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
  - Sinais
  - Filas de mensagens
  - Memória compartilhada
  - Semáforos

Pipes

- Na linguagem de comando (*shell*), os *pipes* são utilizados com freqüência.

- **Exemplo**: ls -l * |grep "filtro".
- Neste caso, a saída do comando ls -l * é passada como entrada do próximo comando grep "filtro".
- O mesmo mecanismo também pode ser usado em programação UNIX para comunicação entre processos.
- Os *pipes* são *buffers* protegidos em memória, acessados segundo a política FIFO.

Pipes

Pipes

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

- **Escrita**:
  - **Primitiva**: `int write(int fd, char *buffer, int nbyte)`
  - **Descrição**: escreve nbyte do buffer apontado por *buffer* no descritor de arquivo fd.
  - **Retorno**:
    - Sucesso: retorna o número de bytes escritos em ***fd***
    - Erro: retorna -1.
- **Leitura**:
  - **Primitiva**: `int read(int fd, char *buffer, int nbyte)`
  - **Descrição**: lê nbyte do arquivo fd para o buffer apontado por *buffer*.
  - **Retorno**:
    - Sucesso: retorna o número de bytes lidos em ***fd***.
    - Erro: retorna -1.

Pipes

- Resumo:
  - cria-se o ***pipe*** (primitiva pipe);
  - cria-se 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].

Pipes

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

int 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);
}
```

Sinais

- Um sinal é uma ***interrupção por software*** enviada aos processos pelo sistema para informá-los da ocorrência de eventos "anormais" dentro do ambiente de execução.
- É uma forma de comunicação assíncrona.
  - Ex: falha de segmentação, violação de memória, erros de entrada e saída, etc.
- Este é um outro mecanismo que possibilita a comunicação e manipulação de processos.

Sinais

P1

```
int main ()
{
    int x;
    int y;
    x = 2;
    y = 3;
    return x+y;
}
```

recebe sinal

```
int trataSinal ()
{
    // Código de
tratamento de sinal
    return x;
}
```

t1

Sinais

Um sinal (com exceção do SIGKILL) pode ser tratado de três maneiras distintas em UNIX:

1. **Ignorado**: o programa pode ignorar as interrupções de teclado geradas pelo usuário (ex: processo executado em *background*).
2. **Interceptado**: na recepção do sinal, a execução de um processo é desviada para o procedimento especificado pelo usuário, para depois retomar a execução no ponto de onde foi interrompido.
3. **Padrão**. Pode ser aplicado a um processo após a recepção de um sinal.

Tipos de Sinal

Os sinais são identificados pelo sistema por um número inteiro.
No linux estão listados no arquivo /usr/include/signal.h

Alguns sinais padrão são:

- **SIGHUP** (1 - ***hangup***) Corte: sinal emitido aos processos associados a um terminal quando este se “desconecta". Este sinal também é emitido a cada processo de um grupo quando o chefe termina sua execução.
- **SIGINT** (2 - ***interrupt***) Interrupção: sinal emitido aos processos do terminal quando as teclas de interrupção (por exemplo: INTR, CTRL+c) do teclado são acionadas.
- **SIGQUIT** (3 - ***quit***)* Abandono: sinal emitido aos processos do terminal quando com a tecla de abandono (QUIT ou CTRL+d) do teclado são acionadas.

Tipos de Sinal

- **SIGILL** (4)* Instrução ilegal: emitido quando uma instrução ilegal é detectada.
- **SIGTRAP** (5)* Problemas com trace: emitido após cada instrução em caso de geração de traces dos processos (utilização da primitiva *ptrace*())
- **SIGIOT** (6)* Problemas de instrução de E/S: emitido em caso de problemas de hardware.
- **SIGEMT** (7) Problemas de instrução no emulador: emitido em caso de erro material dependente da implementação.
- **SIGFPE** (8)* Emitido em caso de erro de cálculo em ponto flutuante, assim como no caso de um número em ponto flutuante em formato ilegal. Indica sempre um erro de programação.

Tipos de Sinal

- **SIGKILL** (9) Destruição: “arma absoluta” para matar os processos. Não pode ser ignorada, tampouco interceptada (existe ainda o SIGTERM para uma morte mais “suave” para processos).
- **SIGBUS** (10)* Emitido em caso de erro sobre o barramento.
- **SIGSEGV** (11)* Emitido em caso de violação da segmentação: tentativa de acesso a um dado fora do domínio de endereçamento do processo.
- **SIGSYS** (12)* Argumento incorreto de uma chamada de sistema.
- **SIGPIPE** (13) Escrita sobre um *pipe* não aberto em leitura.

Tipos de Sinal

- **SIGALRM** (14) Relógio: emitido quando o relógio de um processo pára. O relógio é colocado em funcionamento utilizando a primitiva alarm().
- **SIGTERM** (15) Terminação por software: emitido quando o processo termina de maneira normal. Pode ainda ser utilizado quando o sistema quer por fim à execução de todos os processos ativos.
- **SIGUSR1** (16) Primeiro sinal disponível ao usuário: utilizado para a comunicação entre processos.
- **SIGUSR2** (17) Outro sinal disponível ao usuário: utilizado para comunicação interprocessual.
- **SIGCLD** (18) Morte de um filho: enviado ao pai pela terminação de um processo filho.
- **SIGPWR** (19) Reativação sobre pane elétrica.

Tipos de Sinal

- Observação: Os sinais marcados por * geram um arquivo core no disco quando eles não são corretamente tratados.

- A maioria dos sinais possui tratamento padrão (***default***) dado pelo sistema e causam o término do processo que o recebe.

- Se a ação padrão 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.

Tratamento de Sinais

- Rotina signal:
- Sintaxe:

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

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

- ***kill*** - A primitiva kill() emite ao processo de número ***pid*** o sinal de número ***sig***.

- **Sintaxe**:

