Tópico 4

O Sistema Unix

O SISTEMA UNIX

- Histórico
  - Final da década de 60: o sistema operacional MULTICS (**MULT**iplexed **I**nformation and**C**omputing *S*ervice foi projetado para ser um sistemaoperacional padrão, pelos melhores pesquisadores da época (MIT, GE, Bell Labs). Enorme sucesso acadêmico. Enorme fracasso comercial.. Era um software muito pesado para hardwares com pouca capacidade computacional.
  - UNICS (**UNI**plexed **I**nformation and **C**omputing **S**ervice): sistema projetado por um pesquisador da Bell Labs, que abandonou o projeto no meio (Ken Thompson). Funcionou primeiramente em um PDP-7. Devido ao seu sucesso, seu código foi portado para um PDP-11, reescrito em linguagem de alto nível(B). Ritchie concebe uma nova linguagem, derivada do B: a linguagem C. O sistema UNIX é reescrito em C por Ritchie e Thompson. A Bell Labs dá o sistema Unix às universidades com o código fonte (1974).

HISTÓRICO

- UNIX versão 6 (1976): primeiro padrão do Unix no meio universitário, contou com contribuições de várias universidades. Próxima versão: UNIX versão 7 (1978).
- Como o sistema operacional era escrito em uma linguagem de alto nível, era relativamente simples portá-lo para diferentes arquiteturas. No primeiro trabalho de migração entre arquiteturas, foi projetado um compilador C portátil (Johnson) que poderia, devidamente customizado, gerar código para várias arquiteturas diferentes.
- UNIX system III (1982): lançado comercialmente em 1984 pela AT&T. Fracasso comercial.

HISTÓRICO

- UNIX system V(1983): versão melhorada do system III. Sucesso de mercado.
- UNIX BSD: Desenvolvido a partir do Unix versão 6 pela University of California, em Berkeley. A versão 4BSD incorpora: paginação, sistema de arquivos rápido, comunicação em rede com o protocolo TCP/IP.
- final dos anos 80: duas versões comerciais incompatíveis do UNIX, UNIX 4.3BSD e UNIX System V release III. Unix-likes: Ultrix (DEC), Xenix (Microsoft)

TENTATIVAS DE PADRONIZAÇÃO

- System V Interface Definition (AT&T): definição de chamadas ao sistema e formato de arquivo. Só foi utilizado no System V.
- POSIX (Portable Operating System): pegou as características comuns aos sistemas System V e BSD e propôs padronizações de interfaces de: chamadas de sistema, shell, correspondência, acesso a arquivos, etc.
- OSF (consórcio Open Software Foundation): tentativa da IBM, DEC e HP de produzir um Unix que possua todas as características vigentes. IEEE + X11 + Motif + DCE, etc.
- UI (consórcio Unix International): resposta da AT&T -> AIX (da IBM).

O Sistema Linux

- Seu desenvolvimento foi iniciado em 1991, por um estudante finlandês chamado Linus Torvalds.
- O Linux inicial era um kernel Unix pequeno mas completo para arquiteturas 80386.
- Os códigos do Linux foram disponibilizados no grupo de discussão comp.os.minix.
- A partir daí, o desenvolvimento do Linux contou com participantes de diversas partes do mundo, que se correspondiam pela Internet.

O Sistema Linux

- No ínicio, o desenvolvimento se concentrou no kernel e o desenvolvimento do kernel é até hoje específico do projeto Linux.
- Quando o desenvolvimento do Linux se consolidou, surgiu o conceito de distribuição.
- Uma distribuição inclui todos os componentes do Linux básico acrescidos de um conjunto de ferramentas de administração.

O SISTEMA UNIX - OBJETIVOS

- O Unix é um sistema multi-usuário e multi-tarefa, simples, poderoso e flexível.
- Simplicidade: Tentando evitar um dos principais fatores que causaram o fracasso do MULTICS, os algoritmos do Unix foram selecionados por sua simplicidade, e não por sua velocidade ou sofisticação.
- Flexibilidade: Não foi feito um projeto detalhado do Unix antes de sua implementação, apesar de terem existido alguns princípios de projeto. Assim, foram projetados um pequeno número de elementos básicos que poderiam se combinar de diversas maneiras, resultando em surpreendente flexibilidade.

O Sistema Unix - Objetivos

- Portabilidade: Na implementação do Unix, foi tomado um grande cuidado em manter as estruturas independentes do hardware, na medida do possível. Essa característica, aliada à utilização da linguagem C para a confecção do código fonte, geraram código de portabilidade relativamente simples.
- O Unix é um sistema feito por programadores para programadores. O usuário inexperiente pode dispor de uma interface simples porém maior cuidado foi tomado em oferecer ao programador experiente uma interface concisa e poderosa.

Controle de usuários no UNIX

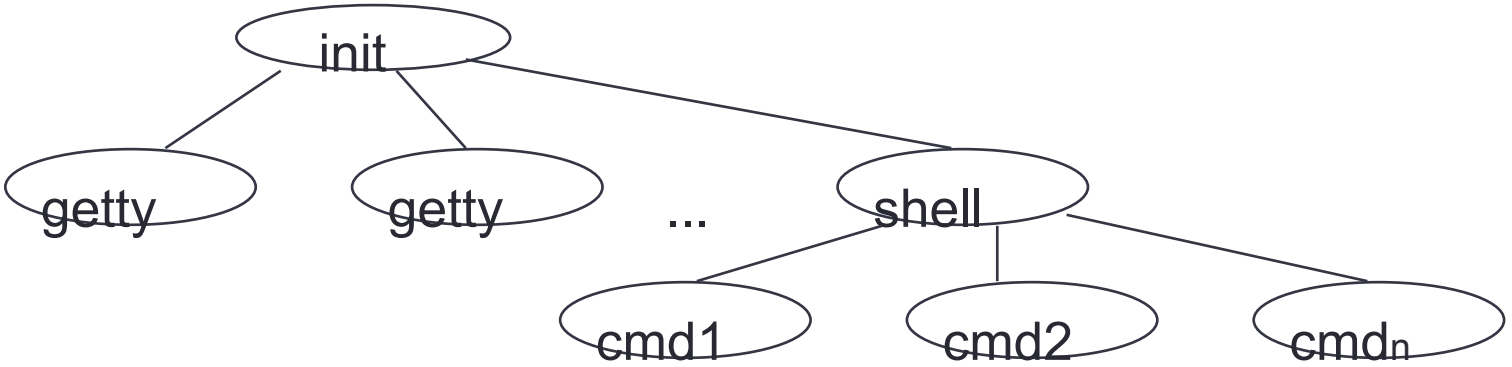
- Cada usuário deve possuir uma conta(login) no Unix para ter acesso ao sistema. Cada conta possui uma senha(password) associada. O par login,password deve constar em um arquivo de senhas (/etc/passwd). O arquivo de senhas não é escondido e todos os usuários podem lê-lo. A password contida neste arquivo está criptografada. Cada usuário é identificado por um número inteiro (uid).
- Conteúdo do arquivo /etc/passwd:
  - username:password:uid:gid:coment:home_dir:shell
- Superusuário:
  - usuário com uid = 0 (root).
  - Poderes do superusuário:
    - ler e escrever em todos os arquivos do sistema
    - executar chamadas de sistema protegidas

Gerência de Processos no Unix

- Os processos no Unix são processos tradicionais (heavyweight). Cada processo possui recursos alocados a ele (ambiente) e registradores (linha de execução). Um processo opera sobre um único conjunto de registradores.
- Os termos heavyweight e lightweight descrevem o custo de uma operação de troca de contexto. Processos heavyweight = grande overhead na troca de contexto.
- O Unix é um sistema operacional multiprogramado. Muitos processos ativos em um dado instante. Muitos processos de um mesmo usuário ativos em um dado instante.

Árvore de Processos

- Os processos são organizados em uma árvore de processos.



- Todo processo tem um número único que o identifica (pid - process id). Todo processo tem um pai (ppid - parent pid). Todo processso é um descendente do processo “init”.

Gerência de Processos Unix

- Tabela de Processos:
  - Guarda as informações referentes aos processos.
  - É acessada pelo pid.
  - As informações relacionadas ao processo são visualizadas através do comando *ps* (shell).

