador local do semáforo;
❷ um *array* de estruturas *sembuf*;
❸ número de estruturas no *array*.

```
struct sembuf {
  short sem_num;        /* indice do semáforo */
  short sem_op;         /* operação requisitada */
  short sem_flag;       /* controle da operação */
}
```


... 2.4 - Comunicação e Sincronização entre Processos no UNIX

- ... o segundo campo da estrutura `sembuf` opera no respectivo semáforo, conforme o teste feito no valor de `sem_op`:

  – **negativo** - se a soma de `sem_op` com o valor do semáforo for não negativa, some `sem_op` ao valor do semáforo e retorne, e se for negativa, bloqueie;
  – **positivo** - some `sem_op` ao valor do semáforo e retorne;
  – **nulo** - retorne se o valor do semáforo for nulo; bloqueie caso contrário.

- ... similar ao *msgctl* e *shmctl*, a chamada de sistema *semctl* é empregada para obter e alterar o *status* de semáforos (permissões, desativação, etc.);

... 2.4 - Comunicação e Sincronização entre Processos no UNIX

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

/* Cria um Semáforo */
int DefSem( key_t key )
{
    int semid;
    union semun arg;

    semid = semget( key, 1, 0777 | IPC_CREAT );
    arg.val = 1;
    semctl( semid, 0, SETVAL, arg );
    return( semid );
}

/* Acessa um Semáforo já criado */
int GetSem( key_t sem )
{
    int semid;
    semid = semget( key, 1, 0777 );
    return( semid );
}

/* Define operação DOWN */
void DOWN( int semid )
{
    struct sembuf psem[2];

    psem[0].sem_num = 0;
    psem[0].sem_op = -1
    psem[0].sem_flg = SEM_UNDO;

    semop( semid, psem, 1 );
}

/* Define operação UP */
void UP( int semid )
{
    struct sembuf vsem[2];

    vsem[0].sem_num = 0;
    vsem[0].sem_op = 1;
    vsem[0].sem_flg = SEM_UNDO;

    semop( semid, vsem, 1 );
}
```


... 2.4 - Comunicação e Sincronização entre Processos no UNIX

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

/* Cria um Semáforo */
int DefSem( key_t key )
{ ... }

/* Acessa um Semáforo já criado */
int GetSem( key_t sem )
{ ... }

/* Define operação DOWN */
void DOWN( int semid )
{ ... }

/* Define operação UP */
void UP( int semid )
{ ... }

int main( int argc, char** argv )
{
    char *poema[6];
    ...
    int i, semid;
    semid = GetSem( 13 );
    if( semid < 0 ) {
        printf("\nSemáforo não criado !\n");
        exit( 0 );
    }

    poema[0] = "Se não houver frutos";
    poema[1] = "Valeu a beleza das flores.";
    poema[2] = "Se não houver flores,";
    poema[3] = "Valeu a beleza das folhas.";
    poema[4] = "Se não houver folhas,";
    poema[5] = "Valeu a intenção da semente.";

    while( 1 ) {
        DOWN( semid );
        for( i - 0; i < 6; i++) {
            printf("\n%s", poema[i] );
            sleep( 1 );
        }
        printf("\n\n");
        UP( semid );
    }
}
```