```c
#include <signal.h>

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

- **Retorno**:
  - 0: sucesso
  - -1: erro

- A primitiva ***kill()*** e na maioria das vezes executada via o comando ***kill*** no ***shell***.

Envio de Sinais (KILL)

- Se o valor inteiro ***sig*** é nulo, nenhum sinal é enviado, e o valor de retorno vai informar se o número de ***pid*** é um número de um processo ou não.
- Utilização do parâmetro ***pid***:
  - **Se pid > 1**: *pid* designará o processo com ID igual a *pid*.
  - **Se pid = 0**: o sinal é enviado a todos os processos do mesmo grupo que o emissor. Esta possibilidade é geralmente utilizada com o comando *shell kill*. O comando ***kill -9 0*** irá matar todos os processos rodando em *background* sem ter de indicar os IDs de todos os processos envolvidos.

Envio de Sinais (KILL)

- **Se pid = 1**:
  - Se o processo pertence ao **super-usuário (SU)**, o sinal é enviado a todos os processos, exceto aos processos do sistema e ao processo que envia o sinal.
  - Senão, o sinal é enviado a todos os processos com ID do usuário real igual ao ID do usuário efetivo do processo que envia o sinal (é uma forma de matar todos os processos que se é proprietário, independente do grupo de processos ao qual se pertence).
- **Se pid < 1**: o sinal é enviado a todos os processos para os quais o ID do grupo de processos (pgid) é igual ao valor absoluto de *pid*.

Tratamento de Sinais

- Exemplo 1 - Tratando Sinais de Usuário SIGUSR:

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void funcao_sigusr1()
{
    printf("recebi sigusr1\n");  }
void funcao_sigusr2()
{
    printf("recebi sigusr2\n");  }

int main()
{
    signal(SIGUSR1, funcao_sigusr1);
    signal(SIGUSR2, funcao_sigusr2);
    kill(getpid(), SIGUSR1);
    kill(getpid(), SIGUSR2);
}
```

exemplo1.c

Tratamento de Sinais

- Exemplo 2 - Tratando o sinal SIGSEGV:

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void funcao_sigsegv() {
   printf("recebi segment fault. Vou morrer!!!\n");
   exit(1);
}

int main() {
  char *p;
  signal(SIGSEGV, funcao_sigsegv);
  /* vou forcar um segment fault */
  printf("%s", *p);
}
```

exemplo2.c

Tratamento de Sinais

- **SIGHUP** (***hangup)***:
  - Sinal emitido aos processos associados a um terminal quando este se "desconecta"
- Pode ser um problema se o usuário deseja que um processo continue a ser executado após o fim de sua seção (aplicação durável).
- Se não for tratado, a aplicação será encerrada no momento em que o usuário fechar sua seção

Tratamento de Sinais

- **SIGHUP** (***hangup)***:
- Como resolver
  1. Utilizar o comando shell ***at*** ou ***@*** que permite de lançar uma aplicação numa certa data, via um processo do sistema, denominado ***deamon***. Neste caso, o sinal SIGHUP não terá nenhuma influência sobre o processo, uma vez que ele não está ligado a nenhum terminal.
  2. Incluir no código da aplicação a recepção do sinal SIGHUP.
  3. Executar o programa em ***background &*** (um processo executado em ***background*** trata automaticamente o sinal SIGHUP)
  4. Executar a aplicação associada ao comando *nohup*, que provocará uma chamada a primitiva *trap*, e que redireciona à saída padrão sobre *nohup.out*.