Tabela de Processos

- Algumas informações contidas:
  - *Flags de estado:* descrevem o estado de execução do processo.
  - *UID:* grupo do usuário que startou o processo
  - *pid:* identificador único do processo
  - *ppid:* pid do pai do processo
  - *CP:* fator de utilização de CPU
  - *PRI*: prioridade do processo
  - *NI:* parâmetro para o escalonador
  - *SZ:*tamanho do segmento de dados + pilha
  - *RSS:* Memória real utilizada pelo processo
  - *WCHAN*: evento pelo qual o processo está esperando
  - *STAT:* status de execução do processo
  - *TT:* terminal associado ao processo
  - *TIME*: tempo de CPU (user+system)
  - *COMMAND*: arquivo executável que gerou o processo

Comando ps -l

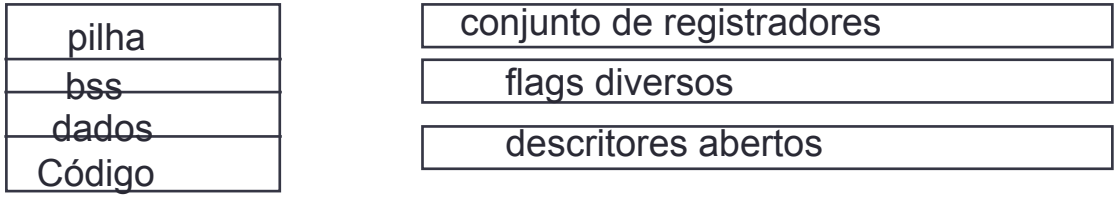
```
Last login: Thu Feb 23 13:05:27 on console


Alba-Melos-MacBook:~ albamelo$ ps -l
  UID   PID  PPID        F CPU PRI NI       SZ    RSS WCHAN     S     ADDR TTY           TIME CMD
  501   700   699     4006   0  31  0  2435468   1068 -       S     5ed0540 ttys000    0:00.04 -bash
  501   723   700     4006   0  31  0  2426560    312 -       S     5efc000 ttys000    0:00.00 ./teste_sleep
  501   726   700     4006   0  31  0  2426560    308 -       R     5564a80 ttys000    0:12.18 ./teste


Alba-Melos-MacBook:~ albamelo$
```

CRIAÇÃO DE PROCESSOS

- Criação de processos:
  - Entidades componentes do processo:

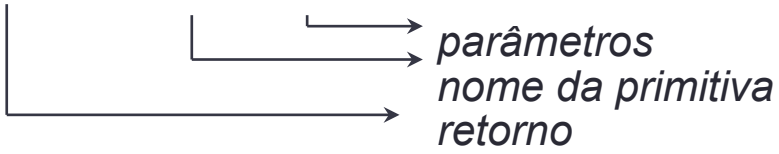


- O contexto de um processo está descrito na tabela de processos
  - A criação de processos no Unix envolve a cópia da maioria dos campos contidos na tabela de processo (registradores, flags de estado, etc) e a cópia dos segmentos de código, dados e pilha do processo pai.

Ao final da criação, os processos pai e filhoexecutam o mesmo arquivo executável.

CRIAÇÃO DE PROCESSOS

- Como o PC (program counter) também foi copiado de processo-pai para processo-filho, o processo-filho começa a sua execução na instrução imediatamente posterior à instrução que causou a sua criação.
- Primitiva de criação de processos: fork
  - Sintaxe: `int fork()`



CRIAÇÃO DE PROCESSOS

- Descrição do fork:
  - O novo processo é uma cópia exata do processo pai, com as seguintes exceções: o pid de cada processo é único; o ppid de cada processo é diferente; o processo filho possui uma cópia dos descritores abertos do pai; todos os semáforos são zerados; os parâmetros de escalonamento são zerados no filho; locks não são herdados; sinais pendentes são descartados no filho
- Retorno:
  - No caso de sucesso, retorna 0 para o filho e o pid do processo filho para o pai.
  - Caso contrário, retorna -1.

CRIAÇÃO DE PROCESSOS

- Exemplo:

```
if ((pid = fork()) < 0)
{ printf("erro no fork\n"); exit (1); }
if (pid == 0)
  printf ("sou o processo filho\n");
else
  printf ("sou o processo pai\n");
```

- Obs: Normalmente, quando nós criamos um novo processo, desejamos executar um novo programa. Nos demais sistemas operacionais, o nome do arquivo executável é dado no momento da criação do processo. No Unix, precisamos fazer em dois passos. Primeiro, devemos usar o fork para gerar um processo filho que é a réplica do processo pai. Depois, devemos executar a primitiva que faz o processo executar um novo programa (ex: execl)

Primitiva de carga de arquivo executável: exec

- Sintaxe:
  - `int execl(char *path, char *argv[0], char *argv [1],..., (char *) 0);`
  - Descrição:
    - O novo arquivo executável é fornecido em *path*. A lista de parâmetros é fornecida em argv. Argv[0] deve ser o nome do programa (sem path). A lista de variáveis de ambiente é fornecida em envp. Estas duas listas devem ser terminadas por NULL.
    - O novo processo mantem o pid, o ppid, os parâmetros do escalonador, o real uid, os descritores de arquivos abertos
    - São alterados: a imagem do processo na memória, o conjunto de registradores e o uid efetivo.
    - Só retorna no caso de erro.
    - No caso de sucesso, não pode retornar pois a imagem foi perdida.

Primitiva de carga de arquivo executável: exec

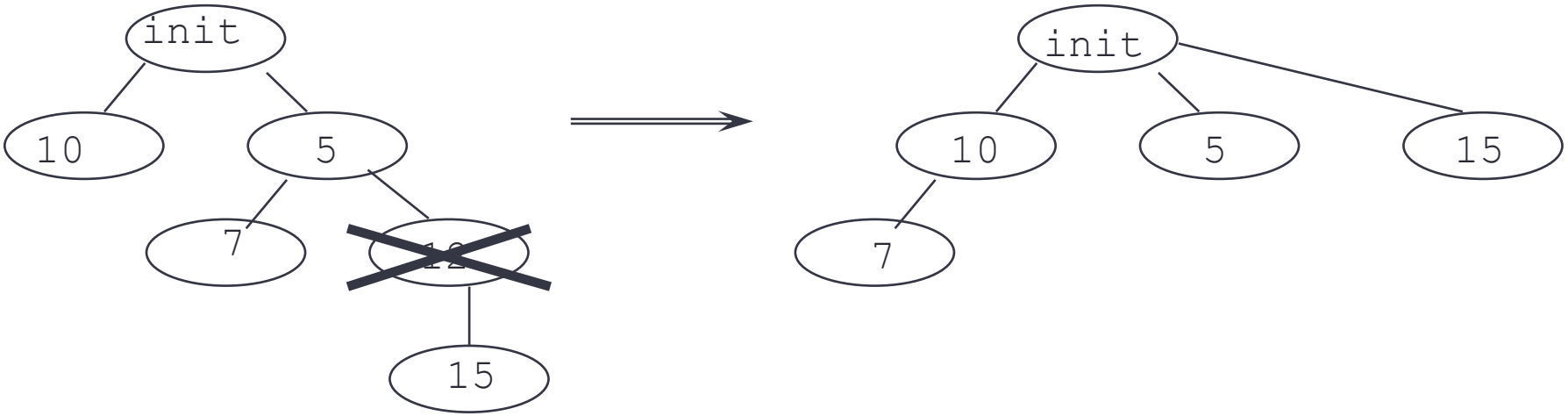
- Exemplo:

```
if ((pid == fork()) < 0)
{ printf("erro no fork\n"); exit(1); }
if (pid == 0)
/* processo filho */
  execl("prfilho", "prfilho", (char *) 0);
printf("continuo o codigo do pai\n");
```

- Algumas primitivas úteis:
  - `int getpid();`
  - `int getuid();`
  - `int getppid();`
  - `int geteuid();`

Término de Processos

- Sempre que um processo filho termina a sua execução, o processo pai deve ser avisado.
- Quando um processo morre, seus filhos passam a ser filhos do processo init. Quando um processo morre e o pai não está esperando o aviso de sua morte, o processo "morrendo" fica em um estado <zombie> ou <defunct> até que o pai seja avisado.



