



Laboratório de Pesquisa em Redes e Multimídia

Sinais no UNIX



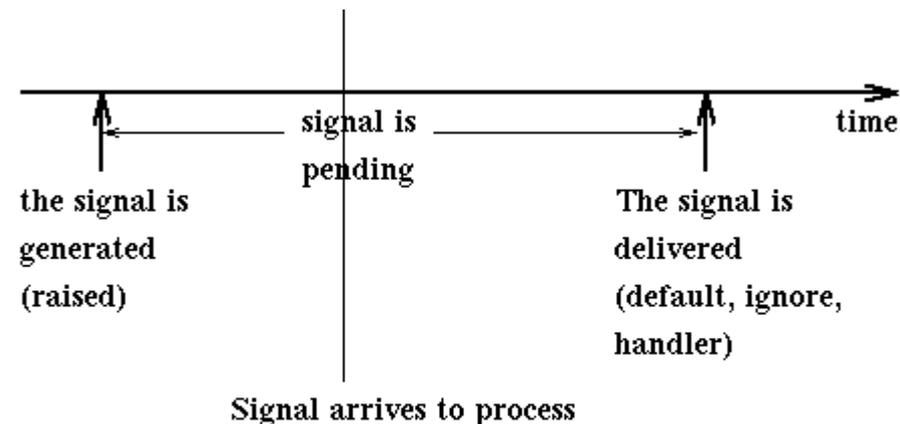
Universidade Federal do Espírito Santo
Departamento de Informática

Modelo de Eventos

- Os processos de nível usuário interagem com o *kernel* por meio de chamadas de sistema
- Entretanto, acontecimentos esporádicos, assíncronos, designados por eventos, levam à necessidade do *kernel* interagir com o usuário. Por exemplo:
 - Situações criadas pelo usuário (ex: término de um temporizador/alarmes, acionamento de uma combinação de teclas/Ctrl-C)
 - Situações geradas por determinadas chamadas ao sistema (ex: término de um processo filho gera um aviso ao processo pai)
- Má alternativa: obrigar todos os programas a verificar periodicamente a ocorrência dessas situações
- A ideia, então, é associar uma rotina para tratar o evento

Modelo de Sinais do Unix (1)

- No Unix, um sinal é uma notificação de software a um processo informando a ocorrência de um evento.
- Neste modelo, um sinal é **gerado** pelo S.O. quando o evento acontece. Ex:
 - Término de um temporizador: sinal SIGALRM é gerado.
 - Acionamento de Ctrl-C: sinal SIGINT é gerado.
- Um sinal é **entregue** quando o processo reage ao sinal, ou seja, executa alguma ação baseada no sinal (ele executa um *tratador do sinal* - *signal handler*).
- Um sinal é dito estar **pendente** se foi gerado mas ainda não entregue.



Modelo de Sinais do Unix (2)

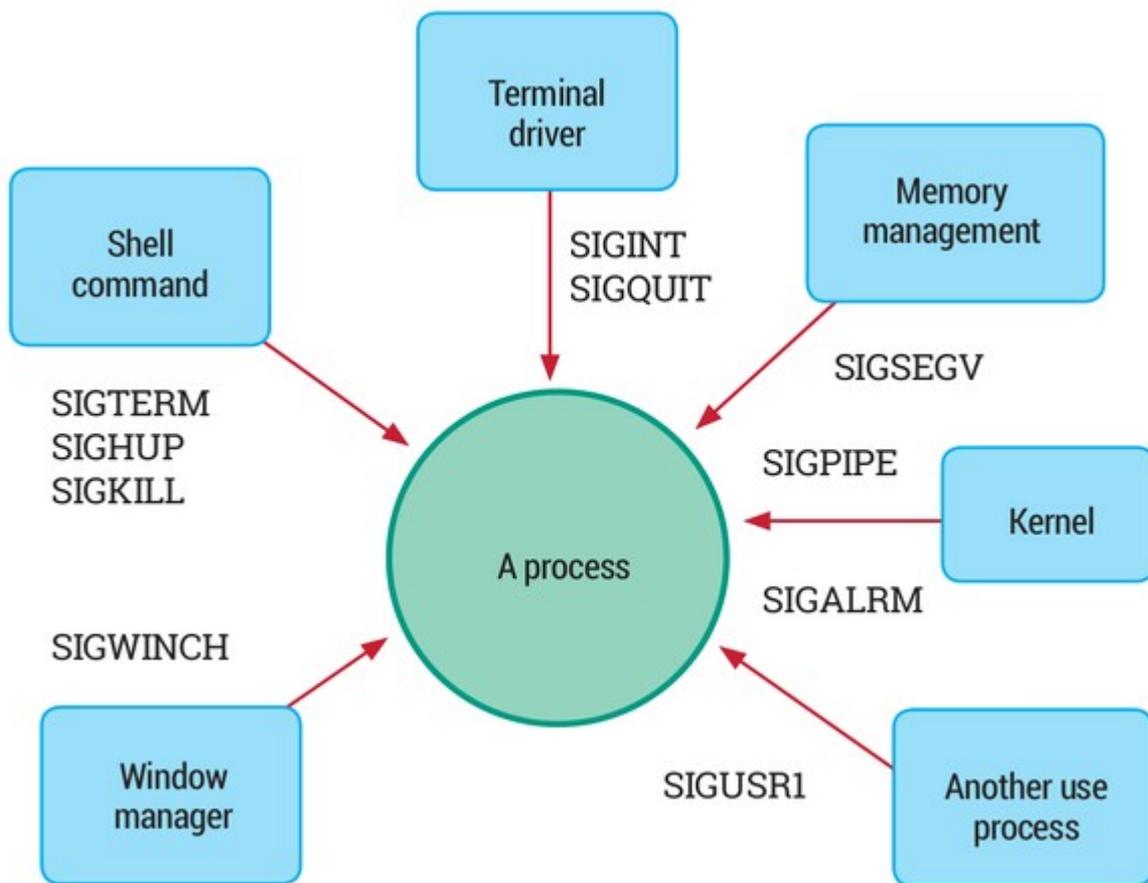
- Na entrega de um sinal, pode ocorrer:
 - (1) **Tratamento default** (definido pelo kernel)
 - (2) **Capturado**: neste caso, é executada a função definida no tratador do sinal definido pelo usuário (**handler**).
 - (3) **Ignorado**: neste caso, nada acontece. Funciona para todos os sinais

*Obs: Nos sinais SIGKILL e SIGSTOP **sempre é executada a ação default**, (eles não podem ser capturados nem ignorados)*

Tipos de Sinais (1)

- O Unix (Posix) define códigos inteiros para um número fixo (30) de sinais (64 no Linux-64bits)
- Cada um é caracterizado por um nome simbólico iniciado com SIG
- O usuário não pode definir novos sinais
 - Unix disponibiliza dois sinais (SIGUSR1 e SIGUSR2) para o usuário utilizar como bem entender
- Alguns sinais são gerados em condições de erro
 - ex: SIGFPE ou SIGSEGV
- ... enquanto outros são gerados por chamadas específicas do S.O.
 - Ex: alarm(), abort() e kill()
- Sinais também são gerados por comandos *shell*
 - kill