Tratamento dos processos zumbis

- O sinal SIGCLD se comporta diferentemente dos outros. Se ele for ignorado, a terminação de um processo filho, sendo que o processo pai não está em espera, não irá acarretar a criação de um processo zumbi.
  Exemplo: O programa a seguir gera um processo zumbi quando o pai é informado da morte do filho por meio do sinal SIGCLD.

```c
#include <stdio.h>
#include <unistd.h>
int main() {
    if (fork() != 0) while(1) ;
    return(0);
}
```

```
# ./test_sigcld &
# ps
```

Tratamento dos processos zumbis

- No programa a seguir o pai ignora o sinal SIGCLD, e seu filho não vai mais se tornar um processo zumbi.

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
int main() {
    signal(SIGCLD,SIG_IGN) ;/* ignora o sinal SIGCLD */
    if (fork() != 0)
        while(1);
    return(0);
}
```

Tratamento dos processos zumbis

- Zumbis temporários

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

int main(void)
{
    pid_t pids[10];
    int i;

    for (i = 9; i >= 0; --i) {
        pids[i] = fork();
        if (pids[i] == 0) {
            sleep(i+1);
            _exit(0);
        }
    }
       for (i = 9; i >= 0; --i)
        waitpid(pids[i], NULL, 0);
    return 0;
}
```

zombie_temp.c

Envio de Sinais (ALARM)

- ***Alarm()*** - A primitiva ***alarm()*** envia um sinal SIGALRM ao processo chamando após um intervalo de tempo ***secs*** (em segundos) passado como argumento, depois reinicia o relógio de alarme. Na chamada da primitiva, o relógio é reiniciado a ***secs*** segundos e é decrementado até 0.
- Somente um alarme por processo
- O tratamento do sinal deve estar previsto no programa, senão o processo será finalizado ao recebê-lo.
- **Sintaxe**:

```
#include <unistd.h>

unsigned int alarm(unsigned int secs);
```

- **Retorno**:
  - tempo restante no relógio se já existir um alarme armado anteriormente
  - 0 se não houver alarme

Envio de Sinais (ALARM)

alarm1.c

```c
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
void int_trata_alarme(int sig) { /* rotina executada na recepcao
de SIGALRM */
    printf("recepcaao do sinal %d :SIGALRM\n",sig);
}
int main() {
    unsigned sec;
    signal(SIGALRM,int_trata_alarme); /* interceptacao do sinal */
    printf("Fazendo alarm(5)\n");
    sec = alarm(5);
    printf("Valor retornado por alarm: %d\n",sec) ;
    printf("Principal em loop infinito (CTRL+c para acabar)\n") ;
    while(1) {
        sleep(6);
        alarm(3);
    }
}
```

Envio de Sinais (ALARM)

PARTE 1/2 alarm2.c

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
void it_horloge(int sig) /* tratamento do desvio sobre SIGALRM */
{
  printf("recepcao do sinal %d : SIGALRM\n",sig) ;
  printf("atencao, o principal reassume o comando\n") ;
}

void it_quit(int sig) /* tratamento do desvio sobre SIGALRM */
{
  printf("recepcao do sinal %d : SIGINT\n",sig) ;
  printf("Por que eu ?\n") ;
}
```

Envio de Sinais (ALARM)

PARTE 2/2 alarm2.c