Término de Processos – Primitiva exit

- O término de processos é implementado através das primitivas `wait` e `exit`.
- Primitiva que termina um processo: _exit
  - Sintaxe: void _exit(int status);
  - Descrição: todos os descritores de arquivos abertos são fechados. Se o processo pai estiver executando um wait, ele é notificado do término do processo filho e tem acesso ao status. Se o pai não estiver esperando, o status é salvo até que o pai execute um wait.
  - Retorno: Nunca retorna, já que causa o término do programa
  - Obs: A maioria dos programas C, chama a rotina exit(), que limpa as bibliotecas standard antes de terminar o programa

Término de Processos - Primitiva wait

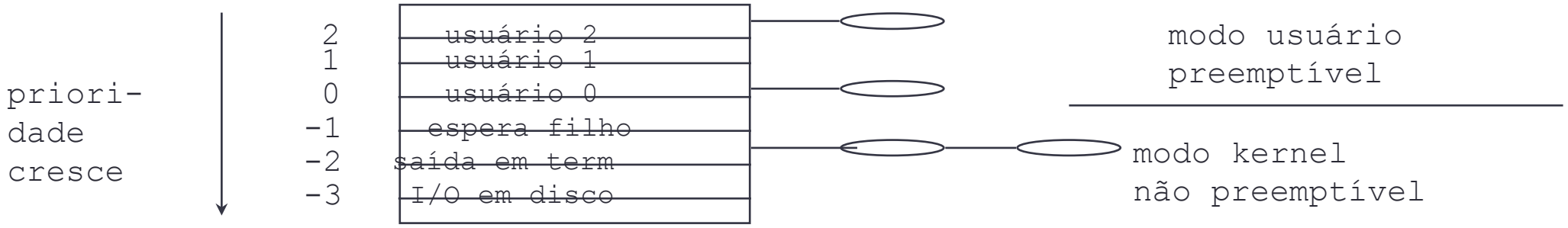
- Primitiva de espera de término de um processo: wait
  - Sintaxe: int wait (int *status);
  - Descrição: bloqueia o processo pai até que um de seus filhos termine ou seja parado para trace. Se exite algum processo quejá morreu mas não foi “esperado” ainda, o retorno é imediato.
  - Retorno:
    - -1: o processo não tem filhos
    - se o primeiro byte = 0177: filho parado
    - se o primeiro byte != 0177 e > 0: filho terminou por causa de um sinal
    - senão filho terminou por um exit.

Estados de Processos

- No Unix, de maneira similar aos outros sistemas operacionais, um processo pode estar pronto para rodar (ready) ou bloqueado (blocked). O processo pode estar bloqueado por várias razões, dentre elas:
  - esperando que a página corrente seja carregada (P);
  - esperando I/O (D);
  - parado por um utilitário de debug (T);
  - dormindo por poucos segundos (S);
  - dormindo por muitos segundos (I);
  - etc.

Escalonamento de Processos

- Os processos prontos para rodar competem pelo uso da CPU, daí a necessidade de um algoritmo de escalonamento de processos.
- Escalonamento de processos no Unix: o algoritmo utilizado é o escalonamento com prioridades dinâmicas. Processos com a mesma prioridade são regidos pela política round-robin. O algoritmo proposto favorece a execução de processos I/O bound (critério = tempo de resposta).



Escalonamento de Processos

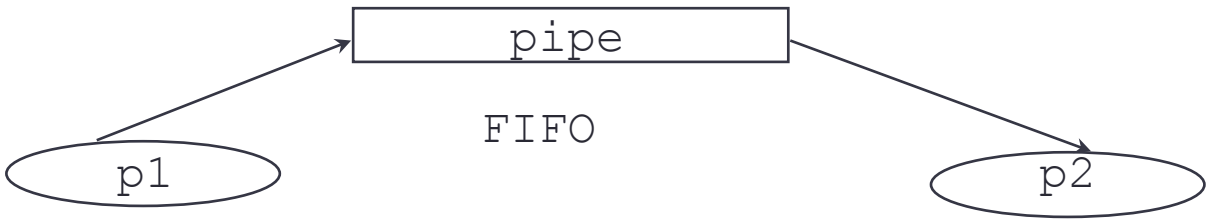
- Cálculo dinâmico da prioridade:
  - A cada "clock tick", o contador de utilização do processador do processo em execução é incrementado na tabela de processos (quantun = *x* clock_ticks).
  - Periodicamente, a prioridade de cada processo é recalculada:
    - pri = base + utilização do processador - *aging*
    - base normalmente é zero, mas o processo pode pedir para rodar em prioridade mais baixa. Nesse caso, base tem um valor positivo
    - Para impedir starvation: o algoritmo do Unix usa a técnica de *aging* (as prioridades dos processos que esperam CPU há muito tempo são decrementadas periodicamente).
  - Primitiva que reduz a prioridade de um processo: nice
    - Sintaxe: `int nice(incr)`
    - Descrição: o fator *base* é substituído por *incr.* Os processos usuário só podem aumentar o nice e consequentemente reduzir a prioridade do processo. Só superusuário pode fornecer valores negativos

Comunicação entre processos

- Os processos Unix, ao longo de sua execução, podem necessitar trocar dados e controles com outros processos. O sistema operacional Unix permite que os processos se comuniquem de diversas maneiras:
  - Pipes;
  - Filas de mensagens;
  - Memória compartilhada;
  - Semáforos;
  - Sinais.

Pipes

- Na linguagem de comando (shell), os pipes são utilizados com freqüência.
- Exemplo: ls -l * |grep May. Neste caso, a saída do comando ls -l * é passada como entrada do próximo comando grep May.
- Na programação Unix, nós utilizamos exatamente o mesmo conceito. Os pipes são buffers protegidos em memória, acessados segundo a política FIFO



Pipes - Criação

- Primitiva de criação de pipes:
  - Sintaxe: int pipe(int fd[2]);
  - Descrição: cria o mecanismo pipe, que é composto de dois descritores de arquivos (fd[0] e fd[1]) para leitura e escrita, respectivamente.
  - Retorno: 0 no caso de sucesso e -1 no caso de erro

Pipes - Manipulação

- *Escrita:*
  - Primitiva: `int write(int fd, char *buf, int nbyte)`
  - Descrição: escreve *nbyte* do buffer apontado por *buf* no descritor de arquivo *fd.*
  - Retorno: Se OK, retorna o número de bytes escritos com sucesso em *fd.* Se não, retorna -1.

- *Leitura:*
  - Primitiva: `int read(int fd, char *buf, int nbyte)`
  - Descrição: lê *nbyte* do arquivo *fd* para o buffer apontado por *buf.*
  - Retorno: Se OK, retorna o número de bytes lidos com sucesso de *fd.* Se não, retorna -1.

Pipes - Resumo

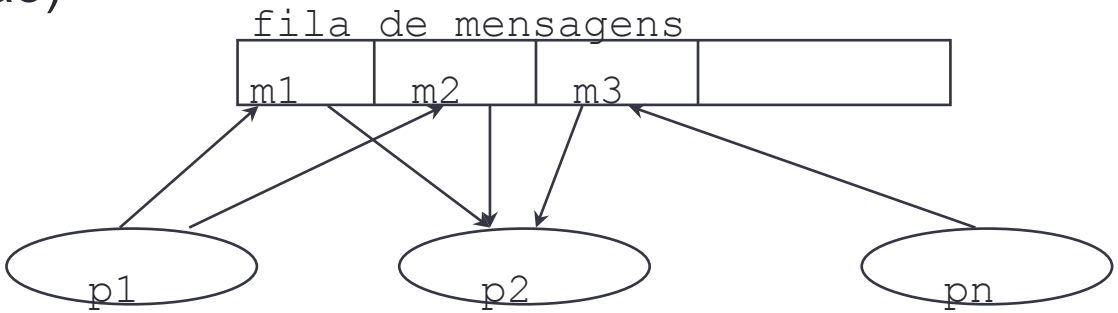
- Mecanismo genérico:
  - cria o pipe (primitiva pipe);
  - cria o processo (primitiva fork). A primitiva fork vai duplicar os descritores de arquivos, assim, o pipe fica disponível para o processo pai e o processo filho;
  - um processo lê e o outro escreve no pipe;
- Observação:
  - A primitiva read lê do pipe <nome>[0] e a primitiva write escreve no pipe <nome>[1].