Tipos de Sinais (2)



Tipos de Sinais (3)

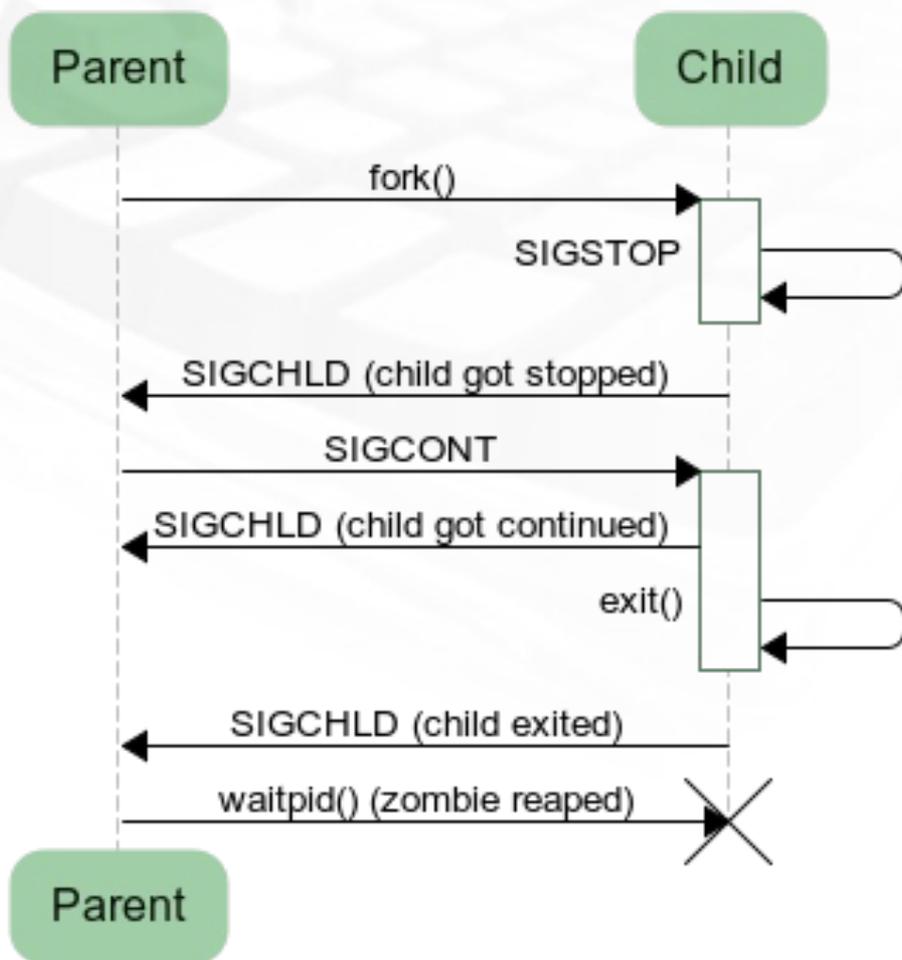
| Nome | Descrição | Origem | Ação Default |
|---------|-------------------------------------|-----------------------|--------------|
| SIGABRT | Terminação anormal | abort () | Terminar |
| SIGALRM | Alarme | alarm () | Terminar |
| SIGCHLD | Filho terminou ou foi suspenso | S.O. | Ignorar |
| SIGCONT | Continuar processo suspenso | S.O. shell (fg, bg) | Continuar |
| SIGFPE | Excepção aritmética | hardware | Terminar |
| SIGILL | Instrução ilegal | hardware | Terminar |
| SIGINT | Interrupção | teclado (^C) | Terminar |
| SIGKILL | Terminação (<i>non catchable</i>) | S.O. | Terminar |
| SIGPIPE | Escrever num <i>pipe</i> sem leitor | S.O. | Terminar |
| SIGQUIT | Saída | teclado (^) | Terminar |
| SIGSEGV | Referência a memória inválida | hardware | Terminar |
| SIGSTOP | Stop (<i>non catchable</i>) | S.O. (shell - stop) | Suspender |
| SIGTERM | Terminação | teclado (^U) | Terminar |
| SIGTSTP | Stop | teclado (^Y, ^Z) | Suspender |
| SIGTTIN | Leitura do teclado em <i>backgd</i> | S.O. (shell) | Suspender |
| SIGTTOU | Escrita no écran em <i>backgd</i> | S.O. (shell) | Suspender |
| SIGUSR1 | Utilizador | de 1 proc. para outro | Terminar |
| SIGUSR2 | Utilizador | idem | Terminar |

Geração de Sinais

- Exceções de hardware
 - Ex: uma divisão por zero gera o sinal SIGFPE, uma violação de memória gera o sinal SIGSEGV.
- Condições de software
 - Ex: o término de um temporizador criado com a função `alarm()` gera o sinal SIGALRM.
- A partir do *shell*, usando o comando `<kill>`
 - Ex: `% kill -USR1 1234` (envia o sinal SIGUSR1 para o processo cujo PID é 1234)
- Usando a função `kill()`
 - Ex: `if (kill(3423,SIGUSR1)==-1) perror("Failed to send the SIGUSR1 signal");`
- Por meio de uma combinação de teclas (interrupção via terminal)
 - Ex: `Ctrl-C=INT (SIGINT)`, `Ctrl-|=QUIT (SIGQUIT)`, `Ctrl-Z=SUSP (SIGSTOP)`, `Ctrl-Y=DSUSP (SIGCONT)`
 - OBS: o comando `stty -a` lista as características do device associado com o *stdin*. Dentre outras coisas, ele associa sinais aos caracteres de controle acima listados.
- No controle de processos
 - Ex: Processo *background* tentando escrever no terminal gera o sinal SIGTTOU, que muda o estado do processo para STOPPED.

No controle de processos

Pais e Filhos (SIGCHLD)



`waitpid(..., ..., WUNTRACED)`

No controle de processos

Gerando core dump... (testem em casa!)