```
int main() {
  unsigned sec;

  signal(SIGINT,it_quit); /* interceptacao do ctrl-c */
  signal(SIGALRM,it_horloge); /* interceptacao do sinal de alarme */

   printf("Armando o alarme para 10 segundos\n");
  sec=alarm(10);

  printf("valor retornado por alarm: %d\n",sec) ;
  printf("Paciencia... Vamos esperar 3 segundos com sleep\n");

  sleep(3) ;
  printf("Rearmando alarm(5) antes de chegar o sinal precedente\n");

  sec=alarm(5);
  printf("novo valor retornado por alarm: %d\n",sec);
  printf("Principal em loop infinito (ctrl-c para parar)\n");

  for(;;);
}
```

Envio de Sinais (ALARM)

- **Observação**: A interceptação do sinal só tem a finalidade de fornecer uma maneira elegante de sair do programa, ou em outras palavras, de permitir um redirecionamento da saída padrão para um arquivo de resultados.

- Pode-se notar que o relógio é reinicializado para o valor de 5 segundos durante a segunda chamada de alarm(), e que mais ainda, o valor retornado e o estado atual do relógio. Finalmente, pode-se observar que o relógio e decrementado ao longo do tempo. As duas últimas linhas da execução são geradas após um sinal CTRL+c do teclado.

- Observação: A função sleep() chama a primitiva alarm(). Deve-se então utilizá-la com maior prudência se o programa já manipula o sinal SIGALRM.

Envio de Sinais (ALARM)

alarm_clock.c

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

void alarm_handler(int signum){
    printf("Buzz Buzz Buzz\n");

    alarm(1); //Novo alarme de 1 segundo
}
int main(){

    signal(SIGALRM, alarm_handler); //Configura o tratamento do
alarme

    alarm(1); // Define o Alarme inicial

    while(1){
        pause(); // Aguarda um sinal
    }
}
```

Envio de Sinais (ALARM)

```c
void sigquit_handler(int signum){
    printf("Alarme Desligao\n");

    //Desliga todos os Alarmes
    alarm(0);

    // Redefine o tratamento de SIGINT
    // Agora Ctrl-C termina o programa
    signal(SIGINT, SIG_DFL);
}

void sigint_handler(int signum){
   printf("Alarme adiado em 5 segundos!
\n");

    //schedule next alarm for 5 seconds
    alarm(5);
}

void alarm_handler(int signum){
    printf("Buzz Buzz Buzz\n");

    //set a new alarm for 1 second
    alarm(1);
}
```

alarm_clock_completo.c

```c
int main(){

    //Define o tratamento do alarme
    signal(SIGALRM, alarm_handler);

    //Define o Tratamento de SIGINT
    signal(SIGINT, sigint_handler);

    //Define o Tratamento de SIGQUIT
    signal(SIGQUIT, sigquit_handler);

    //Define o primeiro alarme
    alarm(1);

    //Pausa no Loop
    while(1){
        pause();
    }
}
```

Envio de Sinais (PAUSE)

- ***Pause()*** - A primitiva ***pause()*** corresponde a uma espera simples. Ela não faz nada, nem espera nada de particular. Entretanto, uma vez que a chegada de um sinal interrompe toda primitiva bloqueada, pode-se dizer que ***pause()*** espera simplesmente a chegada de um sinal.
- Observe o comportamento de retorno clássico de um primitiva bloqueada, isto é o posicionamento de errno em EINTR. Note que, geralmente, o sinal esperado por `pause()` é o relógio de `alarm()`
- **Sintaxe**:

```
#include <unistd.h>

int pause(void)
```

- **Retorno**:
  - sempre = -1

Sleep com Alarm() e Pause();

sleep_pause_alarm.c

```c
void nullfcn() /* define-se aqui uma funcao executada quando */
{ } /* o sinal SIGALRM e interceptado por signal() */
/* esta funcao nao faz absolutamente nada */
void sleep2(int secs) /* dorme por secs segundos */
{
    if( signal(SIGALRM,nullfcn) )
    {
        perror("error: reception signal") ;
        exit(-1) ; }
    alarm(secs) ; /* inicializa o relogio a secs segundos */
    pause(); /* processo em espera por um sinal */
}
int main() /* so para testar sleep2() */
{
    if(fork()==0)
    {
        sleep(3) ;
        printf("hello, sleep\n") ;
    }
    else /* pai */
    {
        sleep2(3) ;
        printf("hello, sleep2\n") ;
    }
    return 0;
}
```

Envio de Sinais (SLEEP)

- ***sleep()*** - A primitiva ***sleep()*** bloqueia o processo por uma determinada quantidade de ***segundos***
- **Sintaxe**:

```
#include <unistd.h>

unsigned int sleep (unsigned int seconds);
```

- **Retorno**:
  - 0 se o tempo especificado passou
  - Tempo restante se tiver sido interrompida por um sinal