Exemplo

```
main()
{
   int  pid, /* pid do processo filho */
        fd[2], /* descritores do pipe */
        estado; /* estado do processo filho */
   char mensagem[30];
   /* cria o pipe */
   if (pipe(fd) < 0)
   { printf("erro na criacao do pipe\n"); exit(1);}
   /* cria o processo */
   if ((pid = fork()) < 0)
   { printf("erro na criacao do processo\n"); exit(1);}
   /* codigo do filho */
   if (pid == 0)
   {
     if (read(fd[0], mensagem, 30) < 0)
       printf("erro na leitura\n");
     else
       printf("valor lido = %s\n", mensagem);
     exit(0);
   }
   /* codigo do pai */
   strcpy(mensagem, "teste do envio no pipe");
   if (write(fd[1], mensagem, sizeof(mensagem)) < 0)
       printf("erro na escrita\n");
   wait(&estado);
   exit(0);
}
```

Filas de mensagem

- As filas de mensagens são mecanismos de comunicação protegidos. Podem estar em memória ou em disco. Só os processos autorizados têm acesso à fila. As filas de mensagens são permanentes, ou seja, elas não são destruídas quando o processo que as criou morre. Necessitam de uma remoção explícita.
- Passos para a utilização de filas:
  1. Criar a fila
  2. Obter o identificador da fila
  3. Ler ou escrever na fila
  4. Remover a fila (ou não)



Criação de Filas

- Primitiva de criação e obtenção de filas: msgget
  - Includes necessários:
    - `#include <sys/types.h>`
    - `#include <sys/ipc.h>`
    - `#include <sys/msg.h>`
  - Criação de filas:
    - Sintaxe:
      - `int msgget(key_t key, int IPC_CREAT | msgflg);`
    - Descrição: o processo que chama esta primitiva cria uma fila com a chave key e as permissoes de acesso em msgflg. O parâmetro IPC_CREAT determina a criação de filas.
    - Retorno:
      - Sucesso: identificador da fila (msqid). Este identificador será utilizado em todas as operações sobre a fila.
      - Erro: -1.
    - Exemplo:

```
if (idfila = msgget(0x1233, IPC_CREAT | 0x1FF) < 0)
{
  printf("erro na criacao da fila\n");
  exit(1);
}
```

Obtenção de Fila

Sintaxe:

```
int msgget(key_t key, int msgflg);
```

- Descrição: o processo que chama esta primitiva obtem o identificador da fila criada com a chave key . As permissoes de acesso são especificadas em msgflg.
- Retorno:
  - Sucesso: identificador da fila (msqid). Este identificador será utilizado em todas as operações sobre a fila.
  - Erro: -1. (erros de permissão de acesso, fila não existe).
- Exemplo:

```
if (idfila = msgget(0x1233, 0x124) < 0)
{
  printf("erro na obtencao da fila\n");
  exit(1);
}
```

Envio de Mensagem

- Sintaxe:
  - `int msgsnd(int msqid, struct msgbuf *msgp, int msgsize, int msgflg);`
- Descrição: o processo que chama esta primitiva envia a mensagem contida em `msgp` de tamanho `msgsize` para a fila `msqid` com os flags `msgflg`.
  - msqid é o identificador obtido com msgget.
  - msgp aponta para a estrutura da mensagem que é a seguinte:
    - long mtype;
    - char mtext[1];
  - Pode ser enviada uma mensagem de tamanho qualquer, contanto que o primeiro campo seja long e descreva o tipo da mensagem.
  - msgsize é o tamanho da mensagem em bytes.
  - msgflg determina o tipo do envio:
    - IPC_NOWAIT: o processo não espera que a mensagem seja colocada na fila e retorna imediatamente
    - 0 : o processo fica bloqueado até que a mensagem possa ser colocada na fila, ou até que a fila seja removida ou até o envio de um sinal que deve ser tratado.
- Retorno:
  - Sucesso: 0; Erro: -1.

Recepção de Mensagem

- Sintaxe:
  - `int msgrcv(int msqid, struct msgbuf *msgp, int msgsize, long msgtyp, int msgflg);`
- Descrição: o processo chama esta primitiva para receber uma mensagem do tipo `msgtyp` de tamanho `msgsize` da fila `msqid` com os flags `msgflg`. A mensagem recebida deve ser colocada no buffer apontado por `msgbuf`.
  - msqid é o identificador obtido com msgget.
  - msgp aponta para a estrutura da mensagem a ser recebida que é a seguinte:
    - long mtype;
    - char mtext[1];
    - Pode ser recebida uma mensagem de tamanho qualquer, contanto que o primeiro campo seja long e descreva o tipo da mensagem.
  - msgsize é o tamanho da mensagem em bytes.
  - msgtyp é o tipo da mensagem (primeiro campo da mensagem). msgp -> mtype deve ser igual a msgtyp.
  - msgflg == IPC_NOWAIT : se não houver mensagem, o processo retorna. msgflg == 0 : se não houver mensagem, o processo fica bloqueado até que chegue uma mensagem, ou a fila seja removida ou seja recebido um sinal tratável.
- Retorno:
  - Sucesso: número de bytes enviados (sem contar o tipo); Erro: -1.

Remoção da Fila de Mensagem

- Sintaxe:
  - `int msgctl(int msqid, IPC_RMID, struct msqid_ds *buf);`
- Descrição: o processo que chama esta primitiva remove a fila de mensagem `msqid`. A remoção da fila só pode ser feita pelo usuário que a criou ou pelo superusuário.
- Retorno:
  - Sucesso: 0
  - Erro: -1.

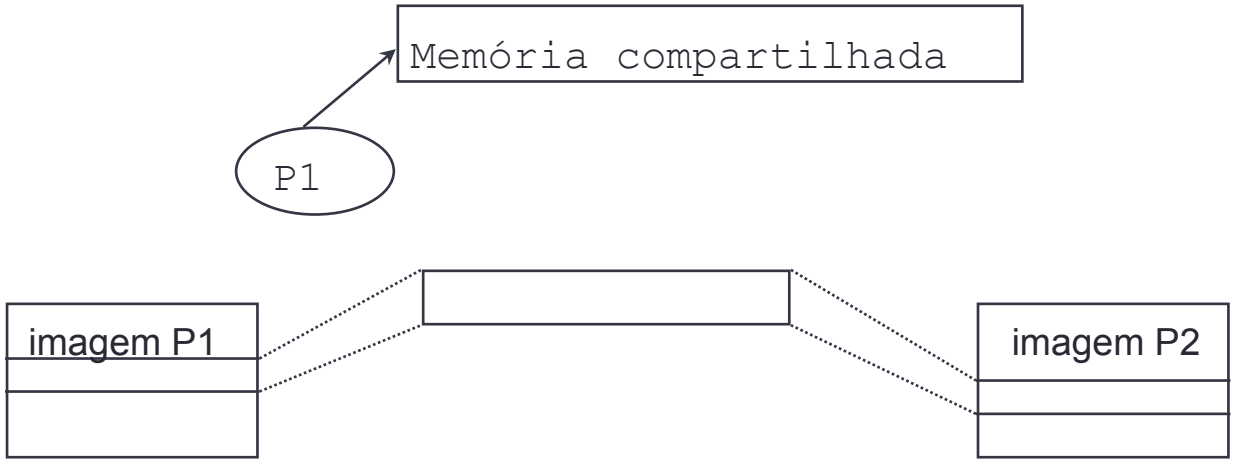
Exemplo

```
#include<errno.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
main()
{
   int pid, idfila, fd[2], estado;
   struct mensagem
   { long pid; char msg[30];};
   struct mensagem mensagem_env, mensagem_rec;
   if ((idfila = msgget(0x1223, IPC_CREAT|0x1ff)) < 0)
   { printf("erro na criacao da fila\n”); exit(1);}
   pid = fork();
   if (pid == 0)
   {
      mensagem_env.pid = getpid();
      strcpy(mensagem_env.msg, "teste de mensagem");
      msgsnd(idfila, &mensagem_env, sizeof(mensagem_env), 0);
      exit (0);
   }
   msgrcv(idfila, &mensagem_rec, sizeof(mensagem_rec), 0, 0);
   printf("mensagem recebida = %d %s\n", mensagem_rec.pid, mensagem_rec.msg);
   wait (&estado);
   exit(0);
}
```

Memória Compartilhada - Passos

- Os processos no Unix podem compartilhar explicitamente um segmento de memória protegido pelo kernel.
- Os passos envolvidos são os seguintes:

❶Criação do segmento de memória compartilhada:



❷Depois da criação, os processos podem dar um attach no segmento compartilhado. O attach bem sucedido retorna um ponteiro para o início da área compartilhada.