Primeiramente é necessário configurar o tamanho do core

```
ulimit -c unlimited
```

Crie um arquivo que gere erro de overflow (dump.c)

```
... int i = 3/0;
```

Compile o arquivo com a opção de debugging

```
gcc -g -o ex1 ex1.c
```

Execute o programa e em seguida acesse o core gerado

```
gdb ex1 core
```

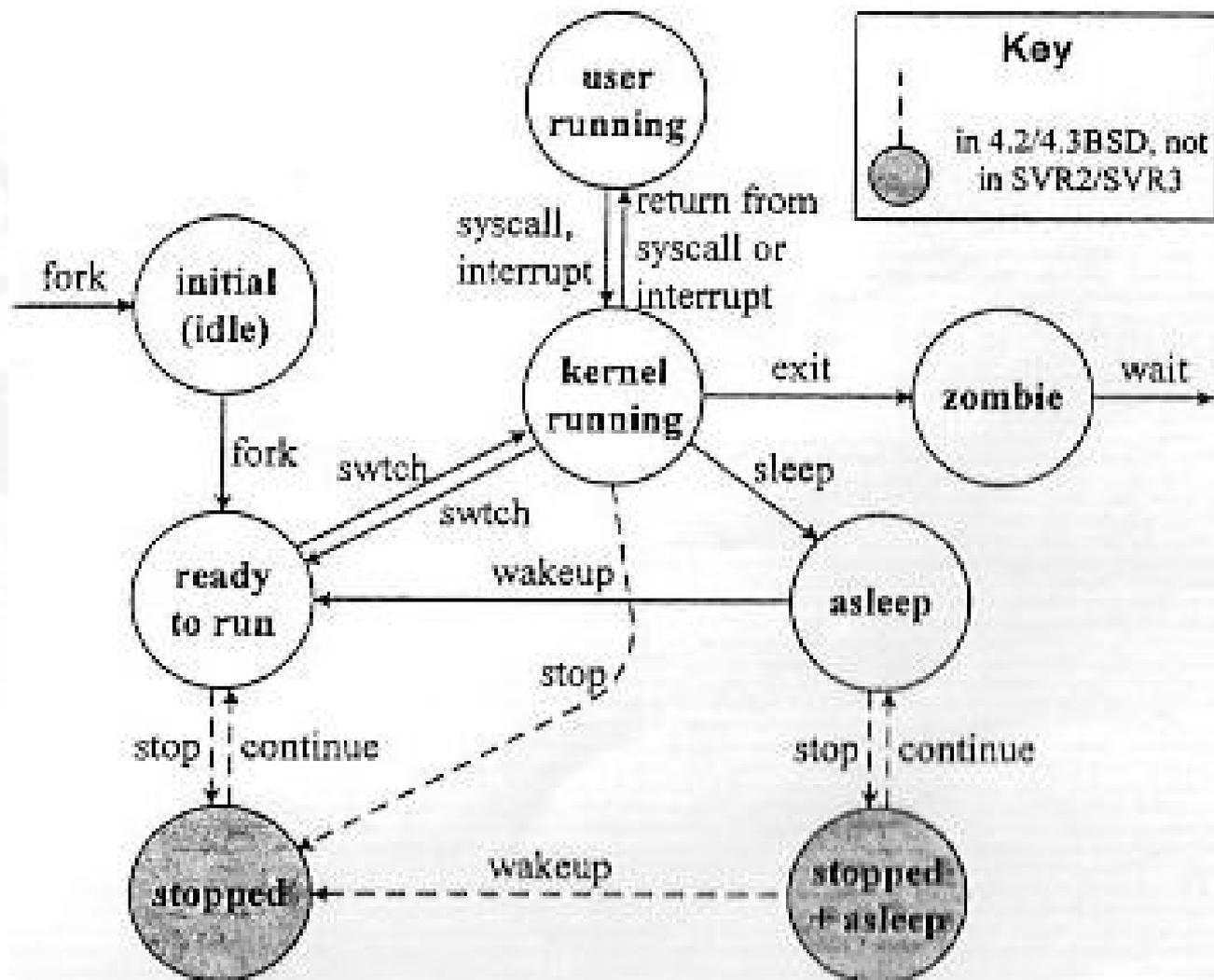


Figure 2-3. Process states and state transitions.

Enviando Sinais: O Comando `kill()`

- O comando `kill` permite o envio de sinais a partir da *shell*.

- Formato:

```
<kill -s pid>
```

- Exemplos:

```
$ kill -9 3423
```

```
$ kill -USR1 3423
```

```
$ kill -l //lista sinais disponíveis
```

- Valores numéricos de sinais:

SIGHUP(1), SIGINT(2), SIGQUIT(3), SIGABRT(6), SIGKILL(9), SIGALRM(14), SIGTERM(15)

Enviando Sinais: a SVC `kill()` ⁽¹⁾

- A função `kill()` é usada dentro de um programa para enviar um sinal para outros processos.

```
#include <sys/types.h>  
#include <signal.h>  
  
int kill(pid_t pid, int sig);
```
- O 1º parâmetro identifica o processo alvo, o segundo indica o sinal.
 - Se `pid>0`: sinal enviado para processo indicado por PID
 - Se `pid=0`: sinal enviado para os processos do grupo do remetente.
 - Se `pid=-1`: sinal enviado para todos os processos para os quais ele pode enviar sinais (depende do UID do usuário).
 - **Se `pid<0` (exceto -1): sinal é enviado para todos os processos com `GroupID` igual a `|pid|`.**

Enviando Sinais: a SVC `kill()` ⁽²⁾

- Se sucesso, `kill` retorna 0. Se erro, retorna -1 e seta a variável `errno`.

| errno | cause |
|--------------|---|
| ESRCH | no process or process group corresponds to pid |
| EPERM | caller does not have the appropriate privileges |
| EINVAL | sig is an invalid or unsupported signal |

- Um processo pode enviar sinais a outro processo apenas se tiver autorização para fazê-lo:
 - Um processo com UID de root pode enviar sinais a qualquer outro processo.
 - Um processo com UID distinto de root apenas pode enviar sinais a outro processo se o *real* UID (ou *effective*) do processo for igual ao *real* UID (ou *effective*) do processo destino.

Enviando Sinais: a SVC `kill()` ⁽³⁾

- Exemplo: enviar SIGUSR1 ao processo 3423

```
if (kill(3423, SIGUSR1) == -1)
    perror("Failed to send the SIGUSR1 signal");
```
- Exemplo: um filho “mata” seu pai

```
if (kill(getppid(), SIGTERM) == -1)
    perror ("Failed to kill parent");
```
- Exemplo: enviar um sinal para si próprio

```
if (kill(getpid(), SIGABRT))
    exit(0);
```

Enviando Sinais: a SVC `kill()`⁽⁴⁾

Exemplo (arquivo `testa_sinais_1.c`)

```
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
int main(void) {
    pid_t pid = fork();
    if (pid == 0) {
        for (;;) {
            printf("pid=%ld\n", getpid());
            sleep(1); }
    }
    else {
        sleep(5);
        kill(pid, SIGTERM); return 0; }
}
```

Enviando Sinais: a `SVC raise()`

- Permite a um processo enviar um sinal para si mesmo. A resposta depende da opção que estiver em vigor para o sinal enviado

```
#include <signal.h>
int raise(int sig);
```

- Exemplo :

```
if (raise(SIGUSR1) != 0)
    perror("Failed to raise SIGUSR1");
```

A SVC alarm()

- A função `alarm()` envia o sinal `SIGALRM` ao processo chamador após decorrido o número especificado de segundos.
- Formato:

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```
- Se no momento da chamada existir uma outra chamada prévia a `alarm()`, a antiga deixa de ter efeito sendo substituída pela nova.
 - `alarm()` retorna 0 normalmente, ou o número de segundos que faltavam a uma possível chamada prévia a `alarm()`.
- Para cancelar uma chamada prévia a `alarm()` sem colocar uma outra ativa pode-se executar `alarm(0)`.
- Se ignorarmos ou não capturarmos `SIGALRM`, a ação *default* é terminar o processo
- Exemplo:

```
void main(void){
    alarm(10); printf ("Looping forever...\n"); for(;;){} }
```

A SVC pause()

- A função `pause()` suspende a execução do processo. O processo só voltará a executar quando receber um sinal qualquer, não ignorado.
 - O serviço `pause()` só retorna se o tratador do sinal recebido também retornar.
- Formato:

```
#include <unistd.h>  
int pause();
```

Tratamento *Default* (1)

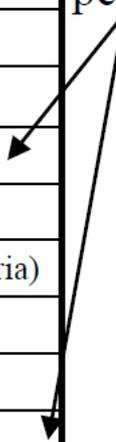
- Sinais apresentam um tratamento *default* de acordo com o evento:
 - **normal kill**
 - **abort**: geração de core dump e término do processo
 - **stop**: o processo é suspenso
 - **continue**: é retomada a execução do processo
- Os **usuários podem alterar o tratamento default**:
 - 1) definindo seus próprios tratadores
 - 2) ignorando o sinal (associando o tratador SIG_IGN ao sinal)
 - 3) bloqueando sinais temporariamente (o sinal se mantém pendente até que seja desbloqueado)
- Para recuperar o tratamento default, basta associar ao sinal o tratador SIG_DFL.
 - `signal(SIGALRM, SIG_DFL); signal(SIGINT, SIG_IGN)`
- SIG_DFL e SIG_IGN são tratadores pré-definidos do Unix.

Tratamento *Default* (2)

Ações Default (por omissão) de alguns sinais definidos no Linux

| Sinal | Código | Ação por omissão | Causa |
|----------------|--------|---------------------|--|
| SIGHUP | 1 | Termina | Terminal ou processo desconectado |
| SIGINT | 2 | Termina | Interrupção no teclado |
| SIGILL | 3 | Termina e gera core | Hardware (instrução ilegal) |
| SIGABRT | 6 | Termina e gera core | Gerado por instrução ABORT |
| <u>SIGKILL</u> | 9 | Termina | Força terminação do processo |
| SIGUSR1 | 10 | Termina | Definido pelo utilizador |
| SIGSEGV | 11 | Termina e gera core | Hardware (referência inválida a memória) |
| SIGALRM | 14 | Termina | Esgotamento do temporizador |
| SIGCHLD | 17 | Ignora | Processo filho termina |
| <u>SIGSTOP</u> | 19 | Suspende | Suspende processo |
| SIGSYS | 31 | Termina e gera core | Chamada inválida a função de sistema |

Tratados apenas pelo núcleo



Tratamento de Sinais (1)

- A SVC `signal()` permite especificar a ação a ser tomada quando um sinal particular é recebido. Em outras palavras, permite **registrar um tratador para o sinal** (signal handler).

```
typedef void (*sighandler_t)(int);  
sighandler_t signal(int signum, sighandler_t handler);
```
- O primeiro parâmetro identifica o código do sinal a tratar. A ação a ser tomada depende do valor do segundo parâmetro:
 - 1) se igual à `SIG_IGN`: o sinal não tem efeito, ele é ignorado;
 - 2) se igual a `SIG_DFL`: é executada a ação/tratador *default* definida pelo kernel para o sinal (p.ex: terminar o processo).
 - 3) se igual à referência de uma função de tratamento, ela é executada (nesse caso, dizemos que o sinal foi capturado);
- *OBS: `signal()` retorna o endereço da função anterior que tratava o sinal.*

Tratamento de Sinais (2)

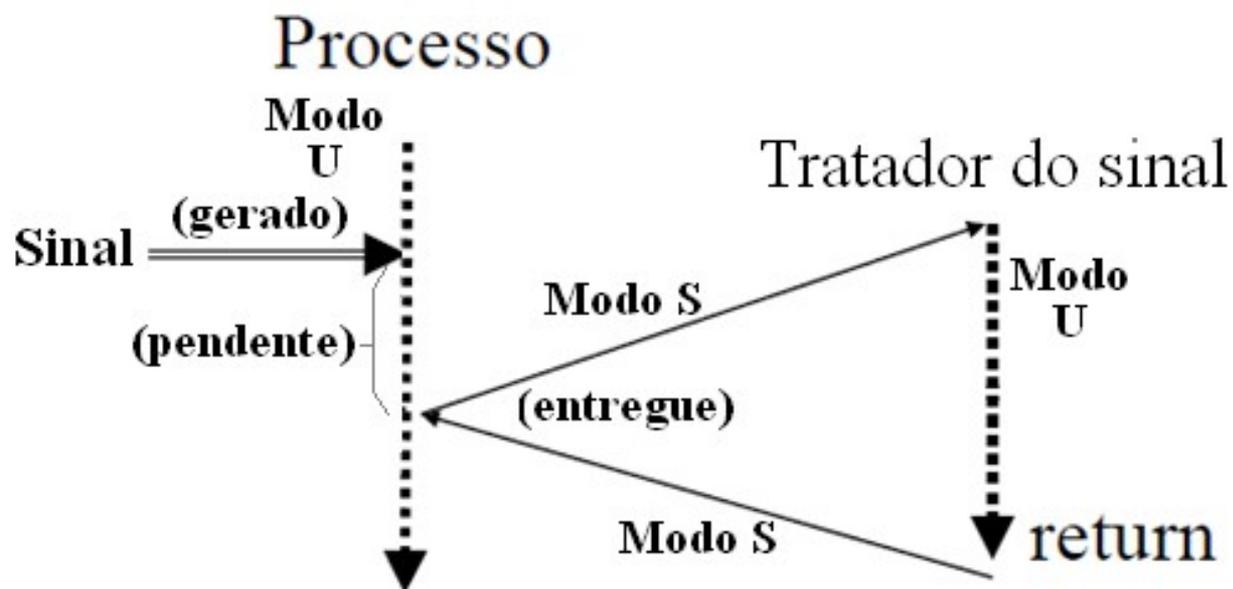
- Procedimento geral para capturar um sinal:

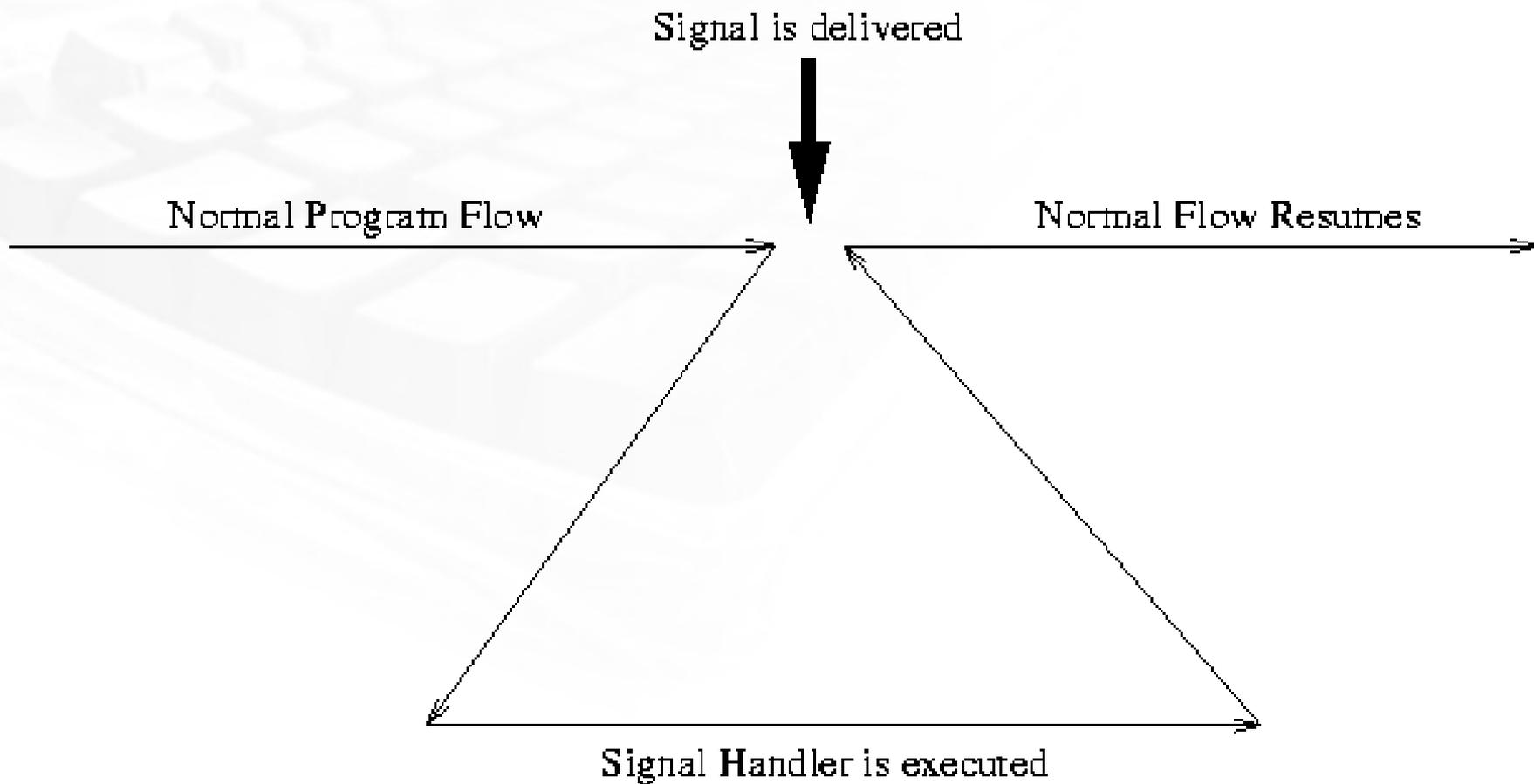
- Escrevemos um tratador para o sinal (p.ex., para o sinal SIGSEGV) ...

```
void trata_SIGSEGV(int signum) {  
    ...  
}
```

- ... e depois instalamos o tratador com a função `signal()`

```
signal(SIGSEGV, trata_SIGSEGV);
```





Tratamento de Sinais (3)

- Como tratar erros deste tipo?

```
int *px = (int*) 0x01010101;  
*px = 0;
```

- Programa recebe um sinal SIGSEGV
- O comportamento padrão é terminar o programa

- E erros deste tipo?

```
int i = 3/0;
```

- Programa recebe um sinal SIGFPE
- O comportamento padrão é terminar o programa

Exemplo 1

```
/* Imprime uma mensagem quando um SIGSEGV é recebido * e restabelece
o tratador padrão. */
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void trata_SIGSEGV(int signum) {
    printf("Acesso indevido `a memória.\n");
    printf("Nao vou esconder este erro. :-)\n");
    signal(SIGSEGV, SIG_DFL);
    raise(SIGSEGV); /* equivale a kill(getpid(), SIGSEGV); */
}

int main() {
    signal(SIGSEGV, trata_SIGSEGV);
    int *px = (int*) 0x01010101;
    *px = 0;
    return 0;
}
```

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

void trata_SIGUSR1(int sig) {
    printf("Tratando SIGUSR1.\n");
}

void trata_SIGUSR2 (int sig) {
    printf("Tratando SIGUSR2.\n");
    raise(SIGUSR1);
    printf("Fim do SIGUSR2.\n");
}

int main (void) {

    signal (SIGUSR1, trata_SIGUSR1);
    signal (SIGUSR2, trata_SIGUSR2);

    raise(SIGUSR2);

    sleep(2);
    return 0;
}
```

Exemplo 2

Sinais encadeados
(arquivo testa_sinais_2.c)

Exemplo 3

programa que trata os sinais do usuário (arquivo testa_sinais_3.c)

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

void tratamento(int sigNumb) {
    if (sigNumb==SIGUSR1) printf("Gerado SIGUSR1\n");
    else if (sigNumb==SIGUSR2) printf("Gerado SIGUSR2\n");
    else printf("Gerado %d\n",sigNumb); }

int main() {
    if (signal(SIGUSR1,tratamento)==SIG_ERR)
        printf("Erro na ligacao USER1\n");
    if (signal(SIGUSR2,tratamento)==SIG_ERR)
        printf("Erro na ligacao USER2\n");
    for(;;) pause(); }
```

Exemplo 4 Definindo um tratador para o sinal SIGALRM

```
#include <stdio.h>
#include <signal.h>
int alarmFlag = 0 /* Global alarm flag */
alarmHandler(); /* Forward declaration of alarm handler */
/*****/
main() {
    signal (SIGALRM, alarmHandler); /* Install signal handler */
    alarm(3); /* Schedule an alarm signal in three seconds */
    printf("Looping...\n");
    while (!alarmFlag) /* Loop until flag set */
    {
        pause(); /* Wait for a signal */
    }
    printf("Loop ends due to alarm signal\n");
}
/*****/
alarmHandler() {
    printf("An alarm clock signal was received\n");
    alarmFlag = 1
}
```

Exemplo 4 (cont.)

```
$handler.exe ...run the program
Looping...
An alarm clock signal was received ...occurs 3 seconds later
Loop ends due to alarm signal
$
```