Memória Compartilhada - Passos

3. Após o attach, o segmento de memória compartilhada pode ser acessado via o ponteiro retornado, com operações normais de read e write.
4. O segmento de memória compartilhado não é destruído quando o processo que o criou morre. Ele necessita de uma remoção explícita.

Criação de Memória Compartilhada

- Includes necessários:
  - `#include <sys/types.h>`
  - `#include <sys/ipc.h>`
  - `#include <sys/shm.h>`
- Primitiva de criação e obtenção de memória compartilhada: shmget
  - Criação de memória:
    - Sintaxe:
      - `int shmget(key_t key, int size, int IPC_CREAT | shmflg);`
    - Descrição: o processo que chama esta primitiva cria um segmento de memória compartilhada de tamanho size com a chave key e as permissões de acesso em shmflg. O parâmetro IPC_CREAT determina a criação de segmentos de memória. O uid e gid da memória são o uid e o gid efetivos do usuário dono do processo.
    - Retorno:
      - Sucesso: identificador da memória compatilhada. Este identificador será utilizado para o attach
      - Erro: -1.

Obtenção de Memória Compartilhada

- Obtenção de memória:
  - Sintaxe:
    - `int shmget(key_t key, int size, int shmflg);`
  - Descrição: o processo que chama esta primitiva obtem o segmento de memória compartilhada de tamanho size com a chave key e as permissões de acesso em shmflg.
  - Retorno:
    - Sucesso: identificador da memória compatilhada. Este identificador será utilizado para o attach
    - Erro: -1.

Attach de Memória Compartilhada

- Mapeamento de memória:shmat
  - Sintaxe:
    - `char *shmat(int shmid, char *shmaddr, int shmflg);`
  - Descrição: o processo que chama esta primitiva mapeia o segmento de memória compartilhada shmid no endereço shmaddr de seu espaço de endereçamento.
    - parâmetros: shmid: descritor de memória compartilhada obtido no shmget. shmaddr: endereço no qual o segmento de memória será mapeado. Se shmaddr for 0, o endereço de mapeamento é selecionado pelo sistema. shmflg: determina o modo de acesso à memória compartilhada (read_only ou read/write).
  - Retorno:
    - Sucesso: endereço do segmento de memória compartilhada
    - Erro: -1.

Detach de Memória Compartilhada

- Unmapping de memória:shmdt
  - Sintaxe:
    - `int shmdt(int shmid);`
  - Descrição: o processo que chama esta primitiva desfaz o mapeamento do segmento de memória compartilhada shmid. A partir do shmdt, o processo não tem mais acesso ao segmento. Deve fazer um novo attach para acessá-lo de novo.
  - Retorno:
    - Sucesso: 0
    - Erro: -1.

Remoção de Memória Compartilhada

- Remoção de memória:shmctl
  - Sintaxe:
    - `int shmctl(int shmid, IPC_RMID, struct shmid_ds *buf);`
  - Descrição: o processo que chama esta primitiva remove o segmento de memória compartilhada shmid. A partir deste momento, nenhum processo possui mais acesso a este segmento. Só o dono ou o superusuário podem remover segmentos.
  - Retorno:
    - Sucesso: 0
    - Erro: -1.

Exemplo

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
main()
{
   int pid, idshm,  estado;
   struct shmid_ds buf;
   int *pshm;

   if ((idshm = shmget(0x1223, sizeof(int),IPC_CREAT|0x1ff)) < 0)
   { printf("erro na criacao da fila\n"); exit(1);}
   if ((pid = fork()) == 0)
   { /* codigo do filho */
     pshm = (int *) shmat(idshm, (char *)0, 0);
     if (pshm == (int *)-1)
     { printf("erro no attach\n"); exit(1);}
     printf("vou escrever\n");
     *pshm = 334;
     exit(0);
   }
   /* codigo do pai */
   pshm = (int *) shmat(idshm, (char *)0, 0);
   if (pshm == (int *) -1)
   { printf("erro no attach\n"); exit(1); }
   sleep(1);
   printf("pai - numero lido = %d\n", *pshm);
   wait(&estado);
   exit (0);
 }
```

Semáforos

- Os semáforos são utilizados para delimitar uma seção crítica, onde somente um processo executará de cada vez. São uma maneira de estabelecer uma certa ordem na execução dos processos.
- Os semáforos no Unix foram implementados de maneira extremamente flexível e, por esta razão, as primitivas resultantes são de grande complexidade.
- Os semáforos são estruturas de dados sobre as quais são executadas operações indivisíveis.

Manipulação de Semáforos

- Passos para a utilização de semáforos:
  1. Criação de um conjunto de semáforos, que será identificado por semid.
  2. Obtenção do identificador do semáforo, verificados todos os problemas de permissão de acesso
  3. Execução de operações sobre os semáforos. Estas operações são na verdade um conjunto de operações definidas pelo programador que serão executadas de maneira indivisível.
  4. Remoção do identificador do semáforo. Os semáforos são também permanentes, ou seja, necessitam de uma remoção explícita.

Criação de Semáforos

- Includes necessários:
  - `#include <sys/types.h>`
  - `#include <sys/ipc.h>`
  - `#include <sys/sem.h>`
- Primitiva de criação e obtenção de um conjunto de semáforos: semget
  - Criação de semáforos:
  - Sintaxe:
    - `int semget(key_t key, int nsems, int IPC_CREAT | semflg);`
  - Descrição: o processo que chama esta primitiva cria um conjunto de nsems com a chave key e as permissões de acesso em shmflg. O parâmetro IPC_CREAT determina a criação de segmentos de memória. O uid e gid da memória são o uid e o gid efetivos do usuário dono do processo.

Operações sobre Semáforos

- Sintaxe:
  - `int semop(int semid, struct sembuf *sops, int nsops);`
- Descrição: esta primitiva executa indivizivelmente um conjunto de operações sobre um conjunto de semáforos identificados por semid. São executadas nsops operações sobre os semáforos. As operações estão descritas na estrutura sembuf, que contem os seguintes campos:
  - short sem_num /* numero do semáforo */
  - short sem_op /* tipo da operação */
  - short sem_flg /*flags */
    - se sem_op < 0: se o valor do semáforo for maior ou igual ao valor absoluto de sem_op, sem_val = sem_val - |sem_op|. Caso contrário, se sem_flg for diferente de IPC_NOWAIT, o processo fica bloqueado.
    - se semop = 0: se o valor do semáforo for 0, retorna; senão fica bloqueado, se sem_flg for diferente de IPC_NOWAIT.
    - se semop > 0: o valor de semop é somado ao valor do semáforo.
- Retorno:
  - Sucesso: 0; Erro: -1.

Obtenção de Semáforos

- Obtenção de semáforos:
  - Sintaxe:
    - `int semget(key_t key, int nsems, int shmflg);`
  - Descrição: o processo que chama esta primitiva obtem o conjunto de nsems semáforos com a chave key e as permissões de acesso em shmflg.
  - Retorno:
    - Sucesso: identificador do conjunto de semáforos.
    - Erro: -1.

Remoção de Semáforos

- Remoção de um conjunto de semáforos:semctl
  - Sintaxe:
    - ```
      int semctl(int shmid,
                 int semnum,
                 int IPC_RMID,
                 union semun {
                  val;
                  struct semid_ds *buf;
                  ushort *array;
                 }arg; shmid_ds *buf);
      ```
  - Descrição: o processo que chama esta primitiva remove o conunto de semáforos identificado por semid. Só o dono ou o superusuário podem remover semáforos.
  - Retorno:
    - Sucesso: 0; Erro: -1.
      ```