Exemplo 5 programa que protege código inibindo e liberando o Ctrl-C (arquivo testa_sinais_5.c)

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

main() {
    int (*oldHandler) ();      /* To hold old handler value */

    printf("I can be Control-C'ed\n");
    sleep(3);

    oldHandler = signal (SIGINT, SIG_IGN); /* Ignore Control-C */
    printf("I'm protected from Control-C now\n");
    sleep(10);

    signal (SIGINT, oldHandler); /* Restore old handler */
    printf("I can be Control-C'ed again\n");
    sleep(10);
    printf("Bye!\n");
}
```

Exemplo 6

Pai exibe mensagem
na morte do filho
(arquivo testa_sinais_6.c)

```
void morte_filho(int n){
printf("morreu o meu filho %d eu sou %d\n",
wait(NULL), getpid());
}

int main() {
//signal(SIGCHLD, morte_filho);
//signal(SIGCHLD, SIG_DFL);
//signal(SIGCHLD, SIG_IGN);
pid_t pid=fork();
if (pid==0) {
if(fork() == 0){
printf("neto..... pid=%d\n",getpid());
sleep(1);
exit(0);
}else{
printf("FILHO... pid=%d\n",getpid());
sleep(30);
exit(0);
}
}
else {
while(1);
exit(0);
}
}
```

Exercício (fonte: Prof. Eduardo Zambon)

Utilizando as funções `kill()` e `signal()`, escreva um programa que cria um processo filho e a seguir tanto o processo pai como o filho ficam em loop infinito incrementando a sua própria cópia de um contador global `i`. Os processos não exibem esse contador na tela durante o loop. Dessa forma você terá um modelo típico de processos orientados a CPU. Instale um handler para o sinal `SIGINT` de forma que um comando de `Ctrl+C` mata ambos os processos. O handler deve exibir na tela o valor do contador de cada processo no momento que ele for terminado. Um possível exemplo de saída fica como abaixo:

```
$ ./ex3
[PARENT]: PID 20019, starts counting
[CHILD]: PID 20020, starts counting
^CProcess 20020 killed, i = 532634516.
Process 20019 killed, i = 530995662.
```

Exemplo 7

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
int delay;
void childhandler(int signo);
void main(int argc, char *argv[])
{
    pid_t pid;
    signal(SIGCHLD, childhandler); /* quando um processo filho terminar envie ao pai o sinal SIGCHLD */
    pid = fork();
    if (pid = 0) /* filho */
        execvp(argv[2], &argv[2]);
    else { /* pai */
        sscanf(argv[1], "%d", &delay); /* transforma string em valor */
        sleep(delay);
        printf("Program %s exceeded limit of %d seconds!\n", argv[2], delay);
        kill(pid, SIGKILL);
    }
}

void childhandler(int signo)
{
    int status;
    pid_t pid;
    pid = wait(&status);
    printf("Child %d terminated within %d seconds.\n", pid, delay);
    exit(0);
}
```

Programa que lança um outro e espera um certo tempo para que o segundo termine. Caso isso não aconteça, deverá terminá-lo de modo forçado.

Exemplo de linha de comando:

```
limit n prog arg1 arg2 arg3
```

n - nº de segundos a esperar

prog - programa a executar

arg1, arg2, ..., argn - argumentos de prog

Bloqueando Sinais (1)

- Um processo pode bloquear temporariamente um sinal, impedindo a sua entrega para tratamento.
- Um processo bloqueia um sinal alterando a sua *máscara de sinais*. Essa é uma estrutura que contém o conjunto corrente de sinais bloqueados do processo.
- Quando um processo bloqueia um sinal, uma ocorrência deste sinal é guardada (mantida) pelo *kernel* até que o sinal seja desbloqueado.

Bloqueando Sinais (2)

- Por que bloquear sinais?
- Uma aplicação pode desejar temporariamente ignorar alguns sinais (ex: evitar Ctrl-C);
- Evitar condições de corrida quando um sinal ocorre no meio do tratamento de outro sinal
- Nota:
 - Não se pode confundir bloquear um sinal com ignorar um sinal. Um **sinal ignorado** é sempre entregue para tratamento mas o tratador a ele associado (SIG_IGN) não faz nada com ele, simplesmente o descarta.

Observações

- Depois de um `fork()` o processo filho herda as configurações de sinais do pai
- Depois de um `exec()` **sinais previamente ignorados permanecem ignorados** mas os tratadores instalados (para sinais capturados) são resetados, voltando ao tratador *default*.
- Portabilidade
 - O padrão POSIX 1003.1 define a interface, mas não regulariza a implementação.
 - SVR2: mecanismo pouco confiável (defeituoso) de notificação de sinais
 - 4.3BSD: mecanismo robusto, mas incompatível com a interface SV
 - SVR4: compatível com POSIX, incorporou várias características do BSD. É compatível com versões mais antigas de SV.
- Banda limitada:
 - Apenas 32 sinais no SVR4 e 4.3BSD, e 64 no AIX e Linux.
- Podem ser usados como mecanismo de comunicação?
 - Sinais são caros porque o emissor tem que fazer *syscall*.
 - Sinais não são empilhados.
 - A resposta não é “imediate”

Máscara de Sinais (1)

- A máscara de sinais bloqueados é definida pelo tipo de dados `sigset_t`. É uma tabela de bits, cada um deles correspondendo a um sinal.

| | | | | | |
|--------|---------|---------|-----|---------|---------|
| SigInt | SigQuit | SigKill | ... | SigCont | SigAbrt |
| 0 | 0 | 1 | ... | 1 | 0 |

- Como bloquear um sinal?
 - Um conjunto de sinais : `sigprocmask()`
 - Um sinal específico : `sigaction()`

Mais detalhes a seguir...

Manipulando as Máscaras (1)

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

- Inicializa a máscara como vazia (sem nenhum sinal)
`int sigemptyset(sigset_t *set);`
- Preenche a máscara com todos os sinais suportados no sistema
`int sigfillset(sigset_t *set);`
- Adiciona um sinal específico à máscara de sinais bloqueados
`int sigaddset(sigset_t *set, int signo);`
- Remove um sinal específico da máscara de bloqueados
`int sigdelset(sigset_t *set, int signo);`
- Testa se um sinal pertence à máscara de bloqueados
`int sigismember(const sigset_t *set, int signo);`

Manipulando as Máscaras (2)

```
sigemptyset // criar uma máscara vazia
```

```
sigaddset //manipular a máscara criada
```

```
sigfillset
```

```
sigdelset
```

```
sigprocmask // alterar a máscara de sinais
```

```
// bloqueados do processo
```

Manipulando as Máscaras (3)

- Exemplo: inicializar um conjunto de sinais

```
sigset_t twosigs;
```

```
if ((sigemptyset(&twosigs) == -1) ||  
    (sigaddset(&twosigs, SIGINT) == -1) ||  
    (sigaddset(&twosigs, SIGQUIT) == -1))  
    perror("Failed to set up signal set");
```

Manipulando as Máscaras (4)

- Tendo construído uma máscara contendo os sinais que nos interessam, podemos bloquear (ou desbloquear) esses sinais usando o serviço `sigprocmask()`.

```
#include <signal.h>
int sigprocmask(int how,
                const sigset_t *restrict set,
                sigset_t *restrict oset);
```

- Se `set` for diferente de `NULL` então a máscara corrente é modificada de acordo com o parâmetro `how`:
 - `SIG_SETMASK`: substitui a máscara atual, que passa a ser dada por `set`.
 - `SIG_BLOCK`: bloqueia os sinais do conjunto `set` (adiciona-os à máscara atual)
 - `SIG_UNBLOCK`: desbloqueia os sinais do conjunto `set`, removendo-os da máscara atual de sinais bloqueados
- Se `oset` é diferente de `NULL` a máscara anterior é retornada em `oset`.