```
#include<errno.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
struct sembuf operacao[2];
int idsem;
int p_sem()
{
     operacao[0].sem_num = 0;
     operacao[0].sem_op = 0;
     operacao[0].sem_flg = 0;
     operacao[1].sem_num = 0;
     operacao[1].sem_op = 1;
     operacao[1].sem_flg = 0;
     if ( semop(idsem, operacao, 2) < 0)
       printf("erro no p=%d\n", errno);
}
int v_sem()
{
     operacao[0].sem_num = 0;
     operacao[0].sem_op = -1;
     operacao[0].sem_flg = 0;
     if ( semop(idsem, operacao, 2) < 0)
       printf("erro no p=%d\n", errno);
}
```

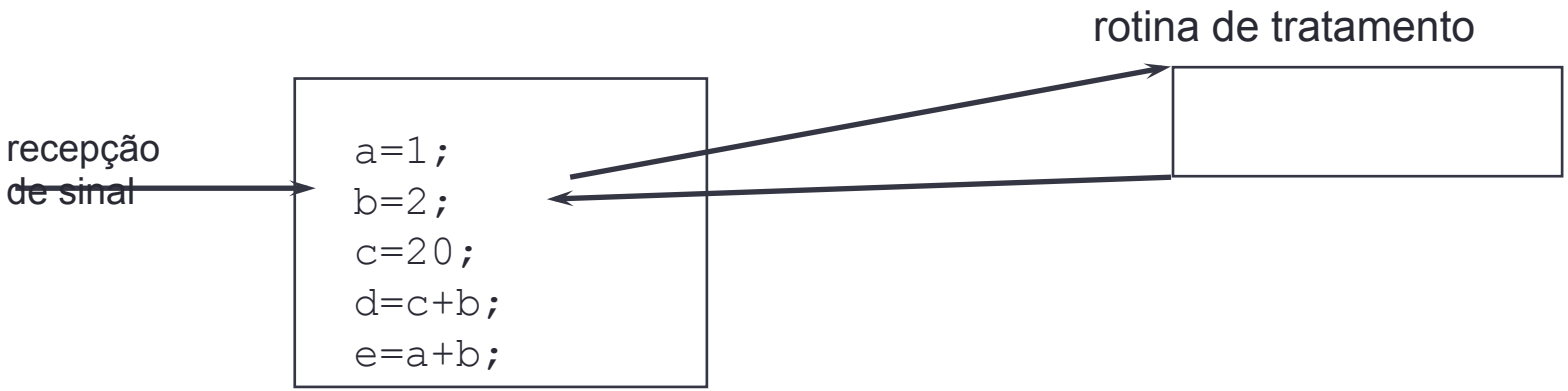
Exemplo – Rotinas P e V (semop)

```
main()
{
   int pid, estado;
   int *psem;
   if ((idsem = semget(0x1223, 1, IPC_CREAT|0x1ff)) < 0)
   { printf("erro na criacao do semaforo\n"); exit(1);}
   if ((pid = fork()) == 0)
   { /* codigo do filho */
     p_sem();
     printf("filho - obtive o semaforo\n");
     printf("filho - vou liberar o semaforo\n");
     v_sem();
      exit(0);
   }
   /* codigo do pai */
     p_sem();
     printf("pai - obtive o semaforo\n");
     printf("pai - vou liberar o semaforo\n");
     v_sem();
   wait(&estado);
   exit (0);
}
```

Exemplo – Programa Principal

Tratamento de Sinais

- Os sinais são interrupções que chegam assincronamente aos processos. No momento de recepção de um sinal "tratável", a execução do programa é desviada para a rotina de tratamento de sinais. Ao final da rotina de tratamento de sinal, o programa volta à instrução imediatamente posterior àquela que estava sendo executada quando o sinal foi recebido.



Sinais do Unix

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3	quit
SIGILL	4	illegal instruction
SIGTRAP	5	trace trap
SIGABRT	6	abort (generated by abort(3) routine)
SIGEMT	7	emulator trap
SIGFPE	8	arithmetic exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
SIGBUS	10	bus error
SIGSEGV	11	segmentation violation
SIGSYS	12	bad argument to system call
SIGPIPE	13	write on a pipe or other socket with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGURG	16	urgent condition present on socket
SIGSTOP	17	stop (cannot be caught, blocked, or ignored)
SIGTSTP	18	stop signal generated from keyboard
SIGCONT	19	continue after stop
SIGCHLD	20	child status has changed
SIGTTIN	21	background read attempted from control terminal
SIGTTOU	22	background write attempted to control terminal

Sinais do Unix

SIGIO	23	I/O is possible on a descriptor
SIGXCPU	24	cpu time limit exceeded
SIGXFSZ	25	file size limit exceeded
SIGVTALRM	26	virtual time alarm
SIGPROF	27	profiling timer alarm
SIGWINCH	28	window changed
SIGLOST	29	resource lost
SIGUSR1	30	user-defined signal 1
SIGUSR2	31	user-defined signal 2

- A maioria destes sinais possui tratamento default dado pelo sistema. A maioria dos sinais causa o término do processo que o recebe, se a ação default for mantida. Com exceção dos sinais SIGKILL e SIGSTOP, todos os outros sinais podem receber um tratamento diferente do default. Os sinais SIGUSR1 e SIGUSR2 não possuem ação pré-definida e são usados para enviar interrupções entre processos de usuário.

Especifica rotina de tratamento - signal

- Rotina signal:
  - Sintaxe:

```
#include <signal.h>
    void (*signal(sig, func))()
    void (*func)();
```

  - Descrição: A rotina signal determina que a função func será executada quando o sinal sig for recebido. A função func pode ser SIG_DFL, SIG_IGN ou uma função qualquer fornecida pelo usuário.
  - Retorno:
    - Sucesso: retorna a ação tomada anteriormente
    - Erro : -1

Atenção: Se o sinal for recebido quando uma operação de I/O estiver sendo executada (arquivos, pipes, filas, semáforos), a operação não é executada até o final e retorna o erro EINTR.

Envio de Sinais - Kill

- Primitiva de envio de sinais: kill
  - Sintaxe:

    #include <signal.h>

    int kill (pid_t pid, int sig);
  - Descrição: O processo que executa o kill envia o sinal sig ao processo de pid pid. Se sig for igual a 0, o kill testa somente a existência do processo de pid pid. Se pid for -1 e o processo for de usuário, o sinal é enviado a todos os processos com uide do usuário. No caso de superusuário, o sinal é enviado a todos os processos do sistema. O processo pode mandar sinal para ele próprio
  - Retorno:
    - 0: sucesso
    - -1: erro

- Exemplo 1: recepção e envio de sinais

```
#include <signal.h>
void funcao_sigusr1()
{
   printf("recebi sigusr1\n");
}
void funcao_sigusr2()
{
   printf("recebi sigusr2\n");
}
main()
{
   signal(SIGUSR1, funcao_sigusr1);
   signal(SIGUSR2, funcao_sigusr2);
   kill(getpid(), SIGUSR1);
   kill(getpid(), SIGUSR2);
}
```

- Exemplo 2: tratamento de exceções

```
#include <signal.h>
void funcao_sigsegv()
{
   printf("recebi segment fault. Vou morrer!!!\n");
   exit(1);
}
main()
{
   char *p;
   signal(SIGSEGV, funcao_sigsegv);
   /* vou forcar um segment fault */
   printf("%s", *p);
}
```

Rotinas alarm e pause

- Função: alarm
  - Sintaxe: unsigned int alarm(unsigned int seconds)
  - Descrição: envia o sinal SIGALARM ao processo que o chamou depois de seconds segundos. alarm(0) cancela os alarms pendentes.
  - Retorno: tempo que faltava para o alarm anterior.
- Função: pause
- Sintaxe: int pause( )
  - Descrição: pára o processo até que um sinal seja recebido.
  - Retorno:
    - Sucesso: não retorna.
    - Erro: -1.

Rotina sleep

- Sintaxe: int sleep(unsigned seconds)
  - Descrição: suspende a execução do processo por no mínimo seconds segundos.
  - Retorno:
    - Sucesso: 0.
    - Se o sleep retornar por recepção de sinal, retorna o número de segundos que o processo deixou de dormir.

Gerência de Memória Unix

- Nas versões mais modernas do Unix, é utilizado o mecanismo de paginação por demanda para a gerência da memória.
- Para rodar, um processo necessita unicamente que sua tabela de páginas esteja em memória. A tabela de páginas dos sistemas Unix é uma geralmente tabela invertida, por isso também é chamada mapa de memória.
- Uma entrada típica na tabela de páginas possui:
  - Índice da próxima entrada, índice da entrada anterior, número do bloco no disco, número do disco, próximo endereço de hash, índice na tabela de processos, base dos segmentos de código, dados e pilha,número da página dentro do segmento, bits de controle, etc.

Gerência de Memória Unix

- Imagem da memória real:

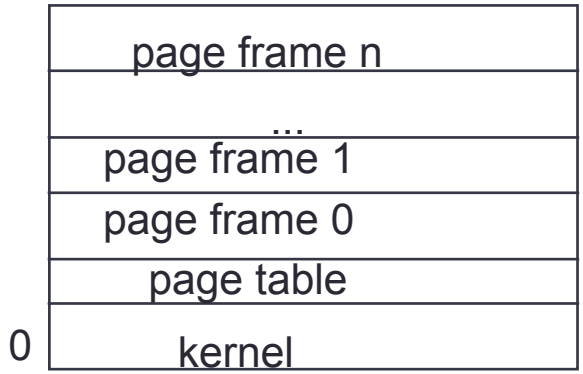
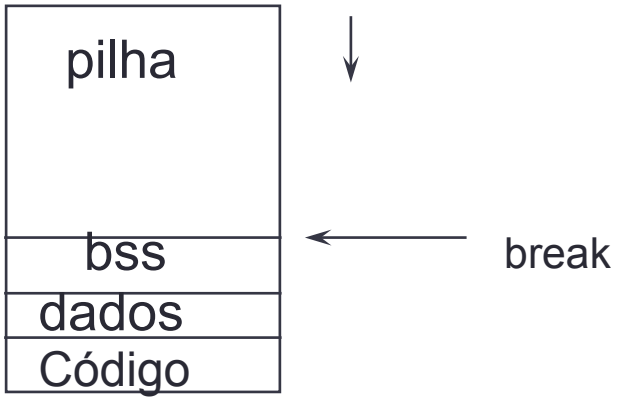


Imagem do processo:



Gerência de Memória Unix - Implementação

- A gerência de memória no Unix é implementada basicamente por três “processos”: o gerente de page fault, o swapper e o paginador.
  - O gerente de page fault é contactado toda vez que uma exceção do tipo page fault ocorre. Ele é o responsável por encontrar uma entrada livre na tabela de páginas e carregar a página que gerou o page fault em memória.
  - O tratamento de page fault é uma operação demorada (da ordem do milisegundo). Para melhorar o desempenho do tratamento do page fault, é utilizado o processo paginador que roda em background e executa o algoritmo de substituição.
  - O swapper é responsável por colocar e retirar processos inteiros na/da memória.

Gerência de Memória Unix - Paginador

- Processo paginador: executa periodicamente (a cada 250ms) e verifica o estado da ocupação da memória. Se houverem muito poucas page frames livres, o paginador tira algumas páginas da memória, liberando assim os page frames (algoritmo de substituição). Assim, diminui a probabilidade de execução do algoritmo de substituição no momento do tratamento do page fault.
  - Esquema de funcionamento:
    - variável min: sempre que a utilização da memória cai abaixo de min, o paginador roda.
    - variável max: o paginador roda até que existam max molduras livres

Gerência de Memória Unix - Implementação

- Algoritmo de substituição de páginas:
  - global: escolhe a página independente do processo que a possui
  - o algoritmo utilizado é o *algoritmo do relógio com duas passagens*. Primeira passagem: zera o bit de utilização. Segunda passagem: as páginas que ainda tiverem o seu bit de utilização zerado são movidas para a lista de espaço livre, mantendo o seu conteúdo. Quando há um page fault, uma das páginas da lista de espaço livre é escolhida. Se uma das páginas contidas na lista de espaço livre for referenciada, ela sai da lista de espaço livre.
  - Se o sistema notar que o paginador está constantemente liberando páginas, o swapper é contactado para remover processos da memória.

Gerência de Memória - Swapper

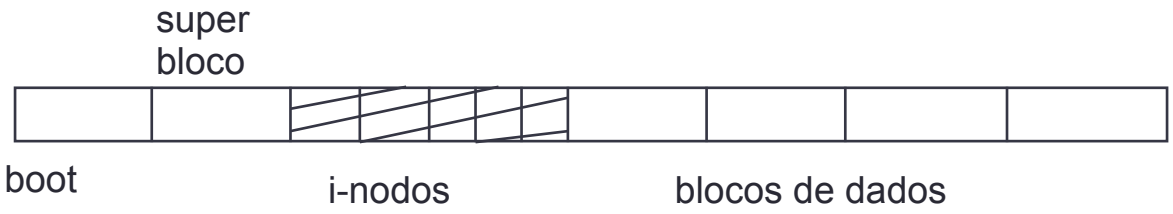
- Processo swapper: Responsável pelas operações de swap in e swap out
  - swap out: se existir processo ocioso por mais de 20 segundos, remove da memória. Se não houver, examina os 4 maiores processos. O que estiver há mais tempo na memória é removido. Repete a operação até que o limite seja atingido.
  - swap in: calcula a prioridade dinâmica dos processos swapped out. Os processos que foram removidos há mais tempo são carregados em memória (swap in). O algoritmo usado evita o movimento de processos muito grandes.

A Chamada brk

- Chamadas de sistema:
  - Rotina: brk
    - Include necessário:
      - #include<sys/types.h>
    - Sintaxe:
      - int brk (caddr_t addr) ou
      - caddr_t sbrk(int incr)
    - Descrição:
      - As duas rotinas setam o endereço mais baixo do segmento de dados. Com brk, o próprio programador deve especificar o novo endereço. Com sbrk, o programador especifica o número de bytes adicionais que ele quer para o seu segmento.

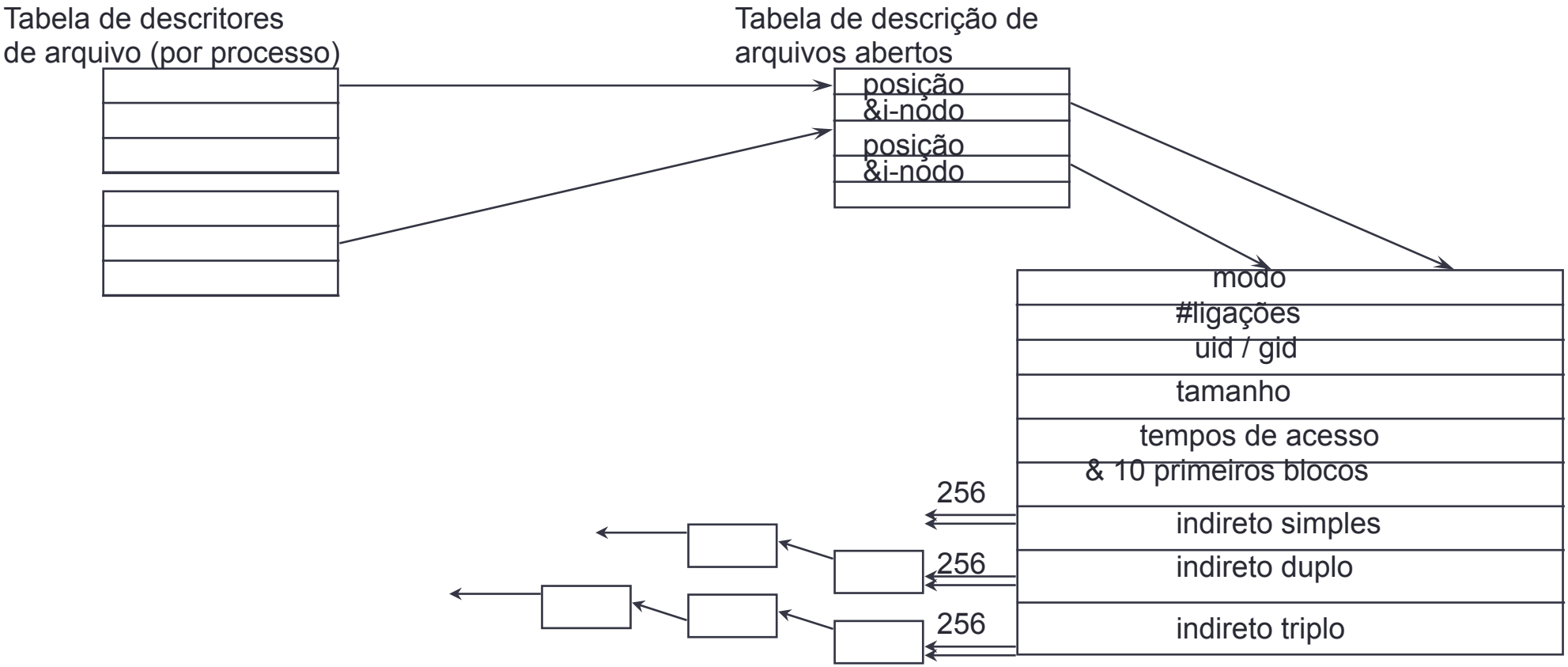
Gerência de Arquivos Unix

- Sistema de Arquivos Unix antigo
  - nomes de arquivos: 14 caracteres