Manipulando as Máscaras (5)

- Exemplo 1 (adiciona SIGINT ao conjunto de sinais bloqueados de um processo)

```
sigset_t newsigset;
```

```
if ((sigemptyset(&newsigset) == -1) ||  
    (sigaddset(&newsigset, SIGINT) == -1))  
    perror("Failed to initialize the signal set");  
else if (sigprocmask(SIG_BLOCK, &newsigset, NULL) == -1)  
    perror("Failed to block SIGINT");
```

Manipulando as Máscaras (6)

- Exemplo 2

...

```
sigemptyset(&intmask);  
sigaddset(&intmask, SIGINT);  
sigprocmask(SIG_BLOCK, &intmask, NULL);
```

...

```
/* se Ctrl-c é pressionado enquanto este código está  
   executando, o sinal SIGINT é bloqueado*/
```

...

```
sigprocmask(SIG_UNBLOCK, &intmask, NULL);
```

```
/* se Ctrl-c é pressionado enquanto este código está  
   executando, ou durante a execução do código acima, o  
   sinal é tratado aqui*/;
```

Manipulando as Máscaras (7)

■ Exemplo 3

...

```
sigfillset(&blockmask);
```

```
sigprocmask(SIG_SETMASK, &blockmask, &oldset);
```

...

```
/* todos os sinais estão bloqueados neste trecho*/
```

...

```
sigprocmask(SIG_SETMASK, &oldset, NULL);
```

...

```
/* a antiga máscara de sinais é reestabelecida aqui */
```

Capturando Sinais com `sigaction()` (1)

- A norma POSIX estabelece um novo serviço para substituir `signal()`. Esse serviço chama-se `sigaction()`.
- Além de permitir examinar ou especificar a ação associada com um determinado sinal, ela também permite, ao mesmo tempo, bloquear outros sinais (dentre outras opções).

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

```
int sigaction(int signo, const struct sigaction *act,  
              struct sigaction *oact);
```

```
struct sigaction {  
    void (*sa_handler)(int); /* SIG_DFL, SIG_IGN ou endereço do  
                               tratador */  
    sigset_t sa_mask; /* sinais adicionais a bloquear durante a  
                       execução do tratador */  
    int sa_flags; /* opções especiais. Se NULL, sa_handler  
                  define a ação a ser executada */  
};
```

Capturando Sinais com `sigaction()` (2)

- Definindo o tratador `mysighand` para o sinal `SIGINT`

```
struct sigaction newact;

newact.sa_handler = mysighand; /* set the new handler */
newact.sa_flags = 0;          /* no special options */
if ((sigemptyset(&newact.sa_mask)== -1) || /*no other signals blocked*/
    (sigaction(SIGINT, &newact, NULL)==-1) )
    perror("Failed to install SIGINT signal handler");
```

- Definindo o tratador *default* (`SIG_DFL`) para o sinal `SIGINT`

```
struct sigaction newact;

newact.sa_handler = SIG_DFL; /* new handler set to default */
sigemptyset(&newact.sa_mask); /* no other signals blocked */
newact.sa_flags = 0;          /* no special options */
if (sigaction(SIGINT, &newact, NULL) == -1)
    perror("Could not set SIGINT handler to default action");
```

Capturando Sinais com `sigaction()` (3)

- Definindo um tratador que captura o sinal `SIGINT` gerado pelo `ctrl-c`

```
void catch_ctrl_c(int signo) {
    char handmsg[] = "I found Ctrl-C\n";
    int msglen = sizeof(handmsg);

    write(STDERR_FILENO, handmsg, msglen);
}

...
struct sigaction act;
act.sa_handler = catch_ctrl_c;
act.sa_flags = 0;

if ((sigemptyset(&act.sa_mask) == -1) ||
    (sigaction(SIGINT, &act, NULL) == -1))
    perror("Failed to set SIGINT to handle Ctrl-C");
```

A SVC `sigsuspend()` ⁽¹⁾

- Suponha que queiramos proteger uma região de código da ocorrência da combinação Ctrl-C e, logo a seguir, esperar por uma dessas ocorrências. Poderíamos ser tentados a escrever o seguinte código:

```
sigset_t newmask, oldmask;
...
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
sigprocmask(SIG_BLOCK, &newmask, &oldmask);

... /* região protegida */

sigprocmask(SIG_SETMASK, &oldmask, NULL);
pause();
...
```

- Este processo ficaria bloqueado se a ocorrência de CTRL-C aparecesse antes da chamada a `pause()`. Para solucionar esse problema o POSIX define a SVC `sigsuspend()`.

A SVC `sigsuspend()` (2)

- Formato:

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

```
int sigsuspend(const sigset_t *sigmask);
```

- `sigsuspend()` põe em vigor a máscara especificada em `sigmask` e bloqueia o processo até este receber um sinal. Após a execução do tratador e o retorno de `sigsuspend()` a máscara original é restaurada. Quando retorna, retorna sempre o valor `-1`.