  - Estrutura do sistema de arquivo no disco:



  - Bloco de boot: não é usado pelo sistema de arquivos
  - Super-bloco (bloco 1): contem informações sobre o sistema de arquivos: número de i-nodos, número de blocos, ponteiro para a lista de blocos livres.
  - i-nodos: cada i-nodo descreve um arquivo (64 bytes). Contem informações gerais (dono, proteção) e a localização do arquivo no disco.
  - diretório: conjunto de entrada de 16 bytes (nome do arquivo + número do i-nodo)

Organização dos inodos



- A primitiva de leitura de um arquivo necessita de um descritor de arquivo. Com este descritor, acessa-se a tabela de descritores e desta tabela acessa-se a tabela de descrições. Lá temos o ponteiro para o i-nodo. No i-nodo, compara-se a posição corrente do arquivo com os endereços dos 10 primeiros blocos (acesso direto). Se a posição corrente não cair lá, o acesso deve ser indireto de um, dois ou três níveis.

FFS (Fast File System) - Berkeley

- Nome do arquivo: 255 caracteres
- O disco é dividido em grupos de cilindros, cada qual com o seu superbloco, i-nodos e blocos de dados. Objetivo -> reduzir o tempo de seek.
- Dois tamanhos de bloco: blocos grandes e blocos pequenos. Os arquivos grandes usam um pequeno conjunto de blocos grandes.

➔transferência eficiente de arquivos grandes

➔eficiência no armazenamento de arquivos pequenos (pouca perda de espaço em disco) e de arquivos grandes (diretório)

➔A maioria dos sistemas Unix modernos usa sistemas de arquivo baseados no FFS

Chamada Open – Criação e/ou Abertura

- Includes necessários:

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

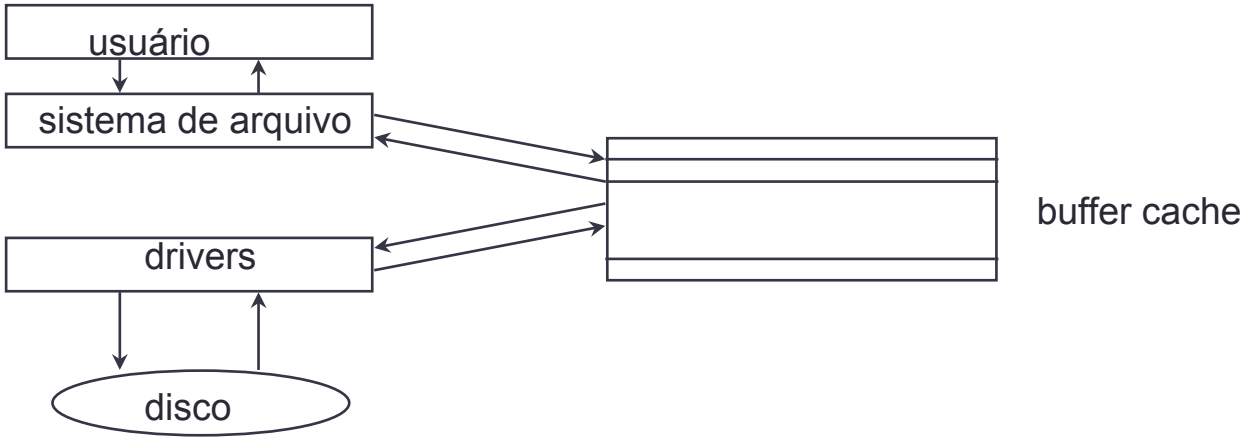
- Sintaxe: int open(char *path, int flags, [mode_t mode])
- Descrição: A primitiva cria o arquivo de nome “path”, se este não existir, se for especificado O_CREAT em flags. O arquivo criado terá as permissões de acesso descritas em mode. O arquivo criado é imediatamente aberto. A operação de abertura de arquivo adiciona uma entrada na tabela de arquivos do processo. O modo de abertura do arquivo está descrito em flags e pode ser:
  - O_RDONLY, O_WRONLY, O_RDWR, O_NDELAY e O_NONBLOCK (usado para pipes e linhas de comunicação), O_SYNC(a escrita só retorna quando o registro tiver sido escrito no disco), O_APPEND, O_EXCL (usado em conjunto com O_CREAT, para retornar erro se o arquivo já existir)
- Retorno: Sucesso: descritor do arquivo aberto, Erro: -1
- Exemplo: fd = open(“arquivo”, O_RDWR | O_CREAT | O_EXCL, 0x777);

Outras chamadas (arquivos)

- Fechamento de arquivos: close
  - Sintaxe: int close (int fd)
  - Descrição: A primitiva close deleta o descritor de arquivos fd da tabela de arquivos do processo.
  - Retorno:
    - Sucesso: 0
    - Erro: -1
- Leitura e escrita de arquivos: sintaxe descrita na parte do curso referente a pipes.
- Manipulação de diretórios: chdir, chmod
- Manipulação de links: link, unlink

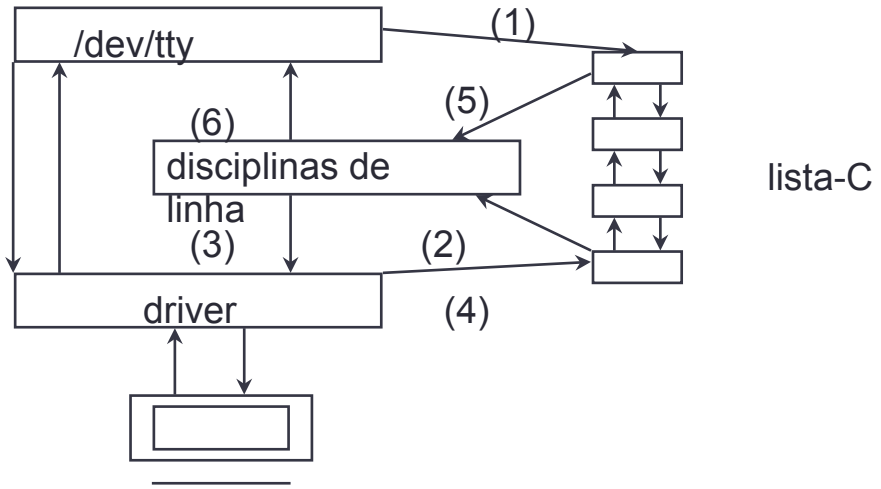
Entrada/Saída no Unix

- Os dispositivos de E/S no Unix são tratados como arquivos => arquivos especiais.
- Cada dispositivo de E/S possui um driver de dispositivo, localizado no diretório /dev, que é dependente do hardware.
- Dois tipos de arquivos especiais: bloco e caracter.
  - Arquivo especial de bloco (e.g. disco): com o objetivo de aumentar o desempenho, existe uma cache entre o sistema de arquivos e o driver (buffer cache). A cada 30 segundos (tempo configurável), os blocos modificados são escritos no driver. O bloco modificado pode ser escrito antes no caso de cache full.



Entrada/Saída no Unix

- Arquivos especiais de caracter (e.g. terminal): Os arquivos a caracter manipulam a estrutura de dados chamada lista-C. Cada elemento desta lista é um pequeno bloco de até 64 caracteres, um contador e um ponteiro para o próximo. Os dados digitados no terminal são filtrados de acordo com disciplinas de linha. São produzidos assim o conjunto de caracteres processado, que será entregue ao processo.



Chamadas Unix de I/O

- Nos sistemas Unix tradicionais, o controle de I/O para os arquivos especiais é feito por uma única chamada ao sistema (ioctl). Assim, usamos a mesma chamada para, por exemplo, alterar a blocagem de transferência de dados entre disco e memória e para alterar a velocidade de um terminal. O padrão POSIX “aboliu” esta chamada e criou um conjunto enorme de outras, específicas a cada função.
  - Exemplo de chamadas POSIX:
    - Setar atributos de um terninal: int tcsetattr(fd, opt, &termio);
    - Setar a velocidade de entrada: int ofsetospeed (&termio, int speed);

Rotina ioctl

- Sintaxe:
  - int ioctl (int fd, int request, caddr_t arg)
- Descrição:
  - A primitiva ioctl executa a função request no arquivo especial descrito por fd. Se forem necessários parâmetros adicionais, eles devem ser fornecidos em arg. A função request é dependente do hardware e é descrita no manual do dispositivo.
- Retorno:
  - Sucesso: != -1
  - Erro: -1

☛**A maioria dos usuários Unix continua a usar ioctl**