- Versão correta do código anterior, usando `sigsuspend()`:

```
sigset_t newmask, oldmask;  
...  
sigemptyset(&newmask);  
sigaddset(&newmask, SIGINT);  
sigprocmask(SIG_BLOCK, &newmask, &oldmask);  
... /* região protegida */  
sigsuspend(&oldmask);  
sigprocmask(SIG_SETMASK, &oldmask, NULL);
```

Implementação do Tratamento de Sinais

(1)

- O kernel necessita manter algum estado na *U area* e na *proc structure*
- A ***U area*** contém informações necessárias para invocar o manipulador de sinais, incluindo:
 - `u_signal []`: Vetor de tratador de sinais para cada sinal
 - `u_sigmask []`: Máscara de sinais bloqueados para cada tratador
- A ***proc structure*** contém campos associados para geração e transferência de sinais, incluindo:
 - `p_cursig`: o sinal corrente sendo tratado
 - `p_sig`: máscara de sinais pendentes.
 - `p_hold`: máscara de sinais bloqueados
 - `p_ignore`: máscara de sinais ignorados

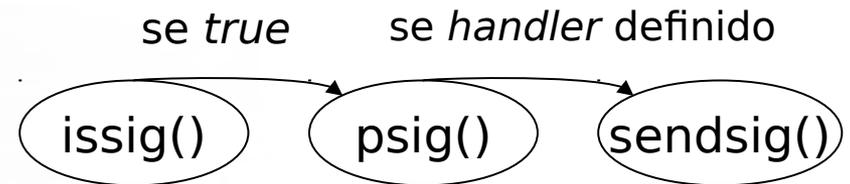
Implementação do Tratamento de Sinais

(2)

- Uma ação de tratamento de um sinal só pode ser executada pelo processo receptor.
- Um processo só pode executar o tratamento quando ele estiver *running*.
- Se, por exemplo, um processo possui baixa prioridade, está suspenso ou bloqueado, pode haver um atraso considerável na execução do tratamento do sinal.
- O processo receptor fica “a par” de um sinal quando o *kernel* chama a função `issig()` (em nome do processo).
 - i.e. o processo está no estado *kernel running*
- Quando `issig()` é chamada?
 - Imediatamente antes de retornar para *user mode* (após uma SVC, interrupção ou exceção)
 - Antes de bloquear o processo em algum evento
 - Após acordar um processo (bloqueado em algum evento)

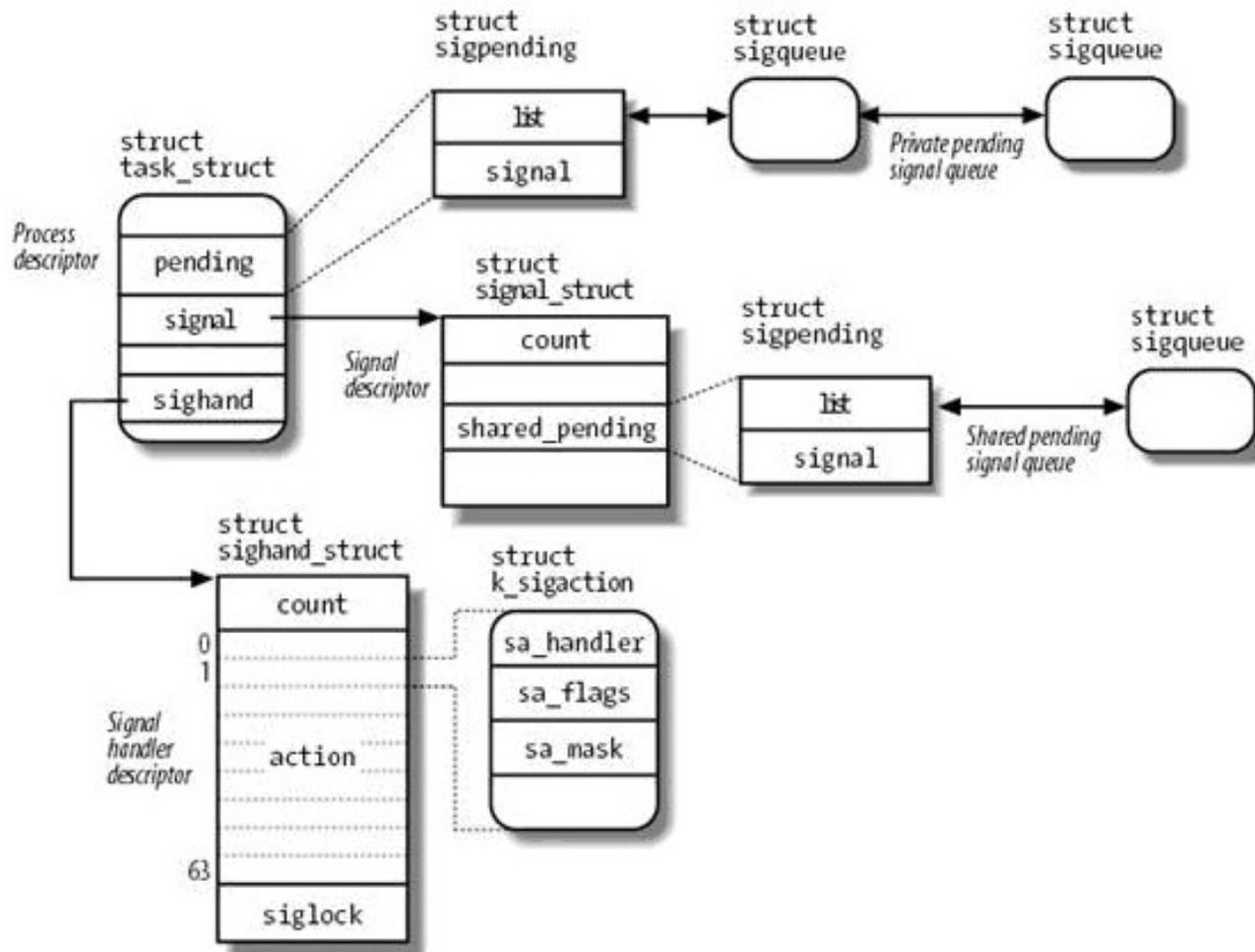
Implementação do Tratamento de Sinais

(2)



- Se `issig()` retornar *true*, o kernel chama `psig()` para despachar o sinal
- `psig()`
 - Termina o processo, gerando *core dump* se requisitado, OU
 - Chama `sendsig()` para invocar o tratador de sinal definido pelo usuário. `sendsig()`:
 - Retorna o processo para *User Mode*
 - Transfere o controle para o tratador de sinal especificado
 - Faz com que após o tratamento do sinal o processo retome a execução normal (de onde foi interrompido)
- Se o sinal chegou quando o processo estava executando uma SVC, geralmente o *kernel* aborta a SVC retornando o código de erro `EINTR`

E no Linux?



Exercício

Vocês devem completar a implementação do programa contido no arquivo `signals.c`. O programa deve contar quantas vezes o usuário envia o sinal SIGINT para o processo em execução. Quando o sinal receber um SIGTSTP (Ctl-Z), ele deve imprimir o número de sinais SIGINT que ele recebeu. Depois de ter recebido 10 SIGINT's, o programa deve “convidar” o usuário a sair (“Really exit (Y/n)?”). Se o usuário não responder em 5 seg., o programa finaliza sozinho.

Referências

- VAHALIA, U. Unix Internals: the new frontiers. Prentice-Hall, 1996.
 - Capítulo 4 (até seção 4.7)
- Deitel H. M.; Deitel P. J.; Choffnes D. R.; "Sistemas Operacionais", 3ª. Edição, Editora Prentice-Hall, 2005
 - Seção 4.7.1
- Silberschatz A. G.; Galvin P. B.; Gagne G.; "Fundamentos de Sistemas Operacionais", 6a. Edição, Editora LTC, 2004.
 - Seção 20.